A Stata Press Publication
StataCorp LLC
College Station, Texas
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase</td>
<td>Erase a disk file</td>
<td>264</td>
</tr>
<tr>
<td>expand</td>
<td>Duplicate observations</td>
<td>266</td>
</tr>
<tr>
<td>expandcl</td>
<td>Duplicate clustered observations</td>
<td>269</td>
</tr>
<tr>
<td>export</td>
<td>Overview of exporting data from Stata</td>
<td>272</td>
</tr>
<tr>
<td>filefilter</td>
<td>Convert ASCII or binary patterns in a file</td>
<td>274</td>
</tr>
<tr>
<td>fillin</td>
<td>Rectangularize dataset</td>
<td>278</td>
</tr>
<tr>
<td>format</td>
<td>Set variables’ output format</td>
<td>280</td>
</tr>
<tr>
<td>frames intro</td>
<td>Introduction to frames</td>
<td>294</td>
</tr>
<tr>
<td>frames</td>
<td>Data frames</td>
<td>306</td>
</tr>
<tr>
<td>frame change</td>
<td>Change identity of current (working) frame</td>
<td>308</td>
</tr>
<tr>
<td>frame copy</td>
<td>Make a copy of a frame</td>
<td>310</td>
</tr>
<tr>
<td>frame create</td>
<td>Create a new frame</td>
<td>312</td>
</tr>
<tr>
<td>frame drop</td>
<td>Drop frame from memory</td>
<td>314</td>
</tr>
<tr>
<td>frame prefix</td>
<td>The frame prefix command</td>
<td>315</td>
</tr>
<tr>
<td>frame put</td>
<td>Copy selected variables or observations to a new frame</td>
<td>317</td>
</tr>
<tr>
<td>frame pwf</td>
<td>Display name of current (working) frame</td>
<td>320</td>
</tr>
<tr>
<td>frame rename</td>
<td>Rename existing frame</td>
<td>322</td>
</tr>
<tr>
<td>frames dir</td>
<td>Display names of all frames in memory</td>
<td>324</td>
</tr>
<tr>
<td>frames reset</td>
<td>Drop all frames from memory</td>
<td>326</td>
</tr>
<tr>
<td>frget</td>
<td>Copy variables from linked frame</td>
<td>328</td>
</tr>
<tr>
<td>frlink</td>
<td>Link frames</td>
<td>335</td>
</tr>
<tr>
<td>generate</td>
<td>Create or change contents of variable</td>
<td>354</td>
</tr>
<tr>
<td>gsort</td>
<td>Ascending and descending sort</td>
<td>360</td>
</tr>
<tr>
<td>hexdump</td>
<td>Display hexadecimal report on file</td>
<td>364</td>
</tr>
<tr>
<td>icd</td>
<td>Introduction to ICD commands</td>
<td>370</td>
</tr>
<tr>
<td>icd9</td>
<td>ICD-9-CM diagnosis codes</td>
<td>377</td>
</tr>
<tr>
<td>icd9p</td>
<td>ICD-9-CM procedure codes</td>
<td>388</td>
</tr>
<tr>
<td>icd10</td>
<td>ICD-10 diagnosis codes</td>
<td>395</td>
</tr>
<tr>
<td>icd10cm</td>
<td>ICD-10-CM diagnosis codes</td>
<td>406</td>
</tr>
<tr>
<td>icd10pcs</td>
<td>ICD-10-PCS procedure codes</td>
<td>418</td>
</tr>
<tr>
<td>import</td>
<td>Overview of importing data into Stata</td>
<td>427</td>
</tr>
<tr>
<td>import dbase</td>
<td>Import and export dBase files</td>
<td>436</td>
</tr>
<tr>
<td>import delimited</td>
<td>Import and export delimited text data</td>
<td>439</td>
</tr>
<tr>
<td>import excel</td>
<td>Import and export Excel files</td>
<td>451</td>
</tr>
<tr>
<td>import fred</td>
<td>Import data from Federal Reserve Economic Data</td>
<td>458</td>
</tr>
<tr>
<td>import haver</td>
<td>Import data from Haver Analytics databases</td>
<td>485</td>
</tr>
<tr>
<td>import sas</td>
<td>Import SAS files</td>
<td>495</td>
</tr>
<tr>
<td>import sasxport5</td>
<td>Import and export data in SAS XPORT Version 5 format</td>
<td>499</td>
</tr>
<tr>
<td>import sasxport8</td>
<td>Import and export data in SAS XPORT Version 8 format</td>
<td>510</td>
</tr>
<tr>
<td>import spss</td>
<td>Import SPSS files</td>
<td>513</td>
</tr>
<tr>
<td>infile (fixed format)</td>
<td>Import text data in fixed format with a dictionary</td>
<td>517</td>
</tr>
<tr>
<td>infile (free format)</td>
<td>Import unformatted text data</td>
<td>536</td>
</tr>
<tr>
<td>infix (fixed format)</td>
<td>Import text data in fixed format</td>
<td>546</td>
</tr>
<tr>
<td>input</td>
<td>Enter data from keyboard</td>
<td>554</td>
</tr>
<tr>
<td>insobs</td>
<td>Add or insert observations</td>
<td>562</td>
</tr>
<tr>
<td>inspect</td>
<td>Display simple summary of data’s attributes</td>
<td>564</td>
</tr>
<tr>
<td>ipolate</td>
<td>Linearly interpolate (extrapolate) values</td>
<td>568</td>
</tr>
<tr>
<td>isid</td>
<td>Check for unique identifiers</td>
<td>571</td>
</tr>
</tbody>
</table>
Contents

j dbc ................................. Load, write, or view data from a database with a Java API 574
joinby ................................. Form all pairwise combinations within groups 585

label ..................................... Manipulate labels 590
label language .......................... Labels for variables and values in multiple languages 600
labelbook ............................... Label utilities 607
list ......................................... List values of variables 618
lookfor .................................. Search for string in variable names and labels 628

memory .................................... Memory management 630
merge ...................................... Merge datasets 636
Missing values .......................... Quick reference for missing values 660
mkdir ...................................... Create directory 661
mvencode ............................... Change missing values to numeric values and vice versa 663
notes ...................................... Place notes in data 667

obs ......................................... Increase the number of observations in a dataset 673
odbc ....................................... Load, write, or view data from ODBC sources 675
order ....................................... Reorder variables in dataset 689
outfile ................................... Export dataset in text format 693
pctile ..................................... Create variable containing percentiles 700
putmata .................................. Put Stata variables into Mata and vice versa 712

range ..................................... Generate numerical range 724
recast ..................................... Change storage type of variable 727
recode .................................... Recode categorical variables 729
rename .................................... Rename variable 738
rename group ................................ Rename groups of variables 740
reshape ................................... Convert data from wide to long form and vice versa 751
rmdir ..................................... Remove directory 769

sample .................................... Draw random sample 771
save ....................................... Save Stata dataset 776
separate .................................. Create separate variables 782
shell ....................................... Temporarily invoke operating system 786
snapshot .................................. Save and restore data snapshots 792
sort ......................................... Sort data 795
split ....................................... Split string variables into parts 804
splitsample ................................ Split data into random samples 809
stack ....................................... Stack data 818
statsby ................................... Collect statistics for a command across a by list 824
sysuse .................................... Use shipped dataset 833
type ....................................... Display contents of a file 836
unicode .................................... Unicode utilities 839
unicode collator ........................ Language-specific Unicode collators 840
unicode convertfile ........................ Low-level file conversion between encodings 842
unicode encoding ........................ Unicode encoding utilities 845
unicode locale .......................... Unicode locale utilities 847
unicode translate ........................ Translate files to Unicode 850
use ........................................ Load Stata dataset 865
Contents

varmanage ........................................ Manage variable labels, formats, and other properties 869
vl ........................................................ Manage variable lists 870
vl create ............................................. Create and modify user-defined variable lists 887
vl drop .................................................. Drop variable lists or variables from variable lists 891
vl list .................................................... List contents of variable lists 894
vl rebuild ............................................. Rebuild variable lists 901
vl set .................................................... Set system-defined variable lists 905
webuse .................................................... Use dataset from Stata website 910
xpose ..................................................... Interchange observations and variables 913
zipfile .................................................. Compress and uncompress files and directories in zip archive format 916
Glossary ...................................................... 918
Subject and author index ........................................ 923
Cross-referencing the documentation

When reading this manual, you will find references to other Stata manuals, for example, [U] 27 Overview of Stata estimation commands; [R] regress; and [D] reshape. The first example is a reference to chapter 27, Overview of Stata estimation commands, in the User’s Guide; the second is a reference to the regress entry in the Base Reference Manual; and the third is a reference to the reshape entry in the Data Management Reference Manual.

All the manuals in the Stata Documentation have a shorthand notation:

[GSM] Getting Started with Stata for Mac
[GSU] Getting Started with Stata for Unix
[GSW] Getting Started with Stata for Windows
[U] Stata User’s Guide
[R] Stata Base Reference Manual
[BAYES] Stata Bayesian Analysis Reference Manual
[FN] Stata Functions Reference Manual
[XT] Stata Longitudinal-Data/Panel-Data Reference Manual
[M] Stata Multiple-Imputation Reference Manual
[SVY] Stata Survey Data Reference Manual
[TABLES] Stata Customizable Tables and Collected Results Reference Manual
[I] Stata Index

Following this entry, [D] *Data management* provides an overview of data management in Stata and of Stata’s data management commands. The other parts of this manual are arranged alphabetically. If you are new to Stata’s data management features, we recommend that you read the following first:

[D]  *Data management* — Introduction to data management commands
[U]  12 Data
[U]  13 Functions and expressions
[U]  11.5 by varlist: construct
[U]  22 Entering and importing data
[U]  23 Combining datasets
[U]  24 Working with strings
[U]  26 Working with categorical data and factor variables
[U]  25 Working with dates and times
[U]  16 Do-files

You can see that most of the suggested reading is in [U]. That is because [U] provides overviews of most Stata features, whereas this is a reference manual and provides details on the usage of specific commands. You will get an overview of features for combining data from [U] 23 *Combining datasets*, but the details of performing a match-merge (merging the records of two files by matching the records on a common variable) will be found here, in [D] *merge*.

Stata is continually being updated, and Stata users are always writing new commands. To ensure that you have the latest features, you should install the most recent official update; see [R] *update*.

Also see

[U]  1.3 What’s new
[R]  *Intro* — Introduction to base reference manual
This manual, called [D], documents Stata’s data management features. See Mitchell (2020) for additional information and examples on data management in Stata.

Data management for statistical applications refers not only to classical data management—sorting, merging, appending, and the like—but also to data reorganization because the statistical routines you will use assume that the data are organized in a certain way. For example, statistical commands that analyze longitudinal data, such as `xtreg`, generally require that the data be in long rather than wide form, meaning that repeated values are recorded not as extra variables, but as extra observations.

Here are the basics everyone should know:

- **`use`** : Load Stata dataset
- **`save`** : Save Stata dataset
- **`describe`** : Describe data in memory or in file
- **`codebook`** : Describe data contents
- **`inspect`** : Display simple summary of data’s attributes
- **`count`** : Count observations satisfying specified conditions
- **`Data types`** : Quick reference for data types
- **`Missing values`** : Quick reference for missing values
- **`Datetime`** : Date and time values and variables
- **`list`** : List values of variables
- **`edit`** : Browse or edit data with Data Editor
- **`varmanage`** : Manage variable labels, formats, and other properties
- **`rename`** : Rename variable
- **`format`** : Set variables’ output format
- **`label`** : Manipulate labels
- **`frames intro`** : Introduction to frames
To work with multiple datasets in memory, see

[D] **frames intro**  Introduction to frames
[D] **frames**  Data frames
[D] **frame change**  Change identity of current (working) frame
[D] **frame copy**  Make a copy of a frame
[D] **frame create**  Create a new frame
[D] **frame drop**  Drop frame from memory
[D] **frame prefix**  The frame prefix command
[D] **frame put**  Copy selected variables or observations to a new frame
[D] **frame pwf**  Display name of current (working) frame
[D] **frame rename**  Rename existing frame
[D] **frames dir**  Display names of all frames in memory
[D] **frames reset**  Drop all frames from memory
[D] **frget**  Copy variables from linked frame
[D] **frlink**  Link frames

You will need to create and drop variables, and here is how:

[D] **generate**  Create or change contents of variable
[D] **egen**  Extensions to generate
[D] **drop**  Drop variables or observations
[D] **clear**  Clear memory
For inputting or importing data, see

[D] use
    Load Stata dataset

[D] sysuse
    Use shipped dataset

[D] webuse
    Use dataset from Stata website

[D] input
    Enter data from keyboard

[D] import
    Overview of importing data into Stata

[D] import dbase
    Import and export dBase files

[D] import delimited
    Import and export delimited text data

[D] import excel
    Import and export Excel files

[D] import fred
    Import data from Federal Reserve Economic Data

[D] import haver
    Import data from Haver Analytics databases

[D] import sas
    Import SAS files

[D] import sasxport5
    Import and export data in SAS XPORT Version 5 format

[D] import sasxport8
    Import and export data in SAS XPORT Version 8 format

[D] import spss
    Import SPSS files

[D] infile (fixed format)
    Import text data in fixed format with a dictionary

[D] infile (free format)
    Import unformatted text data

[D] infix (fixed format)
    Import text data in fixed format

[D] jdbc
    Load, write, or view data from a database with a Java API

[D] odbc
    Load, write, or view data from ODBC sources

[D] hexdump
    Display hexadecimal report on file

[D] icd9
    ICD-9-CM diagnosis codes

[D] icd9p
    ICD-9-CM procedure codes

[D] icd10
    ICD-10 diagnosis codes

[D] icd10cm
    ICD-10-CM diagnosis codes

[D] icd10pcs
    ICD-10-PCS procedure codes

and for exporting data, see

[D] save
    Save Stata dataset

[D] export
    Overview of exporting data from Stata

[D] outfile
    Export dataset in text format

[D] import dbase
    Import and export dBase files

[D] import delimited
    Import and export delimited text data

[D] import excel
    Import and export Excel files

[D] import sasxport5
    Import and export data in SAS XPORT Version 5 format

[D] import sasxport8
    Import and export data in SAS XPORT Version 8 format

[D] jdbc
    Load, write, or view data from a database with a Java API

[D] odbc
    Load, write, or view data from ODBC sources

The ordering of variables and observations (sort order) can be important; see

[D] order
    Reorder variables in dataset

[D] sort
    Sort data

[D] gsort
    Ascending and descending sort
To reorganize or combine data, see

[D] append  
Append datasets

[D] merge  
Merge datasets

[D] fmlink  
Link frames

[D] fmerge  
Copy variables from linked frame

[D] reshape  
Convert data from wide to long form and vice versa

[D] collapse  
Make dataset of summary statistics

[D] contract  
Make dataset of frequencies and percentages

[D] fillin  
Rectangularize dataset

[D] expand  
Duplicate observations

[D] expandcl  
Duplicate clustered observations

[D] stack  
Stack data

[D] joinby  
Form all pairwise combinations within groups

[D] xpose  
Interchange observations and variables

[D] cross  
Form every pairwise combination of two datasets

In the above list, we particularly want to direct your attention to [D] reshape, a useful command that beginners often overlook.

For random sampling, see

[D] sample  
Draw random sample

[D] splitsample  
Split data into random samples

[D] drawnorm  
Draw sample from multivariate normal distribution

For file manipulation, see

[D] type  
Display contents of a file

[D] erase  
Erase a disk file

[D] copy  
Copy file from disk or URL

[D] cd  
Change directory

[D] dir  
Display filenames

[D] mkdir  
Create directory

[D] rmdir  
Remove directory

[D] cf  
Compare two datasets

[D] changeeol  
Convert end-of-line characters of text file

[D] filefilter  
Convert ASCII or binary patterns in a file

[D] checksum  
Calculate checksum of file

[D] zipfile  
Compress and uncompress files and directories in zip archive format
For handling Unicode strings, see

[D]  **unicode**  
Universe utilities

[D]  **unicode translate**  
Translate files to Unicode

[D]  **unicode encoding**  
Unicode encoding utilities

[D]  **unicode locale**  
Unicode locale utilities

[D]  **unicode collator**  
Language-specific Unicode collators

[D]  **unicode convertfile**  
Low-level file conversion between encoding

The entries above are important. The rest are useful when you need them:

[D]  **datasignature**  
Determine whether data have changed

[D]  **type**  
Display contents of a file

[D]  **notes**  
Place notes in data

[D]  **label language**  
Labels for variables and values in multiple languages

[D]  **labelbook**  
Label utilities

[D]  **encode**  
Encode string into numeric and vice versa

[D]  **recode**  
Recode categorical variables

[D]  **ipolate**  
Linearly interpolate (extrapolate) values

[D]  **destring**  
Convert string variables to numeric variables and vice versa

[D]  **mvencode**  
Change missing values to numeric values and vice versa

[D]  **ptile**  
Create variable containing percentiles

[D]  **range**  
Generate numerical range

[D]  **by**  
Repeat Stata command on subsets of the data

[D]  **statsby**  
Collect statistics for a command across a by list

[D]  **dyngen**  
Dynamically generate new values of variables

[D]  **compress**  
Compress data in memory

[D]  **recast**  
Change storage type of variable

[D]  **Datetime display formats**  
Display formats for dates and times

[D]  **Datetime conversion**  
String to numeric date conversion functions

[D]  **Datetime durations**  
Obtaining and working with durations

[D]  **Datetime relative dates**  
Datetime relative dates

[D]  **Datetime values from other software**  
Date and time conversion from other software

[D]  **bcal**  
Business calendar file manipulation

[D]  **Datetime business calendars**  
Business calendars

[D]  **Datetime business calendars creation**  
Business calendars creation
[D] assert Verify truth of claim
[D] assertnested Verify variables nested
[D] clonevar Clone existing variable
[D] compare Compare two variables
[D] corr2data Create dataset with specified correlation structure
[D] ds Compactly list variables with specified properties
[D] duplicates Report, tag, or drop duplicate observations
[D] insobs Add or insert observations
[D] isid Check for unique identifiers
[D] lookfor Search for string in variable names and labels
[D] memory Memory management
[D] putmata Put Stata variables into Mata and vice versa
[D] obs Increase the number of observations in a dataset
[D] rename group Rename groups of variables
[D] separate Create separate variables
[D] shell Temporarily invoke operating system
[D] snapshot Save and restore data snapshots
[D] split Split string variables into parts
[D] vl Manage variable lists
[D] vl create Create and modify user-defined variable lists
[D] vl drop Drop variable lists or variables from variable lists
[D] vl list List contents of variable lists
[D] vl rebuild Rebuild variable lists
[D] vl set Set system-defined variable lists

There are some real jewels in the above, such as [D] notes, [D] compress, and [D] assert, which you will find particularly useful.

References

Also see
[D] Intro — Introduction to data management reference manual
[R] Intro — Introduction to base reference manual
append — Append datasets

Description

append appends Stata-format datasets stored on disk to the end of the dataset in memory. If any filename is specified without an extension, .dta is assumed.

Stata can also join observations from two datasets into one; see [D] merge. See [U] 23 Combining datasets for a comparison of append, merge, and joinby.

Quick start

Append mydata2.dta to mydata1.dta with no data in memory
   append using mydata1 mydata2

As above, but with mydata1.dta in memory
   append using mydata2

As above, and generate newv to indicate source dataset
   append using mydata2, generate(newv)

As above, but do not copy value labels or notes from mydata2.dta
   append using mydata2, generate(newv) nolabel nonotes

Only keep v1, v2, and v3 from mydata2.dta
   append using mydata2, keep(v1 v2 v3)

Menu

Data > Combine datasets > Append datasets
Syntax

```
append using filename [ filename ... ] [ , options ]
```

You may enclose `filename` in double quotes and must do so if `filename` contains blanks or other special characters.

<table>
<thead>
<tr>
<th><code>options</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generate(newvar)</code></td>
<td><code>newvar</code> marks source of resulting observations</td>
</tr>
<tr>
<td><code>keep(varlist)</code></td>
<td>keep specified variables from appending dataset(s)</td>
</tr>
<tr>
<td><code>nolabel</code></td>
<td>do not copy value-label definitions from dataset(s) on disk</td>
</tr>
<tr>
<td><code>nonotes</code></td>
<td>do not copy notes from dataset(s) on disk</td>
</tr>
<tr>
<td><code>force</code></td>
<td>append string to numeric or numeric to string without error</td>
</tr>
</tbody>
</table>

Options

- `generate(newvar)` specifies the name of a variable to be created that will mark the source of observations. Observations from the master dataset (the data in memory before the `append` command) will contain 0 for this variable. Observations from the first using dataset will contain 1 for this variable; observations from the second using dataset will contain 2 for this variable; and so on.

- `keep(varlist)` specifies the variables to be kept from the using dataset. If `keep()` is not specified, all variables are kept.

  The `varlist` in `keep(varlist)` differs from standard Stata varlists in two ways: variable names in `varlist` may not be abbreviated, except by the use of wildcard characters, and you may not refer to a range of variables, such as `price-weight`.

- `nolabel` prevents Stata from copying the value-label definitions from the disk dataset into the dataset in memory. Even if you do not specify this option, label definitions from the disk dataset never replace definitions already in memory.

- `nonotes` prevents notes in the using dataset from being incorporated into the result. The default is to incorporate notes from the using dataset that do not already appear in the master data.

- `force` allows string variables to be appended to numeric variables and vice versa, resulting in missing values from the using dataset. If omitted, `append` issues an error message; if specified, `append` issues a warning message.

Remarks and examples

The disk dataset must be a Stata-format dataset; that is, it must have been created by `save` (see [D] save).

Example 1

We have two datasets stored on disk that we want to combine. The first dataset, called `even.dta`, contains the sixth through eighth positive even numbers. The second dataset, called `odd.dta`, contains the first five positive odd numbers. The datasets are
. use even
(6th through 8th even numbers)
. list

<table>
<thead>
<tr>
<th>number</th>
<th>even</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>6</td>
</tr>
<tr>
<td>2.</td>
<td>7</td>
</tr>
<tr>
<td>3.</td>
<td>8</td>
</tr>
</tbody>
</table>

. use odd
(First five odd numbers)
. list

<table>
<thead>
<tr>
<th>number</th>
<th>odd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
</tr>
</tbody>
</table>

We will append the even data to the end of the odd data. Because the odd data are already in memory (we just used them above), we type append using even. The result is

. append using even
. list

<table>
<thead>
<tr>
<th>number</th>
<th>odd</th>
<th>even</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>7.</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>8.</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

Because the number variable is in both datasets, the variable was extended with the new data from the file even.dta. Because there is no variable called odd in the new data, the additional observations on odd were forward-filled with missing (.). Because there is no variable called even in the original data, the first observations on even were back-filled with missing.
Example 2

The order of variables in the two datasets is irrelevant. Stata always appends variables by name:

```
. use https://www.stata-press.com/data/r17/odd1
(First five odd numbers)
. describe
Contains data from https://www.stata-press.com/data/r17/odd1.dta
 Observations: 5  First five odd numbers
 Variables: 2  9 Jan 2020 08:41
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>odd</td>
<td>float</td>
<td>%9.0g</td>
<td>Odd numbers</td>
</tr>
<tr>
<td>number</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: number

```
. describe using https://www.stata-press.com/data/r17/even
Contains data 6th through 8th even numbers
Observations: 3  9 Jan 2020 08:43
Variables: 2
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>byte</td>
<td>%9.0g</td>
<td></td>
</tr>
<tr>
<td>even</td>
<td>float</td>
<td>%9.0g</td>
<td>Even numbers</td>
</tr>
</tbody>
</table>

Sorted by: number

```
. append using https://www.stata-press.com/data/r17/even
```

```
. list
  odd   number   even
1. 1     1           .
2. 3     2           .
3. 5     3           .
4. 7     4           .
5. 9     5           .
6. .     6           12
7. .     7           14
8. .     8           16
```

The results are the same as those in the first example.

When Stata appends two datasets, the definitions of the dataset in memory, called the master dataset, override the definitions of the dataset on disk, called the using dataset. This extends to value labels, variable labels, characteristics, and date–time stamps. If there are conflicts in numeric storage types, the more precise storage type will be used regardless of whether this storage type was in the master dataset or the using dataset. If a variable is stored as a string in one dataset that is longer than in the other, the longer str# storage type will prevail. If a variable is stored as a strL in one dataset and a str# in another dataset, the strL storage type will prevail.
Technical note

If a variable is a string in one dataset and numeric in the other, Stata issues an error message unless the `force` option is specified. If `force` is specified, Stata issues a warning message before appending the data. If the using dataset contains the string variable, the combined dataset will have numeric missing values for the appended data on this variable; the contents of the string variable in the using dataset are ignored. If the using dataset contains the numeric variable, the combined dataset will have empty strings for the appended data on this variable; the contents of the numeric variable in the using dataset are ignored.

Example 3

Because Stata has five numeric variable types—byte, int, long, float, and double—you may attempt to append datasets containing variables with the same name but of different numeric types; see [U] 12.2.2 Numeric storage types.

Let’s describe the datasets in the example above:

```
. describe using https://www.stata-press.com/data/r17/odd
Contains data First five odd numbers
Observations: 5 9 Jan 2020 08:50
Variables: 2

  Variable Storage Display Value Variable label
name type format label               
  number float %9.0g
  odd    float %9.0g
```

Sorted by:

```
. describe using https://www.stata-press.com/data/r17/even
Contains data 6th through 8th even numbers
Observations: 3 9 Jan 2020 08:43
Variables: 2

  Variable Storage Display Value Variable label
name type format label               
  number byte %9.0g
  even   float %9.0g
```

Sorted by: number

```
. describe using https://www.stata-press.com/data/r17/oddeven
Contains data First five odd numbers
Observations: 8 9 Jan 2020 08:53
Variables: 3

  Variable Storage Display Value Variable label
name type format label               
  number float %9.0g
  odd    float %9.0g
  even   float %9.0g
```

Sorted by:
The number variable was stored as a float in odd.dta but as a byte in even.dta. Because float is the more precise storage type, the resulting dataset, oddeven.dta, had number stored as a float. Had we instead appended odd.dta to even.dta, number would still have been stored as a float:

```
. use https://www.stata-press.com/data/r17/even, clear
(6th through 8th even numbers)
. append using https://www.stata-press.com/data/r17/odd
(variable number was byte, now float to accommodate using data’s values)
. describe
Contains data from https://www.stata-press.com/data/r17/even.dta
Observations: 8 6th through 8th even numbers
Variables: 3 9 Jan 2020 08:43
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
</tr>
<tr>
<td>even</td>
<td>float</td>
<td>%9.0g</td>
<td>Even numbers</td>
</tr>
<tr>
<td>odd</td>
<td>float</td>
<td>%9.0g</td>
<td>Odd numbers</td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.

Example 4

Suppose that we have a dataset in memory containing the variable educ, and we have previously given a label variable educ "Education Level" command so that the variable label associated with educ is “Education Level”. We now append a dataset called newdata.dta, which also contains a variable named educ, except that its variable label is “Ed. Lev”. After appending the two datasets, the educ variable is still labeled “Education Level”. See [U] 12.6.2 Variable labels.

Example 5

Assume that the values of the educ variable are labeled with a value label named educlbl. Further assume that in newdata.dta, the values of educ are also labeled by a value label named educlbl. Thus there is one definition of educlbl in memory and another (although perhaps equivalent) definition in newdata.dta. When you append the new data, you will see the following:

```
. append using newdata
label educlbl already defined
```

If one label in memory and another on disk have the same name, append warns you of the problem and sticks with the definition currently in memory, ignoring the definition in the disk file.

Technical note

When you append two datasets that both contain definitions of the same value label, the codings may not be equivalent. That is why Stata warns you with a message like “label educlbl already defined”. If you do not know that the two value labels are equivalent, you should convert the value-labeled variables into string variables, append the data, and then construct a new coding. `decode` and `encode` make this easy:
. use newdata, clear
. decode educ, gen(edstr)
. drop educ
. save newdata, replace
. use basedata
. decode educ, gen(edstr)
. drop educ
. append using newdata
. encode edstr, gen(educ)
. drop edstr

See [D] encode.

You can specify the nolabel option to force append to ignore all the value-label definitions in the incoming file, whether or not there is a conflict. In practice, you will probably never want to do this.

Example 6

Suppose that we have several datasets containing the populations of counties in various states. We can use append to combine these datasets all at once and use the generate() option to create a variable identifying from which dataset each observation originally came.

. use https://www.stata-press.com/data/r17/capop
. list county pop

<table>
<thead>
<tr>
<th>county</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>9878554</td>
</tr>
<tr>
<td>Orange</td>
<td>2997033</td>
</tr>
<tr>
<td>Ventura</td>
<td>798364</td>
</tr>
</tbody>
</table>

. append using https://www.stata-press.com/data/r17/ilpop
> https://www.stata-press.com/data/r17/txpop, generate(state)
. label define statelab 0 "CA" 1 "IL" 2 "TX"
. label values state statelab
## Video example

How to append files into a single dataset

## Reference


## Also see

[D] **cross** — Form every pairwise combination of two datasets  
[D] **joinby** — Form all pairwise combinations within groups  
[D] **merge** — Merge datasets  
[D] **save** — Save Stata dataset  
[D] **use** — Load Stata dataset  
[U] 23 Combining datasets
assert — Verify truth of claim

Description

assert verifies that \( exp \) is true. If it is true, the command produces no output. If it is not true, assert informs you that the “assertion is false” and issues a return code of 9; see [U] 8 Error messages and return codes.

Quick start

Confirm that \( v1 \) only takes values 0 or 1

\[
\text{assert } v1==0 \mid v1==1
\]

Verify that \( v2 \) is between 100 and 200 and never missing

\[
\text{assert inrange}(v2,100,200)
\]

Verify that \( v2 \) is between 100 and 200 for all nonmissing values

\[
\text{assert inrange}(v2,100,200) \text{ if } !\text{missing}(v2)
\]

Verify that \( v2 \) is between 100 and 200 and never missing when \( \text{catvar} \) equals 2 or 3

\[
\text{assert inrange}(v2,100,200) \text{ if } (\text{catvar}==2 \mid \text{catvar}==3)
\]

Verify that there are 5 observations per cluster identified by \( \text{cvar} \)

\[
\text{by } \text{cvar}: \text{assert } _N==5
\]

As above, but stop checking after the first cluster has fewer than or more than 5 observations

\[
\text{by } \text{cvar}: \text{assert } _N==5, \text{ fast}
\]
Syntax

    assert exp [if] [in] [ , rc0 null fast ]

by is allowed; see [D] by.

Options

rc0 forces a return code of 0, even if the assertion is false.

null forces a return code of 8 on null assertions. A null assertion occurs when an if condition
excludes all observations from being checked by assert. By default, the return code is 0 for null
assertions.

fast forces the command to exit at the first occurrence that exp evaluates to false.

Remarks and examples

assert verifies that the expression provided is true. It is useful because it tells Stata not only
what to do but also what you can expect to find. Groups of assertions are often combined in a do-file
to certify data. If the do-file runs all the way through without complaining, every assertion in the file
is true. Otherwise, assert will provide a count of the contradictions when an assertion is false. It
will also issue an error message along with a return code of 9; see [U] 8 Error messages and return
codes.

assert is seldom used interactively because it is easier to use inspect, summarize, or tabulate
to look for evidence of errors in the dataset. These commands, however, require you to review the
output to spot the error.

Example 1: Observation-level assertions

You and a colleague are analyzing union membership among women. Your colleague imported
data from the National Longitudinal Survey of young women for the years 1968 to 1988. You plan
to include the woman’s age, total work experience, and whether or not she graduated from college
in your model.

Your colleague tells you that the cleaned dataset is called nlswork and that the following things
are true: that the variables recording union membership, age, total experience, and education level
are not missing for any of the observations; that observations taken before a woman turned 18 have
been removed; that total experience is always greater than or equal to 0; and that all college graduates
have at least 14 years of education. Before you begin your analysis, you should verify the accuracy
of these data. To test that the statements above are true, you create a do-file named check.do:

    begin check.do, example 1
    assert age>=18 & !missing(age)
    assert !missing(union)
    assert ttl_exp>=0 & !missing(ttl_exp)
    assert grade>=14 & !missing(grade) if collgrad==1
    end check.do, example 1

You save the above file, read in the data, and then issue the do command to check the assertions:

    . use https://www.stata-press.com/data/r17/nlswork
    (National Longitudinal Survey of Young Women, 14-24 years old in 1968)
    . do check
The output is as follows:

```
. assert age>=18 & !missing(age)
159 contradictions in 28,534 observations
assertion is false
r(9);
end of do-file
r(9);
```

The do-file did not run to completion because it encountered a false assertion—that age is never missing and always at least 18 years.

You should resolve this and any other discrepancies before analyzing the data. You run the do-file again, this time with the `nostop` option, which tells Stata to continue executing the do-file despite any errors.

```
. do check, nostop
```

Once it runs in its entirety, you will have a list of all the data discrepancies to discuss with your colleague. The output is as follows:

```
. assert age>=18 & !missing(age)
159 contradictions in 28,534 observations
assertion is false
r(9);
. assert !missing(union)
9,296 contradictions in 28,534 observations
assertion is false
r(9);
. assert ttl_exp>=0 & !missing(ttl_exp)
. assert grade>=14 & !missing(grade) if collgrad==1
42 contradictions in 4,795 observations
assertion is false
r(9);
```

The output from the false assertions above is helpful. First, the number of contradictions can serve as a clue; a few contradictions may suggest data entry errors, whereas a large number may motivate further investigation. Second, you get a straightforward message that the assertion is false. Finally, you get a return code of 9, which makes it easy to write code based on whether or not an assertion is true.

### Example 2: Speeding up assert

In example 1, we obtained a count of the number of observations where each assertion was false. However, if all you wanted to know was whether or not an assertion was true, you could reduce the amount of time required to check that assertion by specifying the `fast` option, as shown below:

```
. assert age>=18 & !missing(age), fast
assertion is false
r(9);
```

The `fast` option tells Stata to stop checking the assertion when it encounters the first case where it is false, which is why you do not get a count of the contradictions.
Example 3: Assertions by groups

Your assertions in the previous examples were tested in each observation. You spoke with your colleague regarding those assertions, and she has sent you a revised version of the dataset. The next goal is to make sure that age has been recorded correctly over time. Women in the study were observed once per year, and in some years, they were not observed at all. Therefore, you know that age must be increasing with every time period.

Thus, now you want to assess the characteristics of each woman over time, and you can do so with the by: prefix. You include the sort option with the by prefix because the data have not been sorted by woman (idcode) and year already; see [U] 11.5 by varlist: construct. Now you can assert that for each woman, the value of age is greater than it was in the previous year for all years except the first.

You add the following line to check.do:

```
begin check.do, example 3
by idcode (year), sort: assert age>=age[_n-1]+1 if _n>1
end check.do, example 3
```

Upon reissuing the the do check, nostop command, the following output is shown:

```
. by idcode (year), sort: assert age>=age[_n-1]+1 if _n>1
  171 contradictions in 23,823 observations
  assertion is false
  r(9);
.
end of do-file
```

Again, we have found a few errors in the dataset. We might want to check the source of the dataset for any notes on data discrepancies.

Technical note

assert is smart in how it evaluates expressions. When you type something like `assert _N==522` or `assert work[_N]>0`, assert knows that the expression needs to be evaluated only once. When you type `assert female==1 | female==0`, assert knows that the expression needs to be evaluated once for each observation in the dataset.

Here are some more examples demonstrating assert’s intelligence.

```
by female: assert _N==100
```

asserts that there should be 100 observations for every unique value of female. The expression is evaluated once per by-group.

```
by female: assert work[_N]>0
```

asserts that the last observation on work in every by-group should be greater than zero. It is evaluated once per by-group.

```
by female: assert work>0
```

is evaluated once for each observation in the dataset and, in that sense, is formally equivalent to `assert work>0`. 
Reference


Also see

[D] assertnested — Verify variables nested
[P] capture — Capture return code
[P] confirm — Argument verification
[U] 16 Do-files
assertnested — Verify variables nested

**Description**

assertnested verifies that the values of variables are nested within the values of other variables. If they are nested, the command produces no output. If they are not nested, assertnested informs you that they are not and issues an error return code of 459; see [U] 8 Error messages and return codes.

**Quick start**

Confirm that the values of `psu` are nested within `stratum`

```
assertnested stratum psu
```

Confirm that the values of IDs in `student` are nested within `school`, which is nested within `district`

```
assertnested district school student
```

For panel data, where panels are individuals with IDs stored in `panelid`, check that values of `age` and `income` are the same for all observations in each panel

```
assertnested panelid, within(age income)
```

As above, but treat any missing values the same as nonmissing values

```
assertnested panelid, within(age income) missing
```

**Syntax**

```
assertnested varlist [if] [in] [, within(withinvars) missing]
```

The variables in `varlist` are given in the order of biggest grouping to smallest grouping.

`by` is allowed; see [D] by.

**Options**

`within(withinvars)` asserts that the values of `varlist` are nested within each of the variables in `withinvars`. That is, `assertnested varlist, within(w1 w2 ...)` will issue an error if any of `assertnested w1 varlist, assertnested w2 varlist, ...` issue an error.

`missing` specifies that missing values in `varlist` and `withinvars` are to be treated the same as nonmissing values.
assertnested is a convenience command for checking whether variables are nested. We say that \( v2 \) is nested within \( v1 \) if for all observations that have the same value of \( v2 \), the observations also have the same value of \( v1 \).

Here are data that are nested.

```
. list v1 v2, sepby(v1)

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
. assertnested v1 v2
```

assertnested succeeds.

Here are data that are not nested.

```
. list v1 v3, sepby(v1)

<table>
<thead>
<tr>
<th>v1</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
. assertnested v1 v3
```

```
v3 not nested within v1
r(459);
```

assertnested fails.

Running

```
assertnested v1 v2 v3
```

is the same as running

```
assertnested v1 v2
assertnested v2 v3
```

Variables must be specified with the biggest nested grouping first, then the second biggest nested grouping, and so on, to the smallest nested grouping.
Example 1: Nested variables

We have a dataset consisting of two school districts in Texas: the district for the city of College Station and the district for the city of Richardson. The dataset contains the actual names of all the public schools in the variable `school` in these districts, given by variable `district`. The dataset contains fictitious student IDs in the variable `student`.

We want to assert that `student` is nested within `school` and that `school` is nested within `district`.

```
use https://www.stata-press.com/data/r17/schools
assertnested district school student
```

Schools are not nested within district! Are some schools in both districts? That is impossible. But it is possible that both districts have one or more schools with the same name. Let’s find them.

We use `egen`’s `tag()` function to tag one observation for each distinct value of `district` for each school. Then we sum up the number of tags in each school. If the schools were nested within district, there would be only one tag per school. We list the districts and schools with more than one tag.

```
egen tag_district = tag(school district)
bysort school: egen ndistrict = sum(tag_district)
list district school if tag_district == 1 & ndistrict > 1, noobs
```

<table>
<thead>
<tr>
<th>district</th>
<th>school</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richardson</td>
<td>Spring Creek Elementary School</td>
</tr>
<tr>
<td>College Station</td>
<td>Spring Creek Elementary School</td>
</tr>
</tbody>
</table>

Both College Station and Richardson have schools named Spring Creek Elementary School. If we want to check that students are nested within schools, we need to do the check separately by district.

```
bysort district: assertnested school student
```

Or else Texans need to get more creative about naming their schools.

Example 2: Variables constant within panels

Commands that work with panel data in Stata require the data to be in long form. That is, multiple Stata observations for each panel. Saying a variable is constant within each panel is the same as saying the panels are nested within that variable. `assertnested` allows you to assert that variables are constant within each panel.

We illustrate this with choice model data. Choice model data are stored like panel data in that each individual has multiple observations, one for each possible choice. Characteristics of the individual should be constant across observations for an individual.

We load a dataset with consumer choices for purchasing a new car (see [CM] Intro 2 for a description of these data). Then we check that `gender` and `income` are constant for the observations with the same `consumerid` by using the `within()` option.
. use https://www.stata-press.com/data/r17/carchoice, clear
   (Car choice data)
. assertnested consumerid, within(gender income)

The within() option is a convenient way to do multiple assertions. The above is the same as running

. assertnested gender consumerid
. assertnested income consumerid

The option missing can be specified to treat missing values the same as any other value.

. assertnested consumerid, within(gender income) missing
   consumerid not nested within gender
   r(459);

We see that gender is not constant for some consumers when we treat missing values like any other value. Let’s list one person who has missing values for gender:

. list consumerid gender if consumerid == 142, abbrev(10)

<table>
<thead>
<tr>
<th>consumerid</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>509.</td>
<td>142</td>
</tr>
<tr>
<td>510.</td>
<td>142</td>
</tr>
<tr>
<td>511.</td>
<td>142</td>
</tr>
<tr>
<td>512.</td>
<td>142</td>
</tr>
</tbody>
</table>

This person has a missing value for gender for one observation and nonmissing values for other observations. For the data to pass assertnested with the option missing, the variable would have to be either all missing or all nonmissing (and the same value) for each individual.

Also see

[D] assert — Verify truth of claim
[CM] Intro 2 — Data layout
[P] capture — Capture return code
[SVY] Survey — Introduction to survey commands
[XT] xt — Introduction to xt commands
[U] 16 Do-files
**bcal — Business calendar file manipulation**

**Description**

See [D] Datetime business calendars for an introduction to business calendars and dates.

`bcal check` lists the business calendars used by the data in memory, if any.

`bcal dir pattern` lists filenames and directories of all available business calendars matching `pattern`, or all business calendars if `pattern` is not specified.

`bcal describe calname` presents a description of the specified business calendar.

`bcal load calname` loads the specified business calendar. Business calendars load automatically when needed, and thus use of `bcal load` is never required. `bcal load` is used by programmers writing their own business calendars. `bcal load calname` forces immediate loading of a business calendar and displays output, including any error messages due to improper calendar construction.

`bcal create filename, from(varname)` creates a business calendar file based on dates in `varname`. Business holidays are inferred from gaps in `varname`. The qualifiers `if` and `in`, as well as the option `excludemissing()`, can also be used to exclude dates from the new business calendar.

**Quick start**

Create business calendar file `mycal.stbcal` from date variable `tvar` in the dataset in memory

`bcal create mycal, from(tvar)`

As above, and generate business date variable `newt` formatted as `%tbmycal`

`bcal create mycal, from(tvar) generate(newt)`

List directories and filenames of available business calendars

`bcal dir`

Describe range, center date, and number of omitted days in business calendar `mycal.stbcal`

`bcal describe mycal`

Report any `%tb` formats applied to the variables in memory

`bcal check`

**Menu**

Data > Other utilities > Create a business calendar

Data > Other utilities > Manage business calendars

Data > Variables Manager
Syntax

List business calendars used by the data in memory

```
bcal check [varlist] [ , rc0]
```

List filenames and directories of available business calendars

```
bcal dir [pattern]
```

Describe the specified business calendar

```
bcal describe calname
```

Load the specified business calendar

```
bcal load calname
```

Create a business calendar from the current dataset

```
bcal create filename [ if ] [ in ] , from(varname) [ bcal_create_options ]
```

where

- `varlist` is a list of variable names to be checked for whether they use business calendars. If not specified, all variables are checked.
- `pattern` is the name of a business calendar possibly containing wildcards * and ?. If `pattern` is not specified, all available business calendar names are listed.
- `calname` is the name of a business calendar either as a name or as a datetime format; for example, `calname` could be `simple` or `%tbsimple`.
- `filename` is the name of the business calendar file created by `bcal create`.

### bcal_create_options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
<td></td>
</tr>
<tr>
<td>*from(varname)</td>
<td>specify date variable for calendar</td>
</tr>
<tr>
<td>generate(newvar)</td>
<td>generate <code>newvar</code> containing business dates</td>
</tr>
<tr>
<td>excludemissing(varlist [, any])</td>
<td>exclude observations with missing values in <code>varlist</code></td>
</tr>
<tr>
<td>personal</td>
<td>save calendar file in your <code>PERSONAL</code> directory</td>
</tr>
<tr>
<td>replace</td>
<td>replace file if it already exists</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
</tr>
<tr>
<td>purpose(text)</td>
<td>describe purpose of calendar</td>
</tr>
<tr>
<td>dateformat(ymd</td>
<td>specify date format in calendar file</td>
</tr>
<tr>
<td>ydm</td>
<td>specify range of calendar</td>
</tr>
<tr>
<td>mdy</td>
<td>specify center date of calendar</td>
</tr>
<tr>
<td>dmy</td>
<td>specify maximum gap allowed; default is 10 days</td>
</tr>
<tr>
<td>range(fromdate todate)</td>
<td>specify range of calendar</td>
</tr>
<tr>
<td>centerdate(date)</td>
<td>specify center date of calendar</td>
</tr>
</tbody>
</table>

*from(varname) is required.

collect is allowed with all `bcal` commands; see [U] 11.1.10 Prefix commands.
Option for bcal check

RC0 specifies that bcal check is to exit without error (return 0) even if some calendars do not exist or have errors. Programmers can then access the results bcal check stores in r() to get even more details about the problems. If you wish to suppress bcal dir, precede the bcal check command with capture and specify the RC0 option if you wish to access the r() results.

Options for bcal create

FROM(varname) specifies the date variable used to create the business calendar. Gaps between dates in varname define business holidays. The longest gap allowed can be set with the maxgap() option. FROM() is required.

GENERATE(newvar) specifies that newvar be created. newvar is a date variable in %tbcalname format, where calname is the name of the business calendar derived from filename.

EXCLUDEMISSING(varlist [, any]) specifies that the dates of observations with missing values in varlist are business holidays. By default, the dates of observations with missing values in all variables in varlist are holidays. The any suboption specifies that the dates of observations with missing values in any variable in varlist are holidays.

PERSONAL specifies that the calendar file be saved in the PERSONAL directory. This option cannot be used if filename contains the pathname of the directory where the file is to be saved.

REPLACE specifies that the business calendar file be replaced if it already exists.

PURPOSE(text) specifies the purpose of the business calendar being created. text cannot exceed 63 characters.

dateformat(ymd | ydm | myd | dym | dmy) specifies the date format in the new business calendar. The default is dateformat(ymd). dateformat() has nothing to do with how dates will look when variables are formatted with %tbcalname; it specifies how dates are typed in the calendar file.

RANGE(fromdate todate) defines the date range of the calendar being created. fromdate and todate should be in the format specified by the dateformat() option; if not specified, the default ymd format is assumed.

cENTERDATE(date) defines the center date of the new business calendar. If not specified, the earliest date in the calendar is assumed. date should be in the format specified by the dateformat() option; if not specified, the default ymd format is assumed.

MAXGAP(#) specifies the maximum number of consecutive business holidays allowed by bcal create. The default is maxgap(10).

Remarks and examples

bcal check reports on any %tb formats used by the data in memory:

. bcal check
  %tbsimple: defined, used by variable mydate
bcal dir reports on business calendars available:

```
. bcal dir
1 calendar file found:
    simple: C:\Program Files\Stata17\ado\base\s\simple.stbcal
```

bcal describe reports on an individual calendar.

```
. bcal describe simple
Business calendar simple (format %tbsimple):
    purpose: Example for manual
    range: 01nov2011 30nov2011
           18932  18961 in %td units
           0    19 in %tbsimple units
    center: 01nov2011
           18932 in %td units
           0 in %tbsimple units
    omitted: 10 days
    121.8 approx. days/year
    included: 20 days
    243.5 approx. days/year
```

bcal load is used by programmers writing new stbcal-files. See [D] Datetime business calendars creation.

bcal create creates a business calendar file from the current dataset and describes the new calendar. For example, sp500.dta is a dataset installed with Stata that has daily records on the S&P 500 stock market index in 2001. The dataset has observations only for days when trading took place. A business calendar for stock trading in 2001 can be automatically created from this dataset as follows:

```
. sysuse sp500
(S&P 500)
. bcal create sp500, from(date) purpose(S&P 500 for 2001) generate(bizdate)
```

```
Business calendar sp500 (format %tbsp500):
    purpose: S&P 500 for 2001
    range: 02jan2001 31dec2001
           14977  15340 in %td units
           0    247 in %tbsp500 units
    center: 02jan2001
           14977 in %td units
           0 in %tbsp500 units
    omitted: 116 days
    116.4 approx. days/year
    included: 248 days
    248.9 approx. days/year
```

Notes:

business calendar file sp500.stbcal saved
variable bizdate created; it contains business dates in %tbsp500 format
The business calendar file created:

```
* Business calendar "sp500" created by -bcal create-
* Created/replaced on 02 Apr 2021
version 17
purpose "S&P 500 for 2001"
dateformat ymd
range 2001jan02 2001dec31
centerdate 2001jan02
omit dayofweek (Sa Su)
omit date 2001jan15
omit date 2001feb19
omit date 2001apr13
omit date 2001may28
omit date 2001jun04
omit date 2001sep03
omit date 2001sep11
omit date 2001sep12
omit date 2001sep13
omit date 2001sep14
omit date 2001nov22
omit date 2001dec25
```

*bcal create filename, from()* can save the calendar file anywhere in your directory system by specifying a path in *filename*. It is assumed that the directory where the file is to be saved already exists. The pattern of *filename* should be [path] calname [.stbcal]. Here *calname* should be without the %tb prefix; *calname* has to be a valid Stata name but limited to 10 characters. If *path* is not specified, the file is saved in the current working directory. If the .stbcal extension is not specified, it is added.

Save the file in a directory where Stata can find it. Stata automatically searches for stbcal-files in the same way it searches for ado-files. Stata looks for ado-files and stbcal-files in the official Stata directories, your site’s directory (*SITE*), your current working directory, your personal directory (*PERSONAL*), and your directory for materials written by other users (*PLUS*). The option personal specifies that the calendar file be saved in your *PERSONAL* directory, which ensures that the created calendar can be easily found in future work.

**Stored results**

*bcal check* stores the following in *r()*:

Macros

- `r(defined)` : business calendars used, stbcal-file exists, and file contains no errors
- `r(defined)` : business calendars used, but no stbcal-files exist for them
- `r(varlist_calname)` : list of variable names that use business calendar *calname*

Warning to programmers: Specify the *rc0* option to access these returned results. By default, *bcal check* returns code 459 if a business calendar does not exist or if a business calendar exists but has errors; in such cases, the results are not stored.

*bcal dir* stores the following in *r()*:

Macros

- `r(calendars)` : business calendars available
- `r(fn_calname)` : stbcal-file for business calendar *calname*
**bcal** describe and **bcal** create store the following in \( r() \):

**Scalars**

- \( r(\text{min\_date\_td}) \): calendar’s minimum date in \%td units
- \( r(\text{max\_date\_td}) \): calendar’s maximum date in \%td units
- \( r(\text{ctr\_date\_td}) \): calendar’s zero date in \%td units
- \( r(\text{min\_date\_tb}) \): calendar’s minimum date in \%tb units
- \( r(\text{max\_date\_tb}) \): calendar’s maximum date in \%tb units
- \( r(\text{omitted}) \): total number of days omitted from calendar
- \( r(\text{included}) \): total number of days included in calendar
- \( r(\text{omitted\_year}) \): approximate number of days omitted per year from calendar
- \( r(\text{included\_year}) \): approximate number of days included per year in calendar

**Macros**

- \( r(\text{name}) \): pure calendar name (for example, nyse)
- \( r(\text{purpose}) \): short description of calendar’s purpose
- \( r(\text{fn}) \): name of stbcal-file

**bcal** load stores the same results in \( r() \) as **bcal** describe, except it does not store \( r(\text{omitted}) \), \( r(\text{included}) \), \( r(\text{omitted\_year}) \) and \( r(\text{included\_year}) \).

**Reference**


**Also see**

[D] **Datetime** — Date and time values and variables

[D] **Datetime business calendars** — Business calendars

[D] **Datetime business calendars creation** — Business calendars creation
by — Repeat Stata command on subsets of the data

Description

Most Stata commands allow the by prefix, which repeats the command for each group of observations for which the values of the variables in varlist are the same. by without the sort option requires that the data be sorted by varlist; see [D] sort.

Stata commands that work with the by prefix indicate this immediately following their syntax diagram by reporting, for example, “by is allowed; see [D] by” or “bootstrap, by, etc., are allowed; see [U] 11.1.10 Prefix commands”.

by and bysort are really the same command; bysort is just by with the sort option.

The varlist1 (varlist2) syntax is of special use to programmers. It verifies that the data are sorted by varlist1 varlist2 and then performs a by as if only varlist1 were specified. For instance,

```
by pid (time): generate growth = (bp - bp[_n-1])/bp
```

performs the generate by values of pid but first verifies that the data are sorted by pid and time within pid.

Quick start

Generate newv as an observation number within each level of catvar

- by catvar: generate newv = _n

As above, but sort data by catvar first

- by catvar, sort: generate newv = _n

Same as above

- bysort catvar: generate newv = _n

As above, but sort by v within values of catvar

- bysort catvar (v): generate newv = _n

Generate newv as an observation number for each observation in levels of catvar and v

- bysort catvar v: generate newv = _n

Note: Any command that accepts the by prefix may be substituted for generate above.
Syntax

by varlist: stata_cmd

bysort varlist: stata_cmd

The above diagrams show by and bysort as they are typically used. The full syntax of the commands is

by varlist1 [(varlist2)] [ , sort rc0] : stata_cmd

bysort varlist1 [(varlist2)] [ , rc0] : stata_cmd

Options

sort specifies that if the data are not already sorted by varlist, by should sort them.

rc0 specifies that even if the stata_cmd produces an error in one of the by-groups, then by is still to run the stata_cmd on the remaining by-groups. The default action is to stop when an error occurs. rc0 is especially useful when stata_cmd is an estimation command and some by-groups have insufficient observations.

Remarks and examples

Example 1

. use https://www.stata-press.com/data/r17/autornd
   (1978 automobile data)
. keep in 1/20
   (54 observations deleted)
. by mpg: egen mean_w = mean(weight)
   not sorted
   r(5);
. sort mpg
. by mpg: egen mean_w = mean(weight)
by — Repeat Stata command on subsets of the data

by requires that the data be sorted. In the above example, we could have typed by mpg, sort: egen mean_w = mean(weight) or bysort mpg: egen mean_w = mean(weight) rather than the separate sort; all would yield the same results.

For more examples, see [U] 11.1.2 by varlist:, [U] 11.5 by varlist: construct, and [U] 13.7 Explicit subscripting. For extended introductions with detailed examples, see Cox (2002) and Mitchell (2020, chap. 8).

Technical note

by repeats the stata_cmd for each group defined by varlist. If stata_cmd stores results, only the results from the last group on which stata_cmd executes will be stored.

References

Also see

[D] sort — Sort data
[D] statsby — Collect statistics for a command across a by list
[P] byable — Make programs byable
[P] foreach — Loop over items
[P] forvalues — Loop over consecutive values
[P] while — Looping
[U] 11.1.2 by varlist:
[U] 11.1.10 Prefix commands
[U] 11.4 varname and varlists
[U] 11.5 by varlist: construct
cd — Change directory

Description

Stata for Windows: cd changes the current working directory to the specified drive and directory. pwd is equivalent to typing cd without arguments; both display the name of the current working directory. Note: You can shell out to a Windows command prompt; see [D] shell. However, typing !cd directory_name does not change Stata’s current directory; use the cd command to change directories.

Stata for Mac and Stata for Unix: cd (synonym chdir) changes the current working directory to directory_name or, if directory_name is not specified, the home directory. pwd displays the path of the current working directory.

Quick start

Change working directory in Stata for Windows to C:\mydir\myfolder
   cd c:\mydir\myfolder

Change working directory in Stata for Windows to C:\my dir\my folder
   cd "c:\my dir\my folder"

Change working directory in Stata for Mac or Unix to mydir/myfolder
   cd mydir/myfolder

Move up one level in the directory structure
   cd..

Move to myfolder from mydir
   cd myfolder

View current working directory
   pwd

Go to home directory in Stata for Mac or Unix
   cd
Syntax

Stata for Windows

    cd
    cd ["directory_name"]
    cd ["drive:"
    cd ["drive:directory_name"]
    pwd

Stata for Mac and Stata for Unix

    cd
    cd ["directory_name"]
    pwd

If your directory_name contains embedded spaces, remember to enclose it in double quotes.

Remarks and examples

Remarks are presented under the following headings:

    Stata for Windows
    Stata for Mac
    Stata for Unix

Stata for Windows

When you start Stata for Windows, your current working directory is set to the Start in directory specified in Properties. If you want to change this, see [GSW] B.1 The Windows Properties Sheet. You can always see what your working directory is by looking at the status bar at the bottom of the Stata window.

Once you are in Stata, you can change your directory with the cd command.

    . cd
c:\data
    . cd city
c:\data\city
    . cd d:
    D:\
    . cd kande
    D:\kande
    . cd "additional detail"
    D:\kande\additional detail
    . cd c:
    C:\
    . cd data\city
    C:\data\city
When we typed `cd d:`, we changed to the current directory of the D drive. We navigated our way to `d:\kande\additional detail` with three commands: `cd d:`; then `cd kande`, and then `cd "additional detail"`. The double quotes around “additional detail” are necessary because of the space in the directory name. We could have changed to this directory in one command: `cd "d:\kande\additional detail"`.

Notice the last three `cd` commands in the example above. You are probably familiar with the `cd ..` syntax to move up one directory from where you are. The last two `cd` commands above let you move up more than one directory: `cd ...` is shorthand for “`cd ..\..`” and `cd ....` is shorthand for “`cd ..\..\..`”. These shorthand `cd` commands are not limited to Stata; they will work in your Command window under Windows as well.

You can see the current directory (where Stata saves files and looks for files) by typing `pwd`. You can change the current directory by using `cd` or by selecting `File > Change working directory...`. Stata’s `cd` command understands “~” as an abbreviation for the home directory, so you can type things like `cd ~/data`.

If you now wanted to change to `"C:\Users\bill\data\city\ny"`, you could type `cd ny`. If you wanted instead to change to `"C:\Users\bill\data\"`, you could type “`cd ..`”.

**Stata for Mac**

Read [U] 11.6 Filenaming conventions for a description of how filenames are written in a command language before reading this entry.

Invoking an application and then changing folders is an action foreign to most Mac users. If it is foreign to you, you can ignore `cd` and `pwd`. However, they can be useful. You can see the current folder (where Stata saves files and looks for files) by typing `pwd`. You can change the current folder by using `cd` or by selecting `File > Change working directory...`. Stata’s `cd` command understands “~” as an abbreviation for the home directory, so you can type things like `cd ~/data`.

If you now wanted to change to `"/Users/bill/data/city/ny"`, you could type `cd ny`. If you wanted instead to change to `"/Users/bill/data\"`, you could type “`cd ..`”.
Stata for Unix

`cd` and `pwd` are equivalent to Unix’s `cd` and `pwd` commands. Like `csh`, Stata’s `cd` understands “~” as an abbreviation for the home directory `$HOME`, so you can type things like `cd ~/data`; see [U] 11.6 Filenaming conventions.

```
   . pwd
   /usr/bill/proj
   . cd ~/data/city
   /usr/bill/data/city
   .
```

If you now wanted to change to `/usr/bill/data/city/ny`, you could type `cd ny`. If you wanted instead to change to `/usr/bill/data`, you could type “cd ..”.

Also see

[D] copy — Copy file from disk or URL
[D] dir — Display filenames
[D] erase — Erase a disk file
[D] mkdir — Create directory
[D] rmdir — Remove directory
[D] shell — Temporarily invoke operating system
[D] type — Display contents of a file
[U] 11.6 Filenaming conventions
Description

`cf` compares `varlist` of the dataset in memory (the master dataset) with the corresponding variables in `filename` (the using dataset). `cf` returns nothing (that is, a return code of 0) if the specified variables are identical and a return code of 9 if there are any differences. Only the variable values are compared. Variable labels, value labels, notes, characteristics, etc., are not compared.

Quick start

Compare values of v1 and v2 from `mydata1.dta` in memory to `mydata2.dta`

```
cf v1 v2 using mydata2
```

As above, but give a detailed listing of the differences

```
cf v1 v2 using mydata2, verbose
```

As above, but for all variables in memory

```
cf _all using mydata2, verbose
```

Menu

Data > Data utilities > Compare two datasets
Syntax

```
cf varlist using filename [ , all verbose ]
```

Options

all displays the result of the comparison for each variable in `varlist`. Unless `all` is specified, only the results of the variables that differ are displayed. `verbose` gives a detailed listing, by variable, of each observation that differs.

Remarks and examples

`cf` produces messages having the following form:

```
varname: does not exist in using
varname: __ in master but __ in using
varname: __ mismatches
varname: match
```

An example of the second message is “str4 in master but float in using”. Unless `all` is specified, the fourth message does not appear—silence indicates matches.

Example 1

We think the dataset in memory is identical to `mydata.dta`, but we are unsure. We want to understand any differences before continuing:

```
. cf _all using mydata
```

All the variables in the master dataset are in `mydata.dta`, and these variables are the same in both datasets. We might see instead

```
. cf _all using mydata
  mpg: 2 mismatches
  headroom: does not exist in using
  displacement: does not exist in using
  gear_ratio: does not exist in using
r(9);
```

Two changes were made to the `mpg` variable, and the `headroom`, `displacement`, and `gear_ratio` variables do not exist in `mydata.dta`. 
To see the result of each comparison, we could append the all option to our command:

```
.cf _all using mydata, all
make:  match
price:  match
mpg:  2 mismatches
rep78:  match
headroom:  does not exist in using
trunk:  match
weight:  match
length:  match
turn:  match
displacement:  does not exist in using
gear_ratio:  does not exist in using
foreign:  match
r(9);
```

For more details on the mismatches, we can use the verbose option:

```
.cf _all using mydata, verbose
mpg:  2 mismatches
  obs  1.  22 in master;  33 in using
  obs  2.  17 in master;  33 in using
headroom:  does not exist in using
displacement:  does not exist in using
gear_ratio:  does not exist in using
r(9);
```

This example shows us exactly which two observations for mpg differ, as well as the value stored in each dataset.

### Example 2

We want to compare a group of variables in the dataset in memory against the same group of variables in mydata.dta.

```
.cf mpg headroom using mydata
mpg:  2 mismatches
  headroom:  does not exist in using
r(9);
```

### Stored results

`cf` stores the following in `r()`:

- **Macros**
  - `r(Nsum)` number of differences

### Acknowledgment

Speed improvements in `cf` were based on code written by David Kantor.
Also see

[D] compare — Compare two variables
Title

| changeeol — Convert end-of-line characters of text file |

Description

changeeol converts text file \texttt{filename1} to text file \texttt{filename2} with the specified Windows/Unix/Mac/classic Mac-style end-of-line characters. changeeol changes the end-of-line characters from one type of file to another.

Quick start

Create \texttt{mytext2.txt} with Windows end-of-line characters from \texttt{mytext1.txt}

\( \text{changeeol mytext1.txt mytext2.txt, eol(windows)} \)

As above, but convert to Mac-style end-of-line characters

\( \text{changeeol mytext1.txt mytext2.txt, eol(mac)} \)

As above, but convert to Unix-style end-of-line characters

\( \text{changeeol mytext1.txt mytext2.txt, eol(unix)} \)

Syntax

\( \text{changeeol } \textit{filename1} \textit{filename2}, \textit{eol(platform)} \ [ \textit{options} ] \)

\textit{filename1} and \textit{filename2} must be filenames.

Note: Double quotes may be used to enclose the filenames, and the quotes must be used if the filename contains embedded blanks.

<table>
<thead>
<tr>
<th>\textit{options}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*eol(windows)</td>
<td>convert to Windows-style end-of-line characters (\r\n)</td>
</tr>
<tr>
<td>*eol(dos)</td>
<td>synonym for eol(windows)</td>
</tr>
<tr>
<td>*eol(unix)</td>
<td>convert to Unix-style end-of-line characters (\n)</td>
</tr>
<tr>
<td>*eol(mac)</td>
<td>convert to Mac-style end-of-line characters (\n)</td>
</tr>
<tr>
<td>*eol(classicmac)</td>
<td>convert to classic Mac-style end-of-line characters (\r)</td>
</tr>
<tr>
<td>replace</td>
<td>overwrite \textit{filename2}</td>
</tr>
<tr>
<td>force</td>
<td>force to convert \textit{filename1} to \textit{filename2} if \textit{filename1} is a binary file</td>
</tr>
</tbody>
</table>

*eol() is required.
Options

eol(windows|dos|unix|mac|classicmac) specifies to which platform style filename2 is to be converted. eol() is required.

replace specifies that filename2 be replaced if it already exists.

force specifies that filename1 be converted if it is a binary file.

Remarks and examples

changeeol uses hexdump to determine whether filename1 is text or binary. If it is binary, changeeol will refuse to convert it unless the force option is specified.

Examples

Windows:
  . changeeol orig.txt newcopy.txt, eol(windows)

Unix:
  . changeeol orig.txt newcopy.txt, eol(unix)

Mac:
  . changeeol orig.txt newcopy.txt, eol(mac)

Classic Mac:
  . changeeol orig.txt newcopy.txt, eol(classicmac)

Also see

[D] filefilter — Convert ASCII or binary patterns in a file

[D] hexdump — Display hexadecimal report on file
checksum — Calculate checksum of file

### Description

`checksum` creates `filename.sum` files for later use by Stata when it reads files over a network. These optional files are used to reduce the chances of corrupted files going undetected. Whenever Stata reads file `filename.suffix` over a network, whether by `use`, `net`, `update`, etc., it also looks for `filename.sum`. If Stata finds that file, Stata reads it and uses its contents to verify that the first file was received without error. If there are errors, Stata informs the user that the file could not be read.

### Quick start

Calculate checksum of `mydata.dta`

```
checksum mydata.dta
```

As above, and save results to `mydata.sum`

```
checksum mydata.dta, save
```

As above, but save results to `mycheck.sum`

```
checksum mydata.dta, saving(mycheck.sum)
```

As above, but replace `mycheck.sum` if it exists

```
checksum mydata.dta, saving(mycheck.sum, replace)
```

### Syntax

```stata
checksum filename [ , options ]
```

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>save</code></td>
<td>save output to <code>filename.sum</code>; default is to display a report</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>may overwrite <code>filename.sum</code>; use with <code>save</code></td>
</tr>
<tr>
<td><code>saving(filename2 [, replace])</code></td>
<td>save output to <code>filename2</code>; alternative to <code>save</code></td>
</tr>
</tbody>
</table>
checksum calculates a CRC checksum following the POSIX 1003.2 specification and displays the
file size in bytes. checksum produces the same results as the Unix cksum command. Comparing the
checksum of the original file with the received file guarantees the integrity of the received file.

When comparing Stata’s checksum results with those of Unix, do not confuse Unix’s sum and
cksum commands. Unix’s cksum and Stata’s checksum use a more robust algorithm than that used
by Unix’s sum. In some Unix operating systems, there is no cksum command, and the more robust
algorithm is obtained by specifying an option with sum.

Options

save saves the output of the checksum command to the text file filename.sum. The default is to
display a report but not create a file.
replace is for use with save; it permits Stata to overwrite an existing filename.sum file.
saving(filename2 [, replace]) is an alternative to save. It saves the output in the specified
filename. You must supply a file extension if you want one, because none is assumed.

Remarks and examples

Example 1

Say that you wish to put a dataset on your homepage so that colleagues can use it over the Internet
by typing

```
. use http://www.myuni.edu/department/~joe/mydata
```

mydata.dta is important, and even though the chances of the file mydata.dta being corrupted by
the Internet are small, you wish to guard against that. The solution is to create the checksum file
named mydata.sum and place that on your homepage. Your colleagues need type nothing different,
but now Stata will verify that all goes well. When they use the file, they will see either

```
. use http://www.myuni.edu/department/~joe/mydata
(important data from joe)
```
or

```
    file transmission error (checksums do not match)
http://www.myuni.edu/department/~joe/mydata.dta not downloaded
r(639);
```
To make the checksum file, change to the directory where the file is located and type

```
. checksum mydata.dta, save
Checksum for mydata.dta = 263508742, size = 4052
file mydata.sum saved
```
Example 2

Let's use `checksum` on `auto.dta` that is shipped with Stata. We will load the dataset and save it to our current directory.

```stata
. use https://www.stata-press.com/data/r17/auto
   (1978 automobile data)
. save auto
   file auto.dta saved
. checksum auto.dta
   Checksum for auto.dta = 3186221657, size = 12765
```

We see the report produced by `checksum`, but we decide to save this information to a file.

```stata
. checksum auto.dta, save
. type auto.sum
   1 12765 3186221657
```

The first number is the version number (possibly used for future releases). The second number is the file's size in bytes, which can be used with the checksum value to ensure that the file transferred without corruption. The third number is the checksum value. Although two different files can have the same checksum value, two files with the same checksum value almost certainly could not have the same file size.

This example is admittedly artificial. Typically, you would use `checksum` to verify that no file transmission error occurred during a web download. If you want to verify that your own data are unchanged, using `datasignature` is better; see [D] `datasignature`.

```stata

Stored results

`checksum` stores the following in `r()`:

Scalars
- `r(version)` checksum version number
- `r(filelen)` length of file in bytes
- `r(checksum)` checksum value

Also see

[R] `net` — Install and manage community-contributed additions from the Internet
[D] `use` — Load Stata dataset
[D] `datasignature` — Determine whether data have changed

```
clear — Clear memory

Description

clear, by itself, removes data and value labels from memory and is equivalent to typing

. version 17.0
. drop _all  (see [D] drop)
. label drop _all  (see [D] label)

clear mata removes Mata functions and objects from memory and is equivalent to typing

. version 17.0
. mata: mata clear  (see [M-3] mata clear)

clear results eliminates stored results from memory and is equivalent to typing

. version 17.0
. return clear  (see [P] return)
. ereturn clear  (see [P] return)
. sreturn clear  (see [P] return)
. _return drop _all  (see [P] _return)

clear matrix eliminates from memory all matrices created by Stata’s matrix command; it does not eliminate Mata matrices from memory. clear matrix is equivalent to typing

. version 17.0
. return clear  (see [P] return)
. ereturn clear  (see [P] return)
. sreturn clear  (see [P] return)
. _return drop _all  (see [P] _return)
. matrix drop _all  (see [P] matrix utility)
. estimates drop _all  (see [R] estimates)

clear programs eliminates all programs from memory and is equivalent to typing

. version 17.0
. program drop _all  (see [P] program)

clear ado eliminates all automatically loaded ado-file programs from memory (but not programs defined interactively or by do-files). It is equivalent to typing

. version 17.0
. program drop _allado  (see [P] program)

clear rngstream eliminates from memory stored random-number states for all mt64s streams (including the current stream). It resets the mt64s generator to the beginning of every stream, based on the current mt64s seed. clear rngstream does not change the current mt64s seed and stream. The mt64s seed and stream can be set with set seed and set rngstream, respectively.

clear frames eliminates from memory all frames and restores Stata to its initial state of having a single, empty frame named default.

clear collect removes all collections from memory and is equivalent to typing

. version 17.0
. collect clear  (see [TABLES] collect clear)
clear all and clear * are synonyms. They remove all data, value labels, matrices, scalars, constraints, clusters, stored results, frames, sersets, and Mata functions and objects from memory. They also close all open files and postfiles, clear the class system, close any open Graph windows and dialog boxes, drop all programs from memory, and reset all timers to zero. However, they do not call clear rngstream. They are equivalent to typing:

```stata
.clear
.drop _all
.frames reset
.label drop _all
.matrix drop _all
.scalar drop _all
.constraint drop _all
.cluster drop _all
.file close _all
.postutil clear
._return drop _all
.discard
.timer clear
.putdocx clear
.putpdf clear
.mata: mata clear
.python clear
.java clear
```

## Quick start

Remove data and value labels from memory

```
clear
```

Remove Stata matrices from memory

```
clear matrix
```

Remove Mata matrices, Mata objects, and Mata functions from memory

```
clear mata
```

Remove all programs from memory

```
clear programs
```

As above, but only programs automatically loaded by ado-files

```
clear ado
```

Remove results stored in r(), e(), and s() from memory

```
clear results
```

Remove all the above and constraints, clusters, and sersets; reset timers to 0; clear the class system; and close all open files, graph windows, and dialog boxes

```
clear all
```

Same as above

```
clear *
```
clear — Clear memory

Syntax

clear

clear [mata|results|matrix|programs|ado|rngstream|frames|collect]

clear [all|*]

Remarks and examples

You can clear the entire dataset without affecting macros and programs by typing clear. You can also type clear all. This command has the same result as clear by itself but also clears matrices, scalars, constraints, clusters, stored results, sersets, Mata, the class system, business calendars, and programs; closes all open files and postfiles; closes all open Graph windows and dialog boxes; and resets all timers to zero.

Example 1

We load the bpwide dataset to correct a mistake in the data.

. use https://www.stata-press.com/data/r17/bpwide
   (Fictional blood-pressure data)

   . list in 1/5

     patient   sex  agegrp   bp_bef-e   bp_after
     +-----------------+-----------------+-----------------+-----------------+
       1.    1  Male   30-45     143          153
       2.    2  Male   30-45     163          170
       3.    3  Male   30-45     153          168
       4.    4  Male   30-45     153          142
       5.    5  Male   30-45     146          141

   . replace bp_after = 145 in 3
   (1 real change made)

We made another mistake. We meant to change the value of bp_after in observation 4. It is easiest to begin again.

   . clear

   . use https://www.stata-press.com/data/r17/bpwide
      (Fictional blood-pressure data)

Also see

[D] drop — Drop variables or observations
[P] discard — Drop automatically loaded programs
[U] 11 Language syntax
[U] 13 Functions and expressions
**Description**

clonestar generates newvar as an exact copy of an existing variable, varname, with the same storage type, values, and display format as varname. varname’s variable label, value labels, notes, and characteristics will also be copied.

**Quick start**

Copy contents, label, and value label of v1 to newv1

\[ \text{clonevar newv1} = v1 \]

Copy observations from v2 to newv2 where v2 is less than 30

\[ \text{clonevar newv2} = v2 \text{ if } v2 < 30 \]

Copy the first 20 observations of v3 to newv3

\[ \text{clonevar newv3} = v3 \text{ in } f/20 \]

Same as above

\[ \text{clonevar newv3} = v3 \text{ in } 1/20 \]

**Menu**

Data > Create or change data > Other variable-creation commands > Clone existing variable

**Syntax**

\[ \text{clonevar newvar} = \text{varname} [\text{if}] [\text{in}] \]

**Remarks and examples**

clonestar has various possible uses. Programmers may desire that a temporary variable appear to the user exactly like an existing variable. Interactively, you might want a slightly modified copy of an original variable, so the natural starting point is a clone of the original.
Example 1

We have a dataset containing information on modes of travel. These data contain a variable named `mode` that identifies each observation as a specific mode of travel: air, train, bus, or car.

```
    . use https://www.stata-press.com/data/r17/travel
       (Modes of travel)
    . describe mode
    Variable                Storage   Display  Value
            name     type    format   label              Variable label
                mode     byte    %8.0g     travel           Travel mode alternatives
    . label list travel
       travel:
               1 Air
               2 Train
               3 Bus
               4 Car
```

To create an identical variable identifying only observations that contain air or train, we could use `clonevar` with an `if` qualifier.

```
    . clonevar airtrain = mode if mode == 1 | mode == 2
       (420 missing values generated)
    . describe mode airtrain
    Variable                Storage   Display  Value
            name     type    format   label              Variable label
                mode     byte    %8.0g     travel           Travel mode alternatives
                airtrain  byte    %8.0g     travel           Travel mode alternatives
    . list mode airtrain in 1/5

    +----------+-----------------+---------+
   | mode   | airtrain        |
   |--------+-----------------+---------|
   | 1.     | Air             | Air     |
   | 2.     | Train           | Train   |
   | 3.     | Bus             |        |
   | 4.     | Car             |        |
   | 5.     | Air             | Air     |
   +----------+-----------------+---------+
```

The new `airtrain` variable has the same storage type, display format, value label, and variable label as `mode`. If `mode` had any characteristics or notes attached to it, they would have been applied to the new `airtrain` variable, too. The only differences in the two variables are their names and values for bus and car.

Technical note

The `if` qualifier used with the `clonevar` command in example 1 referred to the values of `mode` as 1 and 2. Had we wanted to refer to the values by their associated value labels, we could have typed

```
    . clonevar airtrain = mode if mode == "air":travel | mode == "train":travel
```

For more details, see [U] 13.11 Label values.
**Acknowledgments**

*clonevar* was written by Nicholas J. Cox of the Department of Geography at Durham University, UK, who is coeditor of the *Stata Journal* and author of *Speaking Stata Graphics*. He in turn thanks Michael Blasnik of Nest Labs and Ken Higbee of StataCorp for very helpful comments on a precursor of this command.

**Also see**

[D] *generate* — Create or change contents of variable  
[D] *separate* — Create separate variables
codebook — Describe data contents

Description
codebook examines the variable names, labels, and data to produce a codebook describing the dataset.

Quick start
Codebook of all variables in the dataset
codebook
Codebook of variables v1, v2, and v3
codebook v1 v2 v3
Codebook of all variables starting with code
codebook code*
Include dataset name, last saved date, and variable notes in the codebook
codebook, header notes
Report problems with labels, constant-valued variables, embedded spaces and binary 0 in string variables, and noninteger date variables
codebook, problems
Codebook for dataset with English and Spanish variable and value labels using label languages en and es
codebook, languages(en es)

Menu
Data > Describe data > Describe data contents (codebook)
### Syntax

```
codebook [ `varlist` ] [ `if` ] [ `in` ] [, `options`]
```

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all</code></td>
<td>print complete report without missing values</td>
</tr>
<tr>
<td><code>header</code></td>
<td>print dataset name and last saved date</td>
</tr>
<tr>
<td><code>notes</code></td>
<td>print any notes attached to variables</td>
</tr>
<tr>
<td><code>mv</code></td>
<td>report pattern of missing values</td>
</tr>
<tr>
<td><code>tabulate(#)</code></td>
<td>set tables/summary statistics threshold; default is <code>tabulate(9)</code></td>
</tr>
<tr>
<td><code>problems</code></td>
<td>report potential problems in dataset</td>
</tr>
<tr>
<td><code>detail</code></td>
<td>display detailed report on the variables; only with <code>problems</code></td>
</tr>
<tr>
<td><code>compact</code></td>
<td>display compact report on the variables</td>
</tr>
<tr>
<td><code>dots</code></td>
<td>display a dot for each variable processed; only with <code>compact</code></td>
</tr>
</tbody>
</table>

#### Languages

```
languages [ (namelist) ] use with multilingual datasets; see [D] `label language` for details
```

`collect` is allowed; see [U] 11.1.10 Prefix commands.

### Options

- `all` is equivalent to specifying the `header` and `notes` options. It provides a complete report, which excludes only performing `mv`.
- `header` adds to the top of the output a header that lists the dataset name, the date that the dataset was last saved, etc.
- `notes` lists any notes attached to the variables; see [D] `notes`.
- `mv` specifies that `codebook` search the data to determine the pattern of missing values. This is a CPU-intensive task.
- `tabulate(#)` specifies the number of unique values of the variables to use to determine whether a variable is categorical or continuous. Missing values are not included in this count. The default is 9; when there are more than nine unique values, the variable is classified as continuous. Extended missing values will be included in the tabulation.
- `problems` specifies that a summary report is produced describing potential problems that have been diagnosed:
  - Variables that are labeled with an undefined value label
  - Incompletely value-labeled variables
  - Variables that are constant, including always missing
  - Leading, trailing, and embedded spaces in string variables
  - Embedded binary 0 (\0) in string variables
  - Noninteger-valued date variables

See the discussion of these problems and advice on overcoming them following example 5.
detail may be specified only with the problems option. It specifies that the detailed report on the variables not be suppressed.

compact specifies that a compact report on the variables be displayed. compact may not be specified with any options other than dots.

dots specifies that a dot be displayed for every variable processed. dots may be specified only with compact.

languages \((\text{namelist})\) is for use with multilingual datasets; see \([D]\) label language. It indicates that the codebook pertains to the languages in namelist or to all defined languages if no such list is specified as an argument to languages(). The output of codebook lists the data label and variable labels in these languages and which value labels are attached to variables in these languages.

Problems are diagnosed in all of these languages, as well. The problem report does not provide details in which language problems occur. We advise you to rerun codebook for problematic variables; specify detail to produce the problem report again.

If you have a multilingual dataset but do not specify languages(), all output, including the problem report, is shown in the “active” language.

Remarks and examples

codebook, without arguments, is most usefully combined with \(\text{log}\) to produce a printed listing for enclosure in a notebook documenting the data; see \([U]\) 15 Saving and printing output—log files. codebook is, however, also useful interactively, because you can specify one or a few variables.

Example 1

codebook examines the data in producing its results. For variables that codebook thinks are continuous, it presents the mean; the standard deviation; and the 10th, 25th, 50th, 75th, and 90th percentiles. For variables that it thinks are categorical, it presents a tabulation. In part, codebook makes this determination by counting the number of unique values of the variable. If the number is nine or fewer, codebook reports a tabulation; otherwise, it reports summary statistics.

codebook distinguishes the standard missing values (.\_) and the extended missing values (.a through .z, denoted by .\=*). If extended missing values are found, codebook reports the number of distinct missing value codes that occurred in that variable. Missing values are ignored with the tabulate option when determining whether a variable is treated as continuous or categorical.
codebook — Describe data contents  

. use https://www.stata-press.com/data/r17/educ3
(ccdb46, 52-54)
. codebook fips division, all

    Dataset:  https://www.stata-press.com/data/r17/educ3.dta
    Last saved:  6 Mar 2020 22:20
    Label:  ccdb46, 52-54
    Number of variables: 42
    Number of observations: 956
    Size: 145,312 bytes ignoring labels, etc.

    _dta:
      1.  confirmed data with steve on 7/22

<table>
<thead>
<tr>
<th>fips</th>
<th>state/place code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type: Numeric (long)</th>
<th>Units: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: [10060,560050]</td>
<td></td>
</tr>
<tr>
<td>Unique values: 956</td>
<td>Missing : 0/956</td>
</tr>
<tr>
<td>Mean: 256495</td>
<td></td>
</tr>
<tr>
<td>Std. dev.: 156998</td>
<td></td>
</tr>
<tr>
<td>Percentiles:</td>
<td>10% 25% 50% 75% 90%</td>
</tr>
<tr>
<td>61462</td>
<td>120426</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type: Numeric (int)</th>
<th>Census Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label: division</td>
<td></td>
</tr>
<tr>
<td>Range: [1,9]</td>
<td>Units: 1</td>
</tr>
<tr>
<td>Unique values: 9</td>
<td>Missing : 4/956</td>
</tr>
<tr>
<td>Unique mv codes: 2</td>
<td>Missing .*: 2/956</td>
</tr>
<tr>
<td>Tabulation:</td>
<td>Freq.</td>
</tr>
<tr>
<td>69</td>
<td>1</td>
</tr>
<tr>
<td>97</td>
<td>2</td>
</tr>
<tr>
<td>202</td>
<td>3</td>
</tr>
<tr>
<td>78</td>
<td>4</td>
</tr>
<tr>
<td>115</td>
<td>5</td>
</tr>
<tr>
<td>46</td>
<td>6</td>
</tr>
<tr>
<td>89</td>
<td>7</td>
</tr>
<tr>
<td>59</td>
<td>8</td>
</tr>
<tr>
<td>195</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>.a</td>
</tr>
</tbody>
</table>

Because division has nine unique nonmissing values, codebook reported a tabulation. If division had contained one more unique nonmissing value, codebook would have switched to reporting summary statistics, unless we had included the `tabulate(#)` option.
Example 2

The `mv` option is useful. It instructs `codebook` to search the data to determine patterns of missing values. Different kinds of missing values are not distinguished in the patterns.

```stata
use https://www.stata-press.com/data/r17/citytemp
(City temperature data)
.codebook cooldd heatdd tempjan tempjuly, mv
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Range</th>
<th>Unique values</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Percentiles</th>
<th>Missing ( \rightarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>cooldd</td>
<td>Numeric (int)</td>
<td>[0,4389]</td>
<td>438</td>
<td>1240.41</td>
<td>937.668</td>
<td>411 615 940 1566 2761</td>
<td>cooldd==mv &lt;-&gt; heatdd==mv</td>
</tr>
<tr>
<td>heatdd</td>
<td>Numeric (int)</td>
<td>[0,10816]</td>
<td>471</td>
<td>4425.53</td>
<td>2199.6</td>
<td>1510 2460 4950 6232 6919</td>
<td>cooldd==mv &lt;-&gt; heatdd==mv</td>
</tr>
<tr>
<td>tempjan</td>
<td>Numeric (float)</td>
<td>[2.2,72.6]</td>
<td>310</td>
<td>35.749</td>
<td>14.1881</td>
<td>20.2 25.1 31.3 47.8 55.1</td>
<td>tempjuly==mv &lt;-&gt; tempjan==mv</td>
</tr>
</tbody>
</table>
tempjuly  

Type: Numeric (float)  
Range: [58.1, 93.6]  
Units: .1  
Unique values: 196  
Missing: .: 2/956  
Mean: 75.0538  
Std. dev.: 5.49504  
Percentiles: 10\% 25\% 50\% 75\% 90\%  
68.8 71.8 74.25 78.7 82.3  

Missing values: tempjan==mv <-> tempjuly==mv

codebook reports that if tempjan is missing, tempjuly is also missing, and vice versa. In the output for the cooldd variable, codebook also reports that the pattern of missing values is the same for cooldd and heatdd. In both cases, the correspondence is indicated with “<->”.

For cooldd, codebook also states that “tempjan==mv --> cooldd==mv”. The one-way arrow means that a missing tempjan value implies a missing cooldd value but that a missing cooldd value does not necessarily imply a missing tempjan value.

Another feature of codebook—this one for numeric variables—is that it can determine the units of the variable. For instance, in the example above, tempjan and tempjuly both have units of 0.1, meaning that temperature is recorded to tenths of a degree. codebook handles precision considerations in making this determination (tempjan and tempjuly are floats; see [U] 13.12 Precision and problems therein). If we had a variable in our dataset recorded in 100s (for example, 21,500 or 36,800), codebook would have reported the units as 100. If we had a variable that took on only values divisible by 5 (5, 10, 15, etc.), codebook would have reported the units as 5.

Example 3

We can use the label language command (see [D] label language) and the label command (see [D] label) to create German value labels for our auto dataset. These labels are reported by codebook:

```stata
use https://www.stata-press.com/data/r17/auto
(label 1978 automobile data)
label language en, rename
(label language default renamed en)
label language de, new
(label language de now current language)
label data "1978 Automobile Daten"
label variable foreign "Art Auto"
label values foreign origin_de
label define origin_de 0 "Innen" 1 "Auslandisch"
```
. codebook foreign

<table>
<thead>
<tr>
<th>foreign</th>
<th>Art Auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: Numeric (byte)</td>
<td>Label: origin_de</td>
</tr>
<tr>
<td>Range: [0,1]</td>
<td>Units: 1</td>
</tr>
<tr>
<td>Unique values: 2</td>
<td>Missing : 0/74</td>
</tr>
<tr>
<td>Tabulation: Freq. Numeric Label</td>
<td></td>
</tr>
<tr>
<td>52 0</td>
<td>Innen</td>
</tr>
<tr>
<td>22 1</td>
<td>Ausländisch</td>
</tr>
</tbody>
</table>

. codebook foreign, languages(en de)

<table>
<thead>
<tr>
<th>foreign</th>
<th>in en: Car origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>in de: Art Auto</td>
<td></td>
</tr>
<tr>
<td>Type: Numeric (byte)</td>
<td>Label in en: origin</td>
</tr>
<tr>
<td>Label in de: origin_de</td>
<td></td>
</tr>
<tr>
<td>Range: [0,1]</td>
<td>Units: 1</td>
</tr>
<tr>
<td>Unique values: 2</td>
<td>Missing : 0/74</td>
</tr>
<tr>
<td>Tabulation: Freq. Numeric origin origin_de</td>
<td></td>
</tr>
<tr>
<td>52 0</td>
<td>Domestic Innen</td>
</tr>
<tr>
<td>22 1</td>
<td>Foreign Ausländisch</td>
</tr>
</tbody>
</table>

With the languages() option, the value labels are shown in the specified active and available languages.

Example 4

codebook, compact summarizes the variables in your dataset, including variable labels. It is an alternative to the summarize command.

. use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
. codebook, compact

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Unique</th>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>74</td>
<td>74</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>Make and model</td>
</tr>
<tr>
<td>price</td>
<td>74</td>
<td>74</td>
<td>6165.257</td>
<td>3291</td>
<td>15906</td>
<td>Price</td>
</tr>
<tr>
<td>mpg</td>
<td>74</td>
<td>21</td>
<td>21.2973</td>
<td>12</td>
<td>41</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>rep78</td>
<td>69</td>
<td>5</td>
<td>3.405797</td>
<td>1</td>
<td>5</td>
<td>Repair record 1978</td>
</tr>
<tr>
<td>headroom</td>
<td>74</td>
<td>8</td>
<td>2.993243</td>
<td>1.5</td>
<td>5</td>
<td>Headroom (in.)</td>
</tr>
<tr>
<td>trunk</td>
<td>74</td>
<td>18</td>
<td>13.75676</td>
<td>5</td>
<td>23</td>
<td>Trunk space (cu. ft.)</td>
</tr>
<tr>
<td>weight</td>
<td>74</td>
<td>64</td>
<td>3019.459</td>
<td>1760</td>
<td>4840</td>
<td>Weight (lbs.)</td>
</tr>
<tr>
<td>length</td>
<td>74</td>
<td>47</td>
<td>187.9324</td>
<td>142</td>
<td>233</td>
<td>Length (in.)</td>
</tr>
<tr>
<td>turn</td>
<td>74</td>
<td>18</td>
<td>39.64865</td>
<td>31</td>
<td>51</td>
<td>Turn circle (ft.)</td>
</tr>
<tr>
<td>displacement</td>
<td>74</td>
<td>31</td>
<td>197.2973</td>
<td>79</td>
<td>425</td>
<td>Displacement (cu. in.)</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>74</td>
<td>36</td>
<td>3.014865</td>
<td>2.19</td>
<td>3.89</td>
<td>Gear ratio</td>
</tr>
<tr>
<td>foreign</td>
<td>74</td>
<td>2</td>
<td>.2972973</td>
<td>0</td>
<td>1</td>
<td>Car origin</td>
</tr>
</tbody>
</table>
. summarize

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>74</td>
<td>6165.257</td>
<td>2949.496</td>
<td>3291</td>
<td>15906</td>
</tr>
<tr>
<td>mpg</td>
<td>74</td>
<td>3.405797</td>
<td>.8999323</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>69</td>
<td>2.993243</td>
<td>.8459948</td>
<td></td>
<td></td>
</tr>
<tr>
<td>headroom</td>
<td>74</td>
<td>13.75676</td>
<td>4.277404</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>trunk</td>
<td>74</td>
<td>187.9324</td>
<td>22.26634</td>
<td>142</td>
<td>233</td>
</tr>
<tr>
<td>weight</td>
<td>74</td>
<td>39.64865</td>
<td>4.399354</td>
<td>31</td>
<td>51</td>
</tr>
<tr>
<td>displacement</td>
<td>74</td>
<td>197.2973</td>
<td>91.83722</td>
<td>79</td>
<td>425</td>
</tr>
</tbody>
</table>

| gear_ratio | 74  | 3.014865 | .4652871 | 2.19 | 3.89 |
| foreign    | 74  | .2972973 | .4601885 | 0    | 1    |

Example 5

When codebook determines that neither a tabulation nor a listing of summary statistics is appropriate, for instance, for a string variable or for a numeric variable taking on many labeled values, it reports a few examples instead.

. use https://www.stata-press.com/data/r17/funnyvar
. codebook name

name (unlabeled)

* Type: String (str5), but longest is str3
* Unique values: 10
* Missing "": 0/10
* Examples: "1 0"
* "3"
* "5"
* "7"

Warning: Variable has embedded blanks.

codebook is also on the lookout for common problems that might cause you to make errors when dealing with the data. For string variables, this includes leading, embedded, and trailing blanks and embedded binary 0 (\0). In the output above, codebook informed us that name includes embedded blanks. If name had leading or trailing blanks, it would have mentioned that, too.

When variables are value labeled, codebook performs two checks. First, if a value label labname is associated with a variable, codebook checks whether labname is actually defined. Second, it checks whether all values are value labeled. Partial labeling of a variable may mean that the label was defined incorrectly (for instance, the variable has values 0 and 1, but the value label maps 1 to “male” and 2 to “female”) or that the variable was defined incorrectly (for example, a variable gender with three values). codebook checks whether date variables are integer valued.

If the problems option is specified, codebook does not provide detailed descriptions of each variable but reports only the potential problems in the data.
Potential problems in dataset https://www.stata-press.com/data/r17/ > funnyvar.dta

<table>
<thead>
<tr>
<th>Potential problem</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant (or all missing) vars</td>
<td>human, planet</td>
</tr>
<tr>
<td>vars with nonexisting label</td>
<td>educ</td>
</tr>
<tr>
<td>incompletely labeled vars</td>
<td>gender</td>
</tr>
<tr>
<td>str# vars that may be compressed</td>
<td>name, address, city, country, planet</td>
</tr>
<tr>
<td>string vars with leading blanks</td>
<td>city, country</td>
</tr>
<tr>
<td>string vars with trailing blanks</td>
<td>planet</td>
</tr>
<tr>
<td>string vars with embedded blanks</td>
<td>name, address</td>
</tr>
<tr>
<td>string vars with embedded \0</td>
<td>mugshot</td>
</tr>
<tr>
<td>noninteger-valued date vars</td>
<td>birthdate</td>
</tr>
</tbody>
</table>

In the example above, codebook, problems reported various potential problems with the dataset. These problems include:

- **Constant variables, including variables that are always missing**
  Variables that are constant, taking the same value in all observations, or that are always missing, are often superfluous. Such variables, however, may also indicate problems. For instance, variables that are always missing may occur when importing data with an incorrect input specification. Such variables may also occur if you generate a new variable for a subset of the data, selected with an expression that is false for all observations.

Advice: Carefully check the origin of constant variables. If you are saving a constant variable, be sure to `compress` the variable to use minimal storage.

- **Variables with nonexisting value labels**
  Stata treats value labels as separate objects that can be attached to one or more variables. A problem may arise if variables are linked to value labels that are not yet defined or if an incorrect value label name was used.

Advice: Attach the correct value label, or `label define` the value label. See `[D] label`.

- **Incompletely labeled variables**
  A variable is called “incompletely value labeled” if the variable is value labeled but no mapping is provided for some values of the variable. An example is a variable with values 0, 1, and 2 and value labels for 1, 2, and 3. This situation usually indicates an error, either in the data or in the value label.

Advice: Change either the data or the value label.

- **String variables that may be compressed**
  The storage space used by a string variable is determined by its data type; see `[D] Data types`. For instance, the storage type `str20` indicates that 20 bytes are used per observation. If the declared storage type exceeds your requirements, memory and disk space is wasted.

Advice: Use `compress` to store the data as compactly as possible.
• String variables with leading or trailing blanks

   In most applications, leading and trailing spaces do not affect the meaning of variables but are probably side effects from importing the data or from data manipulation. Spurious leading and trailing spaces force Stata to use more memory than required. In addition, manipulating strings with leading and trailing spaces is harder.

   Advice: Remove leading and trailing blanks from a string variable `s` by typing

   ```
   replace s = strtrim(s)
   ```

   See [FN] String functions.

• String variables with embedded blanks

   String variables with embedded blanks are often appropriate; however, sometimes they indicate problems importing the data.

   Advice: Verify that blanks are meaningful in the variables.

• String variables with embedded binary 0 (\0)

   String variables with embedded binary 0 (\0) are allowed; however, caution should be used when working with them as some commands and functions may only work with the plain-text portion of a binary string, ignoring anything after the first binary 0.

   Advice: Be aware of binary strings in your data and whether you are manipulating them in a way that is only appropriate with plain-text values.

• Noninteger-valued date variables

   Stata’s date and time formats were designed for use with integer values but will work with noninteger values.

   Advice: Carefully inspect the nature of the noninteger values. If noninteger values in a variable are the consequence of roundoff error, you may want to round the variable to the nearest integer.

   ```
   replace time = round(time)
   ```

Of course, more problems not reported by codebook are possible. These might include

• Numerical data stored as strings

   After importing data into Stata, you may discover that some string variables can actually be interpreted as numbers. Stata can do much more with numerical data than with string data. Moreover, string representation usually makes less efficient use of computer resources. `destring` will convert string variables to numeric.

   A string variable may contain a “field” with numeric information. An example is an address variable that contains the street name followed by the house number. The Stata string functions can extract the relevant substring.

• Categorical variables stored as strings

   Most statistical commands do not allow string variables. Moreover, string variables that take only a limited number of distinct values are an inefficient storage method. Use value-labeled numeric values instead. These are easily created with `encode`.

• Duplicate observations

   See [D] duplicates.
Observations that are always missing

Drop observations that are missing for all variables in varlist using the rownonmiss() function:

```stata
gen nos = rownonmiss(varlist)
drop if nos==0
```

Specify _all for varlist if only observations that are always missing should be dropped.

### Stored results

`codebook` stores the following lists of variables with potential problems in `r()`:

**Macros**

- `r(cons)`: constant (or missing)
- `r(labelnotfound)`: undefined value labeled
- `r(notlabeled)`: value labeled but with unlabeled categories
- `r(str_type)`: compressible
- `r(str_leading)`: leading blanks
- `r(str_trailing)`: trailing blanks
- `r(str_embedded)`: embedded blanks
- `r(str_embedded0)`: embedded binary 0 (\0)
- `r(realdate)`: noninteger dates

### References


### Also see

[D] `describe` — Describe data in memory or in file

[D] `ds` — Compactly list variables with specified properties

[D] `inspect` — Display simple summary of data’s attributes

[D] `labelbook` — Label utilities

[D] `notes` — Place notes in data

[D] `split` — Split string variables into parts

[U] 15 Saving and printing output—log files


**Description**

collapse converts the dataset in memory into a dataset of means, sums, medians, etc. *clist* must refer to numeric variables exclusively.

Note: See [D] contract if you want to collapse to a dataset of frequencies.

**Quick start**

Replace dataset in memory with means of v1 and v2

```
collapse v1 v2
```

As above, but calculate statistics separately by each level of *catvar*

```
collapse v1 v2, by(catvar)
```

Dataset of mean, standard deviation, and standard error of the mean of v1

```
collapse (mean) mean1=v1 (sd) sd1=v1 (semean) sem1=v1
```

Mean and standard error of the mean for binomial v2

```
collapse (mean) mean2=v2 (sebinomial) sem2=v2
```

Frequency, median, and interquartile range of v1

```
collapse (count) freq=v1 (p50) p50=v1 (iqr) iqr=v1
```

Weighted and unweighted sum of v2 using frequency weight *wvar*

```
collapse (sum) weighted=v2 (rawsum) unweighted=v2 [fweight=wvar]
```

**Menu**

Data > Create or change data > Other variable-transformation commands > Make dataset of means, medians, etc.
**Syntax**

```
collapse clist [if] [in] [weight] [, options]
```

where `clist` is either

```
[(stat) varlist [(stat)] ...]
[(stat) target_var=varname [target_var=varname ...] [(stat)] ...]
```

or any combination of the `varlist` and `target_var` forms, and `stat` is one of

<table>
<thead>
<tr>
<th>Stat</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>means (default)</td>
</tr>
<tr>
<td>median</td>
<td>medians</td>
</tr>
<tr>
<td>p1</td>
<td>1st percentile</td>
</tr>
<tr>
<td>p2</td>
<td>2nd percentile</td>
</tr>
<tr>
<td>...</td>
<td>3rd–49th percentiles</td>
</tr>
<tr>
<td>p50</td>
<td>50th percentile (same as median)</td>
</tr>
<tr>
<td>...</td>
<td>51st–97th percentiles</td>
</tr>
<tr>
<td>p98</td>
<td>98th percentile</td>
</tr>
<tr>
<td>sd</td>
<td>standard deviations</td>
</tr>
<tr>
<td>semean</td>
<td>standard error of the mean (sd/sqrt(n))</td>
</tr>
<tr>
<td>sebinomial</td>
<td>standard error of the mean, binomial (sqrt(p(1-p)/n))</td>
</tr>
<tr>
<td>sepoisson</td>
<td>standard error of the mean, Poisson (sqrt(mean/n))</td>
</tr>
<tr>
<td>sum</td>
<td>sums, ignoring optionally specified weight except observations with a weight of zero are excluded</td>
</tr>
<tr>
<td>rawsum</td>
<td>sums</td>
</tr>
<tr>
<td>count</td>
<td>number of nonmissing observations</td>
</tr>
<tr>
<td>percent</td>
<td>percentage of nonmissing observations</td>
</tr>
<tr>
<td>max</td>
<td>maximums</td>
</tr>
<tr>
<td>min</td>
<td>minimums</td>
</tr>
<tr>
<td>iqr</td>
<td>interquartile range</td>
</tr>
<tr>
<td>first</td>
<td>first value</td>
</tr>
<tr>
<td>last</td>
<td>last value</td>
</tr>
<tr>
<td>firstnm</td>
<td>first nonmissing value</td>
</tr>
<tr>
<td>lastnm</td>
<td>last nonmissing value</td>
</tr>
<tr>
<td>firstnm</td>
<td>first nonmissing value</td>
</tr>
</tbody>
</table>

If `stat` is not specified, `mean` is assumed.

**options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>by(varlist)</strong></td>
</tr>
<tr>
<td><strong>cw</strong></td>
</tr>
<tr>
<td><strong>fast</strong></td>
</tr>
</tbody>
</table>

`varlist` and `varname` in `clist` may contain time-series operators; see [U] 11.4.4 Time-series varlists.

`aweights`, `fweights`, `iweights`, and `pweights` are allowed; see [U] 11.1.6 weight, and see Weights below. `pweights` may not be used with `sd`, `semean`, `sebinomial`, or `sepoisson`. `iweights` may not be used with `semean`, `sebinomial`, or `sepoisson`. `aweights` may not be used with `sebinomial` or `sepoisson`.

`fast` does not appear in the dialog box.

**Examples:**

```
. collapse age educ income, by(state)
. collapse (mean) age educ (median) income, by(state)
. collapse (mean) age educ income (median) medinc=income, by(state)
. collapse (p25) gpa [fw=number], by(year)
```
Options

by(varlist) specifies the groups over which the means, etc., are to be calculated. If this option is not specified, the resulting dataset will contain 1 observation. If it is specified, varlist may refer to either string or numeric variables.

cw specifies casewise deletion. If cw is not specified, all possible observations are used for each calculated statistic.

The following option is available with collapse but is not shown in the dialog box:
fast specifies that collapse not restore the original dataset should the user press Break. fast is intended for use by programmers.

Remarks and examples

collapse takes the dataset in memory and creates a new dataset containing summary statistics of the original data. collapse adds meaningful variable labels to the variables in this new dataset. Because the syntax diagram for collapse makes using it appear more complicated than it is, collapse is best explained with examples.

Remarks are presented under the following headings:

Introductory examples
Variablewise or casewise deletion
Weights
A final example

Introductory examples

Example 1

Consider the following artificial data on the grade-point average (gpa) of college students:

```
. use https://www.stata-press.com/data/r17/college
. describe
Contains data from https://www.stata-press.com/data/r17/college.dta
Observations: 12
Variables: 4
3 Jan 2020 12:05

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpa</td>
<td>float</td>
<td>%9.0g</td>
<td>gpa for this year</td>
</tr>
<tr>
<td>hour</td>
<td>int</td>
<td>%9.0g</td>
<td>Total academic hours</td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>%9.0g</td>
<td>1 = freshman, 2 = sophomore, 3 = junior, 4 = senior</td>
</tr>
<tr>
<td>number</td>
<td>int</td>
<td>%9.0g</td>
<td>number of students</td>
</tr>
</tbody>
</table>
```

Sorted by: year
To obtain a dataset containing the 25th percentile of gpa’s for each year, we type

```
collapse (p25) gpa [fw=number], by(year)
```

We used frequency weights.

Next we want to create a dataset containing the mean of gpa and hour for each year. We do not have to type `mean` to specify that we want the mean because the mean is reported by default.

```
use https://www.stata-press.com/data/r17/college, clear
collapse gpa hour [fw=number], by(year)
list
```

Now we want to create a dataset containing the mean and median of gpa and hour, and we want the median of gpa and hour to be stored as variables `medgpa` and `medhour`, respectively.

```
use https://www.stata-press.com/data/r17/college, clear
collapse (mean) gpa hour (median) medgpa=gpa medhour=hour [fw=num], by(year)
list
```

Here we want to create a dataset containing a count of gpa and hour and the minimums of gpa and hour. The minimums of gpa and hour will be stored as variables `mingpa` and `minhour`, respectively.
. use https://www.stata-press.com/data/r17/college, clear
. collapse (count) gpa hour (min) mingpa=gpa minhour=hour [fw=num], by(year)
. list

<table>
<thead>
<tr>
<th>year</th>
<th>gpa</th>
<th>hour</th>
<th>mingpa</th>
<th>minhour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>18</td>
<td>2.1</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>12</td>
<td>2.5</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>9</td>
<td>2.2</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>7</td>
<td>2.9</td>
<td>31</td>
</tr>
</tbody>
</table>

Now we replace the values of gpa in 3 of the observations with missing values.

. use https://www.stata-press.com/data/r17/college, clear
. replace gpa = . in 2/4
(3 real changes made, 3 to missing)
. list, sep(4)

<table>
<thead>
<tr>
<th>gpa</th>
<th>hour</th>
<th>year</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>30</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>.</td>
<td>34</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>.</td>
<td>28</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>.</td>
<td>30</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3.8</td>
<td>29</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2.5</td>
<td>30</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2.9</td>
<td>35</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3.7</td>
<td>30</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>35</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3.3</td>
<td>33</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.4</td>
<td>32</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2.9</td>
<td>31</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

If we now want to list the data containing the mean of gpa and hour for each year, collapse uses all observations on hour for year = 1, even though gpa is missing for observations 1–3.

. collapse gpa hour [fw=num], by(year)
. list

<table>
<thead>
<tr>
<th>year</th>
<th>gpa</th>
<th>hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.2</td>
<td>29.44444</td>
</tr>
<tr>
<td>2</td>
<td>2.991667</td>
<td>31.83333</td>
</tr>
<tr>
<td>3</td>
<td>3.233333</td>
<td>32.11111</td>
</tr>
<tr>
<td>4</td>
<td>3.257143</td>
<td>31.71428</td>
</tr>
</tbody>
</table>
If we repeat this process but specify the `cw` option, `collapse` ignores all observations that have missing values.

```
. use https://www.stata-press.com/data/r17/college, clear
. replace gpa = . in 2/4
   (3 real changes made, 3 to missing)
. collapse (mean) gpa hour [fw=num], by(year) cw
. list
```

```
<table>
<thead>
<tr>
<th>year</th>
<th>gpa</th>
<th>hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.2</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>2.99167</td>
<td>31.83333</td>
</tr>
<tr>
<td>3</td>
<td>3.23333</td>
<td>32.11111</td>
</tr>
<tr>
<td>4</td>
<td>3.25714</td>
<td>31.71428</td>
</tr>
</tbody>
</table>
```

Example 2

We have individual-level data from a census in which each observation is a person. Among other variables, the dataset contains the numeric variables `age`, `educ`, and `income` and the string variable `state`. We want to create a 50-observation dataset containing the means of age, education, and income for each state.

```
. collapse age educ income, by(state)
```

The resulting dataset contains means because `collapse` assumes that we want means if we do not specify otherwise. To make this explicit, we could have typed

```
. collapse (mean) age educ income, by(state)
```

Had we wanted the mean for `age` and `educ` and the median for `income`, we could have typed

```
. collapse (mean) age educ (median) income, by(state)
```

or if we had wanted the mean for `age` and `educ` and both the mean and the median for `income`, we could have typed

```
. collapse (mean) age educ income (median) medinc=income, by(state)
```

This last dataset will contain three variables containing means—`age`, `educ`, and `income`—and one variable containing the median of income—`medinc`. Because we typed `(median) medinc=income`, Stata knew to find the median for income and to store those in a variable named `medinc`. This renaming convention is necessary in this example because a variable named `income` containing the mean is also being created.
Variablewise or casewise deletion

Example 3

Let's assume that in our census data, we have 25,000 persons for whom age is recorded but only 15,000 for whom income is recorded; that is, income is missing for 10,000 observations. If we want summary statistics for age and income, collapse will, by default, use all 25,000 observations when calculating the summary statistics for age. If we prefer that collapse use only the 15,000 observations for which income is not missing, we can specify the cw (casewise) option:

```plaintext
  . collapse (mean) age income (median) medinc=income, by(state) cw
```

Weights

collapse allows all four weight types; the default is aweights. Weight normalization affects only the sum, count, sd, semean, and sebinomial statistics.

Let j index observations and i index by-groups. Here are the definitions for count and sum with weights:

count:

- unweighted: $N_i$, the number of observations in group i
- aweight: $N_i$, the number of observations in group i
- fweight, iweight, pweight: $\sum w_j$, the sum of the weights over observations in group i

sum:

- unweighted: $\sum x_j$, the sum of $x_j$ over observations in group i
- aweight: $\sum v_j x_j$ over observations in group i; $v_j =$ weights normalized to sum to $N_i$
- fweight, iweight, pweight: $\sum w_j x_j$ over observations in group i

When the by() option is not specified, the entire dataset is treated as one group.

The sd statistic with weights returns the square root of the bias-corrected variance, which is based on the factor $\sqrt{N_i/(N_i - 1)}$, where $N_i$ is the number of observations. Statistics sd, semean, sebinomial, and sepoisson are not allowed with pweighted data. Otherwise, the statistic is changed by the weights through the computation of the weighted count, as outlined above.

For instance, consider a case in which there are 25 observations in the dataset and a weighting variable that sums to 57. In the unweighted case, the weight is not specified, and the count is 25. In the analytically weighted case, the count is still 25; the scale of the weight is irrelevant. In the frequency-weighted case, however, the count is 57, the sum of the weights.

The rawsum statistic with aweights ignores the weight, with one exception: observations with zero weight will not be included in the sum.
Example 4

Using our same census data, suppose that instead of starting with individual-level data and aggregating to the state level, we started with state-level data and wanted to aggregate to the region level. Also assume that our dataset contains pop, the population of each state.

To obtain unweighted means and medians of age and income, by region, along with the total population, we could type

```
collapse (mean) age income (median) medage=age medinc=income (sum) pop,
> by(region)
```

To obtain weighted means and medians of age and income, by region, along with the total population and using frequency weights, we could type

```
collapse (mean) age income (median) medage=age medinc=income (count) pop
> [fweight=pop], by(region)
```

Note: Specifying (sum) pop would not have worked because that would have yielded the pop-weighted sum of pop. Specifying (count) age would have worked as well as (count) pop because count merely counts the number of nonmissing observations. The counts here, however, are frequency-weighted and equal the sum of pop.

To obtain the same mean and medians as above, but using analytic weights, we could type

```
collapse (mean) age income (median) medage=age medinc=income (rawsum) pop
> [aweight=pop], by(region)
```

Note: Specifying (count) pop would not have worked because, with analytic weights, count would count numbers of physical observations. Specifying (sum) pop would not have worked because sum would calculate weighted sums (with a normalized weight). The rawsum function, however, ignores the weights and sums only the specified variable, with one exception: observations with zero weight will not be included in the sum. rawsum would have worked as the solution to all three cases.

A final example

Example 5

We have census data containing information on each state’s median age, marriage rate, and divorce rate. We want to form a new dataset containing various summary statistics, by region, of the variables:
. use https://www.stata-press.com/data/r17/census5, clear
(1980 Census data by state)
. describe
Contains data from https://www.stata-press.com/data/r17/census5.dta
Observations: 50 1980 Census data by state
Variables: 7 6 Apr 2020 15:43

Variable Storage Display Value
name type format label Variable label
state str14 %14s State
state2 str2 %-2s Two-letter state abbreviation
region int %8.0g cenreg Census region
pop long %10.0g Population
median_age float %9.2f Median age
marriage_rate float %9.0g
divorce_rate float %9.0g

Sorted by: region
. collapse (median) median_age marriage divorce (mean) avgmrate=marriage > avgdrate=divorce [aw=pop], by(region)
. list

```
<table>
<thead>
<tr>
<th>region</th>
<th>median</th>
<th>marriage</th>
<th>divorce</th>
<th>avgmrate</th>
<th>avgdrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>31.90</td>
<td>.0080657</td>
<td>.0035295</td>
<td>.0081472</td>
<td>.0035359</td>
</tr>
<tr>
<td>N Cntrl</td>
<td>29.90</td>
<td>.0093821</td>
<td>.0048636</td>
<td>.0096701</td>
<td>.004961</td>
</tr>
<tr>
<td>South</td>
<td>29.60</td>
<td>.0112609</td>
<td>.0065792</td>
<td>.0117082</td>
<td>.0059439</td>
</tr>
<tr>
<td>West</td>
<td>29.90</td>
<td>.0089093</td>
<td>.0056423</td>
<td>.0125199</td>
<td>.0063464</td>
</tr>
</tbody>
</table>
```
. describe
Contains data
Observations: 4 1980 Census data by state
Variables: 6

Variable Storage Display Value
name type format label Variable label
region int %8.0g cenreg Census region
median_age float %9.2f (p 50) median_age
marriage_rate float %9.0g (p 50) marriage_rate
divorce_rate float %9.0g (p 50) divorce_rate
avgmrate float %9.0g (mean) marriage_rate
avgdrate float %9.0g (mean) divorce_rate

Sorted by: region
Note: Dataset has changed since last saved.

**Acknowledgment**

We thank David Roodman of the Open Philanthropy Project for writing `collapse2`, which inspired several features in `collapse`. 
Also see

[D] contract — Make dataset of frequencies and percentages
[D] egen — Extensions to generate
[D] statsby — Collect statistics for a command across a by list
[R] summarize — Summary statistics
Title

**compare —** Compare two variables

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Menu</th>
<th>Syntax</th>
<th>Remarks and examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

`compare` reports the differences and similarities between `varname_1` and `varname_2`.

Quick start

Describe differences in missing and defined values of `v1` and `v2`

```
compare v1 v2
```

As above, but only for observations where `catvar` is equal to 3

```
compare v1 v2 if catvar==3
```

As above, but for each level of `catvar`

```
by catvar: compare v1 v2
```

Menu

Data > Data utilities > Compare two variables

Syntax

```
compare varname_1 varname_2 [if] [in]
```

by is allowed; see [D] by.

Remarks and examples

> Example 1

One of the more useful accountings made by `compare` is the pattern of missing values:

```
use https://www.stata-press.com/data/r17/fullauto
(Automobile models)
. compare rep77 rep78
```

<table>
<thead>
<tr>
<th>Count</th>
<th>Difference</th>
<th>Minimum</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep77&lt;rep78</td>
<td>16</td>
<td>-3</td>
<td>-1.3125</td>
<td>-1</td>
</tr>
<tr>
<td>rep77=rep78</td>
<td>43</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rep77&gt;rep78</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Jointly defined 66 -3 -.2121212 1
rep77 missing only 3
Jointly missing 5

Total 74

75
We see that both `rep77` and `rep78` are missing in 5 observations and that `rep77` is also missing in 3 more observations.

Technical note

`compare` may be used with numeric variables, string variables, or both. When used with string variables, the summary of the differences (minimum, average, maximum) is not reported. When used with string and numeric variables, the breakdown by `<`, `=`, and `>` is also suppressed.

Also see

[D] `cf` — Compare two datasets
[D] `codebook` — Describe data contents
[D] `inspect` — Display simple summary of data’s attributes
**compress** — Compress data in memory

### Description

`compress` attempts to reduce the amount of memory used by your data.

### Quick start

Reduce the amount of memory used by the current dataset

```
   compress
```

As above, but only reduce memory used by `v1` and `v2`

```
   compress v1 v2
```

Speed up `compress` for large datasets with `strL`-type variables, but possibly reduce the amount of memory saved

```
   compress, nocoalesce
```

### Menu

Data > Data utilities > Optimize variable storage

### Syntax

```
   compress [varlist] [, nocoalesce]
```

### Option

`nocoalesce` specifies that `compress` not try to find duplicate values within `strL` variables in an attempt to save memory. If `nocoalesce` is not specified, `compress` must sort the data by each `strL` variable, which can be time consuming in large datasets.

### Remarks and examples

`compress` reduces the size of your dataset by considering two things. First, it considers demoting

- doubles to longs, ints, or bytes
- floats to ints or bytes
- longs to ints or bytes
- ints to bytes
- str#s to shorter str#s
- strLs to str#s
See 

[D] Data types for an explanation of these storage types.

Second, it considers coalescing strLs within each strL variable. That is to say, if a strL variable takes on the same value in multiple observations, compress can link those values to a single memory location to save memory. To check for this, compress must sort the data on each strL variable. You can use the nocoalesce option to tell compress not to take the time to perform this check. If compress does check whether it can coalesce strL values, it will do whichever saves more memory—coalescing strL values or demoting a strL to a str#—or it will do nothing if it cannot save memory by changing a strL.

compress leaves your data logically unchanged but (probably) appreciably smaller. compress never makes a mistake, results in loss of precision, or hacks off strings.

Example 1

If you do not specify a varlist, compress considers demoting all the variables in your dataset, so typing compress by itself is usually enough:

```
use https://www.stata-press.com/data/r17/compxmp2
(1978 automobile data)
.compress
variable mpg was float now byte
variable price was long now int
variable yenprice was double now long
variable weight was double now int
variable make was str26 now str17
(1,776 bytes saved)
```

If there are no compression possibilities, compress does nothing. For instance, typing compress again results in

```
.compress
(0 bytes saved)
```

Video example

How to optimize the storage of variables

Also see

[D] Data types — Quick reference for data types
[D] recast — Change storage type of variable
**Contract** — Make dataset of frequencies and percentages

### Description

`contract` replaces the dataset in memory with a new dataset consisting of all combinations of `varlist` that exist in the data and a new variable that contains the frequency of each combination.

### Quick start

**Frequency of each combination of v1 and v2 saved in `_freq`**

```
contract v1 v2
```

As above, but name new frequency variable `newf`

```
contract v1 v2, freq(newf)
```

Add percentage of total in `newp`

```
contract v1 v2, freq(newf) percent(newp)
```

Add cumulative frequency `newcf` and cumulative percentage `newcp`

```
contract v1 v2, freq(newf) percent(newp) cfreq(newcf) ///
    cpercent(newcp)
```

**Frequency of combinations excluding missing values**

```
contract v1 v2, nomiss
```

Add combinations with zero observations

```
contract v1 v2, nomiss zero
```

### Menu

- Data > Create or change data > Other variable-transformation commands > Make dataset of frequencies
**Syntax**

```
contract varlist [ if ] [ in ] [ weight ] [ , options ]
```

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>freq(newvar)</code></td>
<td>name of frequency variable; default is <code>_freq</code></td>
</tr>
<tr>
<td><code>cfreq(newvar)</code></td>
<td>create cumulative frequency variable</td>
</tr>
<tr>
<td><code>percent(newvar)</code></td>
<td>create percentage variable</td>
</tr>
<tr>
<td><code>cpercent(newvar)</code></td>
<td>create cumulative percentage variable</td>
</tr>
<tr>
<td><code>float</code></td>
<td>generate percentage variables as type <code>float</code></td>
</tr>
<tr>
<td><code>format(format)</code></td>
<td>display format for new percentage variables; default is <code>format(%8.2f)</code></td>
</tr>
<tr>
<td><code>zero</code></td>
<td>include combinations with frequency zero</td>
</tr>
<tr>
<td><code>nomiss</code></td>
<td>drop observations with missing values</td>
</tr>
</tbody>
</table>

`fweights` are allowed; see [U] 11.1.6 `weight`.

**Options**

- `freq(newvar)` specifies a name for the frequency variable. If not specified, `_freq` is used.
- `cfreq(newvar)` specifies a name for the cumulative frequency variable. If not specified, no cumulative frequency variable is created.
- `percent(newvar)` specifies a name for the percentage variable. If not specified, no percentage variable is created.
- `cpercent(newvar)` specifies a name for the cumulative percentage variable. If not specified, no cumulative percentage variable is created.
- `float` specifies that the percentage variables specified by `percent()` and `cpercent()` will be generated as variables of type `float`. If `float` is not specified, these variables will be generated as variables of type `double`. All generated variables are compressed to the smallest storage type possible without loss of precision; see [D] `compress`.
- `format(format)` specifies a display format for the generated percentage variables specified by `percent()` and `cpercent()`. If `format()` is not specified, these variables will have the display format `%8.2f`.
- `zero` specifies that combinations with frequency zero be included.
- `nomiss` specifies that observations with missing values on any variable in `varlist` be dropped. If `nomiss` is not specified, all observations possible are used.

**Remarks and examples**

`contract` takes the dataset in memory and creates a new dataset containing all combinations of `varlist` that exist in the data and a new variable that contains the frequency of each combination.
Sometimes you may want to collapse a dataset into frequency form. Several observations that have identical values on one or more variables will be replaced by one such observation, together with the frequency of the corresponding set of values. For example, in certain generalized linear models, the frequency of some combination of values is the response variable, so you would need to produce that response variable. The set of covariate values associated with each frequency is sometimes called a covariate class or covariate pattern. Such collapsing is reversible for the variables concerned, because the original dataset can be reconstituted by using `expand` (see `[D] expand`) with the variable containing the frequencies of each covariate class.

### Example 1

Suppose that we wish to collapse `auto2.dta` to a set of frequencies of the variables `rep78`, which takes values labeled “Poor”, “Fair”, “Average”, “Good”, and “Excellent”, and `foreign`, which takes values labeled “Domestic” and “Foreign”.

```
. use https://www.stata-press.com/data/r17/auto2
    (1978 automobile data)
. contract rep78 foreign
. list
```

<table>
<thead>
<tr>
<th>rep78</th>
<th>foreign</th>
<th>_freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poor</td>
<td>Domestic</td>
<td>2</td>
</tr>
<tr>
<td>Fair</td>
<td>Domestic</td>
<td>8</td>
</tr>
<tr>
<td>Average</td>
<td>Domestic</td>
<td>27</td>
</tr>
<tr>
<td>Average</td>
<td>Foreign</td>
<td>3</td>
</tr>
<tr>
<td>Good</td>
<td>Domestic</td>
<td>9</td>
</tr>
<tr>
<td>Good</td>
<td>Foreign</td>
<td>9</td>
</tr>
<tr>
<td>Excellent</td>
<td>Domestic</td>
<td>2</td>
</tr>
<tr>
<td>Excellent</td>
<td>Foreign</td>
<td>9</td>
</tr>
<tr>
<td>.</td>
<td>Domestic</td>
<td>4</td>
</tr>
<tr>
<td>.</td>
<td>Foreign</td>
<td>1</td>
</tr>
</tbody>
</table>

By default, `contract` uses the variable name `_freq` for the new variable that contains the frequencies. If `_freq` is in use, you will be reminded to specify a new variable name via the `freq()` option.
Specifying the zero option requests that combinations with frequency zero also be listed.

```
. use https://www.stata-press.com/data/r17/auto2, clear
   (1978 automobile data)
. contract rep78 foreign, zero
. list
```

<table>
<thead>
<tr>
<th>rep78</th>
<th>foreign</th>
<th>_freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poor</td>
<td>Domestic</td>
<td>2</td>
</tr>
<tr>
<td>Poor</td>
<td>Foreign</td>
<td>0</td>
</tr>
<tr>
<td>Fair</td>
<td>Domestic</td>
<td>8</td>
</tr>
<tr>
<td>Fair</td>
<td>Foreign</td>
<td>0</td>
</tr>
<tr>
<td>Average</td>
<td>Domestic</td>
<td>27</td>
</tr>
<tr>
<td>Average</td>
<td>Foreign</td>
<td>3</td>
</tr>
<tr>
<td>Good</td>
<td>Domestic</td>
<td>9</td>
</tr>
<tr>
<td>Good</td>
<td>Foreign</td>
<td>9</td>
</tr>
<tr>
<td>Excellent</td>
<td>Domestic</td>
<td>2</td>
</tr>
<tr>
<td>Excellent</td>
<td>Foreign</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Domestic</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Foreign</td>
<td>1</td>
</tr>
</tbody>
</table>

Acknowledgments

contract was written by Nicholas J. Cox (1998) of the Department of Geography at Durham University, UK, who is coeditor of the Stata Journal and author of Speaking Stata Graphics. The cfreq(), percent(), cpercent(), float, and format() options were written by Roger Newson of the Imperial College London.

Reference


Also see

[D] expand — Duplicate observations
[D] collapse — Make dataset of summary statistics
[D] duplicates — Report, tag, or drop duplicate observations
copy — Copy file from disk or URL

Description

copy copies an existing file to a file with a new name.

Quick start

Copy mydata.dta from C:\myfolder to C:\otherfolder

```
copy c:\myfolder\mydata.dta c:\otherfolder\
```

As above, but change dataset name to newdata.dta

```
copy c:\myfolder\mydata.dta c:\otherfolder\newdata.dta
```

As above, but replace newdata.dta if it exists

```
copy c:\myfolder\mydata.dta c:\otherfolder\newdata.dta, replace
```

Copy web-based Stata example dataset fullauto.dta to the current working directory

```
```

Syntax

```
copy filename1 filename2 [, options ]
```

filename1 may be a filename or a URL. filename2 may be the name of a file or a directory. If filename2 is a directory name, filename1 will be copied to that directory. filename2 may not be a URL.

Note: Double quotes may be used to enclose the filenames, and the quotes must be used if the filename contains embedded blanks.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>make filename2 readable by all</td>
</tr>
<tr>
<td>text</td>
<td>interpret filename1 as text file and translate to native text format</td>
</tr>
<tr>
<td>replace</td>
<td>may overwrite filename2</td>
</tr>
</tbody>
</table>

replace does not appear in the dialog box.
Options

**public** specifies that *filename* be readable by everyone; otherwise, the file will be created according to the default permissions of your operating system.

**text** specifies that *filename* be interpreted as a text file and be translated to the native form of text files on your computer. Computers differ on how end-of-line is recorded: Unix systems record one line-feed character, Windows computers record a carriage-return/line-feed combination, and Mac computers record just a carriage return. **text** specifies that *filename* be examined to determine how it has end-of-line recorded and that the line-end characters be switched to whatever is appropriate for your computer when the copy is made.

There is no reason to specify **text** when copying a file already on your computer to a different location because the file would already be in your computer’s format.

Do not specify **text** unless you know that the file is a text file; if the file is binary and you specify **text**, the copy will be useless. Most word processors produce binary files, not text files. The term **text**, as it is used here, specifies a particular way of recording textual information.

When other parts of Stata read text files, they do not care how lines are terminated, so there is no reason to translate end-of-line characters on that score. You specify **text** because you may want to look at the file with other software.

The following option is available with **copy** but is not shown in the dialog box:

**replace** specifies that *filename* be replaced if it already exists.

Remarks and examples

Examples:

Windows:

```
. copy orig.dta newcopy.dta
. copy mydir\orig.dta .
. copy orig.dta ../...
. copy "my document" "copy of document"
. copy ../mydir\doc.txt document\doc.tex
. copy https://www.stata.com/examples/simple.dta simple.dta
. copy https://www.stata.com/examples/simple.txt simple.txt, text
```

Mac and Unix:

```
. copy orig.dta newcopy.dta
. copy mydir/orig.dta .
. copy orig.dta ../...
. copy "my document" "copy of document"
. copy ../mydir/doc.txt document/doc.tex
. copy https://www.stata.com/examples/simple.dta simple.dta
. copy https://www.stata.com/examples/simple.txt simple.txt, text
```
Also see

[D] cd — Change directory
[D] dir — Display filenames
[D] erase — Erase a disk file
[D] mkdir — Create directory
[D] rmdir — Remove directory
[D] shell — Temporarily invoke operating system
[D] type — Display contents of a file
[U] 11.6 Filenaming conventions
**Description**

corr2data adds new variables with specified covariance (correlation) structure to the existing dataset or creates a new dataset with a specified covariance (correlation) structure. Singular covariance (correlation) structures are permitted. The purpose of this is to allow you to perform analyses from summary statistics (correlations/covariances and maybe the means) when these summary statistics are all you know and summary statistics are sufficient to obtain results. For example, these summary statistics are sufficient for performing analysis of \( t \) tests, variance, principal components, regression, and factor analysis. The recommended process is

```
. clear  (clear memory)
. corr2data ..., n(#) cov(...) ...  (create artificial data)
. regress ...  (use artificial data appropriately)
```

However, for factor analyses and principal components, the commands `factormat` and `pcamat` allow you to skip the step of using `corr2data`; see [MV] `factor` and [MV] `pca`.

The data created by `corr2data` are artificial; they are not the original data, and it is not a sample from an underlying population with the summary statistics specified. See [D] `drawnorm` if you want to generate a random sample. In a sample, the summary statistics will differ from the population values and will differ from one sample to the next.

The dataset `corr2data` creates is suitable for one purpose only: performing analyses when all that is known are summary statistics and those summary statistics are sufficient for the analysis at hand. The artificial data tricks the analysis command into producing the desired result. The analysis command, being by assumption only a function of the summary statistics, extracts from the artificial data the summary statistics, which are the same summary statistics you specified, and then makes its calculation based on those statistics.

If you doubt whether the analysis depends only on the specified summary statistics, you can generate different artificial datasets by using different seeds of the random-number generator (see the `seed()` option below) and compare the results, which should be the same within rounding error.

**Quick start**

Create dataset with 1,000 observations, \( v1 \) with mean of 3.4 and std. dev. of 1, \( v2 \) with mean of 3 and std. dev. of 0.5, and no correlation between \( v1 \) and \( v2 \)

```
corr2data v1 v2, n(1000) means(3.4 3) sds(1 .5)
```

As above, but with correlation between \( v1 \) and \( v2 \) specified in matrix `mymat`

```
corr2data v1 v2, n(1000) means(3.4 3) sds(1 .5) corr(mymat)
```
Menu
Data > Create or change data > Other variable-creation commands > Create dataset with specified correlation structure

Syntax

corr2data newvarlist [ , options ]

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>replace the current dataset</td>
</tr>
<tr>
<td>double</td>
<td>generate variable type as double; default is float</td>
</tr>
<tr>
<td>n(#)</td>
<td>generate # observations; default is current number</td>
</tr>
<tr>
<td>sds(vector)</td>
<td>standard deviations of generated variables</td>
</tr>
<tr>
<td>corr(matrix</td>
<td>vector)</td>
</tr>
<tr>
<td>cov(matrix</td>
<td>vector)</td>
</tr>
<tr>
<td>cstorage(full)</td>
<td>store correlation/covariance structure as a symmetric $k \times k$ matrix</td>
</tr>
<tr>
<td>cstorage(lower)</td>
<td>store correlation/covariance structure as a lower triangular matrix</td>
</tr>
<tr>
<td>cstorage(upper)</td>
<td>store correlation/covariance structure as an upper triangular matrix</td>
</tr>
<tr>
<td>forcepsd</td>
<td>force the covariance/correlation matrix to be positive semidefinite</td>
</tr>
<tr>
<td>means(vector)</td>
<td>means of generated variables; default is means(0)</td>
</tr>
<tr>
<td>seed(#)</td>
<td>seed for random-number generator</td>
</tr>
</tbody>
</table>

Options

<table>
<thead>
<tr>
<th>Main</th>
<th>clear specifies that it is okay to replace the dataset in memory, even though the current dataset has not been saved on disk.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>double specifies that the new variables be stored as Stata doubles, meaning 8-byte reals. If double is not specified, variables are stored as floats, meaning 4-byte reals. See [D] Data types.</td>
</tr>
<tr>
<td></td>
<td>n(#) specifies the number of observations to be generated; the default is the current number of observations. If n(#) is not specified or is the same as the current number of observations, corr2data adds the new variables to the existing dataset; otherwise, corr2data replaces the dataset in memory.</td>
</tr>
<tr>
<td></td>
<td>sds(vector) specifies the standard deviations of the generated variables. sds() may not be specified with cov().</td>
</tr>
<tr>
<td></td>
<td>corr(matrix</td>
</tr>
<tr>
<td></td>
<td>cov(matrix</td>
</tr>
</tbody>
</table>
cstorage(full | lower | upper) specifies the storage mode for the correlation or covariance structure in corr() or cov(). The following storage modes are supported:

- **full** specifies that the correlation or covariance structure is stored (recorded) as a symmetric \( k \times k \) matrix.
- **lower** specifies that the correlation or covariance structure is recorded as a lower triangular matrix. With \( k \) variables, the matrix should have \( \frac{k(k + 1)}{2} \) elements in the following order:

\[
C_{11} \ C_{21} \ C_{22} \ C_{31} \ C_{32} \ C_{33} \ \cdots \ C_{k1} \ C_{k2} \ \cdots \ C_{kk}
\]

- **upper** specifies that the correlation or covariance structure is recorded as an upper triangular matrix. With \( k \) variables, the matrix should have \( \frac{k(k + 1)}{2} \) elements in the following order:

\[
C_{11} \ C_{12} \ C_{13} \ \cdots \ C_{1k} \ C_{22} \ C_{23} \ \cdots C_{2k} \ \cdots C_{(k-1)k-1} \ C_{(k-1)k} \ C_{kk}
\]

Specifying cstorage(full) is optional if the matrix is square. cstorage(lower) or cstorage(upper) is required for the vectorized storage methods. See Storage modes for correlation and covariance matrices in [D] ddrawnorm for examples.

forcepsd modifies the matrix \( C \) to be positive semidefinite (psd) and to thus be a proper covariance matrix. If \( C \) is not positive semidefinite, it will have negative eigenvalues. By setting the negative eigenvalues to 0 and reconstructing, we obtain the least-squares positive-semidefinite approximation to \( C \). This approximation is a singular covariance matrix.

means(vector) specifies the means of the generated variables. The default is means(0).

Options

- **seed(#)** specifies the seed of the random-number generator used to generate data. # defaults to 0. The random numbers generated inside corr2data do not affect the seed of the standard random-number generator.

Remarks and examples

corr2data is designed to enable analyses of correlation (covariance) matrices by commands that expect variables rather than a correlation (covariance) matrix. corr2data creates variables with exactly the correlation (covariance) that you want to analyze. Apart from means and covariances, all aspects of the data are meaningless. Only analyses that depend on the correlations (covariances) and means produce meaningful results. Thus you may perform a paired \( t \) test ([R] ttest) or an ordinary regression analysis ([R] regress), etc.

If you are not sure that a statistical result depends only on the specified summary statistics and not on other aspects of the data, you can generate different datasets, each having the same summary statistics but other different aspects, by specifying the seed() option. If the statistical results differ beyond what is attributable to roundoff error, then using corr2data is inappropriate.
Example 1

We first run a regression using the auto dataset.

```
. use https://www.stata-press.com/data/r17/auto
  (1978 automobile data)
. regress weight length trunk
```

```
Source | SS        df   MS           Number of obs =  74
        |           |         |                F(2, 71) = 303.95
Model   | 39482774.4 2  19741387.2 Prob > F = 0.0000
Residual| 4611403.95 71  64949.3513 R-squared = 0.8954
        |           |         | Adj R-squared = 0.8925
Total   | 44094178.4 73  604029.841 Root MSE =  254.85
```

```
weight | Coefficient Std. err.   t    P>|t|          [95% conf. interval]
length | 33.83435     1.949751 17.35 0.000      29.94666   37.72204
trunk  | -5.835155   10.14957  -0.57 0.567    -26.07282  14.40252
_cons  | -3258.84     283.3547 -11.50 0.000   -3823.833 -2693.846
```

Suppose that, for some reason, we no longer have the auto dataset. Instead, we know the means and covariance matrices of weight, length, and trunk, and we want to do the same regression again. The matrix of means is

```
. matrix list M
M[1,3]
weight length trunk
_cons  3019.4595 187.93243 13.756757
```

and the covariance matrix is

```
. matrix list V
symmetric V[3,3]
weight       length       trunk
weight 604029.84
length 16370.922   495.78989
trunk 2234.6612    69.202518  18.296187
```

To do the regression analysis in Stata, we need to create a dataset that has the specified correlation structure.

```
. corr2data x y z, n(74) cov(V) means(M)
. regress x y z
```

```
Source | SS        df   MS           Number of obs =  74
        |           |         |                F(2, 71) = 303.95
Model   | 39482773.3 2  19741386.6 Prob > F = 0.0000
Residual| 4611402.75 71  64949.3345 R-squared = 0.8954
        |           |         | Adj R-squared = 0.8925
Total   | 44094176 73  604029.809 Root MSE =  254.85
```

```
x | Coefficient Std. err.   t    P>|t|          [95% conf. interval]
y |  33.83435     1.949751 17.35 0.000      29.94666   37.72204
   z | -5.835155   10.14957  -0.57 0.567    -26.07282  14.40252
  _cons | -3258.84     283.3547 -11.50 0.000   -3823.833 -2693.846
```

The results from the regression based on the generated data are the same as those based on the real data.
Methods and formulas

Two steps are involved in generating the desired dataset. The first step is to generate a zero-mean, zero-correlated dataset. The second step is to apply the desired correlation structure and the means to the zero-mean, zero-correlated dataset. In both steps, we take into account that, given any matrix $A$ and any vector of variables $X$, $\text{Var}(A'X) = A'\text{Var}(X)A$.

Reference


Also see

[D] Data types — Quick reference for data types
[D] drawnorm — Draw sample from multivariate normal distribution
**count — Count observations satisfying specified conditions**

**Description**

`count` counts the number of observations that satisfy the specified conditions. If no conditions are specified, `count` displays the number of observations in the data.

**Quick start**

Count the number of observations

```stata
count
```

As above, but where `catvar` equals 3

```stata
count if catvar==3
```

Count observations for each value of `catvar`

```stata
by catvar: count
```

**Menu**

Data > Data utilities > Count observations satisfying condition

**Syntax**

```
count [if] [in]
```

by and collect are allowed; see [U] 11.1.10 Prefix commands.

**Remarks and examples**

`count` may strike you as an almost useless command, but it can be one of Stata’s handiest.

**Example 1**

How many times have you obtained a statistical result and then asked yourself how it was possible? You think a moment and then mutter aloud, “Wait a minute. Is income ever *negative* in these data?” or “Is sex ever equal to 3?” `count` can quickly answer those questions:

```stata
.use https://www.stata-press.com/data/r17/countxmpl
(1980 Census data by state)
.count
641
```
We have 641 observations. *income* is never negative. *sex*, however, takes on the value 3 once. When we decompose the count by *division*, we see that it takes on that odd value in the Pacific division.

### Stored results

`count` stores the following in `r()`

Scalars

- `r(N)` number of observations

### References


### Also see

[R] `tabulate oneway` — One-way table of frequencies
**Title**

**cross — Form every pairwise combination of two datasets**

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Menu</th>
<th>Syntax</th>
</tr>
</thead>
</table>

**Description**

cross forms every pairwise combination of the data in memory with the data in `filename`. If `filename` is specified without a suffix, `.dta` is assumed.

**Quick start**

Form every pairwise combination of observations from `mydata1.dta` in memory with observations from `mydata2.dta`

```
cross using mydata2
```

**Menu**

Data > Combine datasets > Form every pairwise combination of two datasets

**Syntax**

```
cross using filename
```

**Remarks and examples**

This command is rarely used; also see [D] _joinby_, [D] _merge_, and [D] _append_.

Crossing refers to merging two datasets in every way possible. That is, the first observation of the data in memory is merged with every observation of `filename`, followed by the second, and so on. Thus the result will have $N_1N_2$ observations, where $N_1$ and $N_2$ are the number of observations in memory and in `filename`, respectively.

Typically, the datasets will have no common variables. If they do, such variables will take on only the values of the data in memory.

**Example 1**

We wish to form a dataset containing all combinations of three age categories and two sexes to serve as a stub. The three age categories are 20, 30, and 40. The two sexes are male and female:
. input str6 sex
   sex
   1. male
   2. female
   3. end
. save sex
file sex.dta saved
. drop _all
. input agecat
   agecat
   1. 20
   2. 30
   3. 40
   4. end
. cross using sex
. list

<table>
<thead>
<tr>
<th>agecat</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>male</td>
</tr>
<tr>
<td>30</td>
<td>male</td>
</tr>
<tr>
<td>40</td>
<td>male</td>
</tr>
<tr>
<td>20</td>
<td>female</td>
</tr>
<tr>
<td>30</td>
<td>female</td>
</tr>
<tr>
<td>40</td>
<td>female</td>
</tr>
</tbody>
</table>

References
Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. College Station, TX: Stata Press.

Also see
[D] append — Append datasets
[D] fillin — Rectangularize dataset
[D] joinby — Form all pairwise combinations within groups
[D] merge — Merge datasets
[D] save — Save Stata dataset
Data types — Quick reference for data types

Description

This entry provides a quick reference for data types allowed by Stata. See [U] 12 Data for details.

Remarks and examples

<table>
<thead>
<tr>
<th>Storage type</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Closest to 0 without being 0</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>−127</td>
<td>100</td>
<td>±1</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>−32,767</td>
<td>32,740</td>
<td>±1</td>
<td>2</td>
</tr>
<tr>
<td>long</td>
<td>−2,147,483,647</td>
<td>2,147,483,620</td>
<td>±1</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>−1.70141173319 × 10^{38}</td>
<td>1.70141173319 × 10^{38}</td>
<td>±10^{−38}</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>−8.9884656743 × 10^{307}</td>
<td>8.9884656743 × 10^{307}</td>
<td>±10^{−323}</td>
<td>8</td>
</tr>
</tbody>
</table>

Precision for float is $3.795 \times 10^{−8}$.
Precision for double is $1.414 \times 10^{−16}$.

Strings are stored as str#, for instance, str1, str2, str3, ..., str2045, or as strL. The number after the str indicates the maximum length of the string. A str5 could hold the word “male”, but not the word “female” because “female” has six characters. A strL can hold strings of arbitrary lengths, up to 2000000000 characters, and can even hold binary data containing embedded \0 characters.

Stata keeps data in memory, and you should record your data as parsimoniously as possible. If you have a string variable that has maximum length 6, it would waste memory to store it as a str20. Similarly, if you have an integer variable, it would be a waste to store it as a double.
Precision of numeric storage types

**floats** have about 7 digits of accuracy; the magnitude of the number does not matter. Thus, 1234567 can be stored perfectly as a float, as can 1234567e+20. The number 123456789, however, would be rounded to 123456792. In general, this rounding does not matter.

If you are storing identification numbers, the rounding could matter. If the identification numbers are integers and take 9 digits or less, store them as longs; otherwise, store them as doubles. **doubles** have 16 digits of accuracy.

Stata stores numbers in binary, and this has a second effect on numbers less than 1. 1/10 has no perfect binary representation just as 1/11 has no perfect decimal representation. In float, .1 is stored as .1000000149011612. Note that there are 7 digits of accuracy, just as with numbers larger than 1. Stata, however, performs all calculations in double precision. If you were to store 0.1 in a float called x and then ask, say, list if x==.1, there would be nothing in the list. The .1 that you just typed was converted to double, with 16 digits of accuracy (.100000000000000014...), and that number is never equal to 0.1 stored with float accuracy.

One solution is to type list if x==float(.1). The float() function rounds its argument to float accuracy; see [FN] Programming functions. The other alternative would be store your data as double, but this is probably a waste of memory. Few people have data that is accurate to 1 part in 10 to the 7th. Among the exceptions are banks, who keep records accurate to the penny on amounts of billions of dollars. If you are dealing with such financial data, store your dollar amounts as doubles.

Also see

[D] compress — Compress data in memory
[D] destring — Convert string variables to numeric variables and vice versa
[D] encode — Encode string into numeric and vice versa
[D] format — Set variables’ output format
[D] recast — Change storage type of variable
[U] 12.2.2 Numeric storage types
[U] 12.4 Strings
[U] 12.5 Formats: Controlling how data are displayed
[U] 13.12 Precision and problems therein
**Description**

These commands calculate, display, save, and verify checksums of the data, which taken together form what is called a *signature*. An example signature is `162:11(12321):2725060400:4007406597`. That signature is a function of the values of the variables and their names, and thus the signature can be used later to determine whether a dataset has changed.

- `datasignature` without arguments calculates and displays the signature of the data in memory.
- `datasignature set` does the same, and it stores the signature as a characteristic in the dataset. You should save the dataset afterward so that the signature becomes a permanent part of the dataset.
- `datasignature confirm` verifies that, were the signature recalculated this instant, it would match the one previously set. `datasignature confirm` displays an error message and returns a nonzero return code if the signatures do not match.
- `datasignature report` displays a full report comparing the previously set signature to the current one.

In the above, the signature is stored in the dataset and accessed from it. The signature can also be stored in a separate, small file.

- `datasignature set, saving(filename)` calculates and displays the signature and, in addition to storing it as a characteristic in the dataset, also saves the signature in `filename`.
- `datasignature confirm using filename` verifies that the current signature matches the one stored in `filename`.
- `datasignature report using filename` displays a full report comparing the current signature with the one stored in `filename`.

In all the above, if `filename` is specified without an extension, `.dtasig` is assumed.

- `datasignature clear` clears the signature, if any, stored in the characteristics of the dataset in memory.

**Quick start**

Calculate and display the signature of the dataset in memory

`datasignature`

As above, and store the signature as a characteristic of the data

`datasignature set`

As above, but also save the signature in `datasig.txt`

`datasignature set, saving(datasig.txt)`
Confirm that the data are currently exactly the same as they were when signed
   datasignature confirm

Confirm that the data in memory have the same signature saved in datasig.txt
   datasignature confirm using datasig.txt

Menu
   Data > Other utilities > Manage data signature

Syntax

   datasignature

   datasignature set [, reset]

   datasignature confirm [, strict]

   datasignature report

   datasignature set, saving(filename[, replace]) [reset]

   datasignature confirm using filename [, strict]

   datasignature report using filename

   datasignature clear

   collect is allowed; see [U] 11.1.10 Prefix commands.

Options

   reset is used with datasignature set. It specifies that even though you have previously set a
   signature, you want to erase the old signature and replace it with the current one.

   strict is for use with datasignature confirm. It specifies that, in addition to requiring that the
   signatures match, you also wish to require that the variables be in the same order and that no new
   variables have been added to the dataset. (If any variables were dropped, the signatures would not
   match.)

   saving(filename[, replace]) is used with datasignature set. It specifies that, in addition to
   storing the signature in the dataset, you want a copy of the signature saved in a separate file. If
   filename is specified without a suffix, .dtasig is assumed. The replace suboption allows
   filename to be replaced if it already exists.
Remarks and examples

Remarks are presented under the following headings:

Using datasignature interactively
  Example 1: Verification at a distance
  Example 2: Protecting yourself from yourself
  Example 3: Working with assistants
  Example 4: Working with shared data

Using datasignature in do-files
Interpreting data signatures
The logic of data signatures

Using datasignature interactively

datasignature is useful in the following cases:

1. You and a coworker, separated by distance, have both received what is claimed to be the same dataset. You wish to verify that it is.

2. You work interactively and realize that you could mistakenly modify your data. You wish to guard against that.

3. You want to give your dataset to an assistant to improve the labels and the like. You wish to verify that the data returned to you are the same data.

4. You work with an important dataset served on a network drive. You wish to verify that others have not changed it.

Example 1: Verification at a distance

You load the data and type

```
. datasignature
74:12(71728):3831085005:1395876116
```

Your coworker does the same with his or her copy. You compare the two signatures.

Example 2: Protecting yourself from yourself

You load the data and type

```
. datasignature set
74:12(71728):3831085005:1395876116 (data signature set)
. save, replace
```

From then on, you periodically type

```
. datasignature confirm
(data unchanged since 19feb2021 14:24)
```

One day, however, you check and see the message:

```
. datasignature confirm
(data unchanged since 19feb2021 14:24, except 2 variables have been added)
```
You can find out more by typing 

```
datasignature report
```

(Data signature set on Monday 19feb2021 14:24)

**Data signature summary**

1. Previous data signature 74:12(71728):3831085005:1395876116
2. Same data signature today (same as 1)

**Comparison of current data with previously set data signature**

<table>
<thead>
<tr>
<th>variables</th>
<th>number</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>original # of variables</td>
<td>12</td>
<td>(values unchanged)</td>
</tr>
<tr>
<td>added variables</td>
<td>2</td>
<td>(1)</td>
</tr>
<tr>
<td>dropped variables</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Resulting # of variables 14

(1) Added variables are agesquared logincome.

You could now either drop the added variables or decide to incorporate them:

```
datasignature set
```

(data signature already set -- specify option reset)

```
n(110)
datasignature set, reset
```

74:14(113906):1142538197:2410350265 (data signature reset)

Concerning the detailed report, three data signatures are reported: 1) the stored signature, 2) the signature that would be calculated today on the basis of the same variables in their original order, and 3) the signature that would be calculated today on the basis of all the variables and in their current order.

```
datasignature confirm
```

knew that new variables had been added because signature 1 was equal to signature 2. If some variables had been dropped, however, `datasignature confirm` would not be able to determine whether the remaining variables had changed.

**Example 3: Working with assistants**

You give your dataset to an assistant to have variable labels and the like added. You wish to verify that the returned data are the same data.

Saving the signature with the dataset is inadequate here. Your assistant, having your dataset, could change both your data and the signature and might even do that in a desire to be helpful. The solution is to save the signature in a separate file that you do not give to your assistant:

```
datasignature set, saving(mycopy)
```

74:12(71728):3831085005:1395876116 (data signature set)

(file mycopy.dtasig saved)

You keep file mycopy.dtasig. When your assistant returns the dataset to you, you use it and compare the current signature to what you have stored in mycopy.dtasig:

```
datasignature confirm using mycopy
```

(data unchanged since 19feb2021 15:05)

By the way, the signature is a function of the following:

- The number of observations and number of variables in the data
- The values of the variables
- The names of the variables
The order in which the variables occur in the dataset

The storage types of the individual variables

The signature is not a function of variable labels, value labels, notes, and the like.

Example 4: Working with shared data

You work on a dataset served on a network drive, which means that others could change the data. You wish to know whether this occurs.

The solution here is the same as working with an assistant: you save the signature in a separate, private file on your computer,

```
. datasignature set, saving(private)
74:12(71728):3831085005:1395876116 (data signature set)
(file \texttt{private.dtasig} saved)
```

and then you periodically check the signature by typing

```
. datasignature confirm using private
(data unchanged since 15mar2021 11:22)
```

Using datasignature in do-files

`datasignature confirm` aborts with error if the signatures do not match:

```
. datasignature confirm
  data have changed since 19feb2021 15:05
r(9);
```

This means that, if you use `datasignature confirm` in a do-file, execution of the do-file will be stopped if the data have changed.

You may want to specify the `strict` option. `strict` adds two more requirements: that the variables be in the same order and that no new variables have been added. Without `strict`, these are not considered errors:

```
. datasignature confirm
  (data unchanged since 19feb2021 15:22)
. datasignature confirm, strict
  (data unchanged since 19feb2021 15:05, but order of variables has changed)
r(9);
```

and

```
. datasignature confirm
  (data unchanged since 19feb2021 15:22, except 1 variable has been added)
. datasignature confirm, strict
  (data unchanged since 19feb2021 15:22, except 1 variable has been added)
r(9);
```

If you keep logs of your analyses, issuing `datasignature` or `datasignature confirm` immediately after loading each dataset is a good idea. This way, you have a permanent record that you can use for comparison.
Interpreting data signatures

An example signature is 74:12(71728):3831085005:1395876116. The components are
1. 74, the number of observations;
2. 12, the number of variables;
3. 71728, a checksum function of the variable names and the order in which they occur; and
4. 3831085005 and 1395876116, checksum functions of the values of the variables, calculated two different ways.

Two signatures are equal only if all their components are equal.

Two different datasets will probably not have the same signature, and it is even more unlikely that datasets containing similar values will have equal signatures. There are two data checksums, but do not read too much into that. If either data checksum changes, even just a little, the data have changed. Whether the change in the checksum is large or small—or in one, the other, or both—signifies nothing.

The logic of data signatures

The components of a data signature are known as checksums. The checksums are many-to-one mappings of the data onto the integers. Let's consider the checksums of auto.dta carefully.

The data portion of auto.dta contains 38,184 bytes. There are \(2^{38184}\) such datasets or, equivalently, \(2^{305472}\). The first checksum has \(2^{48}\) possible values, and it can be proven that those values are equally distributed over the \(2^{305472}\) datasets. Thus there are \(2^{305472}/2^{48} - 1 = 2^{305424} - 1\) datasets that have the same first checksum value as auto.dta. The same can be said for the second checksum. It would be difficult to prove, but we believe that the two checksums are conditionally independent, being based on different bit shifts and bit shuffles of the same data. Of the \(2^{305424} - 1\) datasets that have the same first checksum as auto.dta, the second checksum should be equally distributed over them. Thus there are about \(2^{305376} - 1\) datasets with the same first and second checksums as auto.dta.

Now let’s consider those \(2^{305376} - 1\) other datasets. Most of them look nothing like auto.dta. The checksum formulas guarantee that a change of one variable in 1 observation will lead to a change in the calculated result if the value changed is stored in 4 or fewer bytes, and they nearly guarantee it in other cases. When it is not guaranteed, the change cannot be subtle—“Chevrolet” will have to change to binary junk, or a double-precision 1 to \(-6.476678983751e+301\), and so on. The change will be easily detected if you summarize your data and just glance at the minimums and maximums. If the data look at all like auto.dta, which is unlikely, they will look like a corrupted version.

More interesting are offsetting changes across observations. For instance, can you change one variable in 1 observation and make an offsetting change in another observation so that, taken together, they will go undetected? You can fool one of the checksums, but fooling both of them simultaneously will prove difficult. The basic rule is that the more changes you make, the easier it is to create a dataset with the same checksums as auto.dta, but by the time you have done that, the data will look nothing like auto.dta.
Datasignature — Determine whether data have changed

Stored results

Datasignature without arguments and datasignature set store the following in r():

Macros

\[ r(\text{datasignature}) \]
the signature

Datasignature confirm stores the following in r():

Scalars

\[ r(k_{\text{added}}) \]
number of variables added

Macros

\[ r(\text{datasignature}) \]
the signature

datasignature confirm aborts execution if the signatures do not match and so then returns nothing except a return code of 9.

Datasignature report stores the following in r():

Scalars

\[ r(\text{datetime}) \]
\( \%tc \) date–time when set
\[ r(\text{changed}) \]
. if \( r(k_{\text{dropped}}) \neq 0 \), otherwise
0 if data have not changed, 1 if data have changed
\[ r(\text{reordered}) \]
1 if variables reordered, 0 if not reordered.
. if \( r(k_{\text{added}}) \neq 0 \) \( \lor \) \( r(k_{\text{dropped}}) \neq 0 \)
\[ r(k_{\text{original}}) \]
number of original variables
\[ r(k_{\text{added}}) \]
number of added variables
\[ r(k_{\text{dropped}}) \]
number of dropped variables

Macros

\[ r(\text{origdatasignature}) \]
original signature
\[ r(\text{curdatasignature}) \]
current signature on same variables, if it can be calculated
\[ r(\text{fulldatasignature}) \]
current full-data signature
\[ r(\text{varsadded}) \]
variable names added
\[ r(\text{varsdropped}) \]
variable names dropped

datasignature clear stores nothing in r() but does clear it.

Datasignature set stores the signature in the following characteristics:

Characteristic

\[ _\text{dta}[\text{datasignature}_\text{si}] \]
signature
\[ _\text{dta}[\text{datasignature}_\text{dt}] \]
\( \%tc \) date–time when set in \%21x format
\[ _\text{dta}[\text{datasignature}_\text{vl1}] \]
part 1, original variables
\[ _\text{dta}[\text{datasignature}_\text{vl2}] \]
part 2, original variables, if necessary
etc.

To access the original variables stored in \[ _\text{dta}[\text{datasignature}_\text{vl1}] \], etc., from an ado-file, code

\texttt{mata: ado_fromlchar("vars", ",_dta", \"datasignature_vl\")}

Thereafter, the original variable list would be found in ‘vars’.

Methods and formulas

datasignature is implemented using \_datasignature; see \[P\] \_datasignature.
Reference


Also see

[P] _datasignature — Determine whether data have changed

[P] _signestimationsample — Determine whether the estimation sample has changed
**Datetime — Date and time values and variables**

**Description**

This entry provides a complete overview of Stata’s date and time values. We discuss functions used to obtain Stata dates, including string-to-numeric conversions and conversions among different types of dates and times.

Stata’s date and time values need to be formatted so they look like the dates and times we are familiar with. We show basic formatting options here, but more details can be found in [D] Datetime display formats.

[D] Datetime conversion has more details on converting dates and times stored as strings to numerically encoded Stata dates and times.

[D] Datetime values from other software discusses getting Stata dates from dates created by other software.

[D] Datetime durations describes functions designed to get durations (for example, ages) from two Stata dates or to express a duration in different units.

[D] Datetime relative dates describes functions that return dates based on other dates, for example, the date of a birthday in another year.

[D] Datetime business calendars describes business calendars—using dates with nonbusiness days (for example, weekends and holidays) removed. You can use existing calendars or create your own; see [D] Datetime business calendars creation.

For an alphabetical listing of all the datetime functions, see [FN] Date and time functions.

**Quick start**

Convert the string variable `strdate`, with dates such as "January 1, 2020", to a numerically encoded Stata date

```stata
generate numdate = date(strdate, "MDY")
```

Format `numdate` to make it readable when displayed

```stata
format numdate %td
```

Convert the string variable `strtime`, with dates and times such as "January 1,2020 10:30 am", to a numerically encoded Stata datetime variable

```stata
generate double numtime = clock(strtime, "MDYhm")
```

Format `numtime` to make it readable when displayed

```stata
format numtime %tc
```

Convert the string variable `strmonthly`, with monthly dates such as "2012-04", to a Stata date, and format it to make it readable when displayed

```stata
generate nummonth = monthly(strmonthly, "YM")
format nummonth %tm
```
List observations for which `numdate` is prior to February 15, 2013

```stata
list if numdate < td(15/2/2013)
```

Create a monthly date variable from numeric variables `year` and `month`

```stata
generate monthly = ym(year, month)
```

Create a daily date variable from the datetimes stored in `numtime`

```stata
generate dateoftime = dofc(numtime)
```

Create a monthly date variable from the daily dates stored in `numdate`

```stata
generate monthlyofdate = mofd(numdate)
```

Create a new variable with the month of the daily dates stored in `numdate`

```stata
generate monthnum = month(numdate)
```

### Syntax

Syntax is presented under the following headings:

- Types of dates and how they are displayed
- How Stata dates are stored
- Converting dates stored as strings to Stata dates
- Formatting Stata dates for display
- Creating dates from components
- Converting among units
- Extracting time-of-day components from datetimes
- Extracting date components from daily dates
- Typing dates into expressions

### Types of dates and how they are displayed

Dates and times can take many forms; below, we list the types of dates that are supported in Stata. Note that throughout our documentation, we use the term “datetime” to refer to variables that record time or date and time.

<table>
<thead>
<tr>
<th>Date type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime</td>
<td>20jan2010 09:15:22.120</td>
</tr>
<tr>
<td>date</td>
<td>20jan2010, 20/01/2010, ...</td>
</tr>
<tr>
<td>weekly date</td>
<td>2010w3</td>
</tr>
<tr>
<td>monthly date</td>
<td>2010m1</td>
</tr>
<tr>
<td>quarterly date</td>
<td>2010q1</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>2010h1</td>
</tr>
<tr>
<td>yearly date</td>
<td>2010</td>
</tr>
</tbody>
</table>

The styles of the dates in the table above are merely examples; dates can be displayed in a number of ways. Perhaps you prefer 2010.01.20; Jan. 20, 2010; 2010-1; etc.
How Stata dates are stored

Stata dates are numeric values that record durations (positive or negative) from 01jan1960. Below, we list the numeric values corresponding to the dates displayed in the table in the previous section.

<table>
<thead>
<tr>
<th>Stata date type</th>
<th>Examples</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>1,579,598,122,120</td>
<td>milliseconds since 01jan1960 00:00:00.000, assuming 86,400 s/day</td>
</tr>
<tr>
<td>datetime/C</td>
<td>1,579,598,146,120</td>
<td>milliseconds since 01jan1960 00:00:00.000, adjusted for leap seconds*</td>
</tr>
<tr>
<td>date</td>
<td>18,282</td>
<td>days since 01jan1960 (01jan1960 = 0)</td>
</tr>
<tr>
<td>weekly date</td>
<td>2,601</td>
<td>weeks since 1960w1</td>
</tr>
<tr>
<td>monthly date</td>
<td>600</td>
<td>months since 1960m1</td>
</tr>
<tr>
<td>quarterly date</td>
<td>200</td>
<td>quarters since 1960q1</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>100</td>
<td>half-years since 1960h1</td>
</tr>
<tr>
<td>yearly date</td>
<td>2010</td>
<td>years since 0000</td>
</tr>
</tbody>
</table>

* Datetime/C is equivalent to coordinated universal time (UTC). In UTC, leap seconds are periodically inserted because the length of the mean solar day is slowly increasing. See Why there are two datetime encodings in [D] Datetime conversion.

Stata dates are stored as regular Stata numeric variables.

You can convert dates stored as strings to Stata dates by using the string-to-numeric conversion functions; see Converting dates stored as strings to Stata dates.

You can make Stata dates readable by placing the appropriate %fmt on the numeric variable; see Formatting Stata dates for display.

You can convert from one Stata date type to another by using conversion functions; see Converting among units.

Storing dates as numeric values is convenient because you can subtract them to obtain time between dates, for example,

\[
\text{datetime2} - \text{datetime1} = \text{milliseconds between datetime1 and datetime2}
\]

(divide by 1,000 to obtain seconds)

\[
\text{date2} - \text{date1} = \text{days between date1 and date2}
\]

\[
\text{week2} - \text{week1} = \text{weeks between week1 and week2}
\]

\[
\text{month2} - \text{month1} = \text{months between month1 and month2}
\]

\[
\text{half2} - \text{half1} = \text{half-years between half1 and half2}
\]

\[
\text{year2} - \text{year1} = \text{years between year1 and year2}
\]

For time differences in other units, for example, the number of years between date1 and date2, see [D] Datetime durations.
Converting dates stored as strings to Stata dates

To convert dates and times stored as strings to Stata dates and times, use one of the functions listed below.

<table>
<thead>
<tr>
<th>Stata date type</th>
<th>Function</th>
<th>Required variable precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>clock(str, mask)</td>
<td>double</td>
</tr>
<tr>
<td>datetime/C</td>
<td>Clock(str, mask)</td>
<td>double</td>
</tr>
<tr>
<td>date</td>
<td>date(str, mask)</td>
<td>float or long</td>
</tr>
<tr>
<td>weekly date</td>
<td>weekly(str, mask)*</td>
<td>float or int</td>
</tr>
<tr>
<td>monthly date</td>
<td>monthly(str, mask)*</td>
<td>float or int</td>
</tr>
<tr>
<td>quarterly date</td>
<td>quarterly(str, mask)*</td>
<td>float or int</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>halfyearly(str, mask)*</td>
<td>float or int</td>
</tr>
<tr>
<td>yearly date</td>
<td>yearly(str, mask)</td>
<td>float or int</td>
</tr>
</tbody>
</table>

* str is a string variable or a literal string enclosed in quotes.

Within each function, you need to specify the string you want to convert and the order in which the date and time components appear in that string.

The string to be converted with clock(), Clock(), and date() may contain dates and times that are run together or include punctuation marks between the components. However, the functions marked with an asterisk require that the string date contain a space or punctuation between the year and the other component if the string consists only of numbers. For more information on how punctuation is handled and other details related to these conversion functions, see [D] Datetime conversion.

The order of the components is specified within quotes, such as "YMD", and is referred to as a mask. The mask may contain the following elements:

<table>
<thead>
<tr>
<th>Mask element</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>day</td>
</tr>
<tr>
<td>W</td>
<td>week</td>
</tr>
<tr>
<td>M</td>
<td>month</td>
</tr>
<tr>
<td>Q</td>
<td>quarter</td>
</tr>
<tr>
<td>H</td>
<td>half-year</td>
</tr>
<tr>
<td>Y</td>
<td>year</td>
</tr>
<tr>
<td>19Y</td>
<td>two-digit year in the 1900s</td>
</tr>
<tr>
<td>20Y</td>
<td>two-digit year in the 2000s</td>
</tr>
<tr>
<td>h</td>
<td>hour</td>
</tr>
<tr>
<td>m</td>
<td>minute</td>
</tr>
<tr>
<td>s</td>
<td>second</td>
</tr>
<tr>
<td>#</td>
<td>placeholder for something to be ignored</td>
</tr>
</tbody>
</table>
Examples:

1. You have datetimes stored in the string variable `mystr`, an example being `2010.07.12 14:32`. To convert this to a Stata datetime/c variable, you type
   
   ```
   . generate double eventtime = clock(mystr, "YMDhm")
   ```

   The string contains the year, month, and day followed by the hour and minute, so you specify the mask "YMDhm".

2. You have datetimes stored in `mystr`, an example being `2010.07.12 14:32:12`. You type
   
   ```
   . generate double eventtime = clock(mystr, "YMDhms")
   ```

   Mask element s specifies seconds. In example 1, there were no seconds; in this example, there are.

3. You have datetimes stored in `mystr`, an example being `2010 Jul 12 14:32`. You type
   
   ```
   . generate double eventtime = clock(mystr, "YMDhm")
   ```

   This is the same command that you typed in example 1. In the mask, you specify the order of the components; Stata figures out the style for itself. In example 1, months were numeric. In this example, they are spelled out (and happen to be abbreviated).

4. You have datetimes stored in `mystr`, an example being `July 12, 2010 2:32 PM`. You type
   
   ```
   . generate double eventtime = clock(mystr, "MDYhm")
   ```

   Stata automatically looks for AM and PM, in uppercase and lowercase, with and without periods.

5. You have datetimes stored in `mystr`, an example being `7-12-10 14.32`. The 2-digit year is to be interpreted as being prefixed with 20. You type
   
   ```
   . generate double eventtime = clock(mystr, "MD20Yhm")
   ```

6. You have datetimes stored in `mystr`, an example being `14:32 on 7/12/2010`. You type
   
   ```
   . generate double eventtime = clock(mystr, "hm#MDY")
   ```

   The # sign between m and M means “ignore one thing between minute and month”, which in this case is the word “on”. Had you omitted the # from the mask, the new variable `eventtime` would have contained missing values.

7. You have a date stored in `mystr`, an example being `22/7/2010`. In this case, you want to create a Stata date instead of a datetime. You type
   
   ```
   . generate eventdate = date(mystr, "DMY")
   ```

   Typing
   
   ```
   . generate double eventtime = clock(mystr, "DMY")
   ```

   would have worked, too. Variable `eventtime` would contain a different coding from that contained by `eventdate`; namely, it would contain milliseconds from 1jan1960 rather than days (1,595,376,000,000 rather than 18,465). Datetime value 1,595,376,000,000 corresponds to 22jul2010 00:00:00.000.
Formatting Stata dates for display

While Stata dates are stored as regular Stata numeric variables, they are formatted so they look like the dates and times we are familiar with. Each type of date has a corresponding display format, and we list them below:

<table>
<thead>
<tr>
<th>Stata date type</th>
<th>Display format</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>%tc</td>
</tr>
<tr>
<td>datetime/C</td>
<td>%tC</td>
</tr>
<tr>
<td>date</td>
<td>%td</td>
</tr>
<tr>
<td>weekly date</td>
<td>%tw</td>
</tr>
<tr>
<td>monthly date</td>
<td>%tm</td>
</tr>
<tr>
<td>quarterly date</td>
<td>%tq</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>%th</td>
</tr>
<tr>
<td>yearly date</td>
<td>%ty</td>
</tr>
</tbody>
</table>

The display formats above are the simplest forms of each of the Stata dates. You can control how each type of Stata date is displayed; see [D] Datetime display formats.

Examples:

1. You have datetimes stored in string variable mystr, an example being 2010.07.12 14:32. To convert this to a Stata datetime/c variable and make the new variable readable when displayed, you type

   . generate double eventtime = clock(mystr, "YMDhm")
   . format eventtime %tc

2. You have a date stored in mystr, an example being 22/7/2010. To convert this to a Stata date variable and make the new variable readable when displayed, you type

   . generate eventdate = date(mystr, "DMY")
   . format eventdate %td
Creating dates from components

If you have components of your date stored separately, you can use the following functions to create a single date variable. Note that each component used in this function must be numeric; you can specify numeric variables or simply digits.

<table>
<thead>
<tr>
<th>Stata date type</th>
<th>Function to build from components</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>\texttt{mdyhms(M, D, Y, h, m, s)*}</td>
</tr>
<tr>
<td></td>
<td>\texttt{dhms(e_d, h, m, s)*†}</td>
</tr>
<tr>
<td></td>
<td>\texttt{hms(h, m, s)*}</td>
</tr>
<tr>
<td>datetime/C</td>
<td>\texttt{Cmdyhms(M, D, Y, h, m, s)*}</td>
</tr>
<tr>
<td></td>
<td>\texttt{Cdhmse_d, h, m, s)*†}</td>
</tr>
<tr>
<td></td>
<td>\texttt{Chms(h, m, s)*}</td>
</tr>
<tr>
<td>date</td>
<td>\texttt{mdy(M, D, Y)}</td>
</tr>
<tr>
<td>weekly date</td>
<td>\texttt{yw(Y, W)}</td>
</tr>
<tr>
<td>monthly date</td>
<td>\texttt{ym(Y, M)}</td>
</tr>
<tr>
<td>quarterly date</td>
<td>\texttt{yq(Y, Q)}</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>\texttt{yh(Y, H)}</td>
</tr>
<tr>
<td>yearly date</td>
<td>\texttt{y(Y)}</td>
</tr>
</tbody>
</table>

* Stata datetime variables must be stored as doubles.
† \(e_d\) is a Stata date with a month, day, and year component.

Examples:

1. Your dataset has three variables, \(\text{mo}\), \(\text{da}\), and \(\text{yr}\), with each variable containing a date component in numeric form. To create a date variable from these components, you type
   
   \[
   \text{. generate eventdate = mdy(mo, da, yr)} \\
   \text{. format eventdate %td}
   \]

2. Your dataset has two numeric variables, \(\text{mo}\) and \(\text{yr}\). To create a date variable corresponding to the first day of the month, you type
   
   \[
   \text{. generate eventdate = mdy(mo, 1, yr)} \\
   \text{. format eventdate %td}
   \]

3. Your dataset has two numeric variables, \(\text{da}\) and \(\text{yr}\), and one string variable, \(\text{month}\), containing the spelled-out month. In this case, do not use the building-from-component functions. Instead, construct a new string variable with these components, and then convert the string to a Stata date using the conversion functions:
   
   \[
   \text{. generate str work = month + " " + string(da) + " " + string(yr)} \\
   \text{. generate eventdate = date(work, "MDY")} \\
   \text{. format eventdate %td}
   \]
Converting among units

The table below lists the functions for converting one type of date and time to another. Because there are not official functions for every possible conversion, we have also included the functions you can nest instead to obtain those conversions. Similarly, for any other conversion not listed here, you can use two functions, going through date or datetime as appropriate. For example, to obtain a monthly date from a datetime/c variable, you would use `mofd(dofc(varname))`.

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>datetime/c</td>
<td>Cofc()</td>
<td>dofc()</td>
</tr>
<tr>
<td>datetime/C</td>
<td></td>
<td>cofC()</td>
<td>dofC()</td>
</tr>
<tr>
<td>date</td>
<td></td>
<td>cofd()</td>
<td>Cofd()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date</td>
<td>wofd()</td>
<td>mofd()</td>
</tr>
<tr>
<td>weekly</td>
<td></td>
<td>dofW()</td>
<td>mofd(dofW())</td>
</tr>
<tr>
<td>monthly</td>
<td></td>
<td>dofM()</td>
<td>wofd(dofM())</td>
</tr>
<tr>
<td>quarterly</td>
<td></td>
<td>dofQ()</td>
<td>wofd(dofQ())</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>From:</th>
<th>To:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date</td>
<td>hofd()</td>
<td>yofd()</td>
</tr>
<tr>
<td>half-yearly</td>
<td></td>
<td>dofH()</td>
<td></td>
</tr>
<tr>
<td>yearly</td>
<td></td>
<td>dofY()</td>
<td></td>
</tr>
</tbody>
</table>

Note that if you are converting to a date type for which you do not have all the components, those missing elements will be set to their defaults. For example, converting a yearly date to a weekly date would give you the first week of each year. Converting a quarterly date to a monthly date would give you the first month of each quarter, along with the year, of course. Below, we list the defaults for the date and time components:

<table>
<thead>
<tr>
<th>Date component</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>1960</td>
</tr>
<tr>
<td>half-year</td>
<td>1</td>
</tr>
<tr>
<td>quarter</td>
<td>1</td>
</tr>
<tr>
<td>month</td>
<td>1</td>
</tr>
<tr>
<td>week</td>
<td>1</td>
</tr>
<tr>
<td>day</td>
<td>01</td>
</tr>
<tr>
<td>hour</td>
<td>00</td>
</tr>
<tr>
<td>minute</td>
<td>00</td>
</tr>
<tr>
<td>second</td>
<td>00</td>
</tr>
</tbody>
</table>
Examples:

1. You have the Stata datetime/c variable `eventtime` and wish to create the new variable `eventdate` containing just the date from the datetime variable. You type
   
   . generate eventdate = dofc(eventtime)
   . format eventdate %td

2. You have the daily date `eventdate` and wish to create the new datetime/c variable `eventtime` from it. For this unusual case, you can even type
   
   . generate double eventtime = cofd(eventdate)
   . format eventtime %tc

   The time components of the new variable will be set to the default 00:00:00.000.

3. You have the Stata quarterly variable `eventqtr` and wish to create the new Stata date variable `eventdate` from it. You type
   
   . generate eventdate = dofq(eventqtr)
   . format eventdate %tq

   The new variable, `eventdate`, will contain 01jan dates for quarter 1, 01apr dates for quarter 2, 01jul dates for quarter 3, and 01oct dates for quarter 4.

4. You have the datetime/c variable `admittime` and wish to create the quarterly variable `admitqtr` from it. You type
   
   . generate admitqtr = qofd(dofc(admittime))
   . format admitqtr %tq

   Because there is no `qofc()` function, you use `qofd(dofc())`.

Extracting time-of-day components from datetimes

In the table below, we list the functions used to extract time-of-day components from datetimes. If you are working with standard datetimes, use the functions in the datetime/c column. If you are working with leap second–adjusted times, use the functions in the datetime/C column.

<table>
<thead>
<tr>
<th>Desired component</th>
<th>Function</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour of day</td>
<td><code>hh(etc)</code></td>
<td><code>hhC(etc)</code></td>
<td>14</td>
</tr>
<tr>
<td>minutes of day</td>
<td><code>mm(etc)</code></td>
<td><code>mmC(etc)</code></td>
<td>42</td>
</tr>
<tr>
<td>seconds of day</td>
<td><code>ss(etc)</code></td>
<td><code>ssC(etc)</code></td>
<td>57.123</td>
</tr>
<tr>
<td>year, month, day, hour, minute, second, or millisecond</td>
<td><code>clockpart(etc, su)</code></td>
<td><code>Clockpart(etc, su)</code></td>
<td>2020</td>
</tr>
</tbody>
</table>

`etc` is a Stata datetime/c value.

`etc` is a Stata datetime/C value (UTC time with leap seconds).

`su` is a string specifying the time unit. `su` can be string "year" or "y" for year; "month" or "mon" for month; "day" or "d" for day; "hour" or "h" for hour; "minute" or "min" for minute; "second", "sec", or "s" for second; and "millisecond" or "ms" for millisecond (case insensitive).

Notes:

$0 \leq \text{hh}(etc) \leq 23$, $0 \leq \text{hhC}(etc) \leq 23$

$0 \leq \text{mm}(etc) \leq 59$, $0 \leq \text{mmC}(etc) \leq 59$

$0 \leq \text{ss}(etc) < 60$, $0 \leq \text{ssC}(etc) < 61$ (sic)
Example:

1. You have the Stata datetime/c variable `admittime`. You wish to create the new variable `admithour` equal to the hour and fraction of hour within the day of admission. You type

   `. generate admithour = hh(admittime) + mm(admittime)/60 + ss(admittime)/3600`

2. You have the Stata datetime/C variable `admitTime`. You wish to create the new variable `admityear` to record the year of admission. You type

   `. generate admityear = Clockpart(admitTime, "year")`

See [D] Datetime durations for other functions that can be used to calculate durations.

---

### Extracting date components from daily dates

You might be working with dates that have more information than you need. For example, daily dates refer to dates that have a month, day, and year component. If you want to refer only to the month, or year, of a daily date, you can use the extraction functions below.

<table>
<thead>
<tr>
<th>Desired component</th>
<th>Function*</th>
<th>Example†</th>
</tr>
</thead>
<tbody>
<tr>
<td>calendar year</td>
<td><code>year(e_d)</code></td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td><code>datepart(e_d, &quot;year&quot;)</code></td>
<td>2013</td>
</tr>
<tr>
<td>calendar month</td>
<td><code>month(e_d)</code></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td><code>datepart(e_d, &quot;month&quot;)</code></td>
<td>7</td>
</tr>
<tr>
<td>calendar day</td>
<td><code>day(e_d)</code></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td><code>datepart(e_d, &quot;day&quot;)</code></td>
<td>5</td>
</tr>
<tr>
<td>day of week</td>
<td><code>dow(e_d)</code></td>
<td>2</td>
</tr>
<tr>
<td>(0=Sunday)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Julian day of year</td>
<td><code>doy(e_d)</code></td>
<td>186</td>
</tr>
<tr>
<td>(1=first day)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>week within year</td>
<td><code>week(e_d)</code></td>
<td>27</td>
</tr>
<tr>
<td>(1=first week)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quarter within year</td>
<td><code>quarter(e_d)</code></td>
<td>3</td>
</tr>
<tr>
<td>(1=first quarter)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>half within year</td>
<td><code>halfyear(e_d)</code></td>
<td>2</td>
</tr>
<tr>
<td>(1=first half)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* `e_d` is a Stata date with a month, day, and year component.
† All examples are with `e_d = mdy(7,5,2013)`.

All functions require a numeric Stata daily date as an argument. A string variable cannot be specified as the date. To extract components from other Stata date types, use the appropriate conversion function to convert to a daily date. For example, `quarter(dofq(qvar))` would return the quarter of the quarterly date values stored in `qvar`. 
Examples:

1. You wish to obtain the day of week Sunday, Monday, ... corresponding to the daily date variable `eventdate`. You type

   . `generate day_of_week = dow(eventdate)`

   The new variable, `day_of_week`, contains 0 for Sunday, 1 for Monday, ..., 6 for Saturday.

2. You wish to obtain the day of week Sunday, Monday, ... corresponding to the datetime/c variable `eventtime`. You type

   . `generate day_of_week = dow(dofc(eventtime))`

3. You have the daily date variable `evdate` and wish to create the new date variable `evdate_r` from it. `evdate_r` will contain the same date as `evdate` but rounded back to the first of the month. You type

   . `generate evdate_r = mdy(month(evdate), 1, year(evdate))`

   In the above solution, we used the date-component extraction functions `month()` and `year()` and used the build-from-components function `mdy()`.

Typing dates into expressions

You can type date values by just typing the number, such as 16,237 or 1,402,920,000,000, as in

   . `generate before = cond(hiredon < 16237, 1, 0) if !missing(hiredon)`
   . `drop if admittedon < 1402920000000`

Easier to type is

   . `generate before = cond(hiredon < td(15jun2004), 1, 0) if !missing(hiredon)`
   . `drop if admittedon < tc(15jun2004 12:00:00)`

You can type Stata date values by typing the date inside `td()`, as in `td(15jun2004)`. You can type Stata datetime/c values by typing the datetime inside `tc()`, as in `tc(15jun2004 12:00:00)`.

`td()` and `tc()` are called pseudofunctions because they translate what you type into their numerical equivalents. Pseudofunctions require only that you specify the datetime components in the expected order, so rather than `15jun2004` above, we could have specified 15 June 2004, 15-6-2004, or 15/6/2004.

The pseudofunctions and their expected component order are

<table>
<thead>
<tr>
<th>Desired date type</th>
<th>Pseudofunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td><code>tc([day-month-year] hh:mm:[]ss[][.sss][])</code></td>
</tr>
<tr>
<td>datetime/C</td>
<td><code>tc([day-month-year] hh:mm:[]ss[][.sss][])</code></td>
</tr>
<tr>
<td>date</td>
<td><code>td(day-month-year)</code></td>
</tr>
<tr>
<td>weekly date</td>
<td><code>tw(year-week)</code></td>
</tr>
<tr>
<td>monthly date</td>
<td><code>tm(year-month)</code></td>
</tr>
<tr>
<td>quarterly date</td>
<td><code>tq(year-quarter)</code></td>
</tr>
<tr>
<td>half-yearly date</td>
<td><code>th(year-half)</code></td>
</tr>
<tr>
<td>yearly date</td>
<td>none necessary; years are numeric and can be typed directly</td>
</tr>
</tbody>
</table>
Note that the *day-month-year* in `tc()` and `tC()` are optional. If you omit them, 01jan1960 is assumed. Doing so produces time as an offset, which can be useful in, for example,

```
.generate six_hrs_later = eventtime + tc(6:00)
```

Note that string-to-date functions can be used in expressions with literal strings. For example, `date("15jun2004","DMY")` gives the same result as `td(15jun2004)`.

---

### Remarks and examples

Remarks are presented under the following headings:

- **Introduction**
- **Example 1: Converting string datetimes to Stata datetimes**
- **Example 2: Extracting date components**
- **Example 3: Building dates from components**
- **Example 4: Converting among date types**
- **Example 5: Using dates in expressions**

---

### Introduction

To use dates in Stata, you must first convert what you have to a Stata date. Stata dates are numbers, so they can easily be translated from, say, daily dates to monthly dates. Even so, they can be formatted so that they look like the dates you are familiar with. If you have dates stored as strings, you must first convert them to Stata dates.

Converting a string date to a Stata date is as simple as telling Stata the string date and the order of the components. For example, we have a fictional dataset on patients who visited a local hospital. We have their birthdates, the dates of their visits, the reasons for their visits, and the dates they were discharged. All dates and times are stored as strings.

```
.use https://www.stata-press.com/data/r17/visits
(Fictional hospital visit data)
.describe
Contains data from https://www.stata-press.com/data/r17/visits.dta
Observations: 5 Fictional hospital visit data
Variables: 7 27 Aug 2020 22:56
   variable name type format label value label
  ----------- ------ ------- -------- ------------------------
     patid    byte  %9.0g  Patient ID
 dateofbirth str9  %9s Date of birth
      reason str15  %15s Reason for visit
     admit_d str8  %9s Admission date
     admit_t str17  %17s Admission date and time
   discharge_d str9  %9s Discharge date
 discharge_t str14  %14s Discharge date and time
```

Sorted by:
If we wanted to sort our data by birthdates or use these dates to compute a patient’s age, we would need these variables to be numeric, not strings. So let’s create numeric Stata dates from the birthdates and dates of admission:

```
. generate admit = date(admit_d, "YMD")
. generate dob = date(dateofbirth, "MDY")
```

For dates of admission, we told Stata that the string date was stored in `admit_d` and that the date was stored in the following order: year, month, day (YMD). Similarly, for birthdates we specify the string date and the order of the components: month, day, and year (MDY). It does not matter whether the month is written as a number, spelled out completely, or abbreviated to three letters.

You might be surprised by the values listed. The numbers represent the days elapsed since January 1, 1960, Stata’s base date. Most software store dates and times in this manner, but they differ in the date they choose as a base. For us to understand the dates that these values represent, we apply a display format. All datetime display formats begin with a `%t` and contain a second letter representing the type of date: `%td` for daily dates, `%tw` for weekly dates, and so on. In our case, we have daily dates, so we use the `%td` format.

```
. format admit dob %td
. list admit dob
```

If we instead had weekly dates, monthly dates, or quarterly dates, we would use the appropriate string-to-numeric conversion function to create the numeric variable and the appropriate display format. For more ways to format the dates above, see [D] Datetime display formats.

This is a simple example to get us started. The key points are that we want our dates to be stored numerically and formatted so that they look like the dates we are familiar with.
Below, we will discuss how to work with other types of dates. We will explore dates that have a time component, dates with components stored in multiple variables, and dates that have more components than we wish to work with. So whether you need to build, extract, or convert among different types of dates, you will learn how to do so with the examples that follow.

Example 1: Converting string datetimes to Stata datetimes

In this dataset, we also have string variables that record the date and time of admission and discharge:

```
.codebook admit_t discharge_t

<table>
<thead>
<tr>
<th>admit_t</th>
<th>Type: String (str17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique values: 5</td>
<td>Missing &quot;&quot;: 0/5</td>
</tr>
<tr>
<td>Tabulation: Freq. Value</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20110313 8:30:45&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20110409 10:17:08&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20110625 5:15:06&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20120211 10:30:12&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20120801 6:45:59&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Warning: Variable has embedded blanks.

<table>
<thead>
<tr>
<th>discharge_t</th>
<th>Type: String (str14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique values: 5</td>
<td>Missing &quot;&quot;: 0/5</td>
</tr>
<tr>
<td>Tabulation: Freq. Value</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20110326 2:15&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20110409 19:35&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20110629 10:27&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20120216 2:15&quot;</td>
<td></td>
</tr>
<tr>
<td>1 &quot;20120802 11:59&quot;</td>
<td></td>
</tr>
</tbody>
</table>

Warning: Variable has embedded blanks.

Let’s convert these to Stata dates. Regardless if we are working with simple dates or dates and times, the process is the same. We are going to specify the string we want to convert and the order of the components. The only difference between this example and the previous example is the function; because these variables record the date and time, we will now use the `clock()` function, and the variables we generate will be referred to as datetime variables.

```
.generate double admit_time = clock(admit_t, "YMDhms")
.generate double disch_time = clock(discharge_t, "YMDhms")
.format admit_time disch_time %tc
.list admit_time disch_time
```

<table>
<thead>
<tr>
<th>admit_time</th>
<th>disch_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>25jun2011 05:15:06</td>
<td>29jun2011 10:27:00</td>
</tr>
<tr>
<td>13mar2011 08:30:45</td>
<td>26mar2011 02:15:00</td>
</tr>
<tr>
<td>09apr2011 10:17:08</td>
<td>09apr2011 19:35:00</td>
</tr>
<tr>
<td>11feb2012 10:30:12</td>
<td>16feb2012 02:15:00</td>
</tr>
<tr>
<td>01aug2012 06:45:59</td>
<td>02aug2012 11:59:00</td>
</tr>
</tbody>
</table>
Note that the string variable admit\_t contained the hour, minutes, and seconds, whereas the string variable discharge\_t contained only the hour and minutes. This is why we did not specify an s in the list of components for discharge\_t, and it is also why the seconds are set to zero for disch\_time.

These variables now record the milliseconds since 01jan1960 00:00:00.000, assuming 86,400 seconds per day. You might have guessed that these values will be quite large, which is why we need to use the most precise storage type in Stata, double.

We have a lot of information in these variables, but we can choose to view just the portion in which we are interested by modifying the display format. For example, below we specify that we want to display only the hour and minute for the time of discharge, and we list the newly formatted time alongside the original string variable.

```
. format disch_time %tcHH:MM
. list discharge_t disch_time
```

```
<table>
<thead>
<tr>
<th>discharge_t</th>
<th>disch_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20110629</td>
<td>10:27</td>
</tr>
<tr>
<td>20110326</td>
<td>02:15</td>
</tr>
<tr>
<td>20110409</td>
<td>19:35</td>
</tr>
<tr>
<td>20120216</td>
<td>02:15</td>
</tr>
<tr>
<td>20120802</td>
<td>11:59</td>
</tr>
</tbody>
</table>
```

We created the datetime variables above assuming there are 86,400 seconds in a day. This is one way to record time; another way would be to use UTC. UTC times are adjusted for leap seconds and can be obtained by modifying our commands just slightly, as follows:

```
. generate double admit\_Time = Clock(admit\_t, "YMDhms")
. format admit\_Time %tC
```

Notice that the Clock() function and the %tC display format both contain a capital C. When you are working with standard datetimes, you will use functions with a lowercase c, and for UTC times, you will use functions with an uppercase C.

**Example 2: Extracting date components**

Suppose we want to work with just the month or year of admission. We can extract these components from our Stata date variable:

```
. generate admonth = month(admit)
. generate adyear = year(admit)
. list admit admonth adyear
```

```
<table>
<thead>
<tr>
<th>admit</th>
<th>admonth</th>
<th>adyear</th>
</tr>
</thead>
<tbody>
<tr>
<td>25jun2011</td>
<td>6</td>
<td>2011</td>
</tr>
<tr>
<td>13mar2011</td>
<td>3</td>
<td>2011</td>
</tr>
<tr>
<td>09apr2011</td>
<td>4</td>
<td>2011</td>
</tr>
<tr>
<td>11feb2012</td>
<td>2</td>
<td>2012</td>
</tr>
<tr>
<td>01aug2012</td>
<td>8</td>
<td>2012</td>
</tr>
</tbody>
</table>
```
Now, for each year, we can look at the patients that were admitted in the first three months and the reason for their visit:

```stata
bysort adyear: list patid reason if admonth < 4
```

<table>
<thead>
<tr>
<th>adyear</th>
<th>patid</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>2</td>
<td>chest pain</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>adyear</th>
<th>patid</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td>4</td>
<td>abdominal pain</td>
</tr>
</tbody>
</table>

**Example 3: Building dates from components**

If we are concerned only with the month and year of admission, we can also create a monthly date with the two newly created variables above:

```stata
generate monthly = ym(adyear, admonth)
format monthly %tm
list admit monthly
```

<table>
<thead>
<tr>
<th>admit monthly</th>
<th>monthly</th>
</tr>
</thead>
<tbody>
<tr>
<td>25jun2011</td>
<td>2011m6</td>
</tr>
<tr>
<td>13mar2011</td>
<td>2011m3</td>
</tr>
<tr>
<td>09apr2011</td>
<td>2011m4</td>
</tr>
<tr>
<td>11feb2012</td>
<td>2012m2</td>
</tr>
<tr>
<td>01aug2012</td>
<td>2012m8</td>
</tr>
</tbody>
</table>

Because we now have monthly dates, we apply the `%tm` display format.

The `ym()` function shown above is useful when you have components of a date stored separately. In fact, we could have created this monthly date variable by nesting functions:

```stata
generate monthly2 = ym(year(admit), month(admit))
format monthly2 %tm
```

Instead of generating those intermediary variables to extract the month and year of the daily date, we simply used the extraction functions `year()` and `month()` within the `ym()` function. Either of the two methods shown above will give you the same result, but if your goal is to convert a daily date variable to a monthly date, you can use the `mofd()` conversion function, as demonstrated in the next example.
Example 4: Converting among date types

Often, we need to modify the data from its raw form for our purposes. For example, suppose our dataset included only the datetime variable `admit_time` but we were interested only in the date. We could type

```
. generate dateoftime = dofc(admit_time)
. format dateoftime %td
. list admit_time dateoftime
```

<table>
<thead>
<tr>
<th>admit_time</th>
<th>dateoftime</th>
</tr>
</thead>
<tbody>
<tr>
<td>25jun2011 05:15:06</td>
<td>25jun2011</td>
</tr>
<tr>
<td>13mar2011 08:30:45</td>
<td>13mar2011</td>
</tr>
<tr>
<td>09apr2011 10:17:08</td>
<td>09apr2011</td>
</tr>
<tr>
<td>11feb2012 10:30:12</td>
<td>11feb2012</td>
</tr>
<tr>
<td>01aug2012 06:45:59</td>
<td>01aug2012</td>
</tr>
</tbody>
</table>

Or we might want to create a monthly date from the date of admission:

```
. generate monthofdate = mofd(admit)
. format monthofdate %tm
. list admit monthofdate
```

<table>
<thead>
<tr>
<th>admit</th>
<th>monthofdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>25jun2011</td>
<td>2011m6</td>
</tr>
<tr>
<td>13mar2011</td>
<td>2011m3</td>
</tr>
<tr>
<td>09apr2011</td>
<td>2011m4</td>
</tr>
<tr>
<td>11feb2012</td>
<td>2012m2</td>
</tr>
<tr>
<td>01aug2012</td>
<td>2012m8</td>
</tr>
</tbody>
</table>

Several functions are available for converting from one type of date and time to another. But, if one is not available for what you need, you can nest functions to obtain the conversion you want. For example, suppose we would like to convert a monthly date to a quarterly date. There is no direct function for this conversion, so instead we type

```
. generate quarterly = qofd(dofm(monthofdate))
. format quarterly %tq
. list monthofdate quarterly
```

<table>
<thead>
<tr>
<th>monthofdate</th>
<th>quarterly</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011m6</td>
<td>2011q2</td>
</tr>
<tr>
<td>2011m3</td>
<td>2011q1</td>
</tr>
<tr>
<td>2011m4</td>
<td>2011q2</td>
</tr>
<tr>
<td>2012m2</td>
<td>2012q1</td>
</tr>
<tr>
<td>2012m8</td>
<td>2012q3</td>
</tr>
</tbody>
</table>

We use the `dofm()` function to convert the monthly date to a daily date. This daily date will contain the month and year from the monthly date, and the day will be set to 1. This is the general rule with datetime functions; if you are converting from one type of date to another that has more elements, those elements are set to their defaults. The `qofd()` function then converts the resulting daily date to a quarterly date.
Example 5: Using dates in expressions

Besides generating date and time variables, you might use dates in expressions. For example, suppose we wanted to look only at observations after a certain date. Let’s list visit information for any patients who were admitted after February 20, 2012:

```
. list admit patid reason if admit > td(20feb2012)
```

<table>
<thead>
<tr>
<th>admit</th>
<th>patid</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>01aug2012</td>
<td>5</td>
<td>rapid breathing</td>
</tr>
</tbody>
</table>

This `td()` function will convert February 20, 2012, to its numeric form. Our expression is then evaluated by comparing this numeric value with the numeric values stored in `admit`.

If you would like to see that underlying numeric value, you can type

```
. display td(20feb2012)
```

References


Also see

[D] Datetime business calendars — Business calendars
[D] Datetime conversion — Converting strings to Stata dates
[D] Datetime display formats — Display formats for dates and times
[D] Datetime durations — Obtaining and working with durations
[D] Datetime relative dates — Obtaining dates and date information from other dates
[D] Datetime values from other software — Date and time conversion from other software
Description

Stata provides user-definable business calendars.

Syntax

Apply business calendar format

\[ \text{format varlist } \%tbcalname \]

Apply detailed date format with business calendar format

\[ \text{format varlist } \%tbcalname[ :datetime-specifiers ] \]

Convert between business dates and regular dates

\{\text{generate}|\text{replace}\} \ bdate = \text{bofd("calname", regulardate)}
\{\text{generate}|\text{replace}\} \ regulardate = \text{dofb(bdate, "calname")}

File calname.stbcal contains the business calendar definition.

Details of the syntax follow:

1. Definition.
   Business calendars are regular calendars with some dates crossed out:

   November 2011
   \[
   \begin{array}{cccccc}
   \text{Su} & \text{Mo} & \text{Tu} & \text{We} & \text{Th} & \text{Fr} & \text{Sa} \\
   1 & 2 & 3 & 4 & X \\
   X & 7 & 8 & 9 & 10 & 11 & X \\
   X & 14 & 15 & 16 & 17 & 18 & X \\
   X & 21 & 22 & 23 & X & X & X \\
   X & 28 & 29 & 30 & & & \\
   \end{array}
   \]

   A date that appears on the business calendar is called a business date. 11nov2011 is a business date. 12nov2011 is not a business date with respect to this calendar.

   Crossed-out dates are literally omitted. That is,

   \[ 18\text{nov}2011 + 1 = 21\text{nov}2011 \]
   \[ 28\text{nov}2011 - 1 = 23\text{nov}2011 \]

   Stata’s lead and lag operators work the same way.
2. Business calendars are named.
   Assume that the above business calendar is named `simple`.

3. Business calendars are defined in files named `calname.stbcal`, such as `simple.stbcal`. Calendars may be supplied by StataCorp and already installed, obtained from other users directly or via the SSC, or written yourself. Calendars can also be created automatically from the current dataset with the `bcal create` command; see [D] `bcal`. Stbcal-files are treated in the same way as ado-files.

   You can obtain a list of all business calendars installed on your computer by typing `bcal dir`; see [D] `bcal`.

4. Datetime format.
   The date format associated with the business calendar named `simple` is `%tbsimple`, which is to say `/% + t + b + calname`.

   - `%` it is a format
   - `t` it is a datetime
   - `b` it is based on a business calendar
   - `calname` the calendar’s name

5. Format variables the usual way.
   You format variables to have business calendar formats just as you format any variable, using the `format` command.

   . format mydate %tbsimple

   specifies that existing variable `mydate` contains values according to the business calendar named `simple`. See [D] `format`.

   You may format variables `%tbcalname` regardless of whether the corresponding stbcal-file exists. If it does not exist, the underlying numeric values will be displayed in a `%g` format.

6. Detailed date formats.
   You may include detailed datetime format specifiers by placing a colon and the detail specifiers after the calendar’s name.

   . format mydate %tbsimple:CCYY.NN.DD

   would display `21nov2011` as `2011.11.21`. See [D] `Datetime display formats` for detailed datetime format specifiers.

7. Reading business dates.
   To read files containing business dates, ignore the business date aspect and read the files as if they contained regular dates. Convert and format those dates as `%td`; see `Converting dates stored as strings to Stata dates` in [D] `Datetime`. Then convert the regular dates to `%tb` business dates:

   . generate mydate = bofd("simple", regulardate)
   . format mydate %tbsimple
   . assert mydate!=. if regulardate!=.

   The first statement performs the conversion.

   The second statement attaches the `%tbsimple` date format to the new variable `mydate` so that it will display correctly.

   The third statement verifies that all dates recorded in `regulardate` fit onto the business calendar. For instance, `12nov2011` does not appear on the `simple` calendar but, of course, it does appear on the regular calendar. If the data contained `12nov2011`, that would be an error. Function `bofd()` returns missing when the date does not appear on the specified calendar.
8. More on conversion.
   There are only two functions specific to business dates, `bofd()` and `dofb()`. Their definitions are
   
   \[ bdate = bofd("calname", regulardate) \]
   \[ regulardate = dofb(bdate, "calname") \]

   `bofd()` returns missing if `regulardate` is missing or does not appear on the specified business calendar. `dofb()` returns missing if `bdate` contains missing.

9. Obtaining day of week, etc.
   You obtain day of week, etc., by converting business dates to regular dates and then using the standard functions. To obtain the day of week of `bdate` on business calendar `calname`, type
   
   \[ . \ generate \ dow = dow(dofb(bdate, "calname")) \]

   See *Extracting date components from daily dates* in [D] Datetime for the other extraction functions.

10. Stbcal-files.
    The stbcal-file for simple, the calendar shown below,

    | November 2011 |
    |---------------|
    | Su | Mo | Tu | We | Th | Fr | Sa |
    |-----|-----|-----|-----|-----|-----|-----|
    | 1  | 2  | 3  | 4  | X  |     |    |
    | X  | 7  | 8  | 9  | 10 | 11 | X  |
    | X  | 14 | 15 | 16 | 17 | 18 | X  |
    | X  | 21 | 22 | 23 | X  | X  | X  |
    | X  | 28 | 29 | 30 |    |    |    |

    is

    ```
    begin simple.stbcal
    *
    version 1.0.0
    * simple.stbcal
    version 17.0
    purpose "Example for manual"
    dateformat dmy
    range 01nov2011 30nov2011
    centerdate 01nov2011
    omit dayofweek (Sa Su)
    omit date 24nov2011
    omit date 25nov2011
    end simple.stbcal
    ```

    This calendar was so simple that we crossed out the Thanksgiving holidays by specifying the dates to be omitted. In a real calendar, we would change the last two lines,
    
    ```
    omit date 24nov2011
    omit date 25nov2011
    ```
    
    to read
    
    ```
    omit dowinmonth +4 Th of Nov and +1
    ```
    
    which says to omit the fourth (+4) Thursday of November in every year, and omit the day after that (+1), too. See [D] Datetime business calendars creation.
Remarks and examples

See [D] Datetime for an introduction to Stata’s date and time features.

Below we work through an example from start to finish.

Remarks are presented under the following headings:

- **Step 1: Read the data, date as string**
- **Step 2: Convert date variable to %td date**
- **Step 3: Convert %td date to %tb date**
- **Key feature: Each business calendar has its own encoding**
- **Key feature: Omitted dates really are omitted**
- **Key feature: Extracting components from %tb dates**
- **Key feature: Merging on dates**

**Step 1: Read the data, date as string**

File `bcal_simple.raw` on our website provides data, including a date variable, that is to be interpreted according to the business calendar `simple` shown under `Syntax` above.

```
11/4/11 51
11/7/11 9
11/18/11 12
11/21/11 4
11/23/11 17
11/28/11 22
```

We begin by reading the data and then listing the result. Note that we read the date as a string variable:

```
. infile str10 sdate float x using https://www.stata-press.com/data/r17/bcal_simple
(6 observations read)
. list

    sdate   x
  +------------------+
  1.    11/4/11    51
  2.    11/7/11     9
  3.    11/18/11   12
  4.    11/21/11    4
  5.    11/23/11    7
  6.    11/28/11   22
  +------------------+
```

**Step 2: Convert date variable to %td date**

Now we create a numeric date variable from the string date and format it as a date (%td):

```
. generate rdate = date(sdate, "MD20Y")
. format rdate %td
```

See *Converting dates stored as strings to Stata dates* in [D] Datetime. We verify that the conversion went well and drop the string variable of the date:
Step 3: Convert %td date to %tb date

We convert the %td date to a %tbsimple date following the instructions of item 7 of Syntax above.

```
. generate mydate = bofd("simple", rdate)
. format mydate %tbsimple
. assert mydate!=. if rdate!=.
```

Had there been any dates that could not be converted from regular dates to simple business dates, `assert` would have responded, “assertion is false”. Nonetheless, we will list the data to show you that the conversion went well. We would usually drop the %td encoding of the date, but we want it to demonstrate a feature below.

```
. list

<table>
<thead>
<tr>
<th>sdate</th>
<th>x</th>
<th>rdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/4/11</td>
<td>51</td>
<td>04nov2011</td>
</tr>
<tr>
<td>11/7/11</td>
<td>9</td>
<td>07nov2011</td>
</tr>
<tr>
<td>11/18/11</td>
<td>12</td>
<td>18nov2011</td>
</tr>
<tr>
<td>11/21/11</td>
<td>4</td>
<td>21nov2011</td>
</tr>
<tr>
<td>11/23/11</td>
<td>17</td>
<td>23nov2011</td>
</tr>
<tr>
<td>11/28/11</td>
<td>22</td>
<td>28nov2011</td>
</tr>
</tbody>
</table>
```

. drop sdate

```
. list x rdate mydate

<table>
<thead>
<tr>
<th>x</th>
<th>rdate</th>
<th>mydate</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>04nov2011</td>
<td>04nov2011</td>
</tr>
<tr>
<td>9</td>
<td>07nov2011</td>
<td>07nov2011</td>
</tr>
<tr>
<td>12</td>
<td>18nov2011</td>
<td>18nov2011</td>
</tr>
<tr>
<td>4</td>
<td>21nov2011</td>
<td>21nov2011</td>
</tr>
<tr>
<td>17</td>
<td>23nov2011</td>
<td>23nov2011</td>
</tr>
<tr>
<td>22</td>
<td>28nov2011</td>
<td>28nov2011</td>
</tr>
</tbody>
</table>
```

Key feature: Each business calendar has its own encoding

In the listing above, `rdate` and `mydate` appear to be equal. They are not:

```
. format rdate mydate %9.0g
. list

<table>
<thead>
<tr>
<th>x</th>
<th>rdate</th>
<th>mydate</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>18935</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>18938</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>18949</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>18952</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>18954</td>
<td>16</td>
</tr>
<tr>
<td>22</td>
<td>18959</td>
<td>17</td>
</tr>
</tbody>
</table>
```

```
%tb dates each have their own encoding, and those encodings differ from the encoding used by %td dates. It does not matter. Neither encoding is better than the other. Neither do you need to concern yourself with the encoding. If you were curious, you could learn more about the encoding used by %tbsimple by typing bcal describe simple; see [D] bcal.

We will drop variable rdate and put the %tbsimple format back on variable mydate:

```
. drop rdate
. format mydate %tbsimple
```

**Key feature: Omitted dates really are omitted**

In *Syntax*, we mentioned that for the simple business calendar

\[
18\text{nov2011} + 1 = 21\text{nov2011}
\]

\[
28\text{nov2011} - 1 = 23\text{nov2011}
\]

That is true:

```
. generate tomorrow = mydate + 1
. generate yesterday = mydate - 1
. format tomorrow yesterday %tbsimple
. list
```

```
     x      mydate    tomorrow    yesterday
 1. 51  04\text{nov2011}  07\text{nov2011}  03\text{nov2011}
 2.  9  07\text{nov2011}  08\text{nov2011}  04\text{nov2011}
 3. 12 18\text{nov2011}  21\text{nov2011}  17\text{nov2011}
 4.  4  21\text{nov2011}  22\text{nov2011}  18\text{nov2011}
 5. 17 23\text{nov2011}  28\text{nov2011}  22\text{nov2011}
 6. 22 28\text{nov2011}  29\text{nov2011}  23\text{nov2011}
```

```
. drop tomorrow yesterday
```

Stata’s lag and lead operators L.*varname* and F.*varname* work similarly.

**Key feature: Extracting components from %tb dates**

You extract components such as day of week, month, day, and year from business dates using the same extraction functions you use with Stata’s regular %td dates, namely, *dow()* (*month()*), *day()*, and *year()*, and you use function *dofb()* to convert business dates to regular dates. Below we add day of week to our data, list the data, and then drop the new variable:
. generate dow = dow(dofb(mydate, "simple"))
. list

<table>
<thead>
<tr>
<th>x</th>
<th>mydate</th>
<th>dow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>04nov2011</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>07nov2011</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>18nov2011</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>21nov2011</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>23nov2011</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>28nov2011</td>
<td>1</td>
</tr>
</tbody>
</table>

. drop dow

See *Extracting date components from daily dates* in [D] Datetime.

**Key feature: Merging on dates**

It may happen that you have one dataset containing business dates and a second dataset containing regular dates, say, on economic conditions, and you want to merge them. To do that, you create a regular date variable in your first dataset and merge on that:

. generate rdate = dofb(mydate, "simple")
. merge 1:1 rdate using econditions, keep(match)
. drop rdate

**Also see**

[D] bcal — Business calendar file manipulation
[D] Datetime business calendars creation — Business calendars creation
[D] Datetime — Date and time values and variables
Description

Stata provides user-definable business calendars. Business calendars are provided by StataCorp and by other users, and you can write your own. You can also create a business calendar automatically from the current dataset with the `bcal create` command; see [D] `bcal`. This entry concerns writing your own business calendars.

See [D] *Datetime business calendars* for an introduction to business calendars.

Syntax

Business calendar `calname` and corresponding display format `%tbcalname` are defined by the text file `calname.stbcal`, which contains the following:

```
* comments

version version_of_stata
purpose "text"
dateformat { ymd | ydm | myd | mdy | dym | dmy }

range date date
centerdate date
[from { date . } to { date . }): omit ... [ if ]
...
...
```

where

```
omit ... may be
      omit date pdate [ and pmlist ]
      omit dayofweek dowlist
      omit dowinmonth pm# dow [ of monthlist ] [ and pmlist ]

[ if ] may be
      if restriction [ & restriction ... ]

restriction is one of
      dow(dowlist)
      month(monthlist)
      year(yearlist)
```
**date** is a date written with the *year, month, and day* in the order specified by *dateformat*. For instance, if *dateformat* is *dmy*, a *date* can be 12apr2013, 12-4-2013, or 12.4.2013.

*pdate* is a *date* or it is a *date* with character * substituted where the year would usually appear. If *dateformat* is *dmy*, a *pdate* can be 12apr2013, 12-4-2013, or 12.4.2013; or it can be 12apr*, 12-4-*, or 12.4.*. 12apr* means the 12th of April across all years.

*dow* is a day of the week, in English. It may be abbreviated to as few as 2 characters, and capitalization is irrelevant. Examples: Sunday, Mo, tu, Wed, th, Friday, saturday.

*dowlist* is a *dow*, or it is a space-separated list of one or more *dows* enclosed in parentheses. Examples: Sa, (Sa), (Sa Su).

*month* is a month of the year, in English, or it is a month number. It may be abbreviated to the minimum possible, and capitalization is irrelevant. Examples: January, 2, Mar, ap, may, 6, Jul, aug, 9, Octob, nov, 12.

*monthlist* is a *month*, or it is a space-separated list of one or more *months* enclosed in parentheses. Examples: Nov, (Nov), 11, (11), (Nov Dec), (11 12).

*year* is a 4-digit calendar year. Examples: 1872, 1992, 2013, 2050.

*yearlist* is a *year*, or it is a space-separated list of one or more *years* enclosed in parentheses. Examples: 2013, (2013), (2013 2014).

*pm#* is a nonzero integer preceded by a plus or minus sign. Examples: −2, −1, +1. *pm#* appears in *omit dowinmonth pm# dow of monthlist*, where *pm#* specifies which *dow* in the month. *omit dowinmonth +1 Th* means the first Thursday of the month. *omit dowinmonth −1 Th* means the last Thursday of the month.

*pmlist* is a *pm#*, or it is a space-separated list of one or more *pm#s* enclosed in parentheses. Examples: +1, (+1), (+1 +2), (−1 +1 +2). *pmlist* appears in the optional and *pmlist* allowed at the end of *omit date* and *omit dowinmonth*, and it specifies additional dates to be omitted. and +1 means and the day after. and −1 means and the day before.

### Remarks and examples

Remarks are presented under the following headings:

- **Introduction**
- **Concepts**
- **The preliminary commands**
- **The omit commands: from/to and if**
- **The omit commands: and**
- **The omit commands: omit date**
- **The omit commands: omit dayofweek**
- **The omit commands: omit dowinmonth**
- **Creating stbcal-files with bcal create**
- **Where to place stbcal-files**
- **How to debug stbcal-files**
- **Ideas for calendars that may not occur to you**
Introduction

A business calendar is a regular calendar with some dates crossed out, such as

<table>
<thead>
<tr>
<th>Su</th>
<th>Mo</th>
<th>Tu</th>
<th>We</th>
<th>Th</th>
<th>Fr</th>
<th>Sa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The purpose of the stbcal-file is to

1. Specify the range of dates covered by the calendar.
2. Specify the particular date that will be encoded as date 0.
3. Specify the dates from the regular calendar that are to be crossed out.

The stbcal-file for the above calendar could be as simple as

```
begin example_1.stbcal

version 17.0
range 01nov2011 30nov2011
centerdate 01nov2011
omit date 5nov2011
omit date 6nov2011
omit date 12nov2011
omit date 13nov2011
omit date 19nov2011
omit date 20nov2011
omit date 24nov2011
omit date 25nov2011
omit date 26nov2011
omit date 27nov2011

end example_1.stbcal
```

In fact, this calendar can be written more compactly because we can specify to omit all Saturdays and Sundays:

```
begin example_2.stbcal

version 17.0
range 01nov2011 30nov2011
centerdate 01nov2011
omit dayofweek (Sa Su)
omit date 24nov2011
omit date 25nov2011

end example_2.stbcal
```

In this particular calendar, we are omitting 24nov2011 and 25nov2011 because of the American Thanksgiving holiday. Thanksgiving is celebrated on the fourth Thursday of November, and many businesses close on the following Friday as well. It is possible to specify rules like that in stbcal-files:
Understand that this calendar is an artificial example, and it is made all the more artificial because it covers so brief a period. Real stbcal-files cover at least decades, and some cover centuries.

Concepts

You are required to specify four things in an stbcal-file:

1. the version of Stata being used,
2. the range of the calendar,
3. the center date of the calendar, and
4. the dates to be omitted.

Version.

You specify the version of Stata to ensure forward compatibility with future versions of Stata. If your calendar starts with the line `version 17.0`, future versions of Stata will know how to interpret the file even if the definition of the stbcal-file language has greatly changed.

Range.

A calendar is defined over a specific range of dates, and you must explicitly state what that range is. When you or others use your calendar, dates outside the range will be considered invalid, which usually means that they will be treated as missing values.

Center date.

Stata stores dates as integers. In a calendar, 57 might stand for a particular date. If it did, then \( 57 - 1 = 56 \) stands for the day before, and \( 57 + 1 = 58 \) stands for the day after. The previous statement works just as well if we substitute \(-12,739\) for 57, and thus the particular values do not matter except that we must agree upon what values we wish to standardize because we will be storing these values in our datasets.

The standard is called the center date, and here center does not mean the date that corresponds to the middle of your calendar. It means the date that corresponds to the center of integers, which is to say, 0. You must choose a date within the range as the standard. The particular date you choose does not matter, but most authors choose easily remembered ones. Stata’s built-in \%td calendar uses 01jan1960, but that date will probably not be available to you because the center date must be a date on the business calendars, and most businesses were closed on 01jan1960.

It will sometimes happen that you will want to expand the range of your calendar in the future. Today, you make a calendar that covers, say 1990 to 2020, which is good enough for your purposes. Later, you need to expand the range, say back to 1970 or forward to 2030, or both. When you update your calendar, do not change the center date. This way, your new calendar will be backward compatible with your previous one.

Omitted dates.

Obviously you will need to specify the dates to be omitted. You can specify the exact dates to be omitted when need be, but whenever possible, specify the rules instead of the outcome of the rules. Rules change, so learn about the `from/to` prefix that can be used in front of `omit` commands. You can code things like
The preliminary commands

Stbcal-files should begin with these commands:

```
version version_of_stata
purpose "text"
dateformat { ymd | ydm | myd | mdy | dym | dmy }
range date date
centerdate date
```

version version_of_stata
At the time of this writing, you would specify version 17.0. Better still, type command version in Stata to discover the version of Stata you are currently using. Specify that version, and be sure to look at the documentation so that you use the modern syntax correctly.

purpose "text"
This command is optional. The purpose of purpose is not to make comments in your file. If you want comments, include those with a * in front. The purpose sets the text that bcal describe calname will display.

dateformat { ymd | ydm | myd | mdy | dym | dmy }
This command is optional. dateformat ymd is assumed if not specified. This command has nothing to do with how dates will look when variables are formatted with %td. This command specifies how you are typing dates in this stbcal-file on the subsequent commands. Specify the format that you find convenient.

range date date
The date range was discussed in Concepts. You must specify it.

centerdate date
The centering date was discussed in Concepts. You must specify it.

The omit commands: from/to and if

An stbcal-file usually contains multiple omit commands. The omit commands have the syntax

```
[from {date | .} to {date | .}] omit [if]
```

That is, an omit command may optionally be preceded by from/to and may optionally contain an if at the end.

When you do not specify from/to, results are the same as if you specified

```
from . to .. omit ...
```

That is, the omit command applies to all dates from the beginning to the end of the range. In Introduction, we showed the command

```
omit dowinmonth +4 Th of Nov and +1
```
Our sample calendar covered only the month of November, but imagine that it covered a longer period and that the business was open on Fridays following Thanksgiving up until 1998. The Thanksgiving holidays could be coded

\[
\text{from . to 31dec1997: omit downmonth +4 Th of Nov} \\
\text{from 01jan1998 to .: omit downmonth +4 Th of Nov and +1}
\]

The same holidays could also be coded

\[
\text{omit downmonth +4 Th of Nov} \\
\text{from 01jan1998 to .: omit downmonth +4 Th of Nov and +1}
\]

We like the first style better, but understand that the same dates can be omitted from the calendars multiple times and for multiple reasons, and the result is still the same as if the dates were omitted only once.

The optional \textit{if} also determines when the \textit{omit} statement is operational. Let’s think about the Christmas holidays. Let’s say a business is closed on the 24th and 25th of December. That could be coded

\[
\text{omit date 24dec*} \\
\text{omit date 25dec*}
\]

although perhaps that would be more understandable if we coded

\[
\text{from . to .: omit date 24dec*} \\
\text{from . to .: omit date 25dec*}
\]

Remember, \textit{from . to .} is implied when not specified. In any case, we are omitting 24dec and 25dec across all years.

Now consider a more complicated rule. The business is closed on the 24th and 25th of December if the 25th is on Tuesday, Wednesday, Thursday, or Friday. If the 25th is on Saturday or Sunday, the holidays are the preceding Friday and the following Monday. If the 25th is on Monday, the holidays are Monday and Tuesday. The rule could be coded

\[
\text{omit date 25dec* and -1 if dow(Tu We Th Fr)} \\
\text{omit date 25dec* and (-2 -1) if dow(Sa)} \\
\text{omit date 25dec* and (-3 -2) if dow(Su)} \\
\text{omit date 25dec* and +1 if dow(Mo)}
\]

The \textit{if} clause specifies that the \textit{omit} command is only to be executed when 25dec* is one of the specified days of the week. If 25dec* is not one of those days, the \textit{omit} statement is ignored for that year. Our focus here is on the \textit{if} clause. We will explain about the \textit{and} clause in the next section.

Sometimes, you have a choice between using \textit{from/to} or \textit{if}. In such cases, use whichever is convenient. For instance, imagine that the Christmas holiday rule for Monday changed in 2011 and 2012. You could code

\[
\text{from . to 31dec2010: omit date 25dec* and +1 if dow(Mo)} \\
\text{from 01jan2011 to .: omit date ... if dow(Mo)}
\]

or

\[
\text{omit date 25dec* and +1 if dow(Mo) & year(2007 2008 2009 2010)} \\
\text{omit date ... if dow(Mo) & year(2011 2012)}
\]

Generally, we find \textit{from/to} more convenient to code than \textit{if year()}. 
The omit commands: and

The other common piece of syntax that shows up on omit commands is and \textit{pmlist}. We used it above in coding the Christmas holidays,

\begin{verbatim}
omit date 25dec* and -1 if dow(Tu We Th Fr)
omit date 25dec* and (-2 -1) if dow(Sa)
omit date 25dec* and (-3 -2) if dow(Su)
omit date 25dec* and +1 if dow(Mo)
\end{verbatim}

and \textit{pmlist} specifies a list of days also to be omitted if the date being referred to is omitted. The extra days are specified as how many days they are from the date being referred to. Please excuse the inelegant “date being referred to”, but sometimes the date being referred to is implied rather than stated explicitly. For this problem, however, the date being referred to is 25dec across a number of years. The line

\begin{verbatim}
omit date 25dec* and -1 if dow(Tu We Th Fr)
\end{verbatim}

says to omit 25dec and the day before if 25dec is on a Tuesday, Wednesday, etc. The line

\begin{verbatim}
omit date 25dec* and (-2 -1) if dow(Sa)
\end{verbatim}

says to omit 25dec and two days before and one day before if 25dec is Saturday. The line

\begin{verbatim}
omit date 25dec* and (-3 -2) if dow(Su)
\end{verbatim}

says to omit 25dec and three days before and two days before if 25dec is Sunday. The line

\begin{verbatim}
omit date 25dec* and +1 if dow(Mo)
\end{verbatim}

says to omit 25dec and the day after if 25dec is Monday.

Another omit command for solving a different problem reads

\begin{verbatim}
omit dowinmonth -1 We of (Nov Dec) and +1 if year(2009)
\end{verbatim}

Please focus on the and +1. We are going to omit the date being referred to and the date after if the year is 2009. The date being referred to here is -1 We of (Nov Dec), which is to say, the last Wednesday of November and December.

The omit commands: \textit{omit date}

The full syntax of \textit{omit date} is

\begin{verbatim}
[from \{date\}. to \{date\}.:] omit date \textit{pdate} [and \textit{pmlist}] [if]
\end{verbatim}

You may omit specific dates,

\begin{verbatim}
omit date 25dec2010
\end{verbatim}

or you may omit the same date across years:

\begin{verbatim}
omit date 25dec*
\end{verbatim}
The omit commands: omit dayofweek

The full syntax of omit dayofweek is

```
from {date .} to {date .}: omit dayofweek dowlist [if]
```

The specified days of week (Monday, Tuesday, ...) are omitted.

The omit commands: omit dowinmonth

The full syntax of omit dowinmonth is

```
from {date .} to {date .}: omit pm# dow [of monthlist] [and pmlist] [if]
```

dowinmonth stands for day of week in month and refers to days such as the first Monday, second Monday, ..., next-to-last Monday, and last Monday of a month. This is written as +1 Mo, +2 Mo, ..., -2 Mo, and -1 Mo.

Creating stbcal-files with bcal create

Business calendars can be obtained from your Stata installation or from other Stata users. You can also write your own business calendar files or use the `bcal create` command to automatically create a business calendar from the current dataset. With `bcal create`, business holidays are automatically inferred from gaps in the dataset, or they can be explicitly defined by specifying the `if` and in qualifiers, as well as the `excludemissing()` option. You can also edit business calendars created with `bcal create` or obtained from other sources. It is advisable to use `bcal load` or `bcal describe` to verify that a business calendar is well constructed and remains so after editing.

See [D] `bcal` for more information on `bcal create`.

Where to place stbcal-files

Stata automatically searches for stbcal-files in the same way it searches for ado-files. Stata looks for ado-files and stbcal-files in the official Stata directories, your site’s directory (SITE), your current working directory (.), your personal directory (PERSONAL), and your directory for materials written by other users (PLUS). On this writer’s computer, these directories happen to be

```
.sysdir
  STATA: C:\Program Files\Stata17\
  BASE: C:\Program Files\Stata12\ado\base\
  SITE: C:\Program Files\Stata17\ado\site\
  PLUS:  C:\ado\plus\
  PERSONAL:  C:\ado\personal\ 
  OLDPLACE:  C:\ado\ 
```

Place calendars that you write into ., PERSONAL, or SITE. Calendars you obtain from others using `net` or `ssc` will be placed by those commands into PLUS. See [P] `sysdir`, [R] `net`, and [R] `ssc`.

How to debug stbcal-files

Stbcal-files are loaded automatically as they are needed, and because this can happen anytime, even at inopportune moments, no output is produced. If there are errors in the file, no mention is made of the problem, and thereafter Stata simply acts as if it had never found the file, which is to say, variables with `%tbcalname` formats are displayed in `%g` format.
You can tell Stata to load a calendar file right now and to show you the output, including error messages. Type

```
.bcal load calname
```

It does not matter where `calname.stbcal` is stored, Stata will find it. It does not matter whether Stata has already loaded `calname.stbcal`, either secretly or because you previously instructed the file be loaded. It will be reloaded, you will see what you wrote, and you will see any error messages.

**Ideas for calendars that may not occur to you**

Business calendars obviously are not restricted to businesses, and neither do they have to be restricted to days.

Say you have weekly data and want to create a calendar that contains only Mondays. You could code

```
version 17.0
purpose "Mondays only"
range 04jan1960 06jan2020
centerdate 04jan1960
omitdow (Tu We Th Fr Sa Su)
```

```
end mondays.stbcal
```

Say you have semimonthly data and want to include the 1st and 15th of every month. You could code

```
version 17.0
purpose "Semimonthly"
range 01jan1960 15dec2020
centerdate 01jan1960
omit date 2jan*
omit date 3jan*
.
omit date 14jan*
omit date 16jan*
.
omit date 31jan*
omit date 2feb*
.
```

```
end smnth.stbcal
```

Forgive the ellipses, but this file will be long. Even so, you have to create it only once.

As a final example, say that you just want Stata’s `%td` dates, but you wish they were centered on 01jan1970 rather than on 01jan1960. You could code

```
version 17.0
Purpose "%td centered on 01jan1970"
range 01jan1800 31dec2999
centerdate 01jan1970
```

```
end rectr.stbcal
```
Also see

[D] bcal — Business calendar file manipulation
[D] Datetime business calendars — Business calendars
[D] Datetime — Date and time values and variables
Description

These functions convert dates and times recorded as strings to Stata dates. Stata dates are numbers that can be formatted so that they look like the dates you are familiar with. See [D] Datetime for an introduction to Stata’s date and time features.

Quick start

Convert strdate1, with dates such as "Tue January 25, 2013", to a numerically encoded Stata date variable, ignoring the day of the week from the string

```
generate numvar1 = date(strdate1, "#MDY")
```

Convert strdate2, with dates in the 2000s such as "01-25-13", to a Stata date variable

```
generate numvar2 = date(strdate2, "MD20Y")
```

Convert strdate3, with dates such as "15Jan05", to a Stata date variable; expand the two-digit years to the largest year that does not exceed 2006

```
generate numvar3 = date(strdate3, "DMY", 2006)
```

Convert strtime, with times such as "11:15 am", to a numerically encoded Stata datetime/c variable

```
generate double numvar4 = clock(strtime, "hm")
```
Syntax

The string-to-numeric date and time conversion functions are

<table>
<thead>
<tr>
<th>Desired Stata date type</th>
<th>String-to-numeric conversion function</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>clock(str, mask [, toyear])</td>
</tr>
<tr>
<td>datetime/C</td>
<td>Clock(str, mask [, toyear])</td>
</tr>
<tr>
<td>date</td>
<td>date(str, mask [, toyear])</td>
</tr>
<tr>
<td>weekly date</td>
<td>weekly(str, mask [, toyear])</td>
</tr>
<tr>
<td>monthly date</td>
<td>monthly(str, mask [, toyear])</td>
</tr>
<tr>
<td>quarterly date</td>
<td>quarterly(str, mask [, toyear])</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>halfyearly(str, mask [, toyear])</td>
</tr>
<tr>
<td>yearly date</td>
<td>yearly(str, mask [, toyear])</td>
</tr>
</tbody>
</table>

str is the string value to be converted.

mask specifies the order of the date and time components and is a string composed of a sequence of codes (see the next table).

topyear is described in Working with two-digit years, below.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>month</td>
</tr>
<tr>
<td>D</td>
<td>day within month</td>
</tr>
<tr>
<td>Y</td>
<td>4-digit year</td>
</tr>
<tr>
<td>19Y</td>
<td>2-digit year to be interpreted as 19xx</td>
</tr>
<tr>
<td>20Y</td>
<td>2-digit year to be interpreted as 20xx</td>
</tr>
<tr>
<td>W</td>
<td>week (weekly() only)</td>
</tr>
<tr>
<td>Q</td>
<td>quarter (quarterly() only)</td>
</tr>
<tr>
<td>H</td>
<td>half-year (halfyearly() only)</td>
</tr>
<tr>
<td>h</td>
<td>hour of day</td>
</tr>
<tr>
<td>m</td>
<td>minutes within hour</td>
</tr>
<tr>
<td>s</td>
<td>seconds within minute</td>
</tr>
<tr>
<td>#</td>
<td>ignore one element</td>
</tr>
</tbody>
</table>

Blanks are also allowed in mask, which can make the mask easier to read, but they otherwise have no significance.

Examples of masks include the following:

"MDY"   str contains month, day, and year, in that order.

"MD19Y" means the same as "MDY", except that str may contain two-digit years, and when it does, they are to be treated as if they are 4-digit years beginning with 19.

"MDYhms" str contains month, day, year, hour, minute, and second, in that order.

"MDY hms" means the same as "MDYhms"; the blank has no meaning.
"MDY#hms" means that one element between the year and the hour is to be ignored. For example, \texttt{str} contains values like "1-1-2010 at 15:23:17" or values like "1-1-2010 at 3:23:17 PM".

### Remarks and examples

Remarks are presented under the following headings:

- **Introduction**
- **Specifying the mask**
- **How the conversion functions interpret the mask**
- **Working with two-digit years**
- **Working with incomplete dates and times**
- **Converting run-together dates, such as 20060125**
- **Valid times**
- **The clock() and Clock() functions**
- **Why there are two datetime encodings**
- **Advice on using datetime/c and datetime/C**
- **Determining when leap seconds occurred**
- **The date() function**
- **The other conversion functions**

### Introduction

The conversion functions are used to convert string dates, such as 08/12/06, 12-8-2006, 12 Aug 06, 12aug2006 14:23, and 12 aug06 2:23 pm, to Stata dates. The conversion functions are typically used after importing or reading data. You read the date information into string variables and then these functions convert the string into something Stata can use, namely, a numeric Stata date variable.

You use \texttt{generate} to create the Stata date variables. The conversion functions are used in the expressions, such as

```stata
. generate double time_admitted = clock(time_admitted_str, "DMYhms")
. format time_admitted %tc
. generate date_hired = date(date_hired_str, "MDY")
. format date_hired %td
```

Every conversion function—such as \texttt{clock()} and \texttt{date()} above—requires these two arguments:

1. \texttt{str} specifying the string to be converted; and
2. \texttt{mask} specifying the order in which the date and time components appear in \texttt{str}.

Notes:

1. You choose the conversion function \texttt{clock()}, \texttt{Clock()}, \texttt{date()}, etc., according to the type of Stata date you want returned.
2. You specify the mask according to the contents of \texttt{str}.

Usually, you will want to convert \texttt{str} containing 2006.08.13 14:23 to a Stata datetime/c or datetime/C value and convert \texttt{str} containing 2006.08.13 to a Stata date. If you wish, however, it can be the other way around. In that case, the detailed string would convert to a Stata date corresponding to just the date part, 13aug2006, and the less detailed string would convert to a Stata datetime corresponding to 13aug2006 00:00:00.000.
Specifying the mask

An argument *mask* is a string specifying the order of the date and time components in *str*. Examples of string dates and the mask required to convert them include the following:

<table>
<thead>
<tr>
<th>str</th>
<th>Corresponding mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>01dec2006 14:22</td>
<td>&quot;DMYhm&quot;</td>
</tr>
<tr>
<td>01-12-2006 14:22</td>
<td>&quot;DMYhm&quot;</td>
</tr>
<tr>
<td>1dec2006 14:22</td>
<td>&quot;DMYhm&quot;</td>
</tr>
<tr>
<td>1-12-2006 14:22</td>
<td>&quot;DMYhm&quot;</td>
</tr>
<tr>
<td>01dec06 14:22</td>
<td>&quot;DM20Yhm&quot;</td>
</tr>
<tr>
<td>01-12-06 14.22</td>
<td>&quot;DM20Yhm&quot;</td>
</tr>
<tr>
<td>December 1, 2006 14:22</td>
<td>&quot;MDYhm&quot;</td>
</tr>
<tr>
<td>2006 Dec 01 14:22</td>
<td>&quot;YMDhm&quot;</td>
</tr>
<tr>
<td>2006-12-01 14:22</td>
<td>&quot;YMDhm&quot;</td>
</tr>
<tr>
<td>2006-12-01 14:22:43</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 14:22:43.2</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 14:22:43.21</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 14:22:43.213</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 2:22:43.213 pm</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 2:22:43.213 pm.</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 2:22:43.213 P.m.</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>2006-12-01 2:22:43.213 P.M.</td>
<td>&quot;YMDhms&quot;</td>
</tr>
<tr>
<td>20061201 1422</td>
<td>&quot;YMDhm&quot;</td>
</tr>
<tr>
<td>14:22</td>
<td>&quot;hm&quot;</td>
</tr>
<tr>
<td>2006-12-01</td>
<td>&quot;YMD&quot;</td>
</tr>
<tr>
<td>Fri Dec 01 14:22:43 CST 2006</td>
<td>&quot;#MDhms#Y&quot;</td>
</tr>
</tbody>
</table>

Notes:

1. Nothing special needs to be included in *mask* to process a.m. and p.m. markers. When you include code h, the conversion functions automatically watch for meridian markers.

2. You specify the mask according to what is contained in *str*. If that is a subset of what the selected Stata date type could record, the remaining elements are set to their defaults. *clock*("14:22", "hm") produces 01jan1960 14:22:00 and *clock*("2006-12-01", "YMD") produces 01dec2006 00:00:00. *date*("jan 2006", "MY") produces 01jan2006.

*mask* may include spaces so that it is more readable; the spaces have no meaning. Thus, you can type

```
generate double admit = clock(admitstr, "#MDhms#Y")
```
or type

```
generate double admit = clock(admitstr, "# MD hms # Y")
```
and which one you use makes no difference.
How the conversion functions interpret the mask

The conversion functions apply the following rules when interpreting $str$:

1. For each string date to be converted, remove all punctuation except for the period separating seconds from tenths, hundredths, and thousandths of seconds. Replace removed punctuation with a space.

2. Insert a space in the string everywhere that a letter is next to a number, or vice versa.

3. Interpret the resulting elements according to $mask$.

For instance, consider the string

01dec2006 14:22

Under rule 1, the string becomes

01dec2006 14 22

Under rule 2, the string becomes

01 dec 2006 14 22

Finally, the conversion functions apply rule 3. If the mask is "DMYhm", then the functions interpret “01” as the day, “dec” as the month, and so on.

Or consider the string

Wed Dec 01 14:22:43 CST 2006

Under rule 1, the string becomes

Wed Dec 01 14 22 43 CST 2006

Applying rule 2 does not change the string. Now rule 3 is applied. If the mask is "#MDhms#Y", the conversion function skips “Wed”, interprets “Dec” as the month, and so on.

The # code serves a second purpose. If it appears at the end of the mask, it specifies that the rest of $string$ is to be ignored. Consider converting the string

Wed Dec 01 14 22 43 CST 2006 patient 42

The mask code that previously worked when patient 42 was not part of the string, "#MDhms#Y", will result in a missing value in this case. The functions are careful in the conversion, and if the whole string is not used, they return missing. If you end the mask in #, however, the functions ignore the rest of the string. Changing the mask from "#MDhms#Y" to "#MDhms#Y#" will produce the desired result.

Working with two-digit years

Consider converting the string 01-12-06 14:22, which is to be interpreted as 01dec2006 14:22:00, to a Stata datetime value. The conversion functions provide two ways of doing this.

The first is to specify the assumed prefix in the mask. The string 01-12-06 14:22 can be read by specifying the mask "DM20Yhm". If we instead wanted to interpret the year as 1906, we would specify the mask "DM19Yhm". We could even interpret the year as 1806 by specifying "DM18Yhm".

What if our data include 01-12-06 14:22 and include 15-06-98 11:01? We want to interpret the first year as being in 2006 and the second year as being in 1998. That is the purpose of the optional argument $topyear$:

$\text{clock}(\text{string}, \text{mask} [, \text{topyear}])$
When you specify `topyear`, you are stating that when years in `string` are two digits, the full year is to be obtained by finding the largest year that does not exceed `topyear`. Thus, you could code

```stata
. generate double timestamp = clock(timestr, "DMYhm", 2020)
```

The two-digit year 06 would be interpreted as 2006 because 2006 does not exceed 2020. The two-digit year 98 would be interpreted as 1998 because 2098 does exceed 2020.

### Working with incomplete dates and times

The conversion functions do not require that every component of the date and time be specified. Converting 2006-12-01 with mask "YMD" results in 01dec2006 00:00:00.

Converting 14:22 with mask "hm" results in 01jan1960 14:22:00.

Converting 11-2006 with mask "MY" results in 01nov2006 00:00:00.

The default for a component, if not specified in the mask, is

<table>
<thead>
<tr>
<th>Code</th>
<th>Default (if not specified)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>01</td>
</tr>
<tr>
<td>D</td>
<td>01</td>
</tr>
<tr>
<td>Y</td>
<td>1960</td>
</tr>
<tr>
<td>h</td>
<td>00</td>
</tr>
<tr>
<td>m</td>
<td>00</td>
</tr>
<tr>
<td>s</td>
<td>00</td>
</tr>
</tbody>
</table>

Thus, if you have data recording 14:22, meaning a duration of 14 hours and 22 minutes or the time 14:22 each day, you can convert it with `clock(str, "hm")`.

### Converting run-together dates, such as 20060125

The `clock()`, `Clock()`, and `date()` conversion functions will convert dates and times that are run together, such as 20060125, 060125, and 20060125110215 (which is 25jan2006 11:02:15). You do not have to do anything special to convert them:

```stata
. display %d date("20060125", "YMD")
25jan2006
. display %td date("060125", "20YMD")
25jan2006
. display %tc clock("20060125110215", "YMDhms")
25jan2006 11:02:15
```

However, the `weekly()`, `monthly()`, `quarterly()`, and `halfyearly()` functions will convert only dates that are run together if there is a combination of letters and numbers. For example,

```stata
. display %tm monthly("2020m1", "YM")
2020m1
. display %tq quarterly("2020q2", "YQ")
2020q1
```
If your string consists of numbers only, such as 202001, you will need to insert a space or punctuation between the year and the other component before using one of these functions.

In a data context, you could type

```
. generate startdate = date(startdatestr, "YMD")
. generate double starttime = clock(starttimestr, "YMDhms")
```

Remember to read the original date into a string. If you mistakenly read the date as numeric, the best advice is to read the date again. Numbers such as 20060125 and 20060125110215 will be rounded unless they are stored as doubles.

If you mistakenly read the variables as numeric and have verified that rounding did not occur, you can convert the variable from numeric to string by using the `string()` function, which comes in one- and two-argument forms. You will need the two-argument form:

```
. generate str startdatestr = string(startdatedouble, "%10.0g")
. generate str starttimestr = string(starttimedouble, "%16.0g")
```

If you omitted the format, `string()` would produce `2.01e+07` for 20060125 and `2.01e+13` for 20060125110215. The format we used had a width that was two characters larger than the length of the integer number, although using a too-wide format does no harm.

### Valid times

An invalid time is 27:62:90. If you try to convert 27:62:90 to a datetime value, you will obtain a missing value.

Another invalid time is 24:00:00. A correct time would be 00:00:00 of the next day.

In `hh:mm:ss`, the requirements are `0 \leq hh < 24`, `0 \leq mm < 60`, and `0 \leq ss < 60`, although sometimes 60 is allowed. The encoding `31dec2005 23:59:60` is an invalid datetime/c but a valid datetime/C. The encoding `31dec2005 23:59:60` includes an inserted leap second.

Invalid in both datetime encodings is `30dec2005 23:59:60`. Not including a leap second as in `30dec2005 23:59:60` would also be an invalid encoding. A correct datetime would be `31dec2005 00:00:00`.

### The clock() and Clock() functions

Stata provides two separate datetime encodings that we call datetime/c and datetime/C and that others would call “times assuming 86,400 seconds per day” and “times adjusted for leap seconds” or, equivalently, Coordinated Universal Time (UTC).

The syntax of the two functions is the same:

```
clock(str, mask [, topyear])
Clock(str, mask [, topyear])
```

Function `Clock()` is nearly identical to function `clock()`, except that `Clock()` returns a datetime/C value rather than a datetime/c value. For instance,

```
Noon of 23nov2010 = 1,606,132,800,000 in datetime/c
= 1,606,132,824,000 in datetime/C
```

They differ because 24 seconds have been inserted into datetime/C between 01jan1960 and 23nov2010. Correspondingly, `clock()` understands times in which there are leap seconds, such as `30jun1997 23:59:60`. `clock()` would consider `30jun1997 23:59:60` an invalid time and so return a missing value.
Why there are two datetime encodings

Stata provides two different datetime encodings, datetime/c and datetime/C.

The datetime/c encoding assumes that there are $24 \times 60 \times 60 \times 1000$ ms per day, just as an atomic clock does. Atomic clocks count oscillations between the nucleus and the electrons of an atom and thus provide a measurement of the real passage of time.

Time of day measurements have historically been based on astronomical observation, which is a fancy way of saying that the measurements are based on looking at the sun. The sun should be at its highest point at noon, right? So however you might have kept track of time—by falling grains of sand or a wound-up spring—you would have periodically reset your clock and then gone about your business. In olden times, it was understood that the 60 seconds per minute, 60 minutes per hour, and 24 hours per day were theoretical goals that no mechanical device could reproduce accurately. These days, we have more formal definitions for measurements of time. One second is 9,192,631,770 periods of the radiation corresponding to the transition between two levels of the ground state of cesium 133. Obviously, we have better equipment than the ancients, so problem solved, right? Wrong. There are two problems: the formal definition of a second is just a little too short to use for accurately calculating the length of a day, and the Earth’s rotation is slowing down.

Thus, since 1972, leap seconds have been added to atomic clocks once or twice a year to keep time measurements in synchronization with Earth’s rotation. Unlike leap years, however, there is no formula for predicting when leap seconds will occur. Earth may be on average slowing down, but there is a large random component to that. Therefore, leap seconds are determined by committee and announced six months before they are inserted. Leap seconds are added, if necessary, on the end of the day on June 30 and December 31 of the year. The exact times are designated as 23:59:60.

Unadjusted atomic clocks may accurately mark the passage of real time, but you need to understand that leap seconds are every bit as real as every other second of the year. Once a leap second is inserted, it ticks just like any other second and real things can happen during that tick.

You may have heard of terms such as Greenwich Mean Time (GMT) and UTC.

GMT, based on astronomical observation, has been replaced by UTC.

UTC is measured by atomic clocks and is occasionally corrected for leap seconds. UTC is derived from two other times, Universal Time 1 (UT1) and International Atomic Time (TAI). UT1 is the mean solar time with which UTC is kept in sync by the occasional addition of a leap second. TAI is the atomic time on which UTC is based. TAI is a statistical combination of various atomic chronometers, and even it has not ticked uniformly over its history; see http://www.ucolick.org/~sla/leapsecs/timescales.html and especially http://www.ucolick.org/~sla/leapsecs/dutc.html#TAI.

UNK is our term for the time standard most people use. UNK stands for unknown or unknowing. UNK is based on a recent time observation, probably UTC, and it just assumes that there are 86,400 seconds per day after that.

The UNK standard is adequate for many purposes, and when using it you will want to use datetime/c rather than the leap second–adjusted datetime/C encoding. If you are using computer-timestamped data, however, you need to find out whether the timestamping system accounted for leap-second adjustment. Problems can arise even if you do not care about losing or gaining a second here and there.

For instance, you may import from other systems timestamp values recorded in the number of milliseconds that have passed since some agreed-upon date. You may do this, but if you choose the wrong encoding scheme (choose datetime/c when you should choose datetime/C, or vice versa), more recent times will be off by 24 seconds.
To avoid such problems, you may decide to import and export data as strings, such as Fri Aug 18 14:05:36 CDT 2010. This method has advantages, but for datetime/C (UTC) encoding, times such as 23:59:60 are possible. Some systems will refuse to decode such times.

Stata refuses to decode 23:59:60 in the datetime/c encoding (function \texttt{clock()}) and accepts it with datetime/C (function \texttt{Clock()}). When datetime/C function \texttt{Clock()} sees a time with a 60th second, \texttt{Clock()} verifies that the time is one of the official leap seconds. Thus, when converting from printable forms, try assuming datetime/c, and check the result for missing values. If there are none, then you can assume your use of datetime/c was valid. However, if there are missing values and they are due to leap seconds and not some other error, you must use datetime/C \texttt{Clock()} to convert the string value. After that, if you still want to work in datetime/c units, use function \texttt{cofC()} to convert datetime/C values to datetime/c.

If precision matters, the best way to process datetime/C data is simply to treat them that way. The inconvenience is that you cannot assume that there are 86,400 seconds per day. To obtain the duration between dates, you must subtract the two time values involved. The other difficulty has to do with dealing with dates in the future. Under the datetime/C (UTC) encoding, there is no set value for any date more than six months in the future. Below is a summary of advice.

**Advice on using datetime/c and datetime/C**

Stata provides two datetime encodings:

1. datetime/C, also known as UTC, which accounts for leap seconds; and
2. datetime/c, which ignores leap seconds (it assumes 86,400 seconds/day).

Systems vary in how they treat time variables. SAS ignores leap seconds. Oracle includes them. Stata handles either situation. Here is our advice:

- **If you obtain data from a system that accounts for leap seconds, import using Stata’s datetime/C encoding.**
  a. If you later need to export data to a system that does not account for leap seconds, use Stata’s \texttt{cofC()} function to convert time values before exporting.
  b. If you intend to \texttt{tsset} the time variable and the analysis will be at the second level or finer, just \texttt{tsset} the datetime/C variable, specifying the appropriate \texttt{delta()} if necessary—for example, \texttt{delta(1000)} for seconds.
  c. If you intend to \texttt{tsset} the time variable and the analysis will be coarser than the second level (minute, hour, etc.), create a datetime/c variable from the datetime/C variable (\texttt{generate double tctime = cofC(tCtime)}) and \texttt{tsset} that, specifying the appropriate \texttt{delta()} if necessary. You must do that because in a datetime/C variable, there are not necessarily 60 seconds in a minute; some minutes have 61 seconds.

- **If you obtain data from a system that ignores leap seconds, use Stata’s datetime/c encoding.**
  a. If you later need to export data to a system that does account for leap seconds, use Stata’s \texttt{Cofc()} function to convert time values before exporting.
  b. If you intend to \texttt{tsset} the time variable, just \texttt{tsset} it, specifying the appropriate \texttt{delta()}. 
Some users prefer always to use Stata’s datetime/c because %tc values are a little easier to work with. You can always use datetime/c if

- you do not mind having up to 1 second of error; and
- you do not import or export numerical values (clock ticks) from other systems that are using leap seconds, because doing so could introduce nearly 30 seconds of error.

Remember these two things if you use datetime/C variables:

1. The number of seconds between two dates is a function of when the dates occurred. Five days from one date is not simply a matter of adding $5 \times 24 \times 60 \times 60 \times 1000$ ms. You might need to add another 1,000 ms. Three hundred sixty-five days from now might require adding 1,000 or 2,000 ms. The longer the span, the more you might have to add. The best way to add durations to datetime/C variables is to extract the components, add to them, and then reconstruct from the numerical components.

2. You cannot accurately predict datetimes more than six months into the future. We do not know what the datetime/C value of 25dec2026 00:00:00 will be, because every year along the way, the International Earth Rotation Reference Systems Service (IERS) will twice announce whether a leap second will be inserted.

You can help alleviate these inconveniences. Face west and throw rocks. The benefit will be transitory only if the rocks land back on Earth, so you need to throw them really hard. We know what you are thinking, but this does not need to be a coordinated effort.

### Determining when leap seconds occurred

Stata system file `leapseconds.maint` lists the dates on which leap seconds occurred. The file is updated periodically (see [R] update; the file is updated when you update all), and Stata’s datetime/C functions access the file to know when leap seconds occurred.

You can access it, too. To view the file, type

```
    . viewsource leapseconds.maint
```

### The date() function

The syntax of the `date()` function is

```
    date(string, mask [, topyear])
```

The `date()` function is identical to `clock()`, except that `date()` returns a Stata date value rather than a Stata datetime value. The `date()` function is the same as `dofc(clock())`.

daily() is a synonym for `date()`.
The other conversion functions

The other conversion functions are

<table>
<thead>
<tr>
<th>Stata date type</th>
<th>Conversion function</th>
</tr>
</thead>
<tbody>
<tr>
<td>weekly date</td>
<td>weekly(str, mask [, toyear])</td>
</tr>
<tr>
<td>monthly date</td>
<td>monthly(str, mask [, toyear])</td>
</tr>
<tr>
<td>quarterly date</td>
<td>quarterly(str, mask [, toyear])</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>halfyearly(str, mask [, toyear])</td>
</tr>
</tbody>
</table>

str is the value to be converted.
mask specifies the order of the components.
toyear is described in Working with two-digit years, above.

These functions are rarely used because data seldom arrive in these formats.

Each of the functions converts a pair of numbers: weekly() converts a year and a week number (1–52); monthly() converts a year and a month number (1–12); quarterly() converts a year and a quarter number (1–4); and halfyearly() translates a year and a half number (1–2).

The masks allowed are far more limited than the masks for clock(), Clock(), and date():

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>4-digit year</td>
</tr>
<tr>
<td>19Y</td>
<td>2-digit year to be interpreted as 19xx</td>
</tr>
<tr>
<td>20Y</td>
<td>2-digit year to be interpreted as 20xx</td>
</tr>
<tr>
<td>W</td>
<td>week number (weekly() only)</td>
</tr>
<tr>
<td>M</td>
<td>month number (monthly() only)</td>
</tr>
<tr>
<td>Q</td>
<td>quarter number (quarterly() only)</td>
</tr>
<tr>
<td>H</td>
<td>half-year number (halfyearly() only)</td>
</tr>
</tbody>
</table>

The pair of numbers to be converted must be separated by a space or punctuation. No extra characters are allowed.

Reference


Also see

[D] Datetime — Date and time values and variables
[D] Datetime business calendars — Business calendars
[D] Datetime display formats — Display formats for dates and times
[D] Datetime durations — Obtaining and working with durations
[D] Datetime relative dates — Obtaining dates and date information from other dates
[D] Datetime values from other software — Date and time conversion from other software
Datetime display formats — Display formats for dates and times

Description

Stata stores dates and times numerically in one of eight units. The value of a Stata date might be 18,282 or even 1,579,619,730,000. Place the appropriate format on it, and the 18,282 is displayed as 20jan2010 (%td). The 1,579,619,730,000 is displayed as 20jan2010 15:15:30 (%tc).

If you specify additional format characters, you can change how the result is displayed. Rather than 20jan2010, you could change it to 2010.01.20; January 20, 2010; or 1/20/10. Rather than 20jan2010 15:15:30, you could change it to 2010.01.20 15:15; January 20, 2010 3:15 pm; or Wed Jan 20 15:15:30 2010.

See [D] Datetime for an introduction to Stata’s dates and times.

Quick start

Format daily dates stored in datevar to display as 15mar2005
   format datevar %td

Format daily dates stored in datevar to display as 3/15/05
   format datevar %tdnn/DD/YY

Format daily dates stored in datevar to display as Tue Mar. 15
   format datevar %tdDay_Mon._DD

Format dates and times stored in timevar to display as 15mar2005 14:30:00
   format timevar %tc

Format dates and times stored in timevar to display as 14:30
   format timevar %tcHH:MM

Format dates and times stored in timevar to display as 2:30 PM
   format timevar %tchh:mm_AM
Syntax

The formats for displaying Stata dates and times are

<table>
<thead>
<tr>
<th>Stata date type</th>
<th>Display format</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime/c</td>
<td>%tc [details]</td>
</tr>
<tr>
<td>datetime/C</td>
<td>%tC [details]</td>
</tr>
<tr>
<td>date</td>
<td>%td [details]</td>
</tr>
<tr>
<td>weekly date</td>
<td>%tw [details]</td>
</tr>
<tr>
<td>monthly date</td>
<td>%tm [details]</td>
</tr>
<tr>
<td>quarterly date</td>
<td>%tq [details]</td>
</tr>
<tr>
<td>half-yearly date</td>
<td>%th [details]</td>
</tr>
<tr>
<td>yearly date</td>
<td>%ty [details]</td>
</tr>
</tbody>
</table>

The optional details allows you to control how results appear and is composed of a sequence of the following codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>century-1</td>
<td>01–99</td>
</tr>
<tr>
<td>cc</td>
<td>century-1</td>
<td>1–99</td>
</tr>
<tr>
<td>YY</td>
<td>2-digit year</td>
<td>00–99</td>
</tr>
<tr>
<td>yy</td>
<td>2-digit year</td>
<td>0–99</td>
</tr>
<tr>
<td>JJJ</td>
<td>day within year</td>
<td>001–366</td>
</tr>
<tr>
<td>jjj</td>
<td>day within year</td>
<td>1–366</td>
</tr>
<tr>
<td>Mon</td>
<td>month</td>
<td>Jan, Feb, ...</td>
</tr>
<tr>
<td>Month</td>
<td>month</td>
<td>January, February, ...</td>
</tr>
<tr>
<td>mon</td>
<td>month</td>
<td>jan, feb, ...</td>
</tr>
<tr>
<td>month</td>
<td>month</td>
<td>january, february, ...</td>
</tr>
<tr>
<td>NN</td>
<td>month</td>
<td>01–12</td>
</tr>
<tr>
<td>nn</td>
<td>month</td>
<td>1–12</td>
</tr>
<tr>
<td>DD</td>
<td>day within month</td>
<td>01–31</td>
</tr>
<tr>
<td>dd</td>
<td>day within month</td>
<td>1–31</td>
</tr>
<tr>
<td>DAYNAME</td>
<td>day of week</td>
<td>Sunday, Monday, ... (aligned)</td>
</tr>
<tr>
<td>Dayname</td>
<td>day of week</td>
<td>Sunday, Monday, ... (unaligned)</td>
</tr>
<tr>
<td>Day</td>
<td>day of week</td>
<td>Sun, Mon, ...</td>
</tr>
<tr>
<td>Da</td>
<td>day of week</td>
<td>Su, Mo, ...</td>
</tr>
<tr>
<td>day</td>
<td>day of week</td>
<td>sun, mon, ...</td>
</tr>
<tr>
<td>da</td>
<td>day of week</td>
<td>su, mo, ...</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Format</td>
</tr>
<tr>
<td>------</td>
<td>-----------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>h</td>
<td>half</td>
<td>1–2</td>
</tr>
<tr>
<td>q</td>
<td>quarter</td>
<td>1–4</td>
</tr>
<tr>
<td>WW</td>
<td>week</td>
<td>01–52</td>
</tr>
<tr>
<td>ww</td>
<td>week</td>
<td>1–52</td>
</tr>
<tr>
<td>HH</td>
<td>hour</td>
<td>00–23</td>
</tr>
<tr>
<td>Hh</td>
<td>hour</td>
<td>00–12</td>
</tr>
<tr>
<td>hH</td>
<td>hour</td>
<td>0–23</td>
</tr>
<tr>
<td>hh</td>
<td>hour</td>
<td>0–12</td>
</tr>
<tr>
<td>MM</td>
<td>minute</td>
<td>00–59</td>
</tr>
<tr>
<td>mm</td>
<td>minute</td>
<td>0–59</td>
</tr>
<tr>
<td>SS</td>
<td>second</td>
<td>00–60 (sic, due to leap seconds)</td>
</tr>
<tr>
<td>ss</td>
<td>second</td>
<td>0–60 (sic, due to leap seconds)</td>
</tr>
<tr>
<td>.s</td>
<td>tenths</td>
<td>.0–.9</td>
</tr>
<tr>
<td>.ss</td>
<td>hundredths</td>
<td>.00–.99</td>
</tr>
<tr>
<td>.sss</td>
<td>thousandths</td>
<td>.000–.999</td>
</tr>
<tr>
<td>am</td>
<td>show am or pm</td>
<td>am or pm</td>
</tr>
<tr>
<td>a.m.</td>
<td>show a.m. or p.m.</td>
<td>a.m. or p.m.</td>
</tr>
<tr>
<td>AM</td>
<td>show AM or PM</td>
<td>AM or PM</td>
</tr>
<tr>
<td>A.M.</td>
<td>show A.M. or P.M.</td>
<td>A.M. or P.M.</td>
</tr>
<tr>
<td>.</td>
<td>display period</td>
<td>.</td>
</tr>
<tr>
<td>,</td>
<td>display comma</td>
<td>,</td>
</tr>
<tr>
<td>:</td>
<td>display colon</td>
<td>:</td>
</tr>
<tr>
<td>-</td>
<td>display hyphen</td>
<td>-</td>
</tr>
<tr>
<td>_</td>
<td>display space</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>display slash</td>
<td>/</td>
</tr>
<tr>
<td>\</td>
<td>display backslash</td>
<td>\</td>
</tr>
<tr>
<td>!c</td>
<td>display character</td>
<td>c</td>
</tr>
<tr>
<td>+</td>
<td>separator (see note)</td>
<td></td>
</tr>
</tbody>
</table>

Note: + displays nothing; it may be used to separate one code from the next to make the format more readable. + is never necessary. For instance, %tchh:MM+am and %tchh:MMam have the same meaning, as does %tc+hh+:+MM+am.
When *details* is not specified, it is equivalent to specifying

<table>
<thead>
<tr>
<th>Format</th>
<th>Implied (fully specified) format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%tC</td>
<td>%tCDDmonCCYY_HH:MM:SS</td>
</tr>
<tr>
<td>%tc</td>
<td>%tCDDmonCCYY_HH:MM:SS</td>
</tr>
<tr>
<td>%td</td>
<td>%tDDmonCCYY</td>
</tr>
<tr>
<td>%tw</td>
<td>%twCCYY!www</td>
</tr>
<tr>
<td>%tm</td>
<td>%tmCCYY!mnn</td>
</tr>
<tr>
<td>%tq</td>
<td>%tqCCYY!qq</td>
</tr>
<tr>
<td>%th</td>
<td>%thCCYY!hh</td>
</tr>
<tr>
<td>%ty</td>
<td>%tyCCYY</td>
</tr>
</tbody>
</table>

That is, typing

```
.d format mytimevar %tc
```

has the same effect as typing

```
.d format mytimevar %tCDDmonCCYY_HH:MM:SS
```

Format %tCDDmonCCYY_HH:MM:SS is interpreted as

```
% | t | c | DDmonCCYY_HH:MM:SS
   |   |   | formatting codes
   |   | it is a variable formatting codes
   | all formats | datetime format coded in milliseconds | specify how to display value
```

Remarks and examples

Remarks are presented under the following headings:

*Specifying display formats*

*Times are truncated, not rounded, when displayed*

**Specifying display formats**

Rather than using the default format 20jan2010, you could display the daily date in one of these formats:

```
2010.01.20
January 20, 2010
1/20/10
```

Likewise, rather than displaying the datetime/c variable in the default format 20jan2010 15:15:30, you could display it in one of these formats:

```
2010.01.20 15:15
January 20, 2010 3:15 pm
Wed Jan 20 15:15:30 2010
```
Here is how to do it:

1. 2010.01.20
   
   format mytdvar %tdCCYY.NN.DD

2. January 20, 2010
   
   format mytdvar %tdMonth_dd, _CCYY

3. 1/20/10
   
   format mytdvar %tdnn/dd/YY

4. 2010.01.20 15:15
   
   format mytcvar %tcCCYY.NN.DD_HH:MM

5. January 20, 2010 3:15 pm
   
   format mytcvar %tcMonth_dd, _CCYY hh:MM am
   
   Code am at the end indicates that am or pm should be displayed, as appropriate.

6. Wed Jan 20 15:15:30 2010
   
   format mytcvar %tcDay_Mon_DD_HH:MM:SS_CCYY

   In examples 1 to 3, the formats each begin with %td, and in examples 4 to 6, the formats begin with %tc. It is important that you specify the opening correctly—namely, % + t + third_character. The third character indicates the particular encoding type, which is to say, how the numeric value is to be interpreted. You specify %t... for datetime/c variables, %tC... for datetime/C, %td... for date, and so on.

   The default format for datetime/c and datetime/C variables omits the fraction of seconds; 15:15:30.000 is displayed as 15:15:30. If you wish to see the fractional seconds, specify the format
   
   %tcDDmonCCYY_HH:MM:SS.sss
   
   or
   
   %tCDDmonCCYY_HH:MM:SS.sss
   
   as appropriate.

**Times are truncated, not rounded, when displayed**

   Consider the time 11:32:59.999. Other, less precise, ways of writing that time are

   11:32:59.99
   11:32:59.9
   11:32:59
   11:32

   That is, when you suppress the display of more-detailed components of the time, the parts that are displayed are not rounded. Stata displays time just as a digital clock would; the time is 11:32 right up until the instant that it becomes 11:33.
Also see

[D] Datetime — Date and time values and variables
[D] Datetime business calendars — Business calendars
[D] Datetime conversion — Converting strings to Stata dates
[D] Datetime durations — Obtaining and working with durations
[D] Datetime relative dates — Obtaining dates and date information from other dates
[D] Datetime values from other software — Date and time conversion from other software
Title

**Datetime durations — Obtaining and working with durations**

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Syntax</th>
<th>Remarks and examples</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Also see</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

This entry describes functions that calculate durations, such as the number of years between two dates (for example, a person’s age). These functions account for leap years and leap days and produce results that are more consistent than simply taking arithmetic differences of numerical dates and converting to another unit.

This entry also describes functions that convert durations from one unit (for example, milliseconds) to another (for example, hours).

**Quick start**

Calculate age of a subject in integer years on the date of a survey based on a numerically encoded Stata date `dob` that gives the subject’s date of birth and a numerically encoded Stata date `date_of_survey` that gives the subject's date of survey.

```stata
generate subject_age = age(dob, date_of_survey)
```

As above, but calculate the age as a noninteger; that is, include the fractional part.

```stata
generate subject_fage = age_frac(dob, date_of_survey)
```

Calculate age on date `d` for persons born on 29feb as having their birthday on 28feb in nonleap years (rather than the default of 01mar).

```stata
generate celebrate = age(dob, d, "28feb")
```

Calculate the difference in number of months, rounded down to an integer, between two Stata dates, `d1` and `d2`.

```stata
generate diff_months = datediff(d1, d2, "month")
```

As above, but include the fractional part of the difference.

```stata
generate diff_fmonths = datediff_frac(d1, d2, "month")
```

Calculate the difference in number of hours, rounded down to an integer, between two Stata `datetime/c` variables, `t1` and `t2`.

```stata
generate diff_hours = clockdiff(t1, t2, "hour")
```

As above, but include the fractional part of the difference.

```stata
generate diff_fhours = clockdiff_frac(t1, t2, "hour")
```

As above, but use a conversion function to calculate hours with a fractional part.

```stata
generate diff_fhours2 = hours(t2 - t1)
```

Calculate the difference in number of minutes, rounded down to an integer, between two Stata `datetime/C` variables, `tvar1` and `tvar2`.

```stata
generate diff_minutes = clockdiff(tvar1, tvar2, "minute")
```
Syntax

Syntax is presented under the following headings:

Functions for calculating durations
Functions for converting units of a duration

Functions for calculating durations

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>age($e_{d \text{DOB}}, e_d[, s_{nl}]$)</td>
<td>years rounded down to an integer</td>
</tr>
<tr>
<td>age with fraction</td>
<td>age_frac($e_{d \text{DOB}}, e_d[, s_{nl}]$)</td>
<td>years with fractional part</td>
</tr>
<tr>
<td>datetime/C difference</td>
<td>Clockdiff($e_{tC1}, e_{tC2}, s_{tu}$)</td>
<td>integer (rounded down)</td>
</tr>
<tr>
<td>datetime/c difference</td>
<td>clockdiff($e_{tc1}, e_{tc2}, s_{tu}$)</td>
<td>integer (rounded down)</td>
</tr>
<tr>
<td>datetime/C difference with fraction</td>
<td>Clockdiff_frac($e_{tC1}, e_{tC2}, s_{tu}$)</td>
<td>floating point</td>
</tr>
<tr>
<td>datetime/c difference with fraction</td>
<td>clockdiff_frac($e_{tc1}, e_{tc2}, s_{tu}$)</td>
<td>floating point</td>
</tr>
<tr>
<td>date difference</td>
<td>datediff($e_{d1}, e_{d2}, s_{du}[ , s_{nl}]$)</td>
<td>integer (rounded down)</td>
</tr>
<tr>
<td>date difference with fraction</td>
<td>datediff_frac($e_{d1}, e_{d2}, s_{du}[ , s_{nl}]$)</td>
<td>floating point</td>
</tr>
</tbody>
</table>

$e_d$, $e_{d \text{DOB}}$, $e_{d1}$, and $e_{d2}$ are Stata dates.
$e_{tC1}$ and $e_{tC2}$ are Stata datetime/C values.
$e_{tc1}$ and $e_{tc2}$ are Stata datetime/c values.
$s_{nl}$ is a string specifying nonleap-year birthdays or anniversaries of 29feb and may be
"01mar", "1mar", "mar01", or "mar1" (the default); or
"28feb" or "feb28" (case insensitive).
$s_{tu}$ is a string specifying time units:
"day" or "d" for day;
"hour" or "h" for hour;
"minute", "min", or "m" for minute;
"second", "sec", or "s" for second; or
"millisecond" or "ms" for millisecond (case insensitive).
$s_{du}$ is a string specifying date units:
"day" or "d" for day;
"month", "mon", or "m" for month; or
"year" or "y" for year (case insensitive).

Notes:

1. The string $s_{nl}$ specifying nonleap-year birthdays or anniversaries is an optional argument. It rarely needs to be specified. See example 3 below.
2. When $e_d < e_{d \text{DOB}}$, age($e_{d \text{DOB}}, e_d[, s_{nl}]$) and age_frac($e_{d \text{DOB}}, e_d[, s_{nl}]$) return missing (.).
3. Clockdiff($e_{tC1}, e_{tC2}, s_{tu}$) = −Clockdiff($e_{tC2}, e_{tC1}, s_{tu}$).
   clockdiff(), Clockdiff_frac(), clockdiff_frac(), datediff(), and datediff_frac() have the same anticommutative property.
Functions for converting units of a duration

<table>
<thead>
<tr>
<th>Desired conversion</th>
<th>Function</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>milliseconds to hours</td>
<td>hours(ms)</td>
<td>ms/(60 × 60 × 1000)</td>
</tr>
<tr>
<td>milliseconds to minutes</td>
<td>minutes(ms)</td>
<td>ms/(60 × 1000)</td>
</tr>
<tr>
<td>milliseconds to seconds</td>
<td>seconds(ms)</td>
<td>ms/1000</td>
</tr>
<tr>
<td>hours to milliseconds</td>
<td>msofhours(h)*</td>
<td>h × 60 × 60 × 1000</td>
</tr>
<tr>
<td>minutes to milliseconds</td>
<td>msofminutes(m)*</td>
<td>m × 60 × 1000</td>
</tr>
<tr>
<td>seconds to milliseconds</td>
<td>msofseconds(s)*</td>
<td>s × 1000</td>
</tr>
</tbody>
</table>

* Stata datetime values are in milliseconds and must be stored as doubles. When using millisecond results to add to or subtract from a Stata datetime, store the results as doubles.

Remarks and examples

Remarks are presented under the following headings:

- Calculating ages and differences of dates
- Calculating differences of datetimes

We assume you have read [D] Datetime and are familiar with how Stata stores dates and datetimes. String dates and times must be converted into numeric values to become Stata dates and datetimes. Stata date and time values are durations (positive or negative) from 01jan1960. Stata date values record the number of days from 01jan1960. Stata datetime/c values record the number of milliseconds from 01jan1960 00:00:00. Stata datetime/C is the same as datetime/c, except that it accounts for leap seconds and encodes Coordinated Universal Time (UTC).

There are other types of Stata date and time values, ones for weeks, months, quarters, half years, and years, but the functions described here are intended for use with daily dates or datetimes.

Calculating ages and differences of dates

The `age()` function calculates age just as one would expect. Typing

```
   . generate subject_age = age(date_of_birth, current_date)
```

produces integers that are a person’s age in years on `current_date` given birthdate `date_of_birth`. The variables `date_of_birth` and `current_date` must be Stata dates.

The arguments of `age()` need not be variables, but they must be Stata date values, which are numeric. To get Stata date values for literal dates, we can use the date pseudofunction `td()` and use its results as arguments to `age()`. For example,

```
   . display age(td(05feb1927), td(24may2006))
   79
```

shows that an individual born on 05feb1927 was 79 years old on 24may2006.

`age_frac()` returns age including the fractional part. For example, let’s use `age_frac()` with the dates we specified above:

```
   . display age_frac(td(05feb1927), td(24may2006))
   79.29589
```
The `datediff()` and `datediff_frac()` functions produce results in units of years, months, or days. For example, to determine the number of months between `05feb1927` and `24may2006`, first as an integer (rounded down) and as a number including the fractional part, we type

```
. display datediff(td(05feb1927), td(24may2006), "month")
951
. display datediff_frac(td(05feb1927), td(24may2006), "month")
951.6129
```

The optional last argument, `s_nl`, for `age()`, `age_frac()`, `datediff()`, and `datediff_frac()` was not specified in any of the above examples. It applies only to a date of birth (or starting date) on `29feb` when the ending date is not in a leap year. The argument controls whether to use `01mar` (the default) or `28feb` as the birthday (or anniversary) in nonleap years. Setting this argument is important only when the data you are using have a set rule for determining the age of persons born on `29feb`. For example, you might have data on the dates when people first get their driver’s licenses. You would want the argument to match the legal rule for the data. See example 3.

The functions `age()` and `age_frac()` are based on `datediff()` and `datediff_frac()`, respectively,

```
age(e_dDOB,s_nl)=datediff(e_dDOB,e_d,"year",s_nl)
```

and

```
age_frac(e_dDOB,s_nl)=datediff_frac(e_dDOB,e_d,"year",s_nl)
```

when `e_d ≥ e_dDOB`. When `e_d < e_dDOB`, `age()` and `age_frac()` return `missing` (.).

`datediff(...,"year",...)` and `datediff_frac(...,"year",...)` calculate the number of years between two dates just as one would expect. The only wrinkles are leap days and leap years. See *Methods and formulas* in [FN] Date and time functions for details.

The usefulness of these functions is solely in the way they handle leap days and leap years. Suppose, for example, you are doing an analysis of age of onset of some disorder. If you use values from `age_frac()` as time in a survival model, these times will match up perfectly with recorded ages (or ages from `age()` of course). If instead you used

```
. generate time_years = (onset_date - date_of_birth)/365.25
```

as your time variable, there would be minor discrepancies between this time and ages at birthdays. See examples below.

`datediff(...,"month",...)` and `datediff_frac(...,"month",...)` calculate the number of months between two dates as one would expect for starting days 1–28. For example, a starting date on the 28th of the month will have month anniversaries on the 28th of all other months. When the day of the starting date is 29, 30, or 31, other months may not have this day of the month. The last day of February will be 28 or 29. When the starting date is on the 31st, the months ending on the 30th obviously do not have a 31st. In these cases, the first day of the next month is considered the month anniversary. (This is consistent with the default handling of `29feb` start dates when calculating year anniversaries in nonleap years; the nonleap year anniversaries are on `01mar`.)

Fractional months are also a bit tricky because lengths of months vary. There is an example below, and see *Methods and formulas* in [FN] Date and time functions for how they are calculated.

Note that `datediff(...,"year",...)`, `datediff_frac(...,"year",...)`, `datediff(...,"month",...)`, and `datediff_frac(...,"month",...)` all match up. That is, on an ending date on which `datediff(...,"year",...)` increases by one from the previous day, the value of `datediff_frac(...,"year",...)` is exactly an integer and equal to `datediff(...,"year",...)`. On this ending date, `datediff_frac(...,"month",...)` is also an integer and equal to 12 times the year difference.
datediff($e_{d1}, e_{d2}, "day", s_{nl})$ and $\text{datediff\_frac}(e_{d1}, e_{d2}, "day", s_{nl})$ have no complications in how they are calculated. Both are equal to $e_{d2} - e_{d1}$ and are always integers. The optional argument $s_{nl}$ has no bearing on the calculation and is ignored if specified.

> **Example 1: Ages**

Calculating ages is straightforward, but we do need to show how $\text{age\_frac}()$ calculates the fractional part of age. Here is an example.

We have a dataset with string dates. Date of birth is recorded in the variable $\text{str\_dob}$, and the end date for calculating age is in $\text{str\_end\_date}$.

```stata
. use https://www.stata-press.com/data/r17/ages
(Fictional data for calculating ages)
. describe
Contains data from https://www.stata-press.com/data/r17/ages.dta
Observations: 5 Fictional data for calculating ages
Variables: 2 30 Oct 2020 17:35
  Variable Storage Display Value
name type format label Variable label
  str_dob str9 %9s Date of birth
  str_end_date str9 %9s End date
Sorted by:
. list str_dob str_end_date
  str_dob str_end_date
  1. 28/8/1967 27/8/2019
  3. 28/8/1967 29/8/2019
  5. 28/8/1967 29/8/2020
```

We must convert the strings to numeric Stata dates, which we do using the $\text{date()}$ function with a mask of "DMY" because the date components are in the order day, month, year. We format the new encoded date variables using format \%td, the simplest format specification for daily dates.

```stata
. generate dob = date(str_dob, "DMY")
. generate end_date = date(str_end_date, "DMY")
. format dob end_date \%td
. list str_dob dob str_end_date end_date, abbreviate(12)
```

<table>
<thead>
<tr>
<th>str_dob</th>
<th>dob</th>
<th>str_end_date</th>
<th>end_date</th>
</tr>
</thead>
</table>
This person was born on 28aug1967, and we compute his or her age and age with the fractional part on the dates in end_date.

```
. generate age = age(dob, end_date)
. generate double fage = age_frac(dob, end_date)
. format fage %12.0g
. list dob end_date age fage

<table>
<thead>
<tr>
<th>dob</th>
<th>end_date</th>
<th>age</th>
<th>fage</th>
</tr>
</thead>
<tbody>
<tr>
<td>28aug1967</td>
<td>27aug2019</td>
<td>51</td>
<td>51.99726027</td>
</tr>
<tr>
<td>28aug1967</td>
<td>28aug2019</td>
<td>52</td>
<td>52</td>
</tr>
<tr>
<td>28aug1967</td>
<td>29aug2019</td>
<td>52</td>
<td>52.00273224</td>
</tr>
<tr>
<td>28aug1967</td>
<td>28aug2020</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>28aug1967</td>
<td>29aug2020</td>
<td>53</td>
<td>53.00273973</td>
</tr>
</tbody>
</table>
```

Note that the fractional parts on end dates of 29aug2019 and 29aug2020 differ. There are 366 days between 28aug2019 and 28aug2020 because 2020 is a leap year. So the fractional part for 29aug2019 is \( \frac{1}{366} = 0.00273224 \). There are 365 days between 28aug2020 and 28aug2021, so the fractional part for 29aug2020 is \( \frac{1}{365} = 0.00273973 \).

Example 2: Differences in months

Here we show an example of how datediff() and datediff_frac() calculate date differences in units of months.

We load a dataset with Stata date variables start and end. First, we generate months using datediff(start, end, "month") to get the integer difference (rounded down) in months. Then, we generate fmonths using datediff_frac(start, end, "month") to get the difference including the fractional part. We also put datediff(start, end, "day") into a variable to get differences in days to help us see how the fractional parts are calculated.

```
. use https://www.stata-press.com/data/r17/month_differences, clear
   (Fictional data for calculating date differences)
. generate months = datediff(start, end, "month")
. generate double fmonths = datediff_frac(start, end, "month")
. generate days = datediff(start, end, "day")
. format fmonths %12.0g
```
Let's first look at the start date 15jan2019. months increases by one on 15feb2019 and then again on 15mar2019 and 15apr2019. On these days, datediff frac(..., "month") is an integer.

The fractional month difference between 15jan2019 and 16jan2019 is 1/31 = 0.032258. The denominator is 31 because the next month anniversary is 15feb2019, which is 31 days from 15jan2019. The fractional part of the difference between 15jan2019 and 16feb2019 is 1/28 = 0.035714 because there are 28 days between the month anniversaries 15feb2019 and 15mar2019. The fractional part of the difference between 15jan2019 and 16apr2019 is 1/30 = 0.033333 because there are 30 days between the month anniversaries 15apr2019 and 15may2019.

For the start date 31jan2019, monthly anniversaries are 01mar2019, 31mar2019, and 01may2019. Fractional differences are calculated based on the number of days between the monthly anniversaries. For example, there are 29 days between 31jan2019 and 01mar2019, so the fractional difference between 31jan2019 and 01feb2019 is 1/29 = 0.034483.

The optional fourth argument, snl, of datediff(e_d1, e_d2, "month", snl) applies only when the start date, e_d1, falls on 29feb. See the next example for what this option does with ages in years. It works similarly when units are months.

Example 3: Born on a leap day

If you are a “leapling”—born on 29feb—when do you have a birthday in nonleap years? On 28feb or 01mar? Or do you not have a birthday at all in nonleap years (Sullivan 1923)?

In the United Kingdom, a leapling legally becomes 18 on 01mar. In Taiwan, it is 28feb. In the United States, there is no legal statute concerning leap-day birthdates.

The functions age(), age frac(), datediff(), and datediff frac() all have an optional last argument that sets the day of the birthday (or anniversary) in nonleap years. Here is an example using age() and age frac().

We load a dataset with Stata date variables dob (containing date of birth) and end date. We generate age1 using age() with the "01mar" argument (which is the default if it is not specified). The age2 variable is generated using "28feb". We also generate the variables fage1 and fage2 using age frac() with different last arguments.
. use https://www.stata-press.com/data/r17/leap_day, clear
  (Fictional leapling data)
. generate age1 = age(dob, end_date, "01mar")
. generate double fage1 = age_frac(dob, end_date, "01mar")
. generate age2 = age(dob, end_date, "28feb")
. generate double fage2 = age_frac(dob, end_date, "28feb")
. generate year = year(end_date)
. format fage1 fage2 %12.0g
. list dob end_date age1 age2 fage1 fage2, sepby(year)

<table>
<thead>
<tr>
<th></th>
<th>dob</th>
<th>end_date</th>
<th>age1</th>
<th>age2</th>
<th>fage1</th>
<th>fage2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>29feb2004</td>
<td>28feb2019</td>
<td>14</td>
<td>15</td>
<td>14.99726027</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>29feb2004</td>
<td>01mar2019</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15.00273224</td>
</tr>
<tr>
<td>4</td>
<td>29feb2004</td>
<td>28feb2020</td>
<td>15</td>
<td>15</td>
<td>15.99726027</td>
<td>15.99726776</td>
</tr>
<tr>
<td>5</td>
<td>29feb2004</td>
<td>01mar2020</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>29feb2004</td>
<td>01mar2020</td>
<td>16</td>
<td>16</td>
<td>16.00273224</td>
<td>16.00273973</td>
</tr>
</tbody>
</table>

Changes in \texttt{age1} and \texttt{age2} (that is, birthdays) in nonleap years occur on the day specified by the last argument to \texttt{age()}. Note that birthdays in leap years are, of course, on 29feb regardless of the last argument. Fractional parts from \texttt{age\_frac()} differ because they are based on the number of days between birthdays on either side of \texttt{end\_date}, which will be 365 or 366. So fractional parts are multiples of 1/365 or 1/366.

It is worth mentioning again that \texttt{age()}, \texttt{age\_frac()}, \texttt{datediff()}, and \texttt{datediff\_frac()} all match up sensibly, but if there are leaplings, the last argument must be the same (or not be specified) for them to match up. See \textit{Methods and formulas} in [FN] \textbf{Date and time functions}.

\section*{Calculating differences of datetimes}

The \texttt{clockdiff()} function calculates differences of datetime/c values in units of days, hours, minutes, seconds, or milliseconds, with the result rounded down to an integer. The \texttt{Clockdiff()} function does the same, except it calculates differences for datetime/C values (UTC times with leap seconds).

The \texttt{clockdiff\_frac()} and \texttt{Clockdiff\_frac()} functions calculate the corresponding differences for datetime/c and datetime/C values, respectively, but the fractional part of the difference is also included.
Example 4: Differences of datetime/c values

We have a dataset with string datetimes. A start datetime is recorded in the variable `str_start`, and an end datetime is in `str_end`.

```
use https://www.stata-press.com/data/r17/time_differences, clear
(Fictional data for calculating time differences)
list, abbreviate(9)
```

<table>
<thead>
<tr>
<th>str_start</th>
<th>str_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-06-30 00:00:00</td>
<td>2015-06-30 23:59:59</td>
</tr>
<tr>
<td>2015-06-30 00:00:00</td>
<td>2015-06-30 23:59:60</td>
</tr>
<tr>
<td>2015-06-30 00:00:00</td>
<td>2015-07-01 00:00:00</td>
</tr>
<tr>
<td>2015-06-30 00:00:00</td>
<td>2015-07-01 23:59:59</td>
</tr>
<tr>
<td>2015-06-30 00:00:00</td>
<td>2015-07-02 00:00:00</td>
</tr>
</tbody>
</table>
```

We must convert the strings to numeric Stata datetimes, which we do using the `clock()` function with a mask of "YMDhms". We format the new encoded datetime variables using format `%tc`, the simplest format specification for datetime/c.

```
generate double cstart = clock(str_start, "YMDhms")
generate double cend = clock(str_end, "YMDhms")
(1 missing value generated)
format cstart cend %tc
list str_end cend
```

<table>
<thead>
<tr>
<th>str_end</th>
<th>cend</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>2015-07-01 00:00:00</td>
<td>01jul2015 00:00:00</td>
</tr>
<tr>
<td>2015-07-02 00:00:00</td>
<td>02jul2015 00:00:00</td>
</tr>
</tbody>
</table>
```

One of the string values became missing when it was encoded. It was the value "2015-06-30 23:59:60". This is a leap second, which was added to the end of the day on 30jun2015. There is no encoding for leap seconds in datetime/c. That is why it is missing. We snuck in this leap second to illustrate a point later about datetime/C.

We now use clockdiff() to calculate differences in seconds and hours between the datetime/c variables `cstart` and `cend`.

```
generate csecs = clockdiff(cstart, cend, "second")
(1 missing value generated)
generate chours = clockdiff(cstart, cend, "hour")
(1 missing value generated)
list cstart cend csecs chours
```

<table>
<thead>
<tr>
<th>cstart</th>
<th>cend</th>
<th>csecs</th>
<th>chours</th>
</tr>
</thead>
<tbody>
<tr>
<td>30jun2015 00:00:00</td>
<td>30jun2015 23:59:59</td>
<td>86399</td>
<td>23</td>
</tr>
<tr>
<td>30jun2015 00:00:00</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>30jun2015 00:00:00</td>
<td>01jul2015 00:00:00</td>
<td>86400</td>
<td>24</td>
</tr>
<tr>
<td>30jun2015 00:00:00</td>
<td>01jul2015 23:59:59</td>
<td>172799</td>
<td>47</td>
</tr>
<tr>
<td>30jun2015 00:00:00</td>
<td>02jul2015 00:00:00</td>
<td>172800</td>
<td>48</td>
</tr>
</tbody>
</table>
datetime durations — Obtaining and working with durations

clockdiff() calculates values rounded down to integers, and the results are what we expect. Integer hours starting at 30jun2015 00:00:00 are 23 hours at 30jun2015 23:59:59. Integer hours become 24 hours one second later at 01jul2015 00:00:00.

Rather than use clockdiff(), we could take the difference between the datetime/c variables cstart and cend and use the conversion functions seconds() and hours().

```
. generate double csecs2 = seconds(cend - cstart)
   (1 missing value generated)
. generate double chours2 = hours(cend - cstart)
   (1 missing value generated)
. format %12.0g chours2
. list csecs csecs2 chours chours2

   csecs  csecs2   chours   chours2
  ------------- -------------  -------------
     1.    86399   86399       23  23.99972222
     2.        .        .   .           .
     3.    86400   86400       24   24
     4.  172799  172799       47  47.99972222
     5.  172800  172800       48   48
```

The results are consistent with our earlier results. The number of seconds are exactly the same in csecs and csecs2 because they are integers. Hours in chours2 are not integers, but rounded down to integers, they agree with hours produced by clockdiff().

If we want to calculate the difference between cstart and cend in hours with the fractional part, we can use clockdiff_frac() as follows:

```
. generate double fchours = clockdiff_frac(cstart, cend, "hour")
   (1 missing value generated)
. format %12.0g fchours
. list chours chours2 fchours

   chours  chours2   fchours
  -----  -------------  -------------
     1.      23  23.99972222       23.99972222
     2.        .        .           .
     3.      24   24            24
     4.     47  47.99972222      47.99972222
     5.      48   48            48
```

As expected, fchours is the same as chours2.

Example 5: Differences of datetime/C values

What if we are using datetime/C values, that is, datetimes with leap seconds? Let’s redo the previous example encoding the strings using Clock() to produce Cstart and Cend as datetime/C. Then, we generate a variable Csecs using Clockdiff(Cstart, Cend, "second"), Chours using clockdiff(Cstart, Cend, "hour"), and fChours using Clockdiff_frac(Cstart, Cend, "hour").

```
. generate double Cstart = Clock(str_start, "YMDhms")
. generate double Cend = Clock(str_end, "YMDhms")
```
. format Cstart Cend %tC
. generate Csecs = Clockdiff(Cstart, Cend, "second")
. generate Chours = Clockdiff(Cstart, Cend, "hour")
. generate double fChours = Clockdiff_frac(Cstart, Cend, "hour")
. format %12.0g fChours
. list Cstart Cend Csecs Chours fChours

<table>
<thead>
<tr>
<th></th>
<th>Cstart</th>
<th>Cend</th>
<th>Csecs</th>
<th>Chours</th>
<th>fChours</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30jun2015 00:00:00</td>
<td>30jun2015 23:59:59</td>
<td>86399</td>
<td>23</td>
<td>23.9994446</td>
</tr>
<tr>
<td>2</td>
<td>30jun2015 00:00:00</td>
<td>30jun2015 23:59:60</td>
<td>86400</td>
<td>23</td>
<td>23.9997223</td>
</tr>
<tr>
<td>3</td>
<td>30jun2015 00:00:00</td>
<td>01jul2015 00:00:00</td>
<td>86401</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>30jun2015 00:00:00</td>
<td>01jul2015 23:59:59</td>
<td>172800</td>
<td>47</td>
<td>47.99972222</td>
</tr>
<tr>
<td>5</td>
<td>30jun2015 00:00:00</td>
<td>02jul2015 00:00:00</td>
<td>172801</td>
<td>48</td>
<td>48</td>
</tr>
</tbody>
</table>

In the previous example, the difference between the times of the first observation was 23.99972222 hours; now it is 23.99944460 hours. The difference for the first observation in this example is further from 24 hours because there are now two seconds between Cend and 24 hours from Cstart, whereas before there was only one second because the leap second was treated as if it did not exist.

The other difference is the denominator of the fractional part. From the earlier example using datetime/c values and clockdiff_frac(), we note that $1 - 0.99972222 = 0.00027778 = 1/3600$, where 3,600 is the number of seconds in an hour. In this example using datetime/C values and Clockdiff_frac(), we see that $1 - 0.99944460 = 0.00055540 = 2/3601$, where 3,601 is the number of seconds in the hour containing the leap second.

For the second-to-last observation, the fractional part of the difference is 0.99972222, the same as the fractional part in the previous example. So in this example, the hour differences with the fractional part are not evenly spaced, and this would be true even without the second observation with the leap second.
second in the data. If the lack of uniform spacing is a problem and there are no leap seconds in your data, you may want to consider converting your datetime/C data to datetime/c.

Reference


Also see

[D] **Datetime** — Date and time values and variables
[D] **Datetime business calendars** — Business calendars
[D] **Datetime conversion** — Converting strings to Stata dates
[D] **Datetime display formats** — Display formats for dates and times
[D] **Datetime relative dates** — Obtaining dates and date information from other dates
[D] **Datetime values from other software** — Date and time conversion from other software
**Description**

This entry describes functions that calculate dates from other dates, such as the date of a birthday in another year or the next leap year after a given year. It also describes functions that return the current date and current datetime.

**Quick start**

Display today’s date

```stata
display %td today()
```

Save the current date and time in a scalar

```stata
scalar ctime = now()
```

Calculate the date of a birthday in the year given by numeric variable `y` based on a numerically encoded Stata date variable `dob` that gives date of birth

```stata
generate bday_future = birthday(dob, y)
```

As above, but for persons born on 29feb have their birthdays on 28feb in nonleap years (rather than the default of 01mar)

```stata
generate bday_future = birthday(dob, y, "28feb")
```

Calculate the date of the first birthday after Stata date `date_today` based on date of birth `dob`

```stata
generate next_bday = nextbirthday(dob, date_today)
```

Calculate the number of days in the year `y`

```stata
generate ndays = cond(isleapyear(y), 366, 365)
```

Calculate the year of the leap year immediately before the year `y`

```stata
generate yleap = previousleapyear(y)
```

Calculate the number of days in the month on which the values of Stata date variable `d` fall

```stata
generate ndays = daysinmonth(d)
```
## Syntax

<table>
<thead>
<tr>
<th>Description</th>
<th>Function</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>today</td>
<td>today()</td>
<td>Stata date</td>
</tr>
<tr>
<td>current date and time</td>
<td>now()</td>
<td>Stata datetime/c</td>
</tr>
<tr>
<td>birthday in year</td>
<td>birthday($ed$, $Y[,s_{nl}]$)</td>
<td>Stata date</td>
</tr>
<tr>
<td>previous birthday</td>
<td>previousbirthday($ed$, $ed[,s_{nl}]$)</td>
<td>Stata date</td>
</tr>
<tr>
<td>next birthday</td>
<td>nextbirthday($ed$, $ed[,s_{nl}]$)</td>
<td>Stata date</td>
</tr>
<tr>
<td>days in month</td>
<td>daysinmonth($ed$)</td>
<td>28–31</td>
</tr>
<tr>
<td>first day of month</td>
<td>firstdayofmonth($ed$)</td>
<td>Stata date</td>
</tr>
<tr>
<td>last day of month</td>
<td>lastdayofmonth($ed$)</td>
<td>Stata date</td>
</tr>
<tr>
<td>leap year indicator</td>
<td>isleapyear($Y$)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>previous leap year</td>
<td>previousleapyear($Y$)</td>
<td>year</td>
</tr>
<tr>
<td>next leap year</td>
<td>nextleapyear($Y$)</td>
<td>year</td>
</tr>
<tr>
<td>leap second indicator</td>
<td>isleapsecond($etC$)</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

$ed$ and $ed$ are Stata dates.

$etC$ is a Stata datetime/C value (UTC time with leap seconds).

$s_{nl}$ is a string specifying nonleap-year birthdays of 29feb and may be "01mar", "1mar", "mar01", or "mar1" (the default); or "28feb" or "feb28" (case insensitive).

$Y$ is a numeric year.

### Note:

The string $s_{nl}$ specifying nonleap-year birthdays is an optional argument. It rarely needs to be specified. See example 3 in [D] Datetime durations.

## Remarks and examples

Remarks are presented under the following headings:

- Current date and time
- Birthdays and anniversaries
- Months: Number of days, first day, and last day
- Determining leap years
- Determining leap seconds

We assume you have read [D] Datetime and are familiar with how Stata stores and formats dates.
Current date and time

today() and now() return date and datetime/c values for today’s date and the current datetime, respectively. Note that the datetime value returned by now() is not adjusted for leap seconds.

Birthdays and anniversaries

The birthday() function returns a Stata date giving the birthday in a specified year. For example, suppose date_of_birth is a variable containing Stata dates and yvar is a numeric variable containing years; typing

```
generate bday = birthday(date_of_birth, yvar)
```

produces a Stata date variable bday containing birthdays in those years. However, it will not be formatted as a date variable. If you list bday, you will see numbers, not dates. To see dates, you must give it a date format, such as

```
generate bday = birthday(date_of_birth, yvar)
format bday %td
```

We used the format %td, the simplest format specification for daily dates.

Of course, birthday() can be used for more than just birthdays. It can be used to give anniversary dates of any date in different years.

The previousbirthday() and nextbirthday() functions do what their names suggest. Typing

```
generate pbday = previousbirthday(date_of_birth, current_date)
generate nbday = nextbirthday(date_of_birth, current_date)
```

allows birthdays immediately before current_date. Typing

```
generate nbday = nextbirthday(date_of_birth, current_date)
```

allows birthdays immediately after current_date. Note that if current_date is a birthday, previousbirthday() returns the previous birthday, not the value of current_date. Similarly, nextbirthday() returns the next birthday when the argument is a birthday.

The optional last argument, snl, for birthday(), previousbirthday(), and nextbirthday() applies only to a date of birth on 29feb. The argument controls whether to use 01mar (the default) or 28feb as the birthday in nonleap years. See example 3 in [D] Datetime durations and the example below.

Example 1: Birthdays in other years

Here we show how to use birthday() and nextbirthday() to calculate birthdays in other years. We load a dataset with Stata date variables dob and date and a numeric variable year.
. use https://www.stata-press.com/data/r17/birthdays
(Fictional data for calculating birthdays)
. list, sepby(dob)

<table>
<thead>
<tr>
<th>dob</th>
<th>date</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 28 Aug 1967</td>
<td>Sat 28 Aug 2021</td>
<td>2021</td>
</tr>
<tr>
<td>Mon 28 Aug 1967</td>
<td>Mon 29 Aug 2022</td>
<td>2022</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Tue 28 Feb 2023</td>
<td>2023</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Thu 29 Feb 2024</td>
<td>2024</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Sat 01 Mar 2025</td>
<td>2025</td>
</tr>
</tbody>
</table>

To calculate the birthday in year based on date of birth `dob`, we type

. generate bday = birthday(dob, year)
. format bday %tdDay_DD_Mon_CCYY
. list dob year bday, sepby(dob)

<table>
<thead>
<tr>
<th>dob</th>
<th>year</th>
<th>bday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 28 Aug 1967</td>
<td>2022</td>
<td>Sun 28 Aug 2022</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>2023</td>
<td>Wed 01 Mar 2023</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>2024</td>
<td>Thu 29 Feb 2024</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>2025</td>
<td>Sat 01 Mar 2025</td>
</tr>
</tbody>
</table>

We see that for a date of birth of 28 Aug 1967, the birthday in 2020 is on 28 Aug 2020, which is a Friday. For persons born on leap day 29 Feb 1968, their birthdays in nonleap years will be on 01 Mar. In leap years, of course, they will be on 29 Feb.

Note that we used the fancy date format `%tdDay_DD_Mon_CCYY`. The `%td` at the beginning means it is a format for daily dates. `Day` displays the day of the week abbreviated. The underscore (_.) means put in a space. `DD` displays the day with a leading zero. `Mon` displays the month abbreviated. `CCYY` displays the year with the century. See [D] Datetime display formats for all the format variants.

For persons born on leap days ("leaplings"), we can change the day of their birthdays in nonleap years from the default of 01 Mar to 28 Feb by specifying the optional argument "28feb". For example,

. generate abday = birthday(dob, year, "28feb")
. format abday %tdDay_DD_Mon_CCYY
. list dob year abday, sepby(dob)

<table>
<thead>
<tr>
<th>dob</th>
<th>year</th>
<th>abday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 28 Aug 1967</td>
<td>2022</td>
<td>Sun 28 Aug 2022</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>2023</td>
<td>Tue 28 Feb 2023</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>2024</td>
<td>Thu 29 Feb 2024</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>2025</td>
<td>Fri 28 Feb 2025</td>
</tr>
</tbody>
</table>
Birthdays of leaplings are now on 28 Feb in nonleap years. Birthdays for nonleaplings are unaffected by this argument.

Suppose we want a birthday relative to another date. Say we want the date of the first birthday after date. We can do this by typing

```
generate nbday = nextbirthday(dob, date)
format nbday %tdDay_DD_Mon_CCYY
list dob date nbday, sepby(dob)
```

<table>
<thead>
<tr>
<th>dob</th>
<th>date</th>
<th>nbday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 28 Aug 1967</td>
<td>Mon 29 Aug 2022</td>
<td>Mon 28 Aug 2023</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Tue 28 Feb 2023</td>
<td>Wed 01 Mar 2023</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Thu 29 Feb 2024</td>
<td>Sat 01 Mar 2025</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Sat 01 Mar 2025</td>
<td>Sun 01 Mar 2026</td>
</tr>
</tbody>
</table>

We see that the first birthday after 27 Aug 2020 for someone born on 28 Aug is 28 Aug 2020. The first birthday after 28 Aug 2021 (a birthday) for someone born on 28 Aug is the birthday in the next year, 28 Aug 2022.

The first birthday after 29 Feb 2024 for someone born on 29 Feb is 01 Mar 2025. Again, we can specify the argument "28feb" to change the nonleap-year birthdays of leaplings to 28 Feb.

```
generate anbday = nextbirthday(dob, date, "28feb")
format anbday %tdDay_DD_Mon_CCYY
list dob date anbday, sepby(dob)
```

<table>
<thead>
<tr>
<th>dob</th>
<th>date</th>
<th>anbday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 28 Aug 1967</td>
<td>Mon 29 Aug 2022</td>
<td>Mon 28 Aug 2023</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Tue 28 Feb 2023</td>
<td>Thu 29 Feb 2024</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Thu 29 Feb 2024</td>
<td>Fri 28 Feb 2025</td>
</tr>
<tr>
<td>Thu 29 Feb 1968</td>
<td>Sat 01 Mar 2025</td>
<td>Sat 28 Feb 2026</td>
</tr>
</tbody>
</table>

Now the first birthday after 29 Feb 2024 for someone born on 29 Feb is 28 Feb 2025.

---

**Months: Number of days, first day, and last day**

`daysinmonth(ed), firstdayofmonth(ed), and lastdayofmonth(ed)` each take a Stata date `ed` as an argument and determine the month of that date. `daysinmonth()` returns the number of days in that month. `firstdayofmonth()` returns the date of the first day of that month. `lastdayofmonth()` returns the date of the last day of that month.
For example, for any day in the month of February of leap year 2020 (such as 15feb2020), these functions return the following:

```plaintext
. display daysinmonth(mdy(2,15,2020))
29
. display %td firstdayofmonth(mdy(2,15,2020))
01feb2020
. display %td lastdayofmonth(mdy(2,15,2020))
29feb2020
```

**Determining leap years**

`isleapyear(Y), previousleapyear(Y), and nextleapyear(Y)` are functions that make it easier to handle leap years. Each takes a single argument that is a numeric year.

`isleapyear(Y)` returns 1 if `Y` is a leap year and 0 otherwise. The argument `Y` can be a numeric variable or a literal value. Here are some examples with literal values:

```plaintext
. display isleapyear(2020)
1
. display isleapyear(2021)
0
. display isleapyear(2100)
0
. display isleapyear(2400)
1
```

The year 2020 is a leap year, and 2021 is not. The year 2100 is not because it is divisible by 100 and not by 400. The year 2400 is divisible by 400, so it is a leap year.

`previousleapyear(Y)` returns the leap year immediately before year `Y`. `nextleapyear(Y)` returns the first leap year after year `Y`. Here are examples:

```plaintext
. display previousleapyear(2023)
2020
. display nextleapyear(2023)
2024
. display previousleapyear(2024)
2020
. display nextleapyear(2024)
2028
```

As you can see, when the argument is a leap year, these functions return the next leap year or previous leap year and not the leap year argument.
Determining leap seconds

`isleapsecond()` takes a datetime/C value (UTC time) as an argument and returns 1 (true) if that datetime is one of the 1,000 milliseconds of a leap second and 0 (false) otherwise. For example, the first leap second was introduced on 30jun1972, after the last millisecond of the day. Here is what `isleapsecond()` returns at various points in time, including right before the leap second was added on 30jun1972 (at 23:59.999) and right after the leap second was added on 01jul1972 (at 00:00.000). We use `tC()` to create datetime/C values.

```
. display isleapsecond(tC(30jun1972 23:59:59.999))
   0
. display isleapsecond(tC(30jun1972 23:59:60.000))
   1
. display isleapsecond(tC(30jun1972 23:59:60.999))
   1
. display isleapsecond(tC(01jul1972 00:00:0))
   0
```

`isleapsecond()` is useful for determining whether datetime/C values can be converted to datetime/c without any loss of information. Suppose we have a variable `admitTime` that contains times of patient admissions as datetime/C values. We can type the following:

```
. generate anyleapsec = isleapsecond(admitTime)
. tabulate anyleapsec

<table>
<thead>
<tr>
<th></th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,064</td>
<td>100.00</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>1,064</td>
<td>100.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
```

`anyleapsec` is all zero, so no patient was admitted on a leap second, and we can convert `admitTime` to datetime/c without any times being altered.

```
. generate newTime = cofC(admitTime)
```

Had there been leap seconds in the data, `cofC()` would have converted the leap-second times to times one second later. For example,

```
. display %tc cofC(tC(31dec2016 23:59:60))
01jan2017 00:00:00
```

Also see

[D] Datetime — Date and time values and variables
[D] Datetime business calendars — Business calendars
[D] Datetime conversion — Converting strings to Stata dates
[D] Datetime display formats — Display formats for dates and times
[D] Datetime durations — Obtaining and working with durations
[D] Datetime values from other software — Date and time conversion from other software
Description

Most software packages store dates and times numerically as durations from some base date in specified units, but they differ on the base date and the units. In this entry, we discuss how to convert date and time values that you have imported from other packages to Stata dates.

Remarks and examples

Remarks are presented under the following headings:

Introduction
Converting SAS dates
Converting SPSS dates
Converting R dates
Converting Excel dates
  Example 1: Converting Excel dates to Stata dates
Converting OpenOffice dates
Converting Unix time

Introduction

Different software packages use different base dates for storing dates and times numerically. If you are using one of the specialized subcommands for importing data from another package, you do not need to convert your numeric dates after importing them into Stata. `import sas`, `import spss`, and `import excel` will properly convert those dates to Stata dates. However, if you store data from another package into a more general format, like a text file, you will need to do one of two things.

1. If you bring the date variable into Stata as a string, you will have to convert it to a numeric variable.
2. If you import the date variable as a numeric variable, with values representing the underlying numeric values that the other package used, you will have to convert that value to the numeric value for a Stata date.

Below, we discuss the date systems for different software packages and how to convert their date and time values to Stata dates.

Converting SAS dates

If you have data in a SAS-format file, you may want to use the `import sas` command. If the SAS file contains numerically encoded dates, `import sas` will read those dates and properly store them as Stata dates. You do not need to perform any conversion after importing your data with `import sas`.

On the other hand, if you import data originally from SAS that have been saved into another format, such as a text file, dates and datetimes may exist as the underlying numeric values that SAS used. The discussion below concerns converting those numeric values to Stata dates.
SAS provides dates measured as the number of days since 01jan1960 (positive or negative). This is the same coding as used by Stata:

\[
\begin{align*}
&\text{. generate statadate = sasdate} \\
&\text{. format statadate \%td}
\end{align*}
\]

SAS provides datetimes measured as the number of seconds since 01jan1960 00:00:00, assuming 86,400 seconds/day. SAS datetimes do not have leap seconds. To convert to a Stata datetime/c variable, type

\[
\begin{align*}
&\text{. generate double statatime = (sastime*1000)} \\
&\text{. format statatime \%tc}
\end{align*}
\]

It is important that variables containing SAS datetimes, such as `sastime` above, be imported into Stata as doubles.

### Converting SPSS dates

If you have data in an SPSS-format file, you may want to use the `import spss` command. If the SPSS file contains numerically encoded dates, `import spss` will read those dates and properly store them as Stata dates. You do not need to perform any conversion after importing your data with `import spss`.

On the other hand, if you import data originally from SPSS that have been saved into another format, such as a text file, dates and datetimes may exist as the underlying numeric values that SPSS used. The discussion below concerns converting those numeric values to Stata dates.

SPSS provides dates and datetimes measured as the number of seconds since 14oct1582 00:00:00, assuming 86,400 seconds/day. SPSS datetimes do not have leap seconds. To convert to a Stata datetime/c variable, type

\[
\begin{align*}
&\text{. generate double statatime = (spsstime*1000) + tc(14oct1582 00:00)} \\
&\text{. format statatime \%tc}
\end{align*}
\]

To convert to a Stata date, type

\[
\begin{align*}
&\text{. generate statadate = dofc((spsstime*1000) + tc(14oct1582 00:00))} \\
&\text{. format statadate \%td}
\end{align*}
\]

### Converting R dates

R stores dates as days since 01jan1970. To convert to a Stata date, type

\[
\begin{align*}
&\text{. generate statadate = rdate - td(01jan1970)} \\
&\text{. format statadate \%td}
\end{align*}
\]

R stores datetimes as the number of UTC-adjusted seconds (that is, with leap seconds) since 01jan1970 00:00:00. To convert to a Stata datetime/C variable, type

\[
\begin{align*}
&\text{. generate double statatime = rtime - tC(01jan1970 00:00)} \\
&\text{. format statatime \%tC}
\end{align*}
\]

To convert to a Stata datetime/c variable, type

\[
\begin{align*}
&\text{. generate double statatime = cofC(rtime - tC(01jan1970 00:00))} \\
&\text{. format statatime \%tc}
\end{align*}
\]

There are issues of which you need to be aware when working with datetime/C values; see *Why there are two datetime encodings* and *Advice on using datetime/c and datetime/C*, both in [D] Datetime conversion.
Converting Excel dates

If you have data in an Excel format file, you may want to use the `import excel` command. If the Excel file contains numerically encoded dates, `import excel` will read those dates and properly store them as Stata dates. You do not need to perform any conversion after importing your data with `import excel`.

On the other hand, if you are not using `import excel` and you need to manually convert Excel’s numerically encoded dates to Stata dates, you can refer to the discussion below.

Excel has used different date systems across operating systems. Excel for Windows used the “1900 date system”. Excel for Mac used the “1904 date system”. More recently, Excel has been standardizing on the 1900 date system on all operating systems.

Regardless of operating system, Excel can use either encoding. See https://support.microsoft.com/kb/214330 for instructions on converting workbooks between date systems.

Converted dates will be off by four years if you choose the wrong date system.

Converting Excel 1900 date-system dates:

Excel’s 1900 date system stores dates as days since 31Dec1899 (0Jan1900), and it treats 1900 as a leap year, although it was not. Therefore, this date system contains the nonexistent day 29Feb1900, which is not recognized by Stata. You can see http://www.cpearson.com/excel/datetime.htm for more information on how dates and times are handled in Excel.

Because of this behavior, we need to account for that additional day when converting these numerically encoded dates to Stata dates. In other words, to convert Excel dates on or after 01Mar1900 to Stata dates, we instead use 30Dec1899 as the base.

```
. generate statadate = exceldate + td(30dec1899)
. format statadate %td
```

To convert Excel dates on or before 28Feb1900 to Stata dates, we use 31Dec1899 as the base. For an example of working with these dates, see the technical note following example 1.

Stata stores date and datetime values differently, with dates recorded as the number of days since 01Jan1960 and datetimes recorded as the number of milliseconds from 01Jan1960 00:00:00. However, Excel stores date and time values together in a single number. For datetimes on or after 01Mar1900 00:00:00, Excel stores datetimes as days plus fraction of day since 30Dec1899 00:00:00, such as dddddd..tttttt. The integer records the days, and the fractional part records the number of seconds from 00:00:00, the beginning of the day, divided by the number of seconds in 24 hours (24*60*60 = 86400).

To convert with a one-second resolution to a Stata datetime, type

```
. generate double statatime = round((exceltime+td(30dec1899))*86400)*1000
. format statatime %tc
```

Converting Excel 1904 date-system dates:

For dates on or after 01Jan1904, Excel stores dates as days since 01Jan1904. To convert to a Stata date, type

```
. generate statadate = exceldate + td(01jan1904)
. format statadate %td
```
Datetime values from other software — Date and time conversion from other software

For datetimes on or after 01jan1904 00:00:00, Excel stores datetimes as days plus the fraction of the day since 01jan1904 00:00:00. To convert with a one-second resolution to a Stata datetime, type

\. generate double statatime = round((exceltime+td(01jan1904))*86400)*1000
\. format statatime %tc

Example 1: Converting Excel dates to Stata dates

We have some Excel 1900 date-system dates saved in a tab-delimited file. The file contains patients’ ID numbers and their dates of birth. The numeric variable `bdate` contains the numeric values that Excel used to store those dates.

\. clear
\. import delimited "exceldates.txt"
(encoding automatically selected: ISO-8859-1)
(2 vars, 3 obs)
\. list

```
<table>
<thead>
<tr>
<th>patid</th>
<th>bdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33106</td>
</tr>
<tr>
<td>2</td>
<td>31305</td>
</tr>
<tr>
<td>3</td>
<td>37327</td>
</tr>
</tbody>
</table>
```

Stata dates measure the number of days since January 1, 1960. For dates on or after March 1, 1900, Excel’s base date is December 30, 1899. To convert `bdate` to a Stata date, we need to add the number of days from January 1, 1960, to December 30, 1899 (which is a negative number of days).

\. generate statadate = bdate + td(30dec1899)
\. format statadate %td
\. list

```
<table>
<thead>
<tr>
<th>patid</th>
<th>bdate</th>
<th>statadate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33106</td>
<td>21aug1990</td>
</tr>
<tr>
<td>2</td>
<td>31305</td>
<td>15sep1985</td>
</tr>
<tr>
<td>3</td>
<td>37327</td>
<td>12mar2002</td>
</tr>
</tbody>
</table>
```

If you would like to confirm that the conversion has been done properly, you can copy those values of `bdate` into an Excel spreadsheet and format them as dates. You will see the same dates as those listed under `statadate`.

⚠️ Technical note

Suppose we were working with data in Excel that contained dates between January 1, 1900, and February 28, 1900. If we saved these data to a `.txt` or `.csv` file and brought in those numerically encoded dates into Stata, we could not use the conversion function above. The reason these dates are treated differently is that Excel treats 1900 as a leap year, even though it was not; therefore, Excel behaves as if 29feb1900 was an actual date. If you are curious, the purpose of this behavior was to be compatible with a spreadsheet software that was dominant at the time. In short, what this means for us is that if we are working with these particular dates, we need to modify Excel’s base date.
Below, we import a text file with dates between January 1, 1900, and February 28, 1900, to demonstrate.

```
. clear
. import delimited "exceldates2.txt"
(encoding automatically selected: ISO-8859-1)
(2 vars, 3 obs)
. list
```

```
<table>
<thead>
<tr>
<th>patid</th>
<th>bdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
</tr>
</tbody>
</table>
```

Instead of using December 30, 1899, as Excel’s base date, as we did previously, we will now use December 31, 1899.

```
. generate statadate = bdate + td(31dec1899)
. format statadate %td
. list
```

```
<table>
<thead>
<tr>
<th>patid</th>
<th>bdate</th>
<th>statadate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>01jan1900</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>15jan1900</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>12feb1900</td>
</tr>
</tbody>
</table>
```

Now we have a Stata date recording dates between January 1, 1900, and February 28, 1900.

Converting OpenOffice dates

OpenOffice uses the Excel 1900 date system described above.

Converting Unix time

Unix time is stored as the number of seconds since midnight, 01jan1970. To convert to a Stata datetime, type

```
. generate double statatime = unixtime * 1000 + mdyhms(1,1,1970,0,0,0)
```

To convert to a Stata date, type

```
. generate statadate = dofc(unixtime * 1000 + mdyhms(1,1,1970,0,0,0))
```

Reference

Also see

[D] Datetime — Date and time values and variables
[D] Datetime business calendars — Business calendars
[D] Datetime conversion — Converting strings to Stata dates
[D] Datetime display formats — Display formats for dates and times
[D] Datetime durations — Obtaining and working with durations
[D] Datetime relative dates — Obtaining dates and date information from other dates
describe — Describe data in memory or in file

Description

describe produces a summary of the dataset in memory or of the data stored in a Stata-format dataset.

For a compact listing of variable names, use describe, simple.

Quick start

Describe all variables in the dataset
describe

Describe all variables starting with code
describe code*

Describe properties of the dataset
describe, short

Describe without abbreviating variable names
describe, fullnames

Create a dataset containing variable descriptions
describe, replace

Describe contents of mydata.dta without opening the dataset
describe using mydata

Menu

Data > Describe data > Describe data in memory or in a file
Syntax

Describe data in memory

```
describe [varlist] [ , memory_options ]
```

Describe data in file

```
describe [ varlist ] using filename [ , file_options ]
```

<table>
<thead>
<tr>
<th>memory_options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>display only variable names</td>
</tr>
<tr>
<td>short</td>
<td>display only general information</td>
</tr>
<tr>
<td>fullnames</td>
<td>do not abbreviate variable names</td>
</tr>
<tr>
<td>numbers</td>
<td>display variable number along with name</td>
</tr>
<tr>
<td>replace</td>
<td>make dataset, not written report, of description</td>
</tr>
<tr>
<td>clear</td>
<td>for use with replace</td>
</tr>
<tr>
<td>varlist</td>
<td>store r(varlist) and r(sortlist) in addition to usual stored results; programmer’s option</td>
</tr>
</tbody>
</table>

varlist does not appear in the dialog box.

<table>
<thead>
<tr>
<th>file_options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>display only general information</td>
</tr>
<tr>
<td>simple</td>
<td>display only variable names</td>
</tr>
<tr>
<td>varlist</td>
<td>store r(varlist) and r(sortlist) in addition to usual stored results; programmer’s option</td>
</tr>
</tbody>
</table>

varlist does not appear in the dialog box.

collect is allowed; see [U] 11.1.10 Prefix commands.

Options to describe data in memory

- **simple** displays only the variable names in a compact format. **simple** may not be combined with other options.
- **short** suppresses the specific information for each variable. Only the general information (number of observations, number of variables, size, and sort order) is displayed.
- **fullnames** specifies that **describe** display the full names of the variables. The default is to present an abbreviation when the variable name is longer than 15 characters. **describe using** always shows the full names of the variables, so **fullnames** may not be specified with **describe using**.
- **numbers** specifies that **describe** present the variable number with the variable name. If **numbers** is specified, variable names are abbreviated when the name is longer than eight characters. The **numbers** and **fullnames** options may not be specified together. **numbers** may not be specified with **describe using**.
replace and clear are alternatives to the options above. describe usually produces a written report, and the options above specify what the report is to contain. If you specify replace, however, no report is produced; the data in memory are instead replaced with data containing the information that the report would have presented. Each observation of the new data describes a variable in the original data; see describe, replace below.

clear may be specified only when replace is specified. clear specifies that the data in memory be cleared and replaced with the description information, even if the original data have not been saved to disk.

The following option is available with describe but is not shown in the dialog box:

varlist, an option for programmers, specifies that, in addition to the usual stored results, r(varlist) and r(sortlist) be stored, too. r(varlist) will contain the names of the variables in the dataset. r(sortlist) will contain the names of the variables by which the data are sorted.

**Options to describe data in file**

short suppresses the specific information for each variable. Only the general information (number of observations, number of variables, size, and sort order) is displayed.

simple displays only the variable names in a compact format. simple may not be combined with other options.

The following option is available with describe but is not shown in the dialog box:

varlist, an option for programmers, specifies that, in addition to the usual stored results, r(varlist) and r(sortlist) be stored, too. r(varlist) will contain the names of the variables in the dataset. r(sortlist) will contain the names of the variables by which the data are sorted.

Because Stata/MP and Stata/SE can create truly large datasets, there might be too many variables in a dataset for their names to be stored in r(varlist), given the current maximum length of macros, as determined by set maxvar. Should that occur, describe using will issue the error message "too many variables", r(103).

**Remarks and examples**

Remarks are presented under the following headings:

```
describe
describe, replace
```

**describe**

If describe is typed with no operands, the contents of the dataset currently in memory are described.

The varlist in the describe using syntax differs from standard Stata varlists in two ways. First, you cannot abbreviate variable names; that is, you have to type displacement rather than displ. However, you can use the abbreviation character (~) to indicate abbreviations, for example, displ~. Second, you may not refer to a range of variables; specifying price-trunk is considered an error.
Example 1

The basic description includes some general information on the number of variables and observations, along with a description of every variable in the dataset:

```
use https://www.stata-press.com/data/r17/states
(State data)
describe, numbers
Contains data from https://www.stata-press.com/data/r17/states.dta
Observations: 50 State data
Variables: 5 3 Jan 2020 15:17
(_dta has notes)
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. state</td>
<td>str8</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. region</td>
<td>int</td>
<td>%8.0g</td>
<td>reg</td>
<td>Census Region</td>
</tr>
<tr>
<td>3. median~e</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td>Median Age</td>
</tr>
<tr>
<td>4. marria~e</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td>Marriages per 100,000</td>
</tr>
<tr>
<td>5. divorc~e</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td>Divorces per 100,000</td>
</tr>
</tbody>
</table>

Sorted by: region

In this example, the dataset in memory comes from the file `states.dta` and contains 50 observations on 5 variables. The dataset is labeled “State data” and was last modified on January 3, 2018, at 15:17 (3:17 p.m.). The “_dta has notes” message indicates that a note is attached to the dataset; see [U] 12.7 Notes attached to data.

The first variable, `state`, is stored as a `str8` and has a display format of %9s.

The second variable, `region`, is stored as an `int` and has a display format of %8.0g. This variable has associated with it a `value label` called reg, and the variable is labeled Census Region.

The third variable, which is abbreviated `median~e`, is stored as a `float`, has a display format of %9.0g, has no value label, and has a variable label of Median Age. The variables that are abbreviated `marria~e` and `divorc~e` are both stored as `longs` and have display formats of %12.0g. These last two variables are labeled Marriages per 100,000 and Divorces per 100,000, respectively.

The data are sorted by region.

Because we specified the `numbers` option, the variables are numbered; for example, `region` is variable 2 in this dataset.
Example 2

To view the full variable names, we could omit the `numbers` option and specify the `fullnames` option.

```
. describe, fullnames
```

```
Contains data from https://www.stata-press.com/data/r17/states.dta
Observations: 50 State data
Variables: 5 3 Jan 2020 15:17 (_dta has notes)

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str8</td>
<td>%9s</td>
<td></td>
<td>Census Region</td>
</tr>
<tr>
<td>region</td>
<td>int</td>
<td>%8.0g</td>
<td>reg</td>
<td>Median Age</td>
</tr>
<tr>
<td>median_age</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td>Marriages per 100,000</td>
</tr>
<tr>
<td>marriage_rate</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td>Divorces per 100,000</td>
</tr>
<tr>
<td>divorce_rate</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Sorted by: region

Here we did not need to specify the `fullnames` option to see the unabbreviated variable names because the longest variable name is 13 characters. Omitting the `numbers` option results in 15-character variable names being displayed.

Technical note

The `describe` listing above also shows that the size of the dataset is 1,100. If you are curious,

\[ (8 + 2 + 4 + 4 + 4) \times 50 = 1100 \]

The numbers 8, 2, 4, 4, and 4 are the storage requirements for a `str8`, `int`, `float`, `long`, and `long`, respectively; see [U] 12.2.2 Numeric storage types. Fifty is the number of observations in the dataset.

Example 3

If we specify the `short` option, only general information about the data is presented:

```
. describe, short
```

```
Contains data from https://www.stata-press.com/data/r17/states.dta
Observations: 50 State data
Variables: 5 3 Jan 2020 15:17
Sorted by: region
```

If we specify a `varlist`, only the variables in that `varlist` are described.
Example 4

Let’s change datasets. The describe varlist command is particularly useful when combined with the ‘*’ wildcard character. For instance, we can describe all the variables whose names start with pop by typing describe pop*:

```
. use https://www.stata-press.com/data/r17/census
   (1980 Census data by state)
. describe pop*
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>long</td>
<td>%12.0gc</td>
<td>Population</td>
</tr>
<tr>
<td>poplt5</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, &lt; 5 year</td>
</tr>
<tr>
<td>pop5_17</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 5 to 17 years</td>
</tr>
<tr>
<td>pop18p</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 18 and older</td>
</tr>
<tr>
<td>pop65p</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 65 and older</td>
</tr>
<tr>
<td>popurban</td>
<td>long</td>
<td>%12.0gc</td>
<td>Urban population</td>
</tr>
</tbody>
</table>

We can describe the variables state, region, and pop18p by specifying them:

```
. describe state region pop18p
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str14</td>
<td>%-14s</td>
<td>State</td>
</tr>
<tr>
<td>region</td>
<td>int</td>
<td>%-8.0g</td>
<td>cenreg</td>
</tr>
<tr>
<td>pop18p</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 18 and older</td>
</tr>
</tbody>
</table>

Typing describe using filename describes the data stored in filename. If an extension is not specified, .dta is assumed.

Example 5

We can describe the contents of states.dta without disturbing the data that we currently have in memory by typing

```
. describe using https://www.stata-press.com/data/r17/states
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str8</td>
<td>%9s</td>
<td></td>
</tr>
<tr>
<td>region</td>
<td>int</td>
<td>%8.0g</td>
<td>reg</td>
</tr>
<tr>
<td>median_age</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
</tr>
<tr>
<td>marriage_rate</td>
<td>long</td>
<td>%12.0g</td>
<td>Marriages per 100,000</td>
</tr>
<tr>
<td>divorce_rate</td>
<td>long</td>
<td>%12.0g</td>
<td>Divorces per 100,000</td>
</tr>
</tbody>
</table>

Sorted by: region
describe, replace

describe with the replace option is rarely used, although you may sometimes find it convenient.

Think of describe, replace as separate from but related to describe without the replace option. Rather than producing a written report, describe, replace produces a new dataset that contains the same information a written report would. For instance, try the following:

```
. sysuse auto, clear
. describe
(report appears; data in memory unchanged)
. list
(visual proof that data are unchanged)
. describe, replace
(no report appears, but the data in memory are changed!)
. list
(visual proof that data are changed)
```

describe, replace changes the original data in memory into a dataset containing an observation for each variable in the original data. Each observation in the new data describes a variable in the original data. The new variables are

1. position, a variable containing the numeric position of the original variable (1, 2, 3, ...).
2. name, a variable containing the name of the original variable, such as "make", "price", "mpg", ....
3. type, a variable containing the storage type of the original variable, such as "str18", "int", "float", ....
4. isnumeric, a variable equal to 1 if the original variable was numeric and equal to 0 if it was string.
5. format, a variable containing the display format of the original variable, such as "%-18s", "%8.0gc", ....
6. vallab, a variable containing the name of the value label associated with the original variable, if any.
7. varlab, a variable containing the variable label of the original variable, such as "Make and model", "Price", "Mileage (mpg)", ....

In addition, the data contain the following characteristics:

- _dta[_d_filename], the name of the file containing the original data.
- _dta[_d_filedate], the date and time the file was written.
- _dta[_d_N], the number of observations in the original data.
- _dta[_d_sortedby], the variables on which the original data were sorted, if any.
describe — Describe data in memory or in file

Stored results

describe stores the following in r():

Scalars
- r(N) number of observations
- r(k) number of variables
- r(width) width of dataset
- r(changed) flag indicating data have changed since last saved

Macros
- r(datalabel) dataset label
- r(varlist) variables in dataset (if varlist specified)
- r(sortlist) variables by which data are sorted (if varlist specified)

describe, replace stores nothing in r().

References


Also see

[D] ds — Compactly list variables with specified properties
[D] varmanage — Manage variable labels, formats, and other properties
[D] cf — Compare two datasets
[D] codebook — Describe data contents
[D] compare — Compare two variables
[D] compress — Compress data in memory
[D] format — Set variables’ output format
[D] label — Manipulate labels
[D] lookfor — Search for string in variable names and labels
[D] notes — Place notes in data
[D] order — Reorder variables in dataset
[D] rename — Rename variable
[SVY] svydescribe — Describe survey data
[U] 6 Managing memory
**Title**

**destring — Convert string variables to numeric variables and vice versa**

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Options for destring</td>
<td>Options for tostring</td>
</tr>
<tr>
<td>Remarks and examples</td>
<td>Acknowledgment</td>
<td>References</td>
</tr>
<tr>
<td>Also see</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

**destring** converts variables in *varlist* from string to numeric. If *varlist* is not specified, **destring** will attempt to convert all variables in the dataset from string to numeric. Characters listed in `ignore()` are removed. Variables in *varlist* that are already numeric will not be changed.

**destring** treats both empty strings “ “ and “.” as indicating `sysmiss` (.) and interprets the strings “.a”, “.b”, ..., “.z” as the extended missing values .a, .b, ..., .z; see [U] 12.2.1 Missing values. **destring** also ignores any leading or trailing spaces so that, for example, “ ” is equivalent to “ ” and “. ” is equivalent to “.”.

**tostring** converts variables in *varlist* from numeric to string. The most compact string format possible is used. Variables in *varlist* that are already string will not be converted.

**Quick start**

Convert *strg1* from string to numeric, and place result in *num1*

```
destring strg1, generate(num1)
```

As above, but ignore the % character in *strg1*

```
destring strg1, generate(num1) ignore(%)
```

As above, but return . for observations with nonnumeric characters

```
destring strg1, generate(num1) force
```

Convert *num2* from numeric to string, and place result in *strg2*

```
tostring num2, generate(strg2)
```

As above, but format with a leading zero and 3 digits after the decimal

```
tostring num2, generate(strg2) format(%09.3f)
```

**Menu**

**destring**

Data > Create or change data > Other variable-transformation commands > Convert variables from string to numeric

**tostring**

Data > Create or change data > Other variable-transformation commands > Convert variables from numeric to string
**Syntax**

Convert string variables to numeric variables

```
destring [varlist], {generate(newvarlist) | replace} [destring_options]
```

Convert numeric variables to string variables

```
tostring varlist, {generate(newvarlist) | replace} [tostring_options]
```

**destring_options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*generate(newvarlist)</td>
<td>generate newvar₁, ..., newvarₖ for each variable in varlist</td>
</tr>
<tr>
<td>*replace</td>
<td>replace string variables in varlist with numeric variables</td>
</tr>
<tr>
<td>ignore(&quot;chars&quot; [, ignoreopts])</td>
<td>remove specified nonnumeric characters, as characters or as bytes, and illegal Unicode characters</td>
</tr>
<tr>
<td>force</td>
<td>convert nonnumeric strings to missing values</td>
</tr>
<tr>
<td>float</td>
<td>generate numeric variables as type float</td>
</tr>
<tr>
<td>percent</td>
<td>convert percent variables to fractional form</td>
</tr>
<tr>
<td>dpcomma</td>
<td>convert variables with commas as decimals to period-decimal format</td>
</tr>
</tbody>
</table>

* Either *generate(newvarlist)* or replace is required.

**tostring_options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*generate(newvarlist)</td>
<td>generate newvar₁, ..., newvarₖ for each variable in varlist</td>
</tr>
<tr>
<td>*replace</td>
<td>replace numeric variables in varlist with string variables</td>
</tr>
<tr>
<td>force</td>
<td>force conversion ignoring information loss</td>
</tr>
<tr>
<td>format(format)</td>
<td>convert using specified format</td>
</tr>
<tr>
<td>usedisplayformat</td>
<td>convert using display format</td>
</tr>
</tbody>
</table>

* Either *generate(newvarlist)* or replace is required.

**Options for destring**

Either *generate()* or *replace* must be specified. With either option, if any string variable contains nonnumeric characters not specified with *ignore()* then no corresponding variable will be generated, nor will that variable be replaced (unless *force* is specified).

*generate(newvarlist)* specifies that a new variable be created for each variable in varlist. newvarlist must contain the same number of new variable names as there are variables in varlist. If varlist is not specified, destring attempts to generate a numeric variable for each variable in the dataset; newvarlist must then contain the same number of new variable names as there are variables in the dataset. Any variable labels or characteristics will be copied to the new variables created.

*replace* specifies that the variables in varlist be converted to numeric variables. If varlist is not specified, destring attempts to convert all variables from string to numeric. Any variable labels or characteristics will be retained.
ignore("chars" [, ignoreopts ]) specifies nonnumeric characters be removed. ignoreopts may be aschars, asbytes, or illegal. The default behavior is to remove characters as characters, which is the same as specifying aschars. asbytes specifies removal of all bytes included in all characters in the ignore string, regardless of whether these bytes form complete Unicode characters. illegal specifies removal of all illegal Unicode characters, which is useful for removing high-ASCII characters. illegal may not be specified with asbytes. If any string variable still contains any nonnumeric or illegal Unicode characters after the ignore string has been removed, no action will take place for that variable unless force is also specified. Note that to Stata the comma is a nonnumeric character; see also the dpcomma option below.

force specifies that any string values containing nonnumeric characters, in addition to any specified with ignore(), be treated as indicating missing numeric values.

float specifies that any new numeric variables be created initially as type float. The default is type double; see [D] Data types. destring attempts automatically to compress each new numeric variable after creation.

percent removes any percent signs found in the values of a variable, and all values of that variable are divided by 100 to convert the values to fractional form. percent by itself implies that the percent sign, “%”, is an argument to ignore(), but the converse is not true.

dpcomma specifies that variables with commas as decimal values should be converted to have periods as decimal values.

Options for tostring

Either generate() or replace must be specified. If converting any numeric variable to string would result in loss of information, no variable will be produced unless force is specified. For more details, see force below.

generate(newvarlist) specifies that a new variable be created for each variable in varlist. newvarlist must contain the same number of new variable names as there are variables in varlist. Any variable labels or characteristics will be copied to the new variables created.

replace specifies that the variables in varlist be converted to string variables. Any variable labels or characteristics will be retained.

force specifies that conversions be forced even if they entail loss of information. Loss of information means one of two circumstances: 1) The result of real(string(varname, "format")) is not equal to varname; that is, the conversion is not reversible without loss of information; 2) replace was specified, but a variable has associated value labels. In circumstance 1, it is usually best to specify usedisplayformat or format(). In circumstance 2, value labels will be ignored in a forced conversion. decode (see [D] encode) is the standard way to generate a string variable based on value labels.

format(format) specifies that a numeric format be used as an argument to the string() function, which controls the conversion of the numeric variable to string. For example, a format of %7.2f specifies that numbers are to be rounded to two decimal places before conversion to string. See Remarks and examples below and [FN] String functions and [D] format. format() cannot be specified with usedisplayformat.

usedisplayformat specifies that the current display format be used for each variable. For example, this option could be useful when using U.S. Social Security numbers or daily or other dates with some %d or %t format assigned. usedisplayformat cannot be specified with format().
Remarks and examples

Remarks are presented under the following headings:

destring
tostring
Saved characteristics
Video example

destring

Example 1

We read in a dataset, but somehow all the variables were created as strings. The variables contain no nonnumeric characters, and we want to convert them all from string to numeric data types.

```
use https://www.stata-press.com/data/r17/destring1
describe
Contains data from https://www.stata-press.com/data/r17/destring1.dta
Observations: 10
Variables: 5 3 Mar 2020 10:15

Variable Storage Display Value
name type format label Variable label
id str3 %9s
num str3 %9s
code str4 %9s
total str5 %9s
income str5 %9s

Sorted by:
describe
```

```
list
+------------+------------+------------+------------+------------+
|          |          |          |          |          |
| id       | num      | code     | total    | income    |
| 111      | 243      | 1234     | 543      | 23423     |
| 111      | 123      | 2345     | 67854    | 12654     |
| 111      | 234      | 3456     | 345      | 43658     |
| 222      | 345      | 4567     | 57       | 23546     |
| 333      | 456      | 5678     | 23       | 21432     |
| 333      | 567      | 6789     | 23465    | 12987     |
| 333      | 678      | 7890     | 65       | 9823      |
| 444      | 789      | 8976     | 23       | 32980     |
| 444      | 901      | 7654     | 23       | 18565     |
| 555      | 890      | 6543     | 423      | 19234     |

. destring, replace
id: all characters numeric; replaced as int
num: all characters numeric; replaced as int
code: all characters numeric; replaced as int
total: all characters numeric; replaced as long
income: all characters numeric; replaced as long
```
. describe
Contains data from https://www.stata-press.com/data/r17/destring1.dta
Observations: 10
Variables: 5
3 Mar 2020 10:15

| Variable Storage Display Value |
| name type format label Variable label |
| id int %10.0g |
| num int %10.0g |
| code int %10.0g |
| total long %10.0g |
| income long %10.0g |

Sorted by:
Note: Dataset has changed since last saved.

. list

<table>
<thead>
<tr>
<th>id</th>
<th>num</th>
<th>code</th>
<th>total</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>111</td>
<td>243</td>
<td>1234</td>
<td>543</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>123</td>
<td>2345</td>
<td>67854</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>234</td>
<td>3456</td>
<td>345</td>
</tr>
<tr>
<td>4</td>
<td>222</td>
<td>345</td>
<td>4567</td>
<td>57</td>
</tr>
<tr>
<td>5</td>
<td>333</td>
<td>456</td>
<td>5678</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>333</td>
<td>567</td>
<td>6789</td>
<td>23465</td>
</tr>
<tr>
<td>7</td>
<td>333</td>
<td>678</td>
<td>7890</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>444</td>
<td>789</td>
<td>8976</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>444</td>
<td>901</td>
<td>7654</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>555</td>
<td>890</td>
<td>6543</td>
<td>423</td>
</tr>
</tbody>
</table>

Example 2

Our dataset contains the variable date, which was accidentally recorded as a string because of spaces after the year and month. We want to remove the spaces. destring will convert it to numeric and remove the spaces.

. use https://www.stata-press.com/data/r17/destring2, clear
. describe date

| Variable Storage Display Value |
| name type format label Variable label |
| date str14 %10s |
Our dataset contains the variables date, price, and percent. These variables were accidentally read into Stata as string variables because they contain spaces, dollar signs, commas, and percent signs. We want to remove all of these characters and create new variables for date, price, and percent containing numeric values. After removing the percent sign, we want to convert the percent variable to decimal form.
. use https://www.stata-press.com/data/r17/destring2, clear
. describe
Contains data from https://www.stata-press.com/data/r17/destring2.dta
Observations: 10
Variables: 3
3 Mar 2020 22:50

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>str14</td>
<td>%10s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>str11</td>
<td>%11s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>percent</td>
<td>str3</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

. list

<table>
<thead>
<tr>
<th>date</th>
<th>price</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999 12 10</td>
<td>$2,343.68</td>
<td>34%</td>
</tr>
<tr>
<td>2000 07 08</td>
<td>$7,233.44</td>
<td>86%</td>
</tr>
<tr>
<td>1997 03 02</td>
<td>$12,442.89</td>
<td>12%</td>
</tr>
<tr>
<td>1999 09 00</td>
<td>$233,325.31</td>
<td>6%</td>
</tr>
<tr>
<td>1998 10 04</td>
<td>$1,549.23</td>
<td>76%</td>
</tr>
<tr>
<td>2000 03 28</td>
<td>$23,517.03</td>
<td>35%</td>
</tr>
<tr>
<td>2000 08 08</td>
<td>$2.43</td>
<td>69%</td>
</tr>
<tr>
<td>1997 10 20</td>
<td>$9,382.47</td>
<td>32%</td>
</tr>
<tr>
<td>1998 01 16</td>
<td>$289,209.32</td>
<td>45%</td>
</tr>
<tr>
<td>1999 11 12</td>
<td>$8,282.49</td>
<td>1%</td>
</tr>
</tbody>
</table>

. destring date price percent, generate(date2 price2 percent2) ignore("$,%")
> percent
date: character space removed; date2 generated as long
price: characters $, removed; price2 generated as double
percent: character % removed; percent2 generated as double
. describe
Contains data from https://www.stata-press.com/data/r17/destring2.dta
Observations: 10
Variables: 6
3 Mar 2020 22:50

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>str14</td>
<td>%10s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>date2</td>
<td>long</td>
<td>%10.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>str11</td>
<td>%11s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price2</td>
<td>double</td>
<td>%10.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>percent</td>
<td>str3</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>percent2</td>
<td>double</td>
<td>%10.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.
destring — Convert string variables to numeric variables and vice versa

list
date date2 price price2 percent percent2
1. 1999 12 10 19991210 $2,343.68 2343.68 34% .34
2. 2000 07 08 20000708 $7,233.44 7233.44 86% .86
3. 1997 03 02 19970302 $12,442.89 12442.89 12% .12
4. 1999 09 00 19990900 $233,325.31 233325.31 6% .06
5. 1998 10 04 19981004 $1,549.23 1549.23 76% .76
6. 2000 03 28 20000328 $23,517.03 23517.03 35% .35
7. 2000 08 08 20000808 $2.43 2.43 69% .69
8. 1997 10 20 19971020 $9,382.47 9382.47 32% .32
9. 1998 01 16 19980116 $289,209.32 289209.32 45% .45
10. 1999 11 12 19991112 $8,282.49 8282.49 1% .01
tostring

Conversion of numeric data to string equivalents can be problematic. Stata, like most software, holds numeric data to finite precision and in binary form. See the discussion in [U] 13.12 Precision and problems therein. If no format() is specified, tostring uses the format %12.0g. This format is, in particular, sufficient to convert integers held as bytes, ints, or longs to string equivalent without loss of precision.

However, users will often need to specify a format themselves, especially when the numeric data have fractional parts and for some reason a conversion to string is required.

Example 4

Our dataset contains a string month variable and numeric year and day variables. We want to convert the three variables to a %td date.

. use https://www.stata-press.com/data/r17/tostring, clear
. list

id month day year
1. 123456789 jan 10 2001
2. 123456710 mar 20 2001
3. 123456711 may 30 2001
4. 123456712 jun 9 2001
5. 123456713 oct 17 2001
6. 123456714 nov 15 2001
7. 123456715 dec 28 2001
8. 123456716 apr 29 2001
9. 123456717 mar 11 2001
10. 123456718 jul 3 2001

tostring year day, replace
tostring year day, replace
year was float now str4
day was float now str2
generate date = month + "/" + day + "/" + year
generate edate = date(date, "MDY")
.format edate %td
Saved characteristics

Each time the `destring` or `tostring` commands are issued, an entry is made in the characteristics list of each converted variable. You can type `char list` to view these characteristics.

After example 3, we could use `char list` to find out what characters were removed by the `destring` command.

```
. char list
  . date2[destring]: Character removed was: space
  . date2[destring_cmd]: destring date price percent, generate(date2 pri..
  . price2[destring]: Characters removed were: $ ,
  . price2[destring_cmd]: destring date price percent, generate(date2 pri..
  . percent2[destring]: Character removed was: %
  . percent2[destring_cmd]: destring date price percent, generate(date2 pri..
```

Video example

How to convert a string variable to a numeric variable

Acknowledgment

`destring` and `tostring` were originally written by Nicholas J. Cox of the Department of Geography at Durham University, UK, who is coeditor of the *Stata Journal* and author of *Speaking Stata Graphics*.

References


Also see

[D] egen — Extensions to generate
[D] encode — Encode string into numeric and vice versa
[D] generate — Create or change contents of variable
[D] split — Split string variables into parts

[FN] String functions
**Title**

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Syntax</th>
<th>Option</th>
<th>Remarks and examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dir — Display filenames</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

dir and ls—they work the same way—list the names of files in the specified directory; the names of the commands come from names popular on Unix and Windows computers.

**Quick start**

List the names of all files in the current directory using Stata for Windows
```
dir
```
As above, but for Mac or Unix
```
ls
```
List Stata datasets in the current directory using Stata for Windows
```
dir *.dta
```
As above, but for Mac or Unix
```
ls *.dta
```
List dataset name for all .dta files in the C:\ directory using Stata for Windows
```
dir C:\*.dta
```
List dataset name for all .dta files in the home directory using Stata for Windows
```
dir ~\*.dta
```
As above, but for Mac or Unix
```
ls ~/*.dta
```

**Syntax**

```
{dir|ls} ["][filespec]["] [, wide]
```

filespec is any valid Mac, Unix, or Windows file path or file specification (see [U] 11.6 Filenaming conventions) and may include “*” to indicate any string of characters.

Note: Double quotes must be used to enclose filespec if the name contains spaces.
**Option**

*wide* under Mac and Windows produces an effect similar to specifying /w with the DOS *dir* command—it compresses the resulting listing by placing more than one filename on a line. Under Unix, it produces the same effect as typing *ls* -F -C. Without the *wide* option, *ls* is equivalent to typing *ls* -F -l.

**Remarks and examples**

Mac and Unix: The only difference between the Stata and Unix *ls* commands is that piping through the more(1) or pg(1) filter is unnecessary—Stata always pauses when the screen is full.

Windows: Other than minor differences in presentation format, there is only one difference between the Stata and DOS *dir* commands: the DOS /P option is unnecessary, because Stata always pauses when the screen is full.

> **Example 1**

If you use Stata for Windows and wish to obtain a list of all your Stata-format data files, type

```
   . dir *.dta
   3.9k 7/07/15 13:51 auto.dta
   0.6k 8/04/15 10:40 cancer.dta
   3.5k 7/06/08 17:06 census.dta
   3.4k 1/25/08 9:20 hsgn.dta
   0.3k 1/26/08 16:54 kva.dta
   0.7k 4/27/11 11:39 sysage.dta
   0.5k 5/09/07 2:56 systolic.dta
   10.3k 7/13/08 8:37 Household Survey.dta
```

You could also include the *wide* option:

```
   . dir *.dta, wide
   3.9k auto.dta 0.6k cancer.dta 3.5k census.dta
   3.4k hsgn.dta 0.3k kva.dta 0.7k sysage.dta
   0.5k systolic.dta 10.3k Household Survey.dta
```

Unix users will find it more natural to type

```
   . ls *.dta
   -rw-r----- 1 roger 2868 Mar  4 15:34 highway.dta
   -rw-r----- 1 roger 941 Apr  5 09:43 hoyle.dta
   -rw-r----- 1 roger 19312 May 14 10:36 p1.dta
   -rw-r----- 1 roger 11838 Apr 11 13:26 p2.dta
```

but they could type *dir* if they preferred. Mac users may also type either command.

```
   . dir *.dta
   -rw-r----- 1 roger 2868 Mar  4 15:34 highway.dta
   -rw-r----- 1 roger 941 Apr  5 09:43 hoyle.dta
   -rw-r----- 1 roger 19312 May 14 10:36 p1.dta
   -rw-r----- 1 roger 11838 Apr 11 13:26 p2.dta
```
Technical note

There is a macro function named `dir` that allows you to obtain a list of files in a macro for later processing. See *Macro functions for filenames and file paths* in [P] macro.

Also see

[D] `cd` — Change directory
[D] `copy` — Copy file from disk or URL
[D] `erase` — Erase a disk file
[D] `mkdir` — Create directory
[D] `rmdir` — Remove directory
[D] `shell` — Temporarily invoke operating system
[D] `type` — Display contents of a file
[U] 11.6 Filenaming conventions
drawnorm — Draw sample from multivariate normal distribution

Description
drawnorm draws a sample from a multivariate normal distribution with desired means and covariance matrix. The default is orthogonal data with mean 0 and variance 1. The covariance matrix may be singular. The values generated are a function of the current random-number seed or the number specified with set seed(); see [R] set seed.

Quick start
Generate independent variables x and y, where x has mean 2 and standard deviation 0.5 and y has mean 3 and standard deviation 1
drawnorm x y, means(2,3) sds(.5,1)

As above, but create dataset of 1,000 observations on x and y with means stored in vector m and standard deviations stored in vector sd
drawnorm x y, means(m) sds(sd) n(1000)

As above, and set the seed for the random-number generator to reproduce results
drawnorm x y, means(m) sds(sd) n(1000) seed(81625)

Sample from bivariate standard normal distribution with covariance between x and y of 0.5 stored in variance–covariance matrix C
matrix C = (1, .5 \ .5, 1)
drawnorm x y, cov(C)

Sample from a trivariate standard normal distribution with correlation between x and y of 0.4, x and z of 0.3, and y and z of 0.6 stored in correlation matrix C
matrix C = (1, .4, .3 \ .4, 1, .6 \ .3, .6, 1)
drawnorm x y z, corr(C)

Same as above, but avoid typing full matrix by specifying correlations in vector v treated as a lower triangular matrix
matrix v = (1, .4, 1, .3, .6, 1)
drawnorm x y z, corr(v) cstorage(lower)

Menu
Data > Create or change data > Other variable-creation commands > Draw sample from normal distribution
drawnorm — Draw sample from multivariate normal distribution

Syntax

drawnorm newvarlist [ , options ]

options Description

Main

clear replace the current dataset
double generate variable type as double; default is float
n(#) generate # observations; default is current number
sds(vector) standard deviations of generated variables
corr(matrix | vector) correlation matrix
cov(matrix | vector) covariance matrix
cstorage(full) store correlation/covariance structure as a symmetric \( k \times k \) matrix
cstorage(lower) store correlation/covariance structure as a lower triangular matrix
cstorage(upper) store correlation/covariance structure as an upper triangular matrix
forcepsd force the covariance/correlation matrix to be positive semidefinite
means(vector) means of generated variables; default is means(0)

Options

seed(#) seed for random-number generator

Options

clear specifies that the dataset in memory be replaced, even though the current dataset has not been saved on disk.
double specifies that the new variables be stored as Stata doubles, meaning 8-byte reals. If double is not specified, variables are stored as floats, meaning 4-byte reals. See [D] Data types.
n(#) specifies the number of observations to be generated. The default is the current number of observations. If n(#) is not specified or is the same as the current number of observations, drawnorm adds the new variables to the existing dataset; otherwise, drawnorm replaces the data in memory.
sds(vector) specifies the standard deviations of the generated variables. sds() may not be specified with cov().
corr(matrix | vector) specifies the correlation matrix. If neither corr() nor cov() is specified, the default is orthogonal data.
cov(matrix | vector) specifies the covariance matrix. If neither cov() nor corr() is specified, the default is orthogonal data.
cstorage(full | lower | upper) specifies the storage mode for the correlation or covariance structure in corr() or cov(). The following storage modes are supported:

full specifies that the correlation or covariance structure is stored (recorded) as a symmetric \( k \times k \) matrix.
lower specifies that the correlation or covariance structure is recorded as a lower triangular matrix. With \( k \) variables, the matrix should have \( k(k + 1)/2 \) elements in the following order:

\[
C_{11} \ C_{21} \ C_{22} \ C_{31} \ C_{32} \ C_{33} \ldots \ C_{k1} \ C_{k2} \ldots \ C_{kk}
\]

upper specifies that the correlation or covariance structure is recorded as an upper triangular matrix. With \( k \) variables, the matrix should have \( k(k + 1)/2 \) elements in the following order:

\[
C_{11} \ C_{12} \ C_{13} \ldots \ C_{1k} \ C_{22} \ C_{23} \ldots C_{2k} \ldots C_{(k-1)k-1} \ C_{(k-1)k} \ C_{kk}
\]

Specifying cstorage(full) is optional if the matrix is square. cstorage(lower) or cstorage(upper) is required for the vectorized storage methods. See Example 2: Storage modes for correlation and covariance matrices.

forcepsd modifies the matrix \( C \) to be positive semidefinite (psd), and so be a proper covariance matrix. If \( C \) is not positive semidefinite, it will have negative eigenvalues. By setting negative eigenvalues to 0 and reconstructing, we obtain the least-squares positive-semidefinite approximation to \( C \). This approximation is a singular covariance matrix.

means(vector) specifies the means of the generated variables. The default is means(0).

Options

seed(\#) specifies the initial value of the random-number seed used by the runiform() function. The default is the current random-number seed. Specifying seed(\#) is the same as typing set seed \# before issuing the drawnorm command.

Remarks and examples

Example 1

Suppose that we want to draw a sample of 1,000 observations from a normal distribution \( N(M, V) \), where \( M \) is the mean matrix and \( V \) is the covariance matrix:

```
. matrix M = 5, -6, 0.5
. matrix V = (9, 5, 2 \ 5 , 4 , 1 \ 2, 1, 1)
. matrix list M
M[1,3]
c1 c2 c3
r1 5 -6 .5
. matrix list V
symmetric V[3,3]
c1 c2 c3
r1 9
r2 5 4
r3 2 1 1
. drawnorm x y z, n(1000) cov(V) means(M) (obs 1,000)
```
. summarize
Variable | Obs | Mean  | Std. dev. | Min  | Max
---------|-----|-------|-----------|------|------
x        | 1,000 | 4.987428 | 3.013015 | -5.013422 | 16.55432
y        | 1,000 | -6.029116 | 1.998648 | -13.16645 | .0143275
z        | 1,000 | .487068 | 1.056163 | -2.92435 | 3.833934

. correlate, cov
(obs=1,000)

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
</table>
x    | 9.07826 |      |       |
y    | 4.96528 | 3.99459 |       |
z    | 2.18285 | 1.03964 | 1.11548 |

Technical note

The values generated by `drawnorm` are a function of the current random-number seed. To reproduce
the same dataset each time `drawnorm` is run with the same setup, specify the same seed number in
the `seed()` option.

Example 2: Storage modes for correlation and covariance matrices

The three storage modes for specifying the correlation or covariance matrix in `corr2data` and
`drawnorm` can be illustrated with a correlation structure, $C$, of 4 variables. In full storage mode, this
structure can be entered as a $4 \times 4$ Stata matrix:

```stata
. matrix C = ( 1.0000, 0.3232, 0.1112, 0.0066 \ ///
              0.3232, 1.0000, 0.6608, -0.1572 \ ///
              0.1112, 0.6608, 1.0000, -0.1480 \ ///
              0.0066, -0.1572, -0.1480, 1.0000 )
```

Elements within a row are separated by commas, and rows are separated by a backslash, \. We
use the input continuation operator /// for convenient multiline input; see [P] comments. In this
storage mode, we probably want to set the row and column names to the variable names:

```stata
. matrix rownames C = price trunk headroom rep78
. matrix colnames C = price trunk headroom rep78
```

This correlation structure can be entered more conveniently in one of the two vectorized storage
modes. In these modes, we enter the lower triangle or the upper triangle of $C$ in rowwise order; these
two storage modes differ only in the order in which the $k(k+1)/2$ matrix elements are recorded.
The lower storage mode for $C$ comprises a vector with $4(4+1)/2 = 10$ elements, that is, a $1 \times 10$
or $10 \times 1$ Stata matrix, with one row or column,

```stata
. matrix C = ( 1.0000, ///
              0.3232, 1.0000, ///
              0.1112, 0.6608, 1.0000, ///
              0.0066, -0.1572, -0.1480, 1.0000 )
```

or more compactly as

```stata
. matrix C = ( 1, 0.3232, 1, 0.1112, 0.6608, 1, 0.0066, -0.1572, -0.1480, 1 )
```
C may also be entered in upper storage mode as a vector with $4(4 + 1)/2 = 10$ elements, that is, a $1 \times 10$ or $10 \times 1$ Stata matrix,

\[
\begin{array}{c}
\text{. matrix } C = \begin{pmatrix}
1.0000, & 0.3232, & 0.1112, & 0.0066, & \ldots \\
1.0000, & 0.6608, & -0.1572, & \ldots \\
1.0000, & -0.1480, & \ldots \\
1.0000 & \end{pmatrix}
\end{array}
\]

or more compactly as

\[
\begin{array}{c}
\text{. matrix } C = \begin{pmatrix}
1, & 0.3232, & 0.1112, & 0.0066, & 1, & 0.6608, & -0.1572, & 1, & -0.1480, & 1
\end{pmatrix}
\end{array}
\]

**Methods and formulas**

Results are asymptotic. The more observations generated, the closer the correlation matrix of the dataset is to the desired correlation structure.

Let $V = A'A$ be the desired covariance matrix and $M$ be the desired mean matrix. We first generate $X$, such that $X \sim N(0, I)$. Let $Y = A'X + M$, then $Y \sim N(M, V)$.

**References**


**Also see**

[D] corr2data — Create dataset with specified correlation structure

[R] set seed — Specify random-number seed and state
**drop — Drop variables or observations**

**Description**

`drop` eliminates variables or observations from the data in memory.

`keep` works the same way as `drop`, except that you specify the variables or observations to be kept rather than the variables or observations to be deleted.

Warning: `drop` and `keep` are not reversible. Once you have eliminated observations, you cannot read them back in again. You would need to go back to the original dataset and read it in again. Instead of applying `drop` or `keep` for a subset analysis, consider using `if` or `in` to select subsets temporarily. This is usually the best strategy. Alternatively, applying `preserve` followed in due course by `restore` may be a good approach. You can also use `frame put` to place a subset of variables or observations from the current dataset into another frame; see [D] `frame put`.

**Quick start**

Remove `v1`, `v2`, and `v3` from memory

```
drop v1 v2 v3
```

Remove all variables whose name begins with `code` from memory

```
drop code*
```

Remove observations where `v1` is equal to 99

```
drop if v1==99
```

Also drop observations where `v1` equals 88 or `v2` is missing

```
drop if inlist(v1,88,99) | missing(v2)
```

Keep observations where `v3` is not missing

```
keep if !missing(v3)
```

Keep the first observation from each cluster identified by `cvar`  
by `cvar`:  
```
keep if _n==1
```

**Menu**

**Drop or keep variables**

Data > Variables Manager

**Drop or keep observations**

Data > Create or change data > Drop or keep observations
Syntax

Drop variables

drop varlist

Drop observations

drop if exp

Drop a range of observations

drop in range [if exp]

Keep variables

keep varlist

Keep observations that satisfy specified condition

keep if exp

Keep a range of observations

keep in range [if exp]

by and collect are allowed with the second syntax of `drop` and the second syntax of `keep`; see [U] 11.1.10 Prefix commands.

Remarks and examples

You can clear the entire dataset by typing `drop _all` without affecting value labels, macros, and programs. (Also see [U] 12.6 Dataset, variable, and value labels, [U] 18.3 Macros, and [P] program.)
Example 1

We will systematically eliminate data until, at the end, no data are left in memory. We begin by describing the data:

```
. use https://www.stata-press.com/data/r17/census11
(1980 Census data by state)
. describe
Contains data from https://www.stata-press.com/data/r17/census11.dta
Observations: 50  1980 Census data by state
Variables: 15  2 Dec 2020 14:31

Variable name Storage Display Value Variable label
     type format label
--- ------- ------- ------- ----------------------
state  str13  %-13s  State
state2 str2   %-2s  Two-letter state abbreviation
region byte  %-8.0g cenreg Census region
pop    long   %12.0gc Population
poplt5 long   %12.0gc Pop, < 5 year
pop5_17 long   %12.0gc Pop, 5 to 17 years
pop18p long   %12.0gc Pop, 18 and older
pop65p long   %12.0gc Pop, 65 and older
popurban long   %12.0gc Urban population
medage float  %9.2f Median age
depth  long   %12.0gc Number of deaths
marriage long   %12.0gc Number of marriages
divorce long   %12.0gc Number of divorces
mrgrate float  %9.0g Marriage rate
dvcrate float  %9.0g Divorce rate
```

Sorted by: region

We can eliminate all the variables with names that begin with pop by typing `drop pop*`:
. drop pop*
. describe
Contains data from https://www.stata-press.com/data/r17/census11.dta
Observations: 50 1980 Census data by state
Variables: 9 2 Dec 2020 14:31

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str13</td>
<td>%-13s</td>
<td>State</td>
</tr>
<tr>
<td>state2</td>
<td>str2</td>
<td>%-2s</td>
<td>Two-letter state abbreviation</td>
</tr>
<tr>
<td>region</td>
<td>byte</td>
<td>%-8.0g cenreg</td>
<td>Census region</td>
</tr>
<tr>
<td>medage</td>
<td>float</td>
<td>%9.2f</td>
<td>Median age</td>
</tr>
<tr>
<td>death</td>
<td>long</td>
<td>%12.0gc</td>
<td>Number of deaths</td>
</tr>
<tr>
<td>marriage</td>
<td>long</td>
<td>%12.0gc</td>
<td>Number of marriages</td>
</tr>
<tr>
<td>divorce</td>
<td>long</td>
<td>%12.0gc</td>
<td>Number of divorces</td>
</tr>
<tr>
<td>mrgrate</td>
<td>float</td>
<td>%9.0g</td>
<td>Marriage rate</td>
</tr>
<tr>
<td>dvcrate</td>
<td>float</td>
<td>%9.0g</td>
<td>Divorce rate</td>
</tr>
</tbody>
</table>

Sorted by: region
Note: Dataset has changed since last saved.

Let's eliminate more variables and then eliminate observations:

. drop marriage divorce mrgrate dvcrate
. describe
Contains data from https://www.stata-press.com/data/r17/census11.dta
Observations: 50 1980 Census data by state
Variables: 5 2 Dec 2020 14:31

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str13</td>
<td>%-13s</td>
<td>State</td>
</tr>
<tr>
<td>state2</td>
<td>str2</td>
<td>%-2s</td>
<td>Two-letter state abbreviation</td>
</tr>
<tr>
<td>region</td>
<td>byte</td>
<td>%-8.0g cenreg</td>
<td>Census region</td>
</tr>
<tr>
<td>medage</td>
<td>float</td>
<td>%9.2f</td>
<td>Median age</td>
</tr>
<tr>
<td>death</td>
<td>long</td>
<td>%12.0gc</td>
<td>Number of deaths</td>
</tr>
</tbody>
</table>

Sorted by: region
Note: Dataset has changed since last saved.

Next we will drop any observation for which medage is greater than 32.

. drop if medage > 32
(3 observations deleted)

Let's drop the first observation in each region:

. by region: drop if _n==1
(4 observations deleted)

Now we drop all but the last observation in each region:

. by region: drop if _n!=_N
(39 observations deleted)

Let's now drop the first 2 observations in our dataset:

. drop in 1/2
(2 observations deleted)
Finally, let’s get rid of everything:

```
. drop _all
. describe
```

Contains data
Observations: 0
Variables: 0
Sorted by:

Typing `keep in 10/1` is the same as typing `drop in 1/9`.

Typing `keep if x==3` is the same as typing `drop if x !=3`.

`keep` is especially useful for keeping a few variables from a large dataset. Typing `keep myvar1 myvar2` is the same as typing `drop` followed by all the variables in the dataset except `myvar1` and `myvar2`.

⚠️ Technical note

In addition to dropping variables and observations, `drop _all` removes any business calendars; see `[D] Datetime business calendars`.

### Stored results

drop and keep store the following in `r()`:

Scalars

- `r(N_drop)` number of observations dropped
- `r(k_drop)` number of variables dropped

### Also see

[D] clear — Clear memory

[D] frame put — Copy selected variables or observations to a new frame

[D] varmanage — Manage variable labels, formats, and other properties

[U] 11 Language syntax

[U] 13 Functions and expressions
**Description**

ds lists variable names of the dataset currently in memory in a compact or detailed format, and lets you specify subsets of variables to be listed, either by name or by properties (for example, the variables are numeric). In addition, ds leaves behind in `r(varlist)` the names of variables selected so that you can use them in a subsequent command.

ds, typed without arguments, lists all variable names of the dataset currently in memory in a compact form.

**Quick start**

List variables in alphabetical order

ds, alpha

List all string variables

ds, has(type string)

List all numeric variables

ds, has(type numeric)

As above, but exclude date-formatted variables

ds, not(format %td* type string)

List all variables whose label includes the phrase “my text” regardless of case

ds, has(varlabel "*my text*") insensitive

**Menu**

Data > Describe data > Compactly list variable names
Compactly list variables with specified properties

**Syntax**

*Simple syntax*

\[ ds [, \text{alpha}] \]

*Advanced syntax*

\[ ds [\text{varlist}] [, \text{options}] \]

<table>
<thead>
<tr>
<th><strong>options</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>list variables not specified in \text{varlist}</td>
</tr>
<tr>
<td>alpha</td>
<td>list variables in alphabetical order</td>
</tr>
<tr>
<td>detail</td>
<td>display additional details</td>
</tr>
<tr>
<td>\text{varwidth(#)}</td>
<td>display width for variable names; default is \text{varwidth(12)}</td>
</tr>
<tr>
<td>\text{skip(#)}</td>
<td>gap between variables; default is \text{skip(2)}</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
</tr>
<tr>
<td>\text{has(spec)}</td>
<td>describe subset that matches \text{spec}</td>
</tr>
<tr>
<td>\text{not(spec)}</td>
<td>describe subset that does not match \text{spec}</td>
</tr>
<tr>
<td>\text{insensitive}</td>
<td>perform case-insensitive pattern matching</td>
</tr>
<tr>
<td>\text{indent(#)}</td>
<td>indent output; seldom used</td>
</tr>
</tbody>
</table>

\text{collect} is allowed; see [U] 11.1.10 Prefix commands.

\text{insensitive} and \text{indent(#)} are not shown in the dialog box.

<table>
<thead>
<tr>
<th><strong>spec</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{type typelist}</td>
<td>specified types</td>
</tr>
<tr>
<td>\text{format patternlist}</td>
<td>display format matching \text{patternlist}</td>
</tr>
<tr>
<td>\text{varlabel [patternlist]}</td>
<td>variable label or variable label matching \text{patternlist}</td>
</tr>
<tr>
<td>\text{char [patternlist]}</td>
<td>characteristic or characteristic matching \text{patternlist}</td>
</tr>
<tr>
<td>\text{vallabel [patternlist]}</td>
<td>value label or value label matching \text{patternlist}</td>
</tr>
</tbody>
</table>
**typelist** used in `has(type typelist)` and `not(type typelist)` is a list of one or more types, each of which may be numeric, string, str#, strL, byte, int, long, float, or double, or may be a **numlist** such as 1/8 to mean “str1 str2 ... str8”. Examples include

```
has(type int)   is of type int
has(type byte int long) is of integer type
not(type int)   is not of type int
not(type byte int long) is not of the integer types
has(type numeric) is a numeric variable
not(type string) is not a string (str# or strL) variable (same as above)
has(type 1/40)   is str1, str2, ..., str40
has(type str#)   is str1, str2, ..., str2045 but not strL
has(type strL)   is of type strL but not str#
has(type numeric 1/2) is numeric or str1 or str2
```

**patternlist** used in, for instance, `has(format patternlist)`, is a list of one or more **patterns**. A pattern is the expected text with the addition of the characters * and ?. * indicates 0 or more characters go here, and ? indicates exactly 1 character goes here. Examples include

```
has(format *f)   format is %#.#f
has(format %t*)   has time or date format
has(format %-*s)   is a left-justified string
has(varl *weight*) variable label includes word weight
has(varl *weight* *Weight*) variable label has weight or Weight
```

To match a phrase, enclose the phrase in quotes.

```
has(varl "*some phrase*)) variable label has some phrase
```

If instead you used `has(varl *some phrase*)`, then only variables having labels ending in some or starting with phrase would be listed.

### Options

<table>
<thead>
<tr>
<th>Main</th>
</tr>
</thead>
</table>
| not specifies that the variables in **varlist** not be listed. For instance, **ds pop***, not specifies that all variables not starting with the letters pop be listed. The default is to list all the variables in the dataset or, if **varlist** is specified, the variables specified.

**alpha** specifies that the variables be listed in alphabetical order. If the variable contains Unicode characters other than plain ASCII, the sort order is determined strictly by the underlying byte order. See \[U\] 12.4.2.5 Sorting strings containing Unicode characters.

detail specifies that detailed output identical to that of describe be produced. If detail is specified, **varwidth()**, **skip()**, and **indent()** are ignored.

**varwidth(#)** specifies the display width of the variable names; the default is **varwidth(12)**.

**skip(#)** specifies the number of spaces between variable names, where all variable names are assumed to be the length of the longest variable name; the default is **skip(2)**.
Compactly list variables with specified properties

`ds` and `not(spec)` select from the dataset (or from `varlist`) the subset of variables that meet or fail the specification `spec`. Selection may be made on the basis of storage type, variable label, value label, display format, or characteristics. Only one `not`, `has()`, or `not()` option may be specified.

`has(type string)` selects all string variables. Typing `ds, has(type string)` would list all string variables in the dataset, and typing `ds pop*, has(type string)` would list all string variables whose names begin with the letters `pop`.

`has(varlabel)` selects variables with defined variable labels. `has(varlabel *weight*)` selects variables with variable labels including the word “weight”. `not(varlabel)` would select all variables with no variable labels.

`has(vallabel)` selects variables with defined value labels. `has(vallabel yesno)` selects variables whose value label is `yesno`. `has(vallabel *no)` selects variables whose value label ends in the letters `no`.

`has(format patternlist)` specifies variables whose format matches any of the patterns in `patternlist`. `has(format *f)` would select all variables with formats ending in `f`, which presumably would be all `%#, #f, %0#, #f, and %-, #f formats. `has(format *f *fc)` would select all variables with formats ending in `f` or `fc. `not(format %t* %-t*)` would select all variables except those with date or time-series formats.

`has(char)` selects all variables with defined characteristics. `has(char problem)` selects all variables with a characteristic named `problem`.

The following options are available with `ds` but are not shown in the dialog box:

- `insensitive` specifies that the matching of the pattern in `has()` and `not()` be case insensitive. Note that the case insensitivity applies only to ASCII characters.
- `indent(#)` specifies the amount the lines are indented.

**Remarks and examples**

If `ds` is typed without any operands, then a compact list of the variable names for the data currently in memory is displayed.

### Example 1

`ds` can be especially useful if you have a dataset with over 1,000 variables, but you may find it convenient even if you have considerably fewer variables.

```
. use https://www.stata-press.com/data/r17/educ3
   (ccdb46, 52-54)
   . ds
fips   popcol  medhhinc  tlf   emp  clfbls  z
  crimes  perhspls  medfinc  clf  empmanuf  clfuebls  adjinc
  pcrimes  perclpls  state  clffem  emptrade  famnw  perman
  crimrate  prochls  division  clfue  empserv  fam2w  pertrade
  pop25pls  medage  region  empgovt  osigind  famwsamp  perserv
  pophspls  perwhite  dc  empself  osigindp  pop18pls  perother
```
Example 2

You might wonder why you would ever specify a *varlist* with this command. Remember that a *varlist* understands the ‘*’ abbreviation character and the ‘-’ dash notation; see [U] 11.4 varname and varlists.

\[ . \text{ds } p* \]
\[ . \text{pcrimes pophspls perhspls prcolhs pop18pls pertrade perother} \]
\[ . \text{pop25pls popcol perclpls perwhite perman perserv} \]
\[ . \text{ds popcol-clfue} \]
\[ . \text{popcol perclpls medage medhhinc state region tlf clffem} \]
\[ . \text{perhspls prcolhs perwhite medfinc division dc clf clfue} \]

Example 3

Because the primary use of ds is to inspect the names of variables, it is sometimes useful to let ds display the variable names in alphabetical order.

\[ . \text{ds, alpha} \]
\[ . \text{adjinc crimes empmanuf famwsamp osigindp perserv pophspls} \]
\[ . \text{clf crimrate empself fips pcrimes pertrade prcolhs} \]
\[ . \text{clfbls dc empserv medage perclpls perwhite region} \]
\[ . \text{clffem division emptrade medfinc perhspls pop18pls state} \]
\[ . \text{clfue emp fam2w medhhinc perman pop25pls tlf} \]
\[ . \text{clfuebls empgovt famnw osigind perother popcol z} \]

Stored results

ds stores the following in r():

Macros

\[ r(varlist) \] the varlist of found variables

Acknowledgments

ds was originally written by StataCorp. It was redesigned and rewritten by Nicholas J. Cox of the Department of Geography at Durham University, UK, who is coeditor of the *Stata Journal* and author of *Speaking Stata Graphics*. The purpose was to include the selection options *not*, *has()*, and *not()*; to produce better-formatted output; and to be faster. Cox thanks Richard Goldstein, William Gould, Kenneth Higbee, Jay Kaufman, Jean Marie Linhart, and Fred Wolfe for their helpful suggestions on previous versions.
Also see

[D] cf — Compare two datasets
[D] codebook — Describe data contents
[D] compare — Compare two variables
[D] compress — Compress data in memory
[D] describe — Describe data in memory or in file
[D] format — Set variables’ output format
[D] label — Manipulate labels
[D] lookfor — Search for string in variable names and labels
[D] notes — Place notes in data
[D] order — Reorder variables in dataset
[D] rename — Rename variable
**duplicates — Report, tag, or drop duplicate observations**

### Description

duplicates reports, displays, lists, tags, or drops duplicate observations, depending on the subcommand specified. Duplicates are observations with identical values either on all variables if no *varlist* is specified or on a specified *varlist*.

duplicates report produces a table showing observations that occur as one or more copies and indicating how many observations are “surplus” in the sense that they are the second (third, . . .) copy of the first of each group of duplicates.

duplicates examples lists one example for each group of duplicated observations. Each example represents the first occurrence of each group in the dataset.

duplicates list lists all duplicated observations.

duplicates tag generates a variable representing the number of duplicates for each observation. This will be 0 for all unique observations.

duplicates drop drops all but the first occurrence of each group of duplicated observations. The word *drop* may not be abbreviated.

Any observations that do not satisfy specified if and/or in conditions are ignored when you use report, examples, list, or drop. The variable created by tag will have missing values for such observations.

### Quick start

Report the total number of observations and the number of duplicates

`duplicates report`

As above, but only check for duplicates jointly by *v1*, *v2*, and *v3*

`duplicates report v1 v2 v3`

Generate *newv* equal to the number of duplicate observations or 0 for unique observations

`duplicates tag, generate(newv)`

List all duplicate observations

`duplicates list`

As above, but determine duplicates by *v1*, *v2*, and *v3* and separate list by values of *v1*

`duplicates list v1 v2 v3, sepby(v1)`

Drop duplicate observations

`duplicates drop`

Force dropping observations with duplicates for *v1*, *v2*, and *v3* if observations are unique by other variables

`duplicates drop v1 v2 v3, force`
Menu

duplicates report, duplicates examples, and duplicates list
Data > Data utilities > Report and list duplicated observations

duplicates tag
Data > Data utilities > Tag duplicated observations

duplicates drop
Data > Data utilities > Drop duplicated observations

Syntax

Report duplicates

duplicates report [varlist] [if] [in]

List one example for each group of duplicates

duplicates examples [varlist] [if] [in] [, options]

List all duplicates

duplicates list [varlist] [if] [in] [, options]

Tag duplicates

duplicates tag [varlist] [if] [in], generate(newvar)

Drop duplicates

duplicates drop [if] [in]

duplicates drop varlist [if] [in], force
<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
<td></td>
</tr>
<tr>
<td><code>compress</code></td>
<td>compress width of columns in both table and display formats</td>
</tr>
<tr>
<td><code>nocompress</code></td>
<td>use display format of each variable</td>
</tr>
<tr>
<td><code>fast</code></td>
<td>synonym for <code>nocompress</code>; no delay in output of large datasets</td>
</tr>
<tr>
<td><code>abbreviate(#)</code></td>
<td>abbreviate variable names to # characters; default is <code>ab(8)</code></td>
</tr>
<tr>
<td><code>string(#)</code></td>
<td>truncate string variables to # characters; default is <code>string(10)</code></td>
</tr>
<tr>
<td><strong>Options</strong></td>
<td></td>
</tr>
<tr>
<td><code>table</code></td>
<td>force table format</td>
</tr>
<tr>
<td><code>display</code></td>
<td>force display format</td>
</tr>
<tr>
<td><code>header</code></td>
<td>display variable header once; default is table mode</td>
</tr>
<tr>
<td><code>noheader</code></td>
<td>suppress variable header</td>
</tr>
<tr>
<td><code>header(#)</code></td>
<td>display variable header every # lines</td>
</tr>
<tr>
<td><code>clean</code></td>
<td>force table format with no divider or separator lines</td>
</tr>
<tr>
<td><code>divider</code></td>
<td>draw divider lines between columns</td>
</tr>
<tr>
<td><code>separator(#)</code></td>
<td>draw a separator line every # lines; default is <code>separator(5)</code></td>
</tr>
<tr>
<td><code>sepby(varlist)</code></td>
<td>draw a separator line whenever <code>varlist</code> values change</td>
</tr>
<tr>
<td><code>nolabel</code></td>
<td>display numeric codes rather than label values</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td></td>
</tr>
<tr>
<td><code>mean([varlist])</code></td>
<td>add line reporting the mean for each of the (specified) variables</td>
</tr>
<tr>
<td><code>sum([varlist])</code></td>
<td>add line reporting the sum for each of the (specified) variables</td>
</tr>
<tr>
<td><code>N([varlist])</code></td>
<td>add line reporting the number of nonmissing values for each of the (specified) variables</td>
</tr>
<tr>
<td><code>labvar(varname)</code></td>
<td>substitute <code>Mean</code>, <code>Sum</code>, or <code>N</code> for value of <code>varname</code> in last row of table</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
</tr>
<tr>
<td><code>constant([varlist])</code></td>
<td>separate and list variables that are constant only once</td>
</tr>
<tr>
<td><code>notrim</code></td>
<td>suppress string trimming</td>
</tr>
<tr>
<td><code>absolute</code></td>
<td>display overall observation numbers when using by <code>varlist</code>:</td>
</tr>
<tr>
<td><code>nodotz</code></td>
<td>display numerical values equal to <code>.z</code> as field of blanks</td>
</tr>
<tr>
<td><code>subvarname</code></td>
<td>substitute characteristic for variable name in header</td>
</tr>
<tr>
<td><code>linesize(#)</code></td>
<td>columns per line; default is <code>linesize(79)</code></td>
</tr>
</tbody>
</table>

Collect is allowed with all `duplicates` commands; see [U] 11.1.10 Prefix commands.

**Options**

Options are presented under the following headings:

- Options for `duplicates` examples and `duplicates` list
- Option for `duplicates` tag
- Option for `duplicates` drop
Options for duplicates examples and duplicates list

- **Main**
  - compress, nocompress, fast, abbreviate(#), string(#); see [D] list.

- **Options**
  - table, display, header, noheader, header(#), clean, divider, separator(#),
    sepby(varlist), nolabel; see [D] list.

- **Summary**
  - mean[(varlist)], sum[(varlist)], N[(varlist)], labvar(varname); see [D] list.

- **Advanced**
  - constant[(varlist)], notrim, absolute, nodotz, subvarname, linesize(#); see [D] list.

**Option for duplicates tag**

`generate(newvar)` is required and specifies the name of a new variable that will tag duplicates.

**Option for duplicates drop**

`force` specifies that observations duplicated with respect to a named `varlist` be dropped. The `force` option is required when such a `varlist` is given as a reminder that information may be lost by dropping observations, given that those observations may differ on any variable not included in `varlist`.

**Remarks and examples**

Current data management and analysis may hinge on detecting (and sometimes dropping) duplicate observations. In Stata terms, **duplicates** are observations with identical values, either on all variables if no `varlist` is specified or on a specified `varlist`; that is, 2 or more observations that are identical on all specified variables form a group of duplicates. When the specified variables are a set of explanatory variables, such a group is often called a **covariate pattern** or a **covariate class**.

Linguistic purists will point out that duplicate observations are strictly only those that occur in pairs, and they might prefer a more literal term, although the most obvious replacement, “replicates”, already has another statistical meaning. However, the looser term appears in practice to be much more frequently used for this purpose and to be as easy to understand.

Observations may occur as duplicates through some error; for example, the same observations might have been entered more than once into your dataset. In contrast, some researchers deliberately enter a dataset twice. Each entry is a check on the other, and all observations should occur as identical pairs, assuming that one or more variables identify unique records. If there is just one copy, or more than two copies, there has been an error in data entry.

Or duplicate observations may also arise simply because some observations just happen to be identical, which is especially likely with categorical variables or large datasets. In this second situation, consider whether `contract`, which automatically produces a count of each distinct set of observations, is more appropriate for your problem. See [D] contract.
Observations unique on all variables in varlist occur as single copies. Thus there are no surplus observations in the sense that no observation may be dropped without losing information about the contents of observations. (Information will inevitably be lost on the frequency of such observations. Again, if recording frequency is important to you, contract is the better command to use.) Observations that are duplicated twice or more occur as copies, and in each case, all but one copy may be considered surplus.

This command helps you produce a dataset, usually smaller than the original, in which each observation is unique (literally, each occurs only once) and distinct (each differs from all the others). If you are familiar with Unix systems, or with sets of Unix utilities ported to other platforms, you will know the uniq command, which removes duplicate adjacent lines from a file, usually as part of a pipe.

### Example 1

Suppose that we are given a dataset in which some observations are unique (no other observation is identical on all variables) and other observations are duplicates (in each case, at least 1 other observation exists that is identical). Imagine dropping all but 1 observation from each group of duplicates, that is, dropping the surplus observations. Now all the observations are unique. This example helps clarify the difference between 1) identifying unique observations before dropping surplus copies and 2) identifying unique observations after dropping surplus copies (whether in truth or merely in imagination). codebook (see [D] codebook) reports the number of unique values for each variable in this second sense.

Suppose that we have typed in a dataset for 200 individuals. However, a simple describe or count shows that we have 202 observations in our dataset. We guess that we may have typed in 2 observations twice. duplicates report gives a quick report of the occurrence of duplicates:

```
. use https://www.stata-press.com/data/r17/dupxmpl
. duplicates report
Duplicates in terms of all variables

<table>
<thead>
<tr>
<th>Copies</th>
<th>Observations</th>
<th>Surplus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>198</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Our hypothesis is supported: 198 observations are unique (just 1 copy of each), whereas 4 occur as duplicates (2 copies of each; in each case, 1 may be dubbed surplus). We now wish to see which observations are duplicates, so the next step is to ask for a duplicates list.

```
. duplicates list
Duplicates in terms of all variables

<table>
<thead>
<tr>
<th>Group</th>
<th>Obs</th>
<th>id</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
<td>42</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>43</td>
<td>42</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>145</td>
<td>144</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>146</td>
<td>144</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
```
The records for id 42 and id 144 were evidently entered twice. Satisfied, we now issue `duplicates drop`.

```
.duplicates drop
Duplicates in terms of all variables
(2 observations deleted)
```

The `report`, `list`, and `drop` subcommands of `duplicates` are perhaps the most useful, especially for a relatively small dataset. For a larger dataset with many duplicates, a full listing may be too long to be manageable, especially as you see repetitions of the same data. `duplicates examples` gives you a more compact listing in which each group of duplicates is represented by just 1 observation, the first to occur.

A subcommand that is occasionally useful is `duplicates tag`, which generates a new variable containing the number of duplicates for each observation. Thus unique observations are tagged with value 0, and all duplicate observations are tagged with values greater than 0. For checking double data entry, in which you expect just one surplus copy for each individual record, you can generate a tag variable and then look at observations with tag not equal to 1 because both unique observations and groups with two or more surplus copies need inspection.

```
.duplicates tag, gen(tag)
Duplicates in terms of all variables
```

As of Stata 11, the `browse` subcommand is no longer available. To open duplicates in the Data Browser, use the following commands:

```
.duplicates tag, generate(newvar)
.browse if newvar > 0
```

See [D] edit for details on the `browse` command.

### Video example

How to identify and remove duplicate observations

### Stored results

- `duplicates report`, `duplicates examples`, `duplicates list`, `duplicates tag`, and `duplicates drop` store the following in `r()`:
  - Scalars
    - `r(N)` number of observations
  - `duplicates report` also stores the following in `r()`:
    - Scalars
      - `r(unique_value)` number of unique observations
  - `duplicates drop` also stores the following in `r()`:
    - Scalars
      - `r(N_drop)` number of observations dropped
Acknowledgments

duplicates was written by Nicholas J. Cox of the Department of Geography at Durham University, UK, who is coeditor of the Stata Journal and author of Speaking Stata Graphics. He in turn thanks Thomas Steichen (retired) of RJRT for ideas contributed to an earlier jointly written program (Steichen and Cox 1998).

References


Also see

[D] codebook — Describe data contents
[D] contract — Make dataset of frequencies and percentages
[D] edit — Browse or edit data with Data Editor
[D] isid — Check for unique identifiers
[D] list — List values of variables
dyngen — Dynamically generate new values of variables

Description

dyngen replaces the value of variables when two or more variables depend on each other’s lagged values. Use dyngen when the values for the whole set of variables must be computed for an observation before moving to the next observation.

Menu

Data > Create or change data > Dynamically generate new values

Syntax

dyngen {
    update varname1 = exp [if] [ , missval(#)]
    ;
    update varnameN = exp [if] [ , missval(#)]
} [if] [in]

varname_n, n = 1, . . . , N, must already exist in the dataset.
exp must be a valid expression and may include time-series operators; see [U] 11.4.4 Time-series varlists.

Option

missval(#) specifies the value to use in place of missing values when performing calculations. This option is particularly useful when referring to lags that exist prior to the data.

Remarks and examples

Like replace, dyngen modifies the contents of existing variables. However, dyngen works observation by observation. If you are doing a computation only on a single variable that relies only on its own lagged values or those of other variables, you do not need dyngen because generate and replace work their way through the data sequentially. Use dyngen when you need to modify two or more variables at the same time.
The examples in this entry use the following data:

```
. input time x1 x2
  time  x1  x2
  1.  1 3 1
  2.  2 4 4
  3.  3 5 2
  4.  4 5 1
  5.  5 2 1
  6.  end
```

Example 1: Using dyngen

We want to update our values of \( x_1 \) and \( x_2 \) such that \( x_1 \) depends on its current value and the previous value of \( x_2 \), and \( x_2 \) depends on previous values of \( x_1 \) and \( x_2 \). We will be using these same values of \( x_1 \) and \( x_2 \) in subsequent examples, so we do not want to overwrite their values. We create a copy of each in the variables \( d_1 \) and \( d_2 \), where the \( d \) prefix is used to remind us that these variables contain dynamically updated values.

```
. generate d1=x1
. generate d2=x2
```

Because we are using previous values, we need to specify a value for \texttt{dyngen} to substitute in place of missings; in this case, we use the means.

```
. summarize d1 d2

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>5</td>
<td>3.8</td>
<td>1.30384</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>d2</td>
<td>5</td>
<td>1.8</td>
<td>1.30384</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Within the \texttt{dyngen} command, we specify an \texttt{update} statement for \( d_1 \) and \( d_2 \). We also use observation subscripts to indicate the previous values as needed; see [U] \texttt{13.7 Explicit subscripting}. With time-series data, we could also use time-series operators; see example 3 for an illustration.

```
. dyngen {
  .  update d1 = .4*d1 + .1*d2[_n-1], missval(3.8)
  .  update d2 = .2*d1[_n-1] + .3*d2[_n-1], missval(1.8)
}
```

```
. list x1 x2 d*

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>d1</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>3.8</td>
<td>1.8</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.78</td>
<td>1.3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.13</td>
<td>.746</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2.0746</td>
<td>.6498</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>.86498</td>
<td>.60986</td>
</tr>
</tbody>
</table>
```

In observation 1, \texttt{dyngen} has substituted 3.8 for \( d_1 \) and 1.8 for \( d_2 \), values that would otherwise be missing because there are no data preceding the first observation. In observation 2, the updated value of \( d_1 \) is \( 0.4 \times 4 + 0.1 \times 1.8 = 1.78 \) and that of \( d_2 \) is \( 0.2 \times 3.8 + 0.3 \times 1.8 = 1.3 \), and so on.
Example 2: Distinction between dyngen and replace

We can compare the results from example 1 with those from `replace` to see how `dyngen` operates differently.

As in example 1, we create two new variables, `r1` and `r2`, that will hold values we update using `replace`. There is no automatic way to handle missing values with `replace`, so we need to set the first values to the means “by hand” to avoid missing values later. We then have a `replace` command for each variable, restricted to observations 2 through 5.

```
. generate r1=x1
. generate r2=x2
. replace r1 = 3.8 in 1
    (1 real change made)
. replace r2 = 1.8 in 1
    (1 real change made)
. replace r1 = .4*r1 + .1*r2[_n-1] in 2/5
    (4 real changes made)
. replace r2 = .2*r1[_n-1] + .3*r2[_n-1] in 2/5
    (4 real changes made)
```

Now, we can compare the results side by side.

```
. list x* d* r*
    +-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
    | x1 | x2 | d1  | d2  | r1  | r2  |
    +-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
    | 3  | 1  | 3.8 | 1.8 | 3.8 | 1.8 |
    | 4  | 4  | 1.78| 1.3 | 1.78| 1.3 |
    | 5  | 2  | 2.13| .746|.246| .746|
    | 5  | 1  | 2.0746| .6498| .246| .7038|
    | 2  | 1  | .86498| .60986|.9  | .65114|
    +-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

For the first two observations, the inputs are exactly the same, so there is no difference in the outcome. We see differences starting in the third row.

At the time that `replace` is updating the value of `r1` in observation 3, it is making the calculation

\[ 0.4 \times 5 + 0.1 \times 4 = 2.4 \]

because the value of `r2` is still 4, the original value of `x2`. Compare this with the results of `dyngen`, which uses

\[ 0.4 \times 5 + 0.1 \times 1.3 = 2.13 \]

That is, the key distinction is `dyngen` has fully updated observation 2 before moving on to observation 3. `replace` will make a full pass through `r1` before moving on to `r2`.

Example 3: Processing if conditions

Each update statement within the `dyngen` command can take an `if` condition. To illustrate, we replace `d1` and `d2` with the original values of `x1` and `x2` and update them again, this time restricting the updated observations to just those observations where `time` ≥ 3.

```
. replace d1=x1
    (5 real changes made)
. replace d2=x2
    (5 real changes made)
```
Here, we `tsset` the data and use the lag operator instead of subscripting observations, but that is not required.

```stata
.tsset time
Time variable: time, 1 to 5
   Delta: 1 unit
.dyngen {
   . update d1 = .4*d1 + .1*L.d2 if time>=3
   . update d2 = .2*L.d1 + .3*L.d2 if time>=3
   }
.list x* d*
```

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>d1</th>
<th>d2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3.</td>
<td>5</td>
<td>2</td>
<td>2.4</td>
</tr>
<tr>
<td>4.</td>
<td>5</td>
<td>1</td>
<td>2.2</td>
</tr>
<tr>
<td>5.</td>
<td>2</td>
<td>1</td>
<td>.908</td>
</tr>
</tbody>
</table>

When the same `if` condition is specified on all `update` statements, the results are equivalent to specifying one `if` condition on the entire `dyngen` block. We used the same `if` statement on both `update` statements above, so typing the following produces the same results as the code above.

```stata
dyngen {
   update d1 = .4*d1 + .1*L.d2
   update d2 = .2*L.d1 + .3*L.d2
} if time>=3
```

You may also specify an `in` qualifier with the `dyngen` command. If you specify an `if` or `in` qualifier, `dyngen` loops over the observations that meet the `if` condition or `in` range but will reference values outside that range if needed.

---

Also see

[D] `generate` — Create or change contents of variable

[U] 12 Data

[U] 13 Functions and expressions
edit — Browse or edit data with Data Editor

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Menu</th>
<th>Syntax</th>
</tr>
</thead>
</table>

**Description**

edit brings up a spreadsheet-style data editor for entering new data and editing existing data. edit is a better alternative to input; see [D] input.

browse is similar to edit, except that modifications to the data by editing in the grid are not permitted. browse is a convenient alternative to list; see [D] list.

See [GS] 6 Using the Data Editor (GSM, GSU, or GSW) for a tutorial discussion of the Data Editor. This entry provides the technical details.

**Quick start**

Open dataset in the Data Editor for entering new data or editing existing data

edit

As above, but include only v1, v2, and v3

edit v1 v2 v3

As above, but only for observations where v3 is missing

edit v1 v2 v3 if v3 >= .

Open dataset in the Data Editor with no ability to edit data

browse

As above, but include only v1, v2, and v3 and suppress value labels

browse v1 v2 v3, nolabel

**Menu**

edit

Data > Data Editor > Data Editor (Edit)

browse

Data > Data Editor > Data Editor (Browse)
Syntax

Edit using Data Editor

\texttt{edit [varlist] [if] [in] [, nolabel]}

Browse using Data Editor

\texttt{browse [varlist] [if] [in] [, nolabel]}

Option

\texttt{nolabel} causes the underlying numeric values, rather than the label values (equivalent strings), to be displayed for variables with value labels; see [D] \texttt{label}.

Remarks and examples

Remarks are presented under the following headings:

- Modes
- The current observation and current variable
- Assigning value labels to variables
- Changing values of existing cells
- Adding new variables
- Adding new observations
- Copying and pasting
- Logging changes
- Advice

Clicking on Stata’s \texttt{Data Editor (Edit)} button is equivalent to typing \texttt{edit} by itself. Clicking on Stata’s \texttt{Data Editor (Browse)} button is equivalent to typing \texttt{browse} by itself.

\texttt{edit}, typed by itself, opens the Data Editor with all observations on all variables displayed. If you specify a \texttt{varlist}, only the specified variables are displayed in the Editor. If you specify one or both of \texttt{in range} and \texttt{if exp}, only the observations specified are displayed.

Modes

We will refer to the Data Editor in the singular with \texttt{edit} and \texttt{browse} referring to two of its three modes.

\textit{Full-edit mode}. This is the Editor’s mode that you enter when you type \texttt{edit} or type \texttt{edit} followed by a list of variables. All features of the Editor are turned on.

\textit{Filtered mode}. This is the Editor’s mode that you enter when you use \texttt{edit} with or without a list of variables but include \texttt{in range}, \texttt{if exp}, or both, or if you filter the data from within the Editor. A few of the Editor’s features are turned off, most notably, the ability to sort data and the ability to paste data into the Editor.

\textit{Browse mode}. This is the Editor’s mode that you enter when you use \texttt{browse} or when you change the Editor’s mode to \texttt{Browse} after you start the Editor. The ability to type in the Editor, thereby changing data, is turned off, ensuring that the data cannot accidentally be changed. One feature that is left on may surprise you: the ability to sort data. Sorting, in Stata’s mind, is not really a change to the dataset. On the other hand, if you enter using \texttt{browse} and specify \texttt{in range} or \texttt{if exp}, sorting is not allowed. You can think of this as restricted-browse mode.
Actually, the Editor does not set its mode to filtered just because you specify an \texttt{in range} or \texttt{if} \textit{exp}. It sets its mode to filtered if you specify \texttt{in} or \texttt{if} and if this restriction is effective, that is, if the \texttt{in} or \texttt{if} would actually cause some data to be omitted. For instance, typing \texttt{edit if \textit{x}>0} would result in unrestricted full-edit mode if \textit{x} were greater than zero for all observations.

### The current observation and current variable

The Data Editor looks much like a spreadsheet, with rows and columns corresponding to observations and variables, respectively. At all times, one of the cells is highlighted. This is called the current cell. The observation (row) of the current cell is called the current observation. The variable (column) of the current cell is called the current variable.

You change the current cell by clicking with the mouse on another cell or by using the arrow keys.

To help distinguish between the different types of variables in the Editor, string values are displayed in red, value labels are displayed in blue, and all other values are displayed in black. You can change the colors for strings and value labels by right-clicking on the Data Editor window and selecting \textit{Preferences}...

### Assigning value labels to variables

You can assign a value label to a nonstring variable by right-clicking any cell on the variable column, choosing the \texttt{Data > Value Labels} menu, and selecting a value label from the \textit{Attach Value Label to Variable} ‘\textit{varname}’ menu. You can define a value label by right-clicking on the Data Editor window and selecting \texttt{Data > Value Labels > Manage Value Labels}.... You can also accomplish these tasks by using the Properties pane; see [GS] 6 Using the Data Editor (GSM, GSU, or GSW) for details.

### Changing values of existing cells

Make the cell you wish to change the current cell. Type the new value, and press \textit{Enter}. When updating string variables, do not type double quotes around the string. For variables that have a value label, you can right-click on the cell to display a list of values for the value label. You can assign a new value to the cell by selecting a value from the list.

### Technical note

Stata experts will wonder about storage types. Say that variable \textit{mpg} is stored as an \texttt{int} and you want to change the fourth observation to contain 22.5. The Data Editor will change the storage type of the variable. Similarly, if the variable is a \texttt{str4} and you type \texttt{alpha}, it will be changed to \texttt{str5}.

The Editor will not, however, change numeric variable types to strings (unless the numeric variable contains only missing values). This is intentional, as such a change could result in a loss of data and is probably the result of a mistake.

### Technical note

Stata can store long strings in the \texttt{strL} storage type. Although the \texttt{strL} type can hold very long strings, these strings may only be edited if they are 2045 characters or less. Similarly, \texttt{strLs} that hold binary data may not be edited. For more information on storage types, see [D] \textit{Data types}.
Adding new variables

Go to the first empty column, and begin entering your data. The first entry that you make will create the variable and determine whether that variable is numeric or string. The variable will be given a name like `var1`, but you can rename it by using the Properties pane.

Technical note

Stata experts: The storage type will be determined automatically. If you type a number, the created variable will be numeric; if you type a string, it will be a string. Thus if you want a string variable, be sure that your first entry cannot be interpreted as a number. A way to achieve this is to use surrounding quotes so that "123" will be taken as the string "123", not the number 123. If you want a numeric variable, do not worry about whether it is byte, int, float, etc. If a byte will hold your first number but you need a float to hold your second number, the Editor will recast the variable later.

Technical note

If you do not type in the first empty column but instead type in one to the right of it, the Editor will create variables for all the intervening columns.

Adding new observations

Go to the first empty row, and begin entering your data. As soon as you add one cell below the last row of the dataset, an observation will be created.

Technical note

If you do not enter data in the first empty row but, instead, enter data in a row below it, the Data Editor will create observations for all the intervening rows.

Copying and pasting

You can copy and paste data between Stata’s Data Editor and other applications.

First, select the data you wish to copy. In Stata, click on a cell and drag the mouse across other cells to select a range of cells. If you want to select an entire column, click once on the variable name at the top of that column. If you want to select an entire row, click once on the observation number at the left of that row. You can hold down the mouse button after clicking and drag to select multiple columns or rows.

Once you have selected the data, copy the data to the Clipboard. In Stata, right-click on the selected data, and select Copy.

You can copy data to the Clipboard from Stata with or without the variable names at the top of each column by right-clicking on the Data Editor window, selecting Preferences..., and checking or unchecking Include variable names on copy to Clipboard.
You can choose to copy either the value labels or the underlying numeric values associated with the selected data by right-clicking on the Data Editor window, selecting Preferences..., and checking or unchecking Copy value labels instead of numbers. For more information about value labels, see [U] 12.6.3 Value labels and [D] label.

After you have copied data to the Clipboard from Stata’s Data Editor or another spreadsheet, you can paste the data into Stata’s Data Editor. First, select the top-left cell of the area into which you wish to paste the data by clicking on it once. Then right-click on the cell and select Paste. Stata will paste the data from the Clipboard into the Editor, overwriting any data below and to the right of the cell you selected as the top left of the paste area. If the Data Editor is in filtered mode or in browse mode, Paste will be disabled, meaning that you cannot paste into the Data Editor. You can have more control over how data are pasted by selecting Paste Special....

Technical note

If you attempt to paste one or more string values into numeric variables, the original numeric values will be left unchanged for those cells. Stata will display a message box to let you know that this has happened: “You attempted to paste one or more string values into numeric variables. The contents of these cells, if any, are unchanged.”

If you see this message, you should look carefully at the data that you pasted into Stata’s Data Editor to make sure that you pasted into the area that you intended. We recommend that you take a snapshot of your data before pasting into Stata’s Data Editor so that you can restore the data from the snapshot if you make a mistake. See [GS] 6 Using the Data Editor (GSM, GSU, or GSW) to read about snapshots.

Logging changes

When you use edit to enter new data or change existing data, you will find output in the Stata Results window documenting the changes that you made. For example, a line of this output might be

```
. replace mpg = 22.5 in 5
```

The Editor submits a command to Stata for everything you do in it except pasting. If you are logging your results, you will have a permanent record of what you did in the Editor.

Advice

- People who care about data integrity know that editors are dangerous—it is too easy to make changes accidentally. Never use edit when you want to browse.

- Protect yourself when you edit existing data by limiting exposure. If you need to change mpg and need to see model to know which value of mpg to change, do not click on the Data Editor button. Instead, type edit model mpg. It is now impossible for you to change (damage) variables other than model and mpg. Furthermore, if you know that you need to change mpg only if it is missing, you can reduce your exposure even more by typing `edit model mpg if mpg>=.'.

- Stata’s Data Editor is safer than most because it logs changes to the Results window. Use this feature—look at the log afterward, and verify that the changes you made are the changes you wanted to make.
Also see

[D] import — Overview of importing data into Stata
[D] input — Enter data from keyboard
[D] list — List values of variables
[D] save — Save Stata dataset

[GSM] 6 Using the Data Editor
[GSW] 6 Using the Data Editor
[GSU] 6 Using the Data Editor
egen — Extensions to generate

Description

\texttt{egen} creates \textit{newvar} of the optionally specified storage type equal to \textit{fcn(arguments)}. Here \textit{fcn()} is a function specifically written for \texttt{egen}, as documented below or as written by users. Only \texttt{egen} functions may be used with \texttt{egen}, and conversely, only \texttt{egen} may be used to run \texttt{egen} functions.

Depending on \textit{fcn()}, \textit{arguments}, if present, refers to an expression, \textit{varlist}, or a \textit{numlist}, and the \textit{options} are similarly \textit{fcn} dependent. Explicit subscripting (using \_N and \_n), which is commonly used with \texttt{generate}, should not be used with \texttt{egen}; see [\texttt{U}] 13.7 Explicit subscripting.

Quick start

Generate \texttt{newv1} for distinct groups of \texttt{v1} and \texttt{v2}, and create and apply value label \texttt{mylabel}
\begin{verbatim}
egen newv1 = group(v1 v2), label(mylabel)
\end{verbatim}

Generate \texttt{newv2} equal to the minimum of \texttt{v1}, \texttt{v2}, and \texttt{v3} for each observation
\begin{verbatim}
egen newv2 = rowmin(v1 v2 v3)
\end{verbatim}

Generate \texttt{newv3} equal to the overall sum of \texttt{v1}
\begin{verbatim}
egen newv3 = total(v1)
\end{verbatim}

As above, but calculate total within each level of \texttt{catvar}
\begin{verbatim}
egen newv3 = total(v1), by(catvar)
\end{verbatim}

Generate \texttt{newv4} equal to the number of nonmissing numeric values across \texttt{v1}, \texttt{v2}, and \texttt{v3} for each observation
\begin{verbatim}
egen newv4 = rownonmiss(v1 v2 v3)
\end{verbatim}

As above, but allow string values
\begin{verbatim}
egen newv4 = rownonmiss(v1 v2 v3), strok
\end{verbatim}

Generate \texttt{newv5} as the concatenation of numeric \texttt{v1} and string \texttt{v4} separated by a space
\begin{verbatim}
egen newv5 = concat(v1 v4), punct(" ")
\end{verbatim}

Menu

Data > Create or change data > Create new variable (extended)
**Syntax**

```
egen [type] newvar = fcn(arguments) [if] [in] [ , options ]
```

by is allowed with some of the egen functions, as noted below.

where depending on the fcn, arguments refers to an expression, varlist, or numlist, and the options are also fcn dependent, and where fcn is

- `anycount(varlist), values(integer numlist)`
  - may not be combined with by. It returns the number of variables in varlist for which values are equal to any integer value in a supplied numlist. Values for any observations excluded by either if or in are set to 0 (not missing). Also see anyvalue(varname) and anymatch(varlist).

- `anymatch(varlist), values(integer numlist)`
  - may not be combined with by. It is 1 if any variable in varlist is equal to any integer value in a supplied numlist and 0 otherwise. Values for any observations excluded by either if or in are set to 0 (not missing). Also see anyvalue(varname) and anycount(varlist).

- `anyvalue(varname), values(integer numlist)`
  - may not be combined with by. It takes the value of varname if varname is equal to any integer value in a supplied numlist and is missing otherwise. Also see anymatch(varlist) and anycount(varlist).

- `concat(varlist) [ , format( fmt ) decode maxlen( length ) punct( pchars ) ]`
  - may not be combined with by. It concatenates varlist to produce a string variable. Values of string variables are unchanged. Values of numeric variables are converted to string, as is, or are converted using a numeric format under the format( fmt ) option or decoded under the decode option, in which case maxlen( length ) may also be used to control the maximum label length used. By default, variables are added end to end: punct( pchars ) may be used to specify punctuation, such as a space, punct(" "), or a comma, punct(,).

- `count(exp)`
  - creates a constant (within varlist) containing the number of nonmissing observations of exp. Also see rownonmiss() and rowmiss().

- `cut(varname), { at(#,#,...,#) | group(#) } [ icodes label ]`
  - may not be combined with by. It creates a new categorical variable coded with the left-hand ends of the grouping intervals specified in the at() option, which expects an ascending numlist.
  - at(#,#,...,#) supplies the breaks for the groups, in ascending order. The list of breakpoints may be simply a list of numbers separated by commas but can also include the syntax a(b)c, meaning from a to c in steps of size b. newvar is set to missing for observations with varname less than the first number specified in at() and for observations with varname greater than or equal to the last number specified in at(). If no breaks are specified, the command expects the group() option.
  - group(#) specifies the number of equal frequency grouping intervals to be used in the absence of breaks. Specifying this option automatically invokes icodes.
  - icodes requests that the codes 0, 1, 2, etc., be used in place of the left-hand ends of the intervals.
  - label requests that the integer-coded values of the grouped variable be labeled with the left-hand ends of the grouping intervals. Specifying this option automatically invokes icodes.

- `diff(varlist)`
  - may not be combined with by. It creates an indicator variable equal to 1 if the variables in varlist are not equal and 0 otherwise.
ends(strvar) [ , punct(pchars) trim [ head | last | tail ] ]
may not be combined with by. It gives the first “word” or head (with the head option), the last “word” (with the last option), or the remainder or tail (with the tail option) from string variable strvar.

head, last, and tail are determined by the occurrence of pchars, which is by default one space (“ ”).

The head is whatever precedes the first occurrence of pchars, or the whole of the string if it does not occur. For example, the head of “frog toad” is “frog” and that of “frog” is “frog”. With punct(,), the head of “frog, toad” is “frog”.

The last word is whatever follows the last occurrence of pchars or is the whole of the string if a space does not occur. The last word of “frog toad newt” is “newt” and that of “frog” is “frog”. With punct(,), the last word of “frog, toad” is “toad”.

The remainder or tail is whatever follows the first occurrence of pchars, which will be the empty string “” if pchars does not occur. The tail of “frog toad newt” is “toad newt” and that of “frog” is “”. With punct(,), the tail of “frog, toad” is “toad”.

The trim option trims any leading or trailing spaces.

fill(numlist)
may not be combined with by. It creates a variable of ascending or descending numbers or complex repeating patterns. numlist must contain at least two numbers and may be specified using standard numlist notation; see [U] 11.1.8 numlist. if and in are not allowed with fill().

group(varlist) [ , missing autotype label[(lblname [, replace truncate(#) ) )]]
may not be combined with by. It creates one variable taking on values 1, 2, ... for the groups formed by varlist. varlist may contain numeric variables, string variables, or a combination of the two. The order of the groups is that of the sort order of varlist.

missing indicates that missing values in varlist (either . or "") are to be treated like any other value when assigning groups. By default, any observation with a missing value is assigned to the group with newvar equal to missing (.).

autotype specifies that newvar be the smallest type possible to hold the integers generated. The resulting type will be byte, int, long, or double.

label or label(lbname) creates a value label for newvar. The integers in newvar are labeled with the values of varlist or their value labels, if they exist. label(lbname) specifies lbname as the name of the value label. If label alone is specified, the name of the value label is newvar. label(..., replace) allows an existing value label to be redefined. label(..., truncate(#)) truncates the values contributed to the label from each variable in varlist to the length specified by the integer argument #.

iqr(exp) (allows by varlist:)
creates a constant (within varlist) containing the interquartile range of exp. Also see pctile().

kurt(exp) (allows by varlist:)
returns the kurtosis (within varlist) of exp.

mad(exp) (allows by varlist:)
returns the median absolute deviation from the median (within varlist) of exp.

max(exp) [ , missing ] (allows by varlist:)
creates a constant (within varlist) containing the maximum value of exp. missing indicates that missing values be treated like other values.
egen — Extensions to generate

mdev(exp) (allows by varlist:)
returns the mean absolute deviation from the mean (within varlist) of exp.

mean(exp) (allows by varlist:)
creates a constant (within varlist) containing the mean of exp.

median(exp) (allows by varlist:)
creates a constant (within varlist) containing the median of exp. Also see pctl().

min(exp) [ , missing]
creates a constant (within varlist) containing the minimum value of exp. missing indicates that missing values be treated like other values.

mode(varname) [ , minmode maxmode nummode(integer) missing] (allows by varlist:)
produces the mode (within varlist) for varname, which may be numeric or string. The mode is the value occurring most frequently. If two or more modes exist or if varname contains all missing values, the mode produced will be a missing value. To avoid this, the minmode, maxmode, or nummode() option may be used to specify choices for selecting among the multiple modes, and the missing option will treat missing values as categories. minmode returns the lowest value, and maxmode returns the highest value. nummode(#) will return the #th mode, counting from the lowest up. nummode() may not be combined with by. Missing values are excluded from determination of the mode unless missing is specified. Even so, the value of the mode is recorded for observations for which the values of varname are missing unless they are explicitly excluded, that is, by if varname < . or if varname != "".

mtr(year income)
may not be combined with by. It returns the U.S. marginal income tax rate for a married couple with taxable income income in year year, where 1930 ≤ year ≤ 2019. year and income may be specified as variable names or constants; for example, mtr(1993 faminc), mtr(surveyyr 28000), or mtr(surveyyr faminc). A blank or comma may be used to separate income from year.

pc(exp) [ , prop]
returns exp (within varlist) scaled to be a percentage of the total, between 0 and 100. The prop option returns exp scaled to be a proportion of the total, between 0 and 1.

pctl(exp) [ , p(#)] (allows by varlist:)
creates a constant (within varlist) containing the #th percentile of exp. If p(#) is not specified, 50 is assumed, meaning medians. Also see median().

rank(exp) [ , field | track | unique]
creates ranks (within varlist) of exp; by default, equal observations are assigned the average rank. The field option calculates the field rank of exp: the highest value is ranked 1, and there is no correction for ties. That is, the field rank is 1 + the number of values that are higher. The track option calculates the track rank of exp: the lowest value is ranked 1, and there is no correction for ties. That is, the track rank is 1 + the number of values that are lower. The unique option calculates the unique rank of exp: values are ranked 1, . . . , #, and values and ties are broken arbitrarily. Two values that are tied for second are ranked 2 and 3.

rowfirst(varlist)
may not be combined with by. It gives the first nonmissing value in varlist for each observation (row). If all values in varlist are missing for an observation, newvar is set to missing.

rowlast(varlist)
may not be combined with by. It gives the last nonmissing value in varlist for each observation (row). If all values in varlist are missing for an observation, newvar is set to missing.
rowmax(varlist)
may not be combined with by. It gives the maximum value (ignoring missing values) in varlist for each observation (row). If all values in varlist are missing for an observation, newvar is set to missing.

rowmean(varlist)
may not be combined with by. It creates the (row) means of the variables in varlist, ignoring missing values; for example, if three variables are specified and, in some observations, one of the variables is missing, in those observations newvar will contain the mean of the two variables that do exist. Other observations will contain the mean of all three variables. Where none of the variables exist, newvar is set to missing.

rowmedian(varlist)
may not be combined with by. It gives the (row) median of the variables in varlist, ignoring missing values. If all variables in varlist are missing for an observation, newvar is set to missing in that observation. Also see rowpctile().

rowmin(varlist)
may not be combined with by. It gives the minimum value in varlist for each observation (row). If all values in varlist are missing for an observation, newvar is set to missing.

rowmiss(varlist)
may not be combined with by. It gives the number of missing values in varlist for each observation (row).

rownonmiss(varlist) [, strok]
may not be combined with by. It gives the number of nonmissing values in varlist for each observation (row)—this is the value used by rowmean() for the denominator in the mean calculation.

String variables may not be specified unless the strok option is also specified. If strok is specified, string variables will be counted as containing missing values when they contain "". Numeric variables will be counted as containing missing when their value is “≥.”.

rowpctile(varlist) [, p(#)]
may not be combined with by. It gives the #th percentile of the variables in varlist, ignoring missing values. If all variables in varlist are missing for an observation, newvar is set to missing in that observation. If p() is not specified, p(50) is assumed, meaning medians. Also see rowmedian().

rowsd(varlist)
may not be combined with by. It creates the (row) standard deviations of the variables in varlist, ignoring missing values.

rowtotal(varlist) [, missing]
may not be combined with by. It creates the (row) sum of the variables in varlist, treating missing values as 0. If missing is specified and all values in varlist are missing for an observation, newvar is set to missing.

sd(exp) (allows by varlist:)
creates a constant (within varlist) containing the standard deviation of exp. Also see mean().

seq() [, from(#) to(#) block(#) ] (allows by varlist:)
returns integer sequences. Values start from from() (default 1) and increase to to() (the default is the maximum number of values) in blocks (default size 1). If to() is less than the maximum number, sequences restart at from(). Numbering may also be separate within groups defined by varlist or decreasing if to() is less than from(). Sequences depend on the sort order of observations, following three rules: 1) observations excluded by if or in are not
egen — Extensions to generate counted; 2) observations are sorted by varlist, if specified; and 3) otherwise, the order is that when called. No arguments are specified.

skew(exp) (allows by varlist:)
returns the skewness (within varlist) of exp.

std(exp) [ , mean(#) sd(#)] (allows by varlist:)
creates the standardized values (within varlist) of exp. The options specify the desired mean and standard deviation. The default is mean(0) and sd(1), producing a variable with mean 0 and standard deviation 1 (within each group defined by varlist).

tag(varlist) [ , missing]
may not be combined with by. It tags just one observation in each distinct group defined by varlist. When all observations in a group have the same value for a summary variable calculated for the group, it will be sufficient to use just one value for many purposes. The result will be 1 if the observation is tagged and never missing, and 0 otherwise. Values for any observations excluded by either if or in are set to 0 (not missing). Hence, if tag is the variable produced by egen tag = tag(varlist), the idiom if tag is always safe. missing specifies that missing values of varlist may be included.

total(exp) [ , missing] (allows by varlist:)
creates a constant (within varlist) containing the sum of exp treating missing as 0. If missing is specified and all values in exp are missing, newvar is set to missing. Also see mean().

Remarks and examples

Remarks are presented under the following headings:

Summary statistics
Generating patterns
Marking differences among variables
Ranks
Standardized variables
Row functions
Categorical and integer variables
String variables
U.S. marginal income tax rate

See Mitchell (2020) for numerous examples using egen.

Summary statistics

The functions count(), iqr(), kurt(), mad(), max(), mdev(), mean(), median(), min(), mode(), pc(), pctile(), sd(), skew(), and total() create variables containing summary statistics. These functions take a by ...: prefix and, if specified, calculate the summary statistics within each by-group.

Example 1: Without the by prefix

Without the by prefix, the result produced by these functions is a constant for every observation in the data. For instance, we have data on cholesterol levels (chol) and wish to have a variable that, for each patient, records the deviation from the average across all patients:
. use https://www.stata-press.com/data/r17/egenxmpl
. egen avg = mean(chol)
. generate deviation = chol - avg

Example 2: With the by prefix

These functions are most useful when the by prefix is specified. For instance, assume that our
dataset includes dcode, a hospital–patient diagnostic code, and los, the number of days that the
patient remained in the hospital. We wish to obtain the deviation in length of stay from the median
for all patients having the same diagnostic code:

. use https://www.stata-press.com/data/r17/egenxmpl2, clear
. by dcode, sort: egen medstay = median(los)
. generate deltalos = los - medstay

Technical note

Distinguish carefully between Stata’s sum() function and egen’s total() function. Stata’s sum()
function creates the running sum, whereas egen’s total() function creates a constant equal to the
overall sum; for example,

. clear
. set obs 5
Number of observations (_N) was 0, now 5.
. generate a = _n
. generate sum1=sum(a)
. egen sum2=total(a)
. list

<table>
<thead>
<tr>
<th>a</th>
<th>sum1</th>
<th>sum2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Technical note

The definitions and formulas used by these functions are the same as those used by summarize; see [R] summarize. For comparison with summarize, mean() and sd() correspond to the mean and standard deviation. total() is the numerator of the mean, and count() is its denominator. min() and max() correspond to the minimum and maximum. median()—or, equally well, pctile() with p(50)—is the median. pctile() with p(5) refers to the fifth percentile, and so on. iqr() is the difference between the 75th and 25th percentiles.

The mode is the most common value of a dataset, whether it contains numeric or string variables.
It is perhaps most useful for categorical variables (whether defined by integers or strings) or for other
integer-valued values, but mode() can be applied to variables of any type. Nevertheless, the modes
of continuous (or nearly continuous) variables are perhaps better estimated either from inspection of
a graph of a frequency distribution or from the results of some density estimation (see [R] kdensity).
Missing values need special attention. It is possible that missing is the most common value in a variable (whether missing is defined by the period [.] or extended missing values [.a, .b, ..., .z] for numeric variables or the empty string [""] for string variables). However, missing values are by default excluded from determination of modes. If you wish to include them, use the missing option.

In contrast, **egen mode = mode(varname)** allows the generation of nonmissing modes for observations for which **varname** is missing. This allows use of the mode as one simple means of imputing categorical variables. If you want the mode to be missing whenever **varname** is missing, you can specify if **varname** < . or if **varname** != "" or, most generally, if !missing(**varname**).

**mad()** and **mdev()** produce alternative measures of spread. The median absolute deviation from the median and even the mean deviation will both be more resistant than the standard deviation to heavy tails or outliers, in particular from distributions with heavier tails than the normal or Gaussian. The first measure was named the MAD by Andrews et al. (1972) but was already known to K. F. Gauss in 1816, according to Hampel et al. (1986). For more historical and statistical details, see David (1998) and Wilcox (2003, 72–73).

### Generating patterns

To create a sequence of numbers, simply “show” the **fill()** function how the sequence should look. It must be a linear progression to produce the expected results. Stata does not understand geometric progressions. To produce repeating patterns, you present **fill()** with the pattern twice in the **numlist**.

#### Example 3: Sequences produced by **fill()**

Here are some examples of ascending and descending sequences produced by **fill()**:

```
. clear
. set obs 12
   Number of observations (N) was 0, now 12.
. egen i=fill(1 2)
. egen w=fill(100 99)
. egen x=fill(22 17)
. egen y=fill(1 1 2 2)
. egen z=fill(8 8 7 7 7)
. list, sep(4)
```

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>100</td>
<td>22</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>99</td>
<td>17</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>98</td>
<td>12</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td>97</td>
<td>7</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>96</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>6.</td>
<td>6</td>
<td>95</td>
<td>-3</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>7.</td>
<td>7</td>
<td>94</td>
<td>-8</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>8.</td>
<td>8</td>
<td>93</td>
<td>-13</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>9.</td>
<td>9</td>
<td>92</td>
<td>-18</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>10.</td>
<td>10</td>
<td>91</td>
<td>-23</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>11.</td>
<td>11</td>
<td>90</td>
<td>-28</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>12.</td>
<td>12</td>
<td>89</td>
<td>-33</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
Example 4: Patterns produced by fill()

Here are examples of patterns produced by `fill()`:

```
. clear
. set obs 12
Number of observations (_N) was 0, now 12.
. egen a=fill(0 0 1 0 0 1)
. egen b=fill(1 3 8 1 3 8)
. egen c=fill(-3(3)6 -3(3)6)
. egen d=fill(10 20 to 50 10 20 to 50)
. list, sep(4)
```

```
a  b  c  d
1. 0  1 -3 10
2. 0  3  0 20
3. 1  8  3 30
4. 0  1  6 40
5. 0  3 -3 50
6. 1  8  0 10
7. 0  1  3 20
8. 0  3  6 30
9. 1  8 -3 40
10. 0  1  0 50
11. 0  3  3 10
12. 1  8  6 20
```

Example 5: seq()

`seq()` creates a new variable containing one or more sequences of integers. It is useful mainly for quickly creating observation identifiers or automatically numbering levels of factors or categorical variables.

```
. clear
. set obs 12

In the simplest case,

. egen a = seq()
```

is just equivalent to the common idiom

```
. generate a = _n
```
a may also be obtained from

```
. range a 1 _N
```
(the actual value of `_N` may also be used).

In more complicated cases, `seq()` with option calls is equivalent to calls to the versatile functions `int` and `mod`.

```
. egen b = seq(), b(2)
```
produces integers in blocks of 2, whereas

`. egen c = seq(), t(6)

restarts the sequence after 6 is reached.

`. egen d = seq(), f(10) t(12)

shows that sequences may start with integers other than 1, and

`. egen e = seq(), f(3) t(1)

shows that they may decrease.

The results of these commands are shown by

`. list, sep(4)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>3</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>5</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>6</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>6</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

All of these sequences could have been generated in one line with `generate` and with the use of the `int` and `mod` functions. The variables `b` through `e` are obtained with

`. gen b = 1 + int(_n - 1)/2
`. gen c = 1 + mod(_n - 1, 6)
`. gen d = 10 + mod(_n - 1, 3)
`. gen e = 3 - mod(_n - 1, 3)

Nevertheless, `seq()` may save users from puzzling out such solutions or from typing in the needed values.

In general, the sequences produced depend on the sort order of observations, following three rules:

1. observations excluded by `if` or `in` are not counted;
2. observations are sorted by `varlist`, if specified; and
3. otherwise, the order is that specified when `seq()` is called.

The `fill()` and `seq()` functions are alternatives. In essence, `fill()` requires a minimal example that indicates the kind of sequence required, whereas `seq()` requires that the rule be specified through options. There are sequences that `fill()` can produce that `seq()` cannot, and vice versa. `fill()` cannot be combined with `if` or `in`, in contrast to `seq()`, which can.
Marking differences among variables

Example 6: diff()

We have three measures of respondents’ income obtained from different sources. We wish to create the variable differ equal to 1 for disagreements:

```
.use https://www.stata-press.com/data/r17/egenxmpl3, clear
.egen byte differ = diff(inc*)
.list if differ==1
```

<table>
<thead>
<tr>
<th>inc1</th>
<th>inc2</th>
<th>inc3</th>
<th>id</th>
<th>differ</th>
</tr>
</thead>
<tbody>
<tr>
<td>42,491</td>
<td>42,491</td>
<td>42,491</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>26,075</td>
<td>25,075</td>
<td>25,075</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>26,283</td>
<td>25,283</td>
<td>25,283</td>
<td>112</td>
<td>1</td>
</tr>
<tr>
<td>41,780</td>
<td>41,780</td>
<td>41,880</td>
<td>178</td>
<td>1</td>
</tr>
<tr>
<td>25,687</td>
<td>25,687</td>
<td>25,687</td>
<td>200</td>
<td>1</td>
</tr>
</tbody>
</table>

Rather than typing `diff(inc*)`, we could have typed `diff(inc1 inc2 inc3)`.

Ranks

Example 7: rank()

Most applications of `rank()` will be to one variable, but the argument `exp` can be more general, namely, an expression. In particular, `rank(-varname)` reverses ranks from those obtained by `rank(varname)`.

The default ranking and those obtained by using one of the `track`, `field`, and `unique` options differ principally in their treatment of ties. The default is to assign the same rank to tied values such that the sum of the ranks is preserved. The `track` option assigns the same rank but resembles the convention in track events; thus, if one person had the lowest time and three persons tied for second-lowest time, their ranks would be 1, 2, 2, and 2, and the next person(s) would have rank 5. The `field` option acts similarly except that the highest is assigned rank 1, as in field events in which the greatest distance or height wins. The `unique` option breaks ties arbitrarily: its most obvious use is assigning ranks for a graph of ordered values. See also `group()` for another kind of “ranking”.

```
.use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
.keep in 1/10
(64 observations deleted)
.egen rank = rank(mpg)
.egen rank_r = rank(-mpg)
.egen rank_f = rank(mpg), field
```
. egen rank_t = rank(mpg), track
. egen rank_u = rank(mpg), unique
. egen rank_ur = rank(-mpg), unique
. sort rank_u
. list mpg rank*

<table>
<thead>
<tr>
<th>mpg</th>
<th>rank</th>
<th>rank_r</th>
<th>rank_f</th>
<th>rank_t</th>
<th>rank_u</th>
<th>rank_ur</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>6.5</td>
<td>4.5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>6.5</td>
<td>4.5</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>8.5</td>
<td>2.5</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>22</td>
<td>8.5</td>
<td>2.5</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Standardized variables

**Example 8: std()**

We have a variable called age recording the median age in the 50 states. We wish to create the standardized value of age and verify the calculation:

. use https://www.stata-press.com/data/r17/states1, clear
(State data)
. egen stdage = std(age)
. summarize age stdage

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>50</td>
<td>29.54</td>
<td>1.693445</td>
<td>24.2</td>
<td>34.7</td>
</tr>
<tr>
<td>stdage</td>
<td>50</td>
<td>6.41e-09</td>
<td>-3.153336</td>
<td>3.047044</td>
<td></td>
</tr>
</tbody>
</table>

. correlate age stdage
(observ=50)

<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>stdage</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>stdage</td>
<td>1.0000</td>
<td></td>
</tr>
</tbody>
</table>

summarize shows that the new variable has a mean of approximately zero; $10^{-9}$ is the precision of a float and is close enough to zero for all practical purposes. If we wanted, we could have typed egen double stdage = std(age), making stdage a double-precision variable, and the mean would have been $10^{-16}$. In any case, summarize also shows that the standard deviation is 1. correlate shows that the new variable and the original variable are perfectly correlated.
We may optionally specify the mean and standard deviation for the new variable. For instance,

\[
\begin{align*}
\text{. egen newage1 = std(age), sd(2)} \\
\text{. egen newage2 = std(age), mean(2) sd(4)} \\
\text{. egen newage3 = std(age), mean(2)}
\end{align*}
\]

\[
\text{. summarize age newage1-newage3}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>50</td>
<td>29.54</td>
<td>1.693445</td>
<td>24.2</td>
<td>34.7</td>
</tr>
<tr>
<td>newage1</td>
<td>50</td>
<td>1.28e-08</td>
<td>2</td>
<td>-6.306671</td>
<td>6.094089</td>
</tr>
<tr>
<td>newage2</td>
<td>50</td>
<td>2</td>
<td>4</td>
<td>-10.61334</td>
<td>14.18818</td>
</tr>
<tr>
<td>newage3</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>-1.153336</td>
<td>5.047044</td>
</tr>
</tbody>
</table>

\[
\text{. correlate age newage1-newage3}
\]

(\text{obs=50})

<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>newage1</th>
<th>newage2</th>
<th>newage3</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>newage1</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>newage2</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>newage3</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

### Row functions

**Example 9: rowtotal()**

generate’s sum() function creates the vertical, running sum of its argument, whereas egen’s total() function creates a constant equal to the overall sum. egen’s rowtotal() function, however, creates the horizontal sum of its arguments. They all treat missing as zero. However, if the missing option is specified with total() or rowtotal(), then newvar will contain missing values if all values of exp or varlist are missing.

\[
\text{. use https://www.stata-press.com/data/r17/egenxmpl4, clear}
\]

\[
\text{. egen hsum = rowtotal(a b c)}
\]

\[
\text{. generate vsum = sum(hsum)}
\]

\[
\text{. egen sum = total(hsum)}
\]

\[
\text{. list}
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>hsum</th>
<th>vsum</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>.</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>4</td>
<td>.</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>7</td>
<td>8</td>
<td>.</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>33</td>
<td>63</td>
</tr>
</tbody>
</table>
Example 10: rowmean(), rowmedian(), rowpctile(), rowsd(), and rownonmiss()

summarize displays the mean and standard deviation of a variable across observations; program writers can access the mean in \texttt{r(mean)} and the standard deviation in \texttt{r(sd)} (see \texttt{[R] summarize}). \texttt{egen}'s \texttt{rowmean()} function creates the means of observations across variables. \texttt{rowmedian()} creates the medians of observations across variables. \texttt{rowpctile()} returns the \texttt{#th} percentile of the variables specified in \texttt{varlist}. \texttt{rowsd()} creates the standard deviations of observations across variables. \texttt{rownonmiss()} creates a count of the number of nonmissing observations, the denominator of the \texttt{rowmean()} calculation:

\begin{verbatim}
. use https://www.stata-press.com/data/r17/egenxmpl4, clear
. egen avg = rowmean(a b c)
. egen median = rowmedian(a b c)
. egen pct25 = rowpctile(a b c), p(25)
. egen std = rowsd(a b c)
. egen n = rownonmiss(a b c)
. list
\end{verbatim}

\begin{verbatim}
+---+-----+-----+-----+-------+------+---+
|   | a   | b   | c   | avg   | median| n |
|---+-----+-----+-----+-------+------+---+
| 1.| .   | 2   | 3   | 2.5   | 2.5   | 2 |
| 2.| 4   | .   | 6   | 5     | 5     | 4 |
| 3.| 7   | 8   | .   | 7.5   | 7.5   | 7 |
| 4.| 10  | 11  | 12  | 11    | 11    | 10|
\end{verbatim}

Example 11: rowmiss()

\texttt{rowmiss()} returns \(k - \texttt{rownonmiss}()\), where \(k\) is the number of variables specified. \texttt{rowmiss()} can be especially useful for finding casewise-deleted observations caused by missing values.

\begin{verbatim}
. use https://www.stata-press.com/data/r17/auto3, clear
(1978 automobile data)
. correlate price weight mpg
(obs=70)

<table>
<thead>
<tr>
<th>price</th>
<th>weight</th>
<th>mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5309</td>
<td>1.0000</td>
<td></td>
</tr>
<tr>
<td>-0.4478</td>
<td>-0.7985</td>
<td>1.0000</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
. egen excluded = rowmiss(price weight mpg)
. list make price weight mpg if excluded=0
\end{verbatim}

\begin{verbatim}
+---------+-------+-------+-------+
| make    | price | weight| mpg   |
|---------+-------+-------+-------|
| 5.      | Buick Electra | 4,080 | 15    |
| 12.     | Cad. Eldorado | 14,500| 3,900 |
| 40.     | Olds Starfire | 4,195 | .     |
\end{verbatim}
Example 12: rowmin(), rowmax(), rowfirst(), and rowlast()

`rowmin()`, `rowmax()`, `rowfirst()`, and `rowlast()` return the minimum, maximum, first, or last nonmissing value, respectively, for the specified variables within an observation (row).

```
use https://www.stata-press.com/data/r17/egenxmpl5, clear
egen min = rmin(x y z)
(1 missing value generated)
egen max = rmax(x y z)
(1 missing value generated)
egen first = rfirst(x y z)
(1 missing value generated)
egen last = rlast(x y z)
(1 missing value generated)
list, sep(4)
```

```
x  y  z  min  max  first  last
 1.  -1  2  3  -1   3  -1   3
 2.   .  -6  .  -6  -6  -6  -6
 3.   7  .  -5  -5   7   7  -5
 4.   .  .  .  .   .   .   .
 5.   4  .  .   4   4   4   4
 6.   .  .  8  8   8   8   8
 7.   .  3  7  3   7   3   7
 8.   5 -1  6 -1   6   5   6
```

Categorical and integer variables

Example 13: anyvalue(), anymatch(), and anycount()

`anyvalue()`, `anymatch()`, and `anycount()` are for categorical or other variables taking integer values. If we define a subset of values specified by an integer `numlist` (see [U] 11.1.8 `numlist`), `anyvalue()` extracts the subset, leaving every other value missing; `anymatch()` defines an indicator variable (1 if in subset, 0 otherwise); and `anycount()` counts occurrences of the subset across a set of variables. Therefore, with just one variable, `anymatch(varname)` and `anycount(varname)` are equivalent.

With the `auto` dataset, we can generate a variable containing the high values of `rep78` and a variable indicating whether `rep78` has a high value:

```
use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
egen hirep = anyvalue(rep78), v(3/5)
(15 missing values generated)
egen ishirep = anymatch(rep78), v(3/5)
```

Here it is easy to produce the same results with official Stata commands:

```
generate hirep = rep78 if inlist(rep78,3,4,5)
generate byte ishirep = inlist(rep78,3,4,5)
```
However, as the specification becomes more complicated or involves several variables, the `egen` functions may be more convenient.

> Example 14: `group()`

`group()` maps the distinct groups of a `varlist` to a categorical variable that takes on integer values from 1 to the total number of groups. The order of the groups is that of the sort order of `varlist`. The `varlist` may be of numeric variables, string variables, or a mixture of the two. The resulting variable can be useful for many purposes, including stepping through the distinct groups easily and systematically and cleaning up an untidy ordering. Suppose that the actual (and arbitrary) codes present in the data are 1, 2, 4, and 7, but we desire equally spaced numbers, as when the codes will be values on one axis of a graph. `group()` maps these to 1, 2, 3, and 4.

We have a variable `agegrp` that takes on the values 24, 40, 50, and 65, corresponding to age groups 18–24, 25–40, 41–50, and 51 and above. Perhaps we created this coding using the `recode()` function (see ["U 13.3 Functions and "U 26 Working with categorical data and factor variables"] from another age-in-years variable:

```
. generate agegrp=recode(age,24,40,50,65)
```

We now want to change the codes to 1, 2, 3, and 4:

```
. egen agegrp2 = group(agegrp)
```

> Example 15: `group()` with missing values

We have two categorical variables, `race` and `sex`, which may be string or numeric. We want to use `ir` (see [R Epitab]) to create a Mantel–Haenszel weighted estimate of the incidence rate. `ir`, however, allows only one variable to be specified in its `by()` option. We type

```
. use https://www.stata-press.com/data/r17/egenxmpl6, clear
. egen racesex = group(race sex)
  (2 missing values generated)
. ir deaths smokes pyears, by(racesex)
  (output omitted)
```

The new numeric variable, `racesex`, will be missing wherever `race` or `sex` is missing (meaning . for numeric variables and "" for string variables), so missing values will be handled correctly. When we list some of the data, we see

```
. list race sex racesex in 1/7, sep(0)
+-------------+---------+-----------+
<table>
<thead>
<tr>
<th>race</th>
<th>sex</th>
<th>racesex</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Female</td>
<td>1</td>
</tr>
<tr>
<td>White</td>
<td>Male</td>
<td>2</td>
</tr>
<tr>
<td>Black</td>
<td>Female</td>
<td>3</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>.</td>
<td>Female</td>
<td>.</td>
</tr>
<tr>
<td>Black</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
+-------------+---------+-----------+
```

group() began by putting the data in the order of the grouping variables and then assigned the numeric codes. Observations 6 and 7 were assigned to `racesex = .` because, in one case, `race` was not known, and in the other, `sex` was not known. (These observations were not used by `ir`.)
If we wanted the unknown groups to be treated just as any other category, we could have typed

```stata
. egen rs2 = group(race sex), missing
. list race sex rs2 in 1/7, sep(0)
```

<table>
<thead>
<tr>
<th>race</th>
<th>sex</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Female</td>
<td>1</td>
</tr>
<tr>
<td>White</td>
<td>Male</td>
<td>2</td>
</tr>
<tr>
<td>Black</td>
<td>Female</td>
<td>3</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>.</td>
<td>Female</td>
<td>6</td>
</tr>
<tr>
<td>Black</td>
<td>.</td>
<td>5</td>
</tr>
</tbody>
</table>

The resulting variable from `group()` does not have value labels. Therefore, the values carry no indication of meaning. Interpretation requires comparison with the original `varlist`. To get value labels, we specify the option `label`.

```stata
. egen rs3 = group(race sex), missing label
. list race sex rs3 in 1/7, sep(0)
```

<table>
<thead>
<tr>
<th>race</th>
<th>sex</th>
<th>rs3</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Female</td>
<td>White Female</td>
</tr>
<tr>
<td>White</td>
<td>Male</td>
<td>White Male</td>
</tr>
<tr>
<td>Black</td>
<td>Female</td>
<td>Black Female</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>Black Male</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>Black Male</td>
</tr>
<tr>
<td>.</td>
<td>Female</td>
<td>. Female</td>
</tr>
<tr>
<td>Black</td>
<td>.</td>
<td>. Black</td>
</tr>
</tbody>
</table>

The numeric values of the generated variable `rs3` are the same as `rs2`, but `rs3` has a value label that indicates the categories of `race` and `sex` that define the groups. The value label created by `group()` uses the actual values of the categorical variables or their value labels, if they exist. In this case, the categorical variables `race` and `sex` are numeric variables with value labels, so their value labels were used to create the value label for `rs3`.

### String variables

Concatenation of string variables is provided in Stata. In context, Stata understands the addition symbol `+` as specifying concatenation or adding strings end to end. "soft" + "ware" produces "software", and given string variables `s1` and `s2`, `s1 + s2` indicates their concatenation.

The complications that may arise in practice include wanting 1) to concatenate the string versions of numeric variables and 2) to concatenate variables, together with some separator such as a space or a comma. Given numeric variables `n1` and `n2`,

```stata
. generate newstr = s1 + string(n1) + string(n2) + s2
```

shows how numeric values may be converted to their string equivalents before concatenation, and

```stata
. generate newstr = s1 + " " + s2 + " " + s3
```

shows how spaces may be added between variables. Stata will automatically assign the most appropriate data type for the new string variables.
Example 16: concat()

concat() allows us to do everything in one line concisely.

. egen newstr = concat(s1 n1 n2 s2)

carries with it an implicit instruction to convert numeric values to their string equivalents, and the appropriate string data type is worked out within concat() by Stata's automatic promotion. Moreover,

. egen newstr = concat(s1 s2 s3), p(" ")
specifies that spaces be used as separators. (The default is to have no separation of concatenated strings.)

As an example of punctuation other than a space, consider

. egen fullname = concat(surname forename), p("", "")

Noninteger numerical values can cause difficulties, but

. egen newstr = concat(n1 n2), format(%9.3f) p(" ")
specifies the use of format %9.3f. This is equivalent to

. generate str1 newstr = ""
. replace newstr = string(n1,"%9.3f") + " " + string(n2,"%9.3f")

See [FN] String functions for more about string().

As a final flourish, the decode option instructs concat() to use value labels. With that option, the maxlength() option may also be used. For more details about decode, see [D] encode. Unlike the decode command, however, concat() uses string(varname), not "", whenever values of varname are not associated with value labels, and the format() option, whenever specified, applies to this use of string().

Example 17: ends()

The ends(strvar) function is used for subdividing strings. The approach is to find specified separators by using the strpos() string function and then to extract what is desired, which either precedes or follows the separators, using the substr() string function.

By default, substrings are considered to be separated by individual spaces, so we will give definitions in those terms and then generalize.

The head of the string is whatever precedes the first space or is the whole of the string if no space occurs. This could also be called the first “word”. The tail of the string is whatever follows the first space. This could be nothing or one or more words. The last word in the string is whatever follows the last space or is the whole of the string if no space occurs.

To clarify, let's look at some examples. The quotation marks here just mark the limits of each string and are not part of the strings.

<table>
<thead>
<tr>
<th></th>
<th>head</th>
<th>tail</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;frog&quot;</td>
<td>&quot;frog&quot;</td>
<td>'&quot;'</td>
<td>&quot;frog&quot;</td>
</tr>
<tr>
<td>&quot;frog toad&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad&quot;</td>
<td>&quot;toad&quot;</td>
</tr>
<tr>
<td>&quot;frog toad newt&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad newt&quot;</td>
<td>&quot;newt&quot;</td>
</tr>
<tr>
<td>&quot;frog toad newt&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad newt&quot;</td>
<td>&quot;newt&quot;</td>
</tr>
<tr>
<td>&quot;frog toad newt&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad newt&quot;</td>
<td>&quot;newt&quot;</td>
</tr>
</tbody>
</table>
The main subtlety is that these functions are literal, so the tail of "frog toad newt", in which two spaces follow "frog", includes the second of those spaces, and is thus "toad newt". Therefore, you may prefer to use the trim option to trim the result of any leading or trailing spaces, producing "toad newt" in this instance.

The punct(pchars) option may be used to specify separators other than spaces. The general definitions of the head, tail, and last options are therefore interpreted in terms of whatever separator has been specified; that is, they are relative to the first or last occurrence of the separator in the string value. Thus, with punct(,) and the string "Darwin, Charles Robert", the head is "Darwin", and the tail and the last are both "Charles Robert". Note again the leading space in this example, which may be trimmed with trim. The punctuation (here the comma,“,,”) is discarded, just as it is with one space.

pchars, the argument of punct(), will usually, but not always, be one character. If two or more characters are specified, these must occur together; for example, punct(,:;) would mean that words are separated by a colon followed by a semicolon (that is, :;). It is not implied, in particular, that the colon and semicolon are alternatives. To do that, you would have to modify the programs presented here or resort to first principles by using split; see [D] split.

With personal names, the head or last option might be applied to extract surnames if strings were similar to "Darwin, Charles Robert" or "Charles Robert Darwin", with the surname coming first or last. What then happens with surnames like "von Neumann" or "de la Mare"? "von Neumann, John" is no problem, if the comma is specified as a separator, but the last option is not intelligent enough to handle "Walter de la Mare" properly.

U.S. marginal income tax rate

mtr(year income) (Schmidt 1993, 1994) returns the U.S. marginal income tax rate for a married couple with taxable income income in year year, where 1930 ≤ year ≤ 2019.

Example 18: mtr()

Schmidt (1993) examines the change in the progressivity of the U.S. tax schedule over the period from 1930 to 1990. As a measure of progressivity, he calculates the difference in the marginal tax rates at the 75th and 25th percentiles of income, using a dataset of percentiles of taxable income developed by Hakkio, Rush, and Schmidt (1996). (Certain aspects of the income distribution are imputed in these data.) A subset of the data contains the following:

```
.m. describe
Contains data from incomel.dta
Observations: 61
Variables: 4
12 Feb 2021 03:33

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>float</td>
<td>%9.0g</td>
<td>Year</td>
</tr>
<tr>
<td>inc25</td>
<td>float</td>
<td>%9.0g</td>
<td>25th percentile</td>
</tr>
<tr>
<td>inc50</td>
<td>float</td>
<td>%9.0g</td>
<td>50th percentile</td>
</tr>
<tr>
<td>inc75</td>
<td>float</td>
<td>%9.0g</td>
<td>75th percentile</td>
</tr>
</tbody>
</table>
```

Sorted by:
Given the series for income and the four-digit year, we can generate the marginal tax rates corresponding to the 25th and 75th percentiles of income:

- `egen mtr25 = mtr(year inc25)`
- `egen mtr75 = mtr(year inc75)`
- `summarize mtr25 mtr75`

### Methods and formulas

Stata users have written many extra functions for `egen`. Type `net search egen` to locate Internet sources of programs.

### Acknowledgments

The `mtr()` function of `egen` was written by Timothy J. Schmidt formerly with the Federal Reserve Bank of Kansas City.

The `cut()` function was written by David Clayton (retired) of the Cambridge Institute for Medical Research and Michael Hills (retired) of the London School of Hygiene and Tropical Medicine.

Many of the other `egen` functions were written by Nicholas J. Cox of the Department of Geography at Durham University, UK, and coeditor of the `Stata Journal` and author of `Speaking Stata Graphics`.

### References


https://doi.org/10.1214/ss/1028905831.


Also see

[D] collapse — Make dataset of summary statistics
[D] generate — Create or change contents of variable
[U] 13.3 Functions
**encode — Encode string into numeric and vice versa**

**Description**

`encode` creates a new variable named `newvar` based on the string variable `varname`, creating, adding to, or just using (as necessary) the value label `newvar` or, if specified, `name`. Do not use `encode` if `varname` contains numbers that merely happen to be stored as strings; instead, use `generate newvar =real(varname)` or `destring`; see [U] 24.2 Categorical string variables, [FN] String functions, and [D] `destring`.

`decode` creates a new string variable named `newvar` based on the “encoded” numeric variable `varname` and its value label.

**Quick start**

Generate numeric `newv1` from string `v1`, using the values of `v1` to create a value label that is applied to `newv1`

```stata
encode v1, generate(newv1)
```

As above, but name the value label `mylabel1`

```stata
encode v1, generate(newv1) label(mylabel1)
```

As above, but refuse to encode `v1` if values exist in `v1` that are not present in preexisting value label `mylabel1`

```stata
encode v1, generate(newv1) label(mylabel1) noextend
```

Convert numeric `v2` to string `newv2` using the value label applied to `v2` to generate values of `newv2`

```stata
decode v2, generate(newv2)
```

**Menu**

- `encode`
  - Data > Create or change data > Other variable-transformation commands > Encode value labels from string variable
- `decode`
  - Data > Create or change data > Other variable-transformation commands > Decode strings from labeled numeric variable

257
Syntax

String variable to numeric variable

```
encode varname [if] [in], generate(newvar) [label(name) noextend]
```

Numeric variable to string variable

```
decode varname [if] [in], generate(newvar) [maxlength(#)]
```

Options for encode

`generate(newvar)` is required and specifies the name of the variable to be created.

`label(name)` specifies the name of the value label to be created or used and added to if the named value label already exists. If `label()` is not specified, `encode` uses the same name for the label as it does for the new variable.

`noextend` specifies that `varname` not be encoded if there are values contained in `varname` that are not present in `label(name)`. By default, any values not present in `label(name)` will be added to that label.

Options for decode

`generate(newvar)` is required and specifies the name of the variable to be created.

`maxlength(#)` specifies how many bytes of the value label to retain; # must be between 1 and 32,000. The default is `maxlength(32000)`.

Remarks and examples

Remarks are presented under the following headings:

`encode`  
`decode`  
`Video example`

**encode**

`encode` is most useful in making string variables accessible to Stata’s statistical routines, most of which can work only with numeric variables. `encode` is also useful in reducing the size of a dataset. If you are not familiar with value labels, read [U] 12.6.3 Value labels.

The maximum number of associations within each value label is 65,536. Each association in a value label maps a string of up to 32,000 bytes to a number. For plain ASCII text, the number of bytes is equal to the number of characters. If your string has other Unicode characters, the number of bytes is greater than the number of characters. See [U] 12.4.2 Handling Unicode strings. If your variable contains string values longer than 32,000 bytes, then only the first 32,000 bytes are retained and assigned as a value label to a number.
Example 1

We have a dataset on high blood pressure, and among the variables is `sex`, a string variable containing either “male” or “female”. We wish to run a regression of high blood pressure on race, sex, and age group. We type `regress hbp race sex age_grp` and get the message “no observations”.

```
use https://www.stata-press.com/data/r17/hbp2
regress hbp sex race age_grp
no observations
r(2000);
```

Stata’s statistical procedures cannot directly deal with string variables; as far as they are concerned, all observations on `sex` are missing. `encode` provides the solution:

```
encode sex, gen(gender)
regress hbp gender race age_grp
```

```
Source       | SS    | df   | MS     | Number of obs = 1,121  
Model        | 2.01013476 | 3 | .67004492 | F(3, 1117) = 15.15  
Residual     | 49.3886164 | 1,117 | .044215413 | Prob > F = 0.0000  
Total        | 51.3987511 | 1,120 | .045891742 | R-squared = 0.0391  
Adj R-squared | .0365  
Root MSE     | .21027

hbp          | Coefficient| Std. err. | t     | P>|t|   | [95% conf. interval]   
gender       | .0394747   | .0130022 | 3.04 | 0.002 | .0139633 -.0649861    
race         | -.0409453  | .0113721 | -3.60 | 0.000 | -.0632584 -.0186322   
age_grp      | .0241484   | .00624  | 3.87 | 0.000 | .0119049 .0363919     
_cons         | -.016815   | .0389167 | -0.43 | 0.666 | -.093173 .059543      
```

`encode` looks at a string variable and makes an internal table of all the values it takes on, here “male” and “female”. It then alphabetizes that list and assigns numeric codes to each entry. Thus 1 becomes “female” and 2 becomes “male”. It creates a new `int` variable (gender) and substitutes a 1 where `sex` is “female”, a 2 where `sex` is “male”, and a `missing` (.) where `sex` is `null` ("""). It creates a value label (also named `gender`) that records the mapping 1 ↔ female and 2 ↔ male. Finally, `encode` labels the values of the new variable with the value label.

Example 2

It is difficult to distinguish the result of `encode` from the original string variable. For instance, in our last two examples, we typed `encode sex, gen(gender)`. Let’s compare the two variables:

```
list sex gender in 1/4
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>female female</td>
</tr>
<tr>
<td>2.</td>
<td>.</td>
</tr>
<tr>
<td>3.</td>
<td>male  male</td>
</tr>
<tr>
<td>4.</td>
<td>male  male</td>
</tr>
</tbody>
</table>

They look almost identical, although you should notice the missing value for gender in the second observation.
The difference does show, however, if we tell `list` to ignore the value labels and show how the data really appear:

```
. list sex gender in 1/4, nolabel

<table>
<thead>
<tr>
<th>sex</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>female</td>
<td>1</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>male</td>
<td>2</td>
</tr>
<tr>
<td>male</td>
<td>2</td>
</tr>
</tbody>
</table>
```

We could also ask to see the underlying value label:

```
. label list gender

gender: 1 female
         2 male
```

gender really is a numeric variable, but because all Stata commands understand value labels, the variable displays as “male” and “female”, just as the underlying string variable `sex` would.

> Example 3

We can drastically reduce the size of our dataset by encoding strings and then discarding the underlying string variable. We have a string variable, `sex`, that records each person’s sex as “male” and “female”. Because female has six characters, the variable is stored as a `str6`.

We can encode the `sex` variable and use `compress` to store the variable as a byte, which takes only 1 byte. Because our dataset contains 1,130 people, the string variable takes 6,780 bytes, but the encoded variable will take only 1,130 bytes.

```
. use https://www.stata-press.com/data/r17/hbp2, clear
. describe
Contains data from https://www.stata-press.com/data/r17/hbp2.dta
Observations: 1,130
Variables: 7 3 Mar 2020 06:47

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>str10</td>
<td>%10s</td>
<td>Record identification number</td>
</tr>
<tr>
<td>city</td>
<td>byte</td>
<td>%8.0g</td>
<td>City</td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>%8.0g</td>
<td>Year</td>
</tr>
<tr>
<td>age_grp</td>
<td>byte</td>
<td>%8.0g</td>
<td>agefmt Age group</td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td>racefmt Race</td>
</tr>
<tr>
<td>hbp</td>
<td>byte</td>
<td>%8.0g</td>
<td>yn High blood pressure</td>
</tr>
<tr>
<td>sex</td>
<td>str6</td>
<td>%9s</td>
<td>Sex</td>
</tr>
</tbody>
</table>

Sorted by:
. encode sex, generate(gender)
```
. list sex gender in 1/5

<table>
<thead>
<tr>
<th></th>
<th>sex</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>female</td>
<td>female</td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>male</td>
<td>male</td>
</tr>
<tr>
<td>4.</td>
<td>male</td>
<td>male</td>
</tr>
<tr>
<td>5.</td>
<td>female</td>
<td>female</td>
</tr>
</tbody>
</table>

. drop sex
. rename gender sex
. compress

variable sex was long now byte
(3,390 bytes saved)

. describe
Contains data from https://www.stata-press.com/data/r17/hbp2.dta
Observations: 1,130
Variables: 7

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>str10</td>
<td>%10s</td>
<td>Record identification number</td>
</tr>
<tr>
<td>city</td>
<td>byte</td>
<td>%8.0g</td>
<td>City</td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>%8.0g</td>
<td>Year</td>
</tr>
<tr>
<td>age_grp</td>
<td>byte</td>
<td>%8.0g</td>
<td>agefmt Age group</td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td>racefmt Race</td>
</tr>
<tr>
<td>hbp</td>
<td>byte</td>
<td>%8.0g</td>
<td>yn High blood pressure</td>
</tr>
<tr>
<td>sex</td>
<td>byte</td>
<td>%8.0g</td>
<td>gender Sex</td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.

The size of our dataset has fallen from 24,860 bytes to 19,210 bytes.

⚠️ Technical note

In the examples above, the value label did not exist before `encode` created it, because that is not required. If the value label does exist, `encode` uses your encoding as far as it can and adds new mappings for anything not found in your value label. For instance, if you wanted “female” to be encoded as 0 rather than 1 (possibly for use in linear regression), you could type

. label define gender 0 "female"
. encode sex, gen(gender)

You can also specify the name of the value label. If you do not, the value label is assumed to have the same name as the newly created variable. For instance,

. label define sexlbl 0 "female"
. encode sex, gen(gender) label(sexlbl)
decode

decode is used to convert numeric variables with associated value labels into true string variables.

Example 4

We have a numeric variable named `female` that records the values 0 and 1. `female` is associated with a value label named `sexlbl` that says that 0 means male and 1 means female:

```
. use https://www.stata-press.com/data/r17/hbp3, clear
. describe female
```

```
Variable name Storage type Display format Value label Variable label
female byte %8.0g sexlbl Female
```

```
. label list sexlbl
sexlbl:
  0 Male
  1 Female
```

We see that `female` is stored as a byte. It is a numeric variable. Nevertheless, it has an associated value label describing what the numeric codes mean, so if we tabulate the variable, for instance, it appears to contain the strings “male” and “female”:

```
. tabulate female
```

```
Female Freq. Percent Cum.
Male 695 61.61 61.61
Female 433 38.39 100.00
Total 1,128 100.00
```

We can create a real string variable from this numerically encoded variable by using `decode`:

```
. decode female, gen(sex)
. describe sex
```

```
Variable name Storage type Display format Value label Variable label
sex str6 %9s Female
```

We have a new variable called `sex`. It is a string, and Stata automatically created the shortest possible string. The word “female” has six characters, so our new variable is a str6. `female` and `sex` appear indistinguishable:

```
. list female sex in 1/4
```

```
female  sex
1. Female Female
2. . .
3. Male Male
4. Male Male
```
But when we add nolabel, the difference is apparent:

```
. list female sex in 1/4, nolabel

<table>
<thead>
<tr>
<th>female</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Female</td>
</tr>
<tr>
<td>0</td>
<td>Male</td>
</tr>
</tbody>
</table>
```

Example 5

decode is most useful in instances when we wish to match-merge two datasets on a variable that has been encoded inconsistently.

For instance, we have two datasets on individual states in which one of the variables (state) takes on values such as “CA” and “NY”. The state variable was originally a string, but along the way the variable was encoded into an integer with a corresponding value label in one or both datasets.

We wish to merge these two datasets, but either 1) one of the datasets has a string variable for state and the other an encoded variable or 2) although both are numeric, we are not certain that the codings are consistent. Perhaps “CA” has been coded 5 in one dataset and 6 in another.

Because decode will take an encoded variable and turn it back into a string, decode provides the solution:

```
use first  
decode state, gen(st)  
drop state  
sort st  
save first, replace  
use second  
decode state, gen(st)  
drop state  
sort st  
merge 1:1 st using first
```

Video example

How to convert categorical string variables to labeled numeric variables

References


Also see

[D] compress — Compress data in memory
[D] destring — Convert string variables to numeric variables and vice versa
[D] generate — Create or change contents of variable
[U] 12.6.3 Value labels
[U] 24.2 Categorical string variables
**Description**

The `erase` command erases files stored on disk. `rm` is a synonym for `erase` for the convenience of Mac and Unix users.

Stata for Mac users: `erase` is permanent; the file is not moved to the Trash but is immediately removed from the disk.

Stata for Windows users: `erase` is permanent; the file is not moved to the Recycle Bin but is immediately removed from the disk.

**Quick start**

Delete `mylog.smcl` from current directory in Stata for Windows

```
erase mylog.smcl
```

Same as above for Mac and Unix

```
rm mylog.smcl
```

Delete `mydata.dta` from current directory in Stata for Windows

```
erase mydata.dta
```

Same as above for Mac and Unix

```
rm mydata.dta
```

Delete `mylog.smcl` from `C:\my dir\my folder` in Stata for Windows

```
erase "c:\my dir\my folder\mylog.smcl"
```

Same as above for Mac and Unix

```
rm "-/my dir/my folder/mylog.smcl"
```

**Syntax**

```
{ erase | rm } ["filename"]
```

Note: Double quotes must be used to enclose `filename` if the name contains spaces.

**Remarks and examples**

The only difference between Stata’s `erase` (`rm`) command and the Windows command prompt `DEL` or Unix `rm(1)` command is that we may not specify groups of files. Stata requires that we erase files one at a time.

Mac users may prefer to discard files by dragging them to the Trash.

Windows users may prefer to discard files by dragging them to the Recycle Bin.
Example 1

Stata provides seven operating system equivalent commands: `cd`, `copy`, `dir`, `erase`, `mkdir`, `rmdir`, and `type`, or, from the Unix perspective, `cd`, `copy`, `ls`, `rm`, `mkdir`, `rmdir`, and `cat`. These commands are provided for Mac users, too. Stata users can also issue any operating system command by using Stata’s `shell` command, so you should never have to exit Stata to perform some housekeeping detail.

Suppose that we have the file `mydata.dta` stored on disk and we wish to permanently eliminate it:

```stata
. erase mydata
  file mydata not found
  r(601);
. erase mydata.dta
```

Our first attempt, `erase mydata`, was unsuccessful. Although Stata ordinarily supplies the file extension for you, it does not do so when you type `erase`. You must be explicit. Our second attempt eliminated the file. Unix users could have typed `rm mydata.dta` if they preferred.

Also see

[D] `cd` — Change directory
[D] `copy` — Copy file from disk or URL
[D] `dir` — Display filenames
[D] `mkdir` — Create directory
[D] `rmdir` — Remove directory
[D] `shell` — Temporarily invoke operating system
[D] `type` — Display contents of a file
[U] 11.6 Filenaming conventions
expand replaces each observation in the dataset with $n$ copies of the observation, where $n$ is equal to the required expression rounded to the nearest integer. If the expression is less than 1 or equal to `missing`, it is interpreted as if it were 1, and the observation is retained but not duplicated.

**Quick start**

Duplicate each observation 3 times, resulting in the original and 2 copies

```
expand 3
```

Duplicate each observation the number of times stored in $v$

```
expand v
```

As above, but flag duplicated observations using generated `newv`

```
expand v, generate(newv)
```

As above, but only duplicate observations where `catvar` equals 4

```
expand v if catvar==4, generate(newv)
```

**Menu**

Data > Create or change data > Other variable-transformation commands > Duplicate observations
Syntax

```
expand [ =] exp [ if ] [ in ] [ , generate( newvar ) ]
```

Option

`generate( newvar )` creates new variable `newvar` containing 0 if the observation originally appeared in the dataset and 1 if the observation is a duplicate. For instance, after an `expand`, you could revert to the original observations by typing `keep if newvar==0`.

Remarks and examples

### Example 1

`expand` is, admittedly, a strange command. It can, however, be useful in tricky programs or for reformatting data for survival analysis (see examples in [R] Epitab). Here is a silly use of `expand`:

```
. use https://www.stata-press.com/data/r17/expandxmpl
. list
    n  x
 1. -1 1
 2.  0 2
 3.  1 3
 4.  2 4
 5.  3 5
. expand n
(1 negative count ignored; observation not deleted)
(1 zero count ignored; observation not deleted)
(3 observations created)
. list
   n  x
 1. -1 1
 2.  0 2
 3.  1 3
 4.  2 4
 5.  3 5
 6.  2 4
 7.  3 5
 8.  3 5
```

The new observations are added to the end of the dataset. `expand` informed us that it created 3 observations. The first 3 observations were not replicated because `n` was less than or equal to 1. `n` is 2 in the fourth observation, so `expand` created one replication of this observation, bringing the total number of observations of this type to 2. `expand` created two replications of observation 5 because `n` is 3.

Because there were 5 observations in the original dataset and because `expand` adds new observations onto the end of the dataset, we could now undo the expansion by typing `drop in 6/1`. 
References


Also see

[D] `contract` — Make dataset of frequencies and percentages
[D] `expandcl` — Duplicate clustered observations
[D] `fillin` — Rectangularize dataset
**expandcl** — Duplicate clustered observations

### Description

`expandcl` duplicates clusters of observations and generates a new variable that identifies the clusters uniquely.

`expandcl` replaces each cluster in the dataset with $n$ copies of the cluster, where $n$ is equal to the required expression rounded to the nearest integer. The expression is required to be constant within cluster. If the expression is less than 1 or equal to `missing`, it is interpreted as if it were 1, and the cluster is retained but not duplicated.

### Quick start

Duplicate each set of observations on clusters identified by `cvar` 3 times, and store new cluster identifier in `newcv`

```
expandcl 3, cluster(cvar) generate(newcv)
```

Duplicate each cluster of observations the number of times stored in `v`

```
expandcl `v`, cluster(cvar) generate(newcv)
```

### Menu

Data > Create or change data > Other variable-transformation commands > Duplicate clustered observations
Syntax

```
expandcl [ = ] exp [ if ] [ in ], cluster(varlist) generate(newvar)
```

Options

`cluster(varlist)` is required and specifies the variables that identify the clusters before expanding the data.

`generate(newvar)` is required and stores unique identifiers for the duplicated clusters in `newvar`. `newvar` will identify the clusters by using consecutive integers starting from 1.

Remarks and examples

Example 1

We will show how `expandcl` works by using a small dataset with five clusters. In this dataset, cl identifies the clusters, x contains a unique value for each observation, and n identifies how many copies we want of each cluster.

```
. use https://www.stata-press.com/data/r17/expclxmpl
. list, sepby(cl)
    +---------+--+-+--
   | cl | x | n |
   +---------+--+-+--
   | 1 | 10 | 1 | 1 |
   | 2 | 10 | 2 | 1 |
   | 3 | 20 | 3 | 0 |
   | 4 | 20 | 4 | 0 |
   | 5 | 30 | 5 | 1 |
   | 6 | 30 | 6 | 1 |
   | 7 | 40 | 7 | 2.7 |
   | 8 | 40 | 8 | 2.7 |
   | 9 | 50 | 9 | 3 |
   | 10 | 50 | 10 | 3 |
   | 11 | 60 | 11 | . |
   | 12 | 60 | 12 | . |
   +---------+--+-+--
```

. expandcl n, generate(newcl) cluster(cl)
   (2 missing counts ignored; observations not deleted)
   (2 noninteger counts rounded to integer)
   (2 negative counts ignored; observations not deleted)
   (2 zero counts ignored; observations not deleted)
   (8 observations created)
. sort newcl cl x
The first three clusters were not replicated because \( n \) was less than or equal to 1. \( n \) is 2.7 in the fourth cluster, so `expandcl` created two replications (2.7 was rounded to 3) of this cluster, bringing the total number of clusters of this type to 3. `expandcl` created two replications of cluster 50 because \( n \) is 3. Finally, `expandcl` did not replicate the last cluster because \( n \) was missing.

Also see

[D] `expand` — Duplicate observations
[R] `bsample` — Sampling with replacement
Description

This entry provides a quick reference for determining which method to use for exporting Stata data from memory to other formats.

Remarks and examples

Remarks are presented under the following headings:

Summary of the different methods
- export excel
- export delimited
- jdbc
- odbc
- outfile
- export sasxport5 and export sasxport8
- export dbase

Summary of the different methods

export excel

- export excel creates Microsoft Excel worksheets in .xls and .xlsx files.
- Entire worksheets can be exported, or custom cell ranges can be overwritten.
- See [D] import excel.

export delimited

- export delimited creates comma-separated or tab-delimited files that many other programs can read.
- A custom delimiter may also be specified.
- The first line of the file can optionally contain the names of the variables.
- See [D] import delimited.

jdbc

- Java Database Connectivity (JDBC) is an application programming interface for the programming language Java. The jdbc command allows you to connect to, load data from, insert data into, and execute queries on a database using JDBC.
- See [D] jdbc.
export — Overview of exporting data from Stata

odbc

- ODBC, an acronym for Open DataBase Connectivity, is a standard for exchanging data between programs. Stata supports the ODBC standard for exporting data via the odbc command and can write to any ODBC data source on your computer.
  - See [D] odbc.

outfile

- outfile creates text-format datasets.
  - The data can be written in space-separated or comma-separated format.
  - Alternatively, the data can be written in fixed-column format.
  - See [D] outfile.

export sasxport5 and export sasxport8

- export sasxport5 saves SAS XPORT Version 5 Transport format files.
  - export sasxport5 can also write value-label information to a formats.xpf XPORT file.
- export sasxport8 saves SAS XPORT Version 8 Transport format files.
  - export sasxport8 can also write value-label information to a SAS command (.sas) file.
  - See [D] import sasxport5 and [D] import sasxport8.

export dbase

- export dbase saves version IV dBase (.dbf) files.
  - See [D] import dbase.

Also see

[D] import — Overview of importing data into Stata
[M-5] _docx*() — Generate Office Open XML (.docx) file
[M-5] xl() — Excel file I/O class
[RPT] dyndoc — Convert dynamic Markdown document to HTML or Word (.docx) document
[RPT] putdocx intro — Introduction to generating Office Open XML (.docx) files
[RPT] putexcel — Export results to an Excel file
[RPT] putpdf intro — Introduction to generating PDF files
filefilter — Convert ASCII or binary patterns in a file

Description

filefilter reads an input file, searching for oldpattern. Whenever a matching pattern is found, it is replaced with newpattern. All resulting data, whether matching or nonmatching, are then written to the new file.

Because of the buffering design of filefilter, arbitrarily large files can be converted quickly. filefilter is also useful when traditional editors cannot edit a file, such as when unprintable ASCII characters are involved. In fact, converting end-of-line characters between Macintosh, Windows, and Unix is convenient with the EOL codes.

Unicode is not directly supported, but UTF-8 encoded files can be operated on by using byte-sequence methods in some cases.

Although it is not mandatory, you may want to use quotes to delimit a pattern, protecting the pattern from Stata’s parsing routines. A pattern that contains blanks must be in quotes.

Quick start

Create newfile.txt from oldfile.txt by replacing all tabs with semicolons
    filefilter oldfile.txt newfile.txt, from(\t) to(;;)

Create newfile.txt from oldfile.txt by replacing all instances of “The” with “the”
    filefilter oldfile.txt newfile.txt, from("The") to("the")
**Syntax**

```
filefilter oldfile newfile,
   \{ from(oldpattern) to(newpattern) | ascii2ebcdic | ebcdic2ascii \} [ options ]
```

where `oldpattern` and `newpattern` for ASCII characters are

```
"string" or string

string  ::= [char[char[...]]]
char     ::= regchar | code
regchar  ::= ASCII 32–91, 93–127, or extended ASCII 128, 161–255; excludes ‘\’

options Description
\* from(oldpattern) find `oldpattern` to be replaced
\* to(newpattern) use `newpattern` to replace occurrences of `from()`
\* ascii2ebcdic convert file from ASCII to EBCDIC
\* ebcdic2ascii convert file from EBCDIC to ASCII
replace replace `newfile` if it already exists
```

* Both `from(oldpattern)` and `to(newpattern)` are required, or `ascii2ebcdic` or `ebcdic2ascii` is required. collect is allowed; see [U] 11.1.10 Prefix commands.

**Options**

`from(oldpattern)` specifies the pattern to be found and replaced. It is required unless `ascii2ebcdic` or `ebcdic2ascii` is specified.

`to(newpattern)` specifies the pattern used to replace occurrences of `from()`. It is required unless `ascii2ebcdic` or `ebcdic2ascii` is specified.

`ascii2ebcdic` specifies that characters in the file be converted from ASCII coding to EBCDIC coding. `from()`, `to()`, and `ebcdic2ascii` are not allowed with `ascii2ebcdic`.

`ebcdic2ascii` specifies that characters in the file be converted from EBCDIC coding to ASCII coding. `from()`, `to()`, and `ascii2ebcdic` are not allowed with `ebcdic2ascii`.

replace specifies that `newfile` be replaced if it already exists.
Remarks and examples

Convert Classic Mac-style EOL characters to Windows-style

    . filefilter macfile.txt winfile.txt, from(\M) to(\W) replace

Convert left quote (‘) characters to the string “left quote”

    . filefilter autol.csv autol2.csv, from(\LQ) to("left quote")

Convert the character with hexadecimal code 60 to the string “left quote”

    . filefilter autol.csv autol2.csv, from(\60h) to("left quote")

Convert the character with decimal code 96 to the string “left quote”

    . filefilter autol.csv autol2.csv, from(\096d) to("left quote")

Convert strings beginning with hexadecimal code 6B followed by “Text” followed by decimal character 100 followed by “Text” to an empty string (remove them from the file)

    . filefilter file1.txt file2.txt, from(\6BhText\100dText") to(""

Convert file from EBCDIC to ASCII encoding

    . filefilter ebcdicfile.txt asciifile.txt, ebcdic2ascii

Technical note

Unicode is not directly supported, but you can try to operate on a UTF-8 encoded Unicode file by working on the byte sequence representation of the UTF-8 encoded Unicode character. For example, the Unicode character é, the Latin small letter “e” with an acute accent (Unicode code point \u00e9), has the byte sequence representation (195,169). You can obtain the byte sequence by using `tobytes("é")`. Although you may use 195 and 169 in `regchar` and `code`, they will be treated as two separate bytes instead of one character é (195 followed by 169). In short, this goes beyond the original design of the command and is technically unsupported. If you try to use filefilter in this way, you might encounter problems.

Stored results

    filefilter stores the following in r():

    Scalars
        r(occurrences)  number of oldpattern found
        r(bytes_from)   # of bytes represented by oldpattern
        r(bytes_to)     # of bytes represented by newpattern

Reference

Also see

[P] file — Read and write text and binary files
[D] changeeol — Convert end-of-line characters of text file
[D] hexdump — Display hexadecimal report on file
fillin — Rectangularize dataset

Description

fillin adds observations with missing data so that all interactions of varlist exist, thus making a complete rectangularization of varlist. fillin also adds the variable _fillin to the dataset. _fillin is 1 for observations created by using fillin and 0 for previously existing observations. varlist may not contain strLs.

Quick start

Add observations so that all possible interactions of v1 and v2 exist and flag new observations with _fillin = 1

fillin v1 v2

As above, but also include interactions with v3

fillin v1 v2 v3

Menu

Data > Create or change data > Other variable-transformation commands > Rectangularize dataset

Syntax

fillin varlist
Remarks and examples

Example 1

We have data on something by sex, race, and age group. We suspect that some of the combinations of sex, race, and age do not exist, but if so, we want them to exist with whatever remaining variables there are in the dataset set to missing. For example, rather than having a missing observation for black females aged 20–24, we want to create an observation that contains missing values:

```
. use https://www.stata-press.com/data/r17/fillin1
. list

<table>
<thead>
<tr>
<th>sex</th>
<th>race</th>
<th>age_gr-p</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>female</td>
<td>white</td>
<td>20-24</td>
<td>20393</td>
</tr>
<tr>
<td>2.</td>
<td>male</td>
<td>white</td>
<td>25-29</td>
<td>32750</td>
</tr>
<tr>
<td>3.</td>
<td>female</td>
<td>black</td>
<td>30-34</td>
<td>39399</td>
</tr>
</tbody>
</table>
```

```
. fillin sex race age_group
. list, sepby(sex)

<table>
<thead>
<tr>
<th>sex</th>
<th>race</th>
<th>age_gr-p</th>
<th>x1</th>
<th>x2</th>
<th>_fillin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>female</td>
<td>white</td>
<td>20-24</td>
<td>20393</td>
<td>14.5</td>
</tr>
<tr>
<td>2.</td>
<td>female</td>
<td>white</td>
<td>25-29</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3.</td>
<td>female</td>
<td>white</td>
<td>30-34</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4.</td>
<td>female</td>
<td>black</td>
<td>20-24</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5.</td>
<td>female</td>
<td>black</td>
<td>25-29</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6.</td>
<td>female</td>
<td>black</td>
<td>30-34</td>
<td>39399</td>
<td>14.2</td>
</tr>
<tr>
<td>7.</td>
<td>male</td>
<td>white</td>
<td>20-24</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8.</td>
<td>male</td>
<td>white</td>
<td>25-29</td>
<td>32750</td>
<td>12.7</td>
</tr>
<tr>
<td>9.</td>
<td>male</td>
<td>white</td>
<td>30-34</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>10.</td>
<td>male</td>
<td>black</td>
<td>20-24</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>11.</td>
<td>male</td>
<td>black</td>
<td>25-29</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>12.</td>
<td>male</td>
<td>black</td>
<td>30-34</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
```

References

Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. College Station, TX: Stata Press.


Also see

[D] cross — Form every pairwise combination of two datasets
[D] expand — Duplicate observations
[D] joinby — Form all pairwise combinations within groups
[D] save — Save Stata dataset
**format** — Set variables’ output format

### Description

format **varlist %fmt** and format **%fmt varlist** are the same commands. They set the display format associated with the variables specified. The default formats are a function of the type of the variable:

- byte %8.0g
- int %8.0g
- long %12.0g
- float %9.0g
- double %10.0g
- str# %#s
- strL %9s

**set dp** sets the symbol that Stata uses to represent the decimal point. The default is period, meaning that one and a half is displayed as 1.5.

**format [varlist]** displays the current formats associated with the variables. **format** by itself lists all variables that have formats too long to be listed in their entirety by **describe**. **format varlist** lists the formats for the specified variables regardless of their length. **format *** lists the formats for all the variables.

### Quick start

Show 10-digit v1 as whole numbers with commas

format v1 %15.0gc

Same as above

format %15.0gc v1

Left-align string variable v2 of type str20

format v2 %-20s

Show 3-digit v3 with 1 digit after the decimal

format v3 %4.1f

Left-align v4 and v5, and show with leading zeros if less than 4 digits in length

format v4 v5 %-04.0f

Show v6 in Stata default date format like 19jun2014

format v6 %td

As above, but show v6 in a date format like 06/14/2014

format v6 %tdNN/DD/CCYY

### Menu

Data > Variables Manager
Syntax

Set formats

format varlist %fmt

format %fmt varlist

Set style of decimal point

set dp \{ comma|period \} [, permanently]

Display long formats

format [ varlist ]

where %fmt can be a numerical, date, business calendar, or string format.

<table>
<thead>
<tr>
<th>Numerical %fmt</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>right-justified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%#.##g</td>
<td>general</td>
<td>%9.0g</td>
</tr>
<tr>
<td>%#.##f</td>
<td>fixed</td>
<td>%9.2f</td>
</tr>
<tr>
<td>%#.##e</td>
<td>exponential</td>
<td>%10.7e</td>
</tr>
<tr>
<td>%21x</td>
<td>hexadecimal</td>
<td>%21x</td>
</tr>
<tr>
<td>%16H</td>
<td>binary, hilo</td>
<td>%16H</td>
</tr>
<tr>
<td>%16L</td>
<td>binary, lohi</td>
<td>%16L</td>
</tr>
<tr>
<td>%8H</td>
<td>binary, hilo</td>
<td>%8H</td>
</tr>
<tr>
<td>%8L</td>
<td>binary, lohi</td>
<td>%8L</td>
</tr>
<tr>
<td>right-justified with commas</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%#.##gc</td>
<td>general</td>
<td>%9.0gc</td>
</tr>
<tr>
<td>%#.##fc</td>
<td>fixed</td>
<td>%9.2fc</td>
</tr>
<tr>
<td>right-justified with leading zeros</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%0#.##f</td>
<td>fixed</td>
<td>%09.2f</td>
</tr>
<tr>
<td>left-justified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%-.##g</td>
<td>general</td>
<td>%-9.0g</td>
</tr>
<tr>
<td>%-.##f</td>
<td>fixed</td>
<td>%-9.2f</td>
</tr>
<tr>
<td>%-.##e</td>
<td>exponential</td>
<td>%-10.7e</td>
</tr>
<tr>
<td>left-justified with commas</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%-#.##gc</td>
<td>general</td>
<td>%-9.0gc</td>
</tr>
<tr>
<td>%-#.##fc</td>
<td>fixed</td>
<td>%-9.2fc</td>
</tr>
</tbody>
</table>

You may substitute comma (,) for period (.) in any of the above formats to make comma the decimal point. In %9,2fc, 1000.03 is 1.000.03. Or you can set dp comma.
### Date Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%tc</td>
<td>date/time</td>
<td>%tc</td>
</tr>
<tr>
<td>%tC</td>
<td>date/time</td>
<td>%tC</td>
</tr>
<tr>
<td>%td</td>
<td>date</td>
<td>%td</td>
</tr>
<tr>
<td>%tw</td>
<td>week</td>
<td>%tw</td>
</tr>
<tr>
<td>%tm</td>
<td>month</td>
<td>%tm</td>
</tr>
<tr>
<td>%tq</td>
<td>quarter</td>
<td>%tq</td>
</tr>
<tr>
<td>%th</td>
<td>half-year</td>
<td>%th</td>
</tr>
<tr>
<td>%ty</td>
<td>year</td>
<td>%ty</td>
</tr>
<tr>
<td>%tg</td>
<td>generic</td>
<td>%tg</td>
</tr>
</tbody>
</table>

### Left-Justified Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%-tc</td>
<td>date/time</td>
<td>%-tc</td>
</tr>
<tr>
<td>%-tC</td>
<td>date/time</td>
<td>%-tC</td>
</tr>
<tr>
<td>%-td</td>
<td>date</td>
<td>%-td</td>
</tr>
</tbody>
</table>

There are many variations allowed. See [D] Datetime display formats.

### Business Calendar Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%tbc</td>
<td>a business calendar defined in calname.stbcal</td>
<td>%tbsimple</td>
</tr>
</tbody>
</table>

See [D] Datetime business calendars.

### String Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%#s</td>
<td>string</td>
<td>%15s</td>
</tr>
<tr>
<td>%-#s</td>
<td>string</td>
<td>%-20s</td>
</tr>
<tr>
<td>%-#s</td>
<td>string</td>
<td>%-12s</td>
</tr>
</tbody>
</table>

The centered format is for use with display only.

### Option

`permanent`ly specifies that, in addition to making the change right now, the dp setting be remembered and become the default setting when you invoke Stata.
Remarks and examples

Remarks are presented under the following headings:

Setting formats
Setting European formats
Details of formats
  The \%f format
  The \%fc format
  The \%g format
  The \%gc format
  The \%e format
  The \%21x format
  The \%16H and \%16L formats
  The \%8H and \%8L formats
  The \%t format
  The \%s format
Other effects of formats
Displaying current formats
Video example

Setting formats

See \[U\] 12.5 Formats: Controlling how data are displayed for an explanation of \%fmt. To review: Stata’s three numeric formats are denoted by a leading percent sign, \%, followed by the string w.d (or w,d for European format), where w and d stand for two integers. The first integer, w, specifies the width of the format. The second integer, d, specifies the number of digits that are to follow the decimal point; d must be less than w. Finally, a character denoting the format type (e, f, or g) is appended. For example, \%9.2f specifies the f format that is nine characters wide and has two digits following the decimal point. For f and g, a c may also be suffixed to indicate comma formats. Other “numeric” formats known collectively as the \%t formats are used to display dates and times; see \[D\] Datetime display formats. String formats are denoted by \%ws, where w indicates the width of the format.

Example 1

We have census data by region and state on median age and population in 1980.

. use https://www.stata-press.com/data/r17/census10
   (1980 Census data by state)
. describe

Contains data from https://www.stata-press.com/data/r17/census10.dta
Observations:  50  1980 Census data by state
Variables:  4  9 Apr 2020 08:05

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str14</td>
<td>%14s</td>
<td>State</td>
</tr>
<tr>
<td>region</td>
<td>int</td>
<td>%8.0g</td>
<td>Census region</td>
</tr>
<tr>
<td>pop</td>
<td>long</td>
<td>%11.0g</td>
<td>Population</td>
</tr>
<tr>
<td>medage</td>
<td>float</td>
<td>%9.0g</td>
<td>Median age</td>
</tr>
</tbody>
</table>

Sorted by:
. list in 1/8

<table>
<thead>
<tr>
<th>state</th>
<th>region</th>
<th>pop</th>
<th>medage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>South</td>
<td>3893888</td>
<td>29.3</td>
</tr>
<tr>
<td>Alaska</td>
<td>West</td>
<td>401851</td>
<td>26.1</td>
</tr>
<tr>
<td>Arizona</td>
<td>West</td>
<td>2718215</td>
<td>29.2</td>
</tr>
<tr>
<td>Arkansas</td>
<td>South</td>
<td>2286435</td>
<td>30.6</td>
</tr>
<tr>
<td>California</td>
<td>West</td>
<td>23667902</td>
<td>29.9</td>
</tr>
<tr>
<td>Colorado</td>
<td>West</td>
<td>2889964</td>
<td>28.6</td>
</tr>
<tr>
<td>Connecticut</td>
<td>NE</td>
<td>3107576</td>
<td>32</td>
</tr>
<tr>
<td>Delaware</td>
<td>South</td>
<td>594338</td>
<td>29.8</td>
</tr>
</tbody>
</table>

The `state` variable has a display format of `%14s`. To left-align the state data, we type

```
. format state %-14s
```

. list in 1/8

<table>
<thead>
<tr>
<th>state</th>
<th>region</th>
<th>pop</th>
<th>medage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>South</td>
<td>3893888</td>
<td>29.3</td>
</tr>
<tr>
<td>Alaska</td>
<td>West</td>
<td>401851</td>
<td>26.1</td>
</tr>
<tr>
<td>Arizona</td>
<td>West</td>
<td>2718215</td>
<td>29.2</td>
</tr>
<tr>
<td>Arkansas</td>
<td>South</td>
<td>2286435</td>
<td>30.6</td>
</tr>
<tr>
<td>California</td>
<td>West</td>
<td>23667902</td>
<td>29.9</td>
</tr>
<tr>
<td>Colorado</td>
<td>West</td>
<td>2889964</td>
<td>28.6</td>
</tr>
<tr>
<td>Connecticut</td>
<td>NE</td>
<td>3107576</td>
<td>32</td>
</tr>
<tr>
<td>Delaware</td>
<td>South</td>
<td>594338</td>
<td>29.8</td>
</tr>
</tbody>
</table>

Although it seems like `region` is a string variable, it is really a numeric variable with an attached value label. You do the same thing to left-align a numeric variable as you do a string variable: insert a negative sign.

```
. format region %-8.0g
```

. list in 1/8

<table>
<thead>
<tr>
<th>state</th>
<th>region</th>
<th>pop</th>
<th>medage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>South</td>
<td>3893888</td>
<td>29.3</td>
</tr>
<tr>
<td>Alaska</td>
<td>West</td>
<td>401851</td>
<td>26.1</td>
</tr>
<tr>
<td>Arizona</td>
<td>West</td>
<td>2718215</td>
<td>29.2</td>
</tr>
<tr>
<td>Arkansas</td>
<td>South</td>
<td>2286435</td>
<td>30.6</td>
</tr>
<tr>
<td>California</td>
<td>West</td>
<td>23667902</td>
<td>29.9</td>
</tr>
<tr>
<td>Colorado</td>
<td>West</td>
<td>2889964</td>
<td>28.6</td>
</tr>
<tr>
<td>Connecticut</td>
<td>NE</td>
<td>3107576</td>
<td>32</td>
</tr>
<tr>
<td>Delaware</td>
<td>South</td>
<td>594338</td>
<td>29.8</td>
</tr>
</tbody>
</table>
The pop variable would probably be easier to read if we inserted commas by appending a ‘c’:

```stata
.format pop %11.0gc
.list in 1/8
```

<table>
<thead>
<tr>
<th>state</th>
<th>region</th>
<th>pop</th>
<th>medage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>South</td>
<td>3,893,888</td>
<td>29.3</td>
</tr>
<tr>
<td>Alaska</td>
<td>West</td>
<td>401,851</td>
<td>26.1</td>
</tr>
<tr>
<td>Arizona</td>
<td>West</td>
<td>2,718,215</td>
<td>29.2</td>
</tr>
<tr>
<td>Arkansas</td>
<td>South</td>
<td>2,286,435</td>
<td>30.6</td>
</tr>
<tr>
<td>California</td>
<td>West</td>
<td>23,667,902</td>
<td>29.9</td>
</tr>
<tr>
<td>Colorado</td>
<td>West</td>
<td>2,889,964</td>
<td>28.6</td>
</tr>
<tr>
<td>Connecticut</td>
<td>NE</td>
<td>3,107,576</td>
<td>32.0</td>
</tr>
<tr>
<td>Delaware</td>
<td>South</td>
<td>594,338</td>
<td>29.8</td>
</tr>
</tbody>
</table>

Look at the value of pop for observation 5. There are no commas. This number was too large for Stata to insert commas and still respect the current width of 11. Let’s try again:

```stata
.format pop %12.0gc
.list in 1/8
```

<table>
<thead>
<tr>
<th>state</th>
<th>region</th>
<th>pop</th>
<th>medage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>South</td>
<td>3,893,888</td>
<td>29.3</td>
</tr>
<tr>
<td>Alaska</td>
<td>West</td>
<td>401,851</td>
<td>26.1</td>
</tr>
<tr>
<td>Arizona</td>
<td>West</td>
<td>2,718,215</td>
<td>29.2</td>
</tr>
<tr>
<td>Arkansas</td>
<td>South</td>
<td>2,286,435</td>
<td>30.6</td>
</tr>
<tr>
<td>California</td>
<td>West</td>
<td>23,667,902</td>
<td>29.9</td>
</tr>
<tr>
<td>Colorado</td>
<td>West</td>
<td>2,889,964</td>
<td>28.6</td>
</tr>
<tr>
<td>Connecticut</td>
<td>NE</td>
<td>3,107,576</td>
<td>32.0</td>
</tr>
<tr>
<td>Delaware</td>
<td>South</td>
<td>594,338</td>
<td>29.8</td>
</tr>
</tbody>
</table>

Finally, medage would look better if the decimal points were vertically aligned.

```stata
.format medage %8.1f
.list in 1/8
```

<table>
<thead>
<tr>
<th>state</th>
<th>region</th>
<th>pop</th>
<th>medage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alabama</td>
<td>South</td>
<td>3,893,888</td>
<td>29.3</td>
</tr>
<tr>
<td>Alaska</td>
<td>West</td>
<td>401,851</td>
<td>26.1</td>
</tr>
<tr>
<td>Arizona</td>
<td>West</td>
<td>2,718,215</td>
<td>29.2</td>
</tr>
<tr>
<td>Arkansas</td>
<td>South</td>
<td>2,286,435</td>
<td>30.6</td>
</tr>
<tr>
<td>California</td>
<td>West</td>
<td>23,667,902</td>
<td>29.9</td>
</tr>
<tr>
<td>Colorado</td>
<td>West</td>
<td>2,889,964</td>
<td>28.6</td>
</tr>
<tr>
<td>Connecticut</td>
<td>NE</td>
<td>3,107,576</td>
<td>32.0</td>
</tr>
<tr>
<td>Delaware</td>
<td>South</td>
<td>594,338</td>
<td>29.8</td>
</tr>
</tbody>
</table>

Display formats are permanently attached to variables by the `format` command. If we save the data, the next time we use it, state will still be formatted as %-14s, region will still be formatted as %-8.0g, etc.
Example 2

Suppose that we have an employee identification variable, empid, and that we want to retain the leading zeros when we list our data. format has a leading-zero option that allows this.

```
. use https://www.stata-press.com/data/r17/fmtxmpl, clear
. describe empid
Variable name Storage Display Value
    name  type    format   label          Variable label
empid      float   %9.0g
. list empid in 83/87
empid
  83.  98
  84.  99
  85. 100
  86. 101
  87. 102
. format empid %05.0f
. list empid in 83/87
empid
  83. 00098
  84. 00099
  85. 00100
  86. 00101
  87. 00102
```

Technical note

The syntax of the format command allows a varlist and not just one variable name. Thus you can attach the %9.2f format to the variables myvar, thisvar, and thatvar by typing

```
. format myvar thisvar thatvar %9.2f
```

Example 3

We have employee data that includes hiredate and login and logout times. hiredate is stored as a float, but we were careful to store login and logout as doubles. We need to attach a date format to these three variables.

```
. use https://www.stata-press.com/data/r17/fmtxmpl2, clear
. format hiredate login logout
      Variable name       Display format
hiredate          %9.0g
login              %10.0g
logout             %10.0g
```
We remember that the project manager requested that hire dates be presented in the same form as they were previously.

```
format hiredate %td
list hiredate in 1/5
```

```
hiredate
1. 24jan1986
2. 10mar1994
3. 29sep2006
4. 14apr2006
5. 03dec1999
```

Setting European formats

Do you prefer that one and one half be written as 1,5 and that one thousand one and a half be written as 1.001,5? Stata will present numbers in that format if, when you set the format, you specify ‘,’ rather than ‘.’ as follows:

```
use https://www.stata-press.com/data/r17/census10, clear
(1980 Census data by state)
format pop %12,0gc
format medage %9,2f
```
You can also leave the formats just as they were and instead type `set dp comma`. That tells Stata to interpret all formats as if you had typed the comma instead of the period:

```
.set dp comma
```

You can return to using periods by typing

```
.set dp period
```

Setting a variable’s display format to European affects how the variable’s values are displayed by `list` and in a few other places. Setting `dp` to `comma` affects every bit of Stata.

Also, `set dp comma` affects only how Stata displays output, not how it gets input. When you need to type one and a half, you must type `1.5` regardless of context.

### Technical note

`set dp comma` makes drastic changes inside Stata, and we mention this because some older, user-written programs may not be able to deal with those changes. If you are using an older, user-written program, you might `set dp comma` only to find that the program does not work and instead presents some sort of syntax error.

If, using any program, you get an unanticipated error, try setting `dp` back to `period`.

Even with `set dp comma`, you might still see some output with the decimal symbol shown as a period rather than a comma. There are two places in Stata where Stata ignores `set dp comma` because the features are generally used to produce what will be treated as input, and `set dp comma` does not affect how Stata inputs numbers. First,
local x = sqrt(2)

stores the string “1.414213562373095” in x and not “1,414213562373095”, so if some program were to display ‘x’ as a string in the output, the period would be displayed. Most programs, however, would use ‘x’ in subsequent calculations or, at the least, when the time came to display what was in ‘x’, would display it as a number. They would code

```
display ... ‘x’ ...
```

and not

```
display ... "’x’" ...
```

so the output would be

```
... 1,4142135 ...
```

The other place where Stata ignores `set dp comma` is the `string()` function. If you type

```
. generate res = string(numvar)
```

new variable `res` will contain the string representation of numeric variable `numvar`, with the decimal symbol being a period, even if you have previously `set dp comma`. Of course, if you explicitly ask that `string()` use European format,

```
. generate res = string(numvar,"%09,0g")
```

then `string()` honors your request; `string()` merely ignores the global `set dp comma`.

Details of formats

The %f format

In `%w.df`, `w` is the total output width, including sign and decimal point, and `d` is the number of digits to appear to the right of the decimal point. The result is right-justified.

The number 5.139 in `%12.2f` format displays as

```
-----+----1--
  5.14
```

When `d = 0`, the decimal point is not displayed. The number 5.14 in `%12.0f` format displays as

```
-----+----1--
  5
```

`%w.df` works the same way, except that the output is left-justified in the field. The number 5.139 in `%12.2f` displays as

```
-----+----1--
  5.14
```

The %fc format

`%w.dfc` works like `%w.df` except that commas are inserted to make larger numbers more readable. `w` records the total width of the result, including commas.

The number 5.139 in `%12.2fc` format displays as

```
-----+----1--
  5.14
```
The number 5203.139 in \%12.2fc format displays as

\[
------+----1--
\]
\[
5,203.14
\]

As with \%f, if \(d = 0\), the decimal point is not displayed. The number 5203.139 in \%12.0fc format displays as

\[
------+----1--
\]
\[
5,203
\]

As with \%f, a minus sign may be inserted to left justify the output. The number 5203.139 in \%-12.0fc format displays as

\[
------+----1--
\]
\[
5,203
\]

The \%g format

In \%w.dg, \(w\) is the overall width, and \(d\) is usually specified as 0, which leaves up to the format the number of digits to be displayed to the right of the decimal point. If \(d \neq 0\) is specified, then not more than \(d\) digits will be displayed. As with \%f, a minus sign may be inserted to left-justify results.

\%g differs from \%f in that 1) it decides how many digits to display to the right of the decimal point, and 2) it will switch to a \%e format if the number is too large or too small.

The number 5.139 in \%12.0g format displays as

\[
------+----1--
\]
\[
5.139
\]

The number 5231371222.139 in \%12.0g format displays as

\[
------+----1--
\]
\[
5231371222
\]

The number 52313712223.139 displays as

\[
------+----1--
\]
\[
5.23137e+10
\]

The number 0.0000029394 displays as

\[
------+----1--
\]
\[
2.93940e-06
\]

The \%gc format

\%w.dgc is \%w.dg with commas. It works in the same way as the \%g and \%fc formats.

The \%e format

\%w.de displays numeric values in exponential format. \(w\) records the width of the format. \(d\) records the number of digits to be shown after the decimal place. \(w\) should be greater than or equal to \(d+7\) or, if 3-digit exponents are expected, \(d+8\).

The number 5.139 in \%12.4e format is

\[
------+----1--
\]
\[
5.1390e+00
\]
The number $5.139 \times 10^{220}$ is

```
      +1.48e5604189375X+002
```

The %21x format

The %21x format is for those, typically programmers, who wish to analyze routines for numerical roundoff error. There is no better way to look at numbers than how the computer actually records them.

The number 5.139 in %21x format is

```
      +1.4800000000000X+002
```

The number 5.125 is

```
      +1.48e5604189375X+002
```

Reported is a signed, base-16 number with base-16 point, the letter X, and a signed, 3-digit base-16 integer. Call the two numbers $f$ and $e$. The interpretation is $f \times 2^e$.

The %16H and %16L formats

The %16H and %16L formats show the value in the IEEE floating point, double-precision form. %16H shows the value in most-significant-byte-first (hilo) form. %16L shows the number in least-significant-byte-first (lohi) form.

The number 5.139 in %16H is

```
40148e5604189375
```

The number 5.139 in %16L is

```
75931804568e1440
```

The format is sometimes used by programmers who are simultaneously studying a hexadecimal dump of a binary file.

The %8H and %8L formats

%8H and %8L are similar to %16H and %16L but show the number in IEEE single-precision form.

The number 5.139 in %8H is

```
40a472b0
```

The number 5.139 in %8L is

```
b072a440
```

The %t format

The %t format displays numerical variables as dates and times. See [D] Datetime display formats.
The `%s` format

The `%ws` format displays a string in a right-justified field of width \( w \). `%ws` displays the string left-justified.

“Mary Smith” in `%16s` format is

```
-------+-------
Mary Smith
```

“Mary Smith” in `%16s` format is

```
-------+-------
Mary Smith
```

Also, in some contexts, particularly display (see \[P\] display), `%ws` is allowed, which centers the string. “Mary Smith” in `%16s` format is

```
-------+-------
Mary Smith
```

Other effects of formats

You have data on the age of employees, and you type `summarize age` to obtain the mean and standard deviation. By default, Stata uses its default `g` format to provide as much precision as possible.

```stata
. use https://www.stata-press.com/data/r17/fmtxmpl, clear
. summarize age
```

```
  Variable | Obs  Mean | Std. dev. |         Min |         Max
-------------|---------|-----------|-------------|-------------
    age      | 204     | 30.18627  | 10.38067    | 18.00       | 66.00       
```

If you attach a `%9.2f` format to the variable and specify the `format` option, Stata uses that specification to format the results:

```stata
. format age %9.2f
. summarize age, format
```

```
  Variable | Obs  Mean | Std. dev. |         Min |         Max
-------------|---------|-----------|-------------|-------------
    age      | 204     | 30.19     | 10.38       | 18.00       | 66.00       
```

Displaying current formats

`format varlist` is not often used to display the formats associated with variables because using `describe` (see \[D\] describe) is easier and provides more information. The exceptions are date variables. Unless you use the default `%tc, %tC, ...` formats (and most people do), the format specifier itself can become very long, such as

```stata
. format admittime `%tcDDmonCCYY_HH:MM:SS.sss
```

Such formats are too long for `describe` to display, so it gives up. In such cases, you can use `format` to display the format:

```stata
. format admittance %tcDDmonCCYY_HH:MM:SS.sss
```

Type `format *` to see the formats for all the variables.
**Video example**

How to change the display format of a variable

**References**


**Also see**

[D] Datetime business calendars — Business calendars

[D] Datetime display formats — Display formats for dates and times

[D] list — List values of variables

[D] varmanage — Manage variable labels, formats, and other properties

[P] display — Display strings and values of scalar expressions

[U] 12.5 Formats: Controlling how data are displayed

[U] 12.6 Dataset, variable, and value labels
Description

Frames, also known as data frames, allow you to simultaneously store multiple datasets in memory. The datasets in memory are stored in frames, and Stata allows multiple frames. You can switch between them and even link data in them to data in other frames. How this works is presented below.

Remarks and examples

Remarks are presented under the following headings:

What frames can do for you
- Use frames to multitask
- Use frames to perform tasks integral to your work
- Use frames to work with separate datasets simultaneously
- Use frames to record statistics gathered from simulations
- Frames make Stata (preserve/restore) faster
- Other uses will occur to you that we should have listed

Learning frames
- The current frame
- Creating new frames
- Type frame or frames, it does not matter
- Switching frames
- Copying frames
- Dropping frames
- Resetting frames
- Frame prefix command
- Linking frames
- Ignore the _frval() function
- Posting new observations to frames

Programming with frames
- Ado-programming with frames
- Mata programming with frames

What frames can do for you

Frames let you have multiple datasets in memory simultaneously. Here are a few ways you can use them.

Use frames to multitask

You can create a new frame, load another dataset into it, perform some task, switch back, and discard the frame.
You are working. The phone rings. Something has to be handled right now.

```
+ frame create interruption // you create new frame ...
+ frame change interruption // and switch to it
+ use another_dataset // you load a dataset
  // you do what needs doing
+ frame change default // you switch back
+ frame drop interruption // you delete the new frame
```

You are back to work just as if you had never been interrupted.

**Use frames to perform tasks integral to your work**

You need to calculate a value from the data and add it to the data. This is troublesome because making the calculation requires modifying the data, the same data that need to be unmodified and have the result added to them.

You have loaded yourdata.dta into memory and have already made some updates to it. You have not yet saved those changes. You set about calculating the troublesome value.

```
+ frame copy default subtask // create & copy current data to new frame
+ frame change subtask // switch to the new frame
  // begin result calculation
  omitted steps
  keep if mark1 | mark2 // drop observations!
  omitted steps
  regress dmpg dw if mod(_n,2) // calculate troublesome value
+ frame change default // switch back to previous frame
+ gen dwc = cond(foreign,_b[ DW],0) // save result in yourdata.dta
+ frame drop subtask // drop new frame
```

You could have used preserve and restore to perform this task. Using frames, however, is usually more convenient, if for no other reason than you can switch back and forth between them. You cannot do that with a preserved dataset and the modified copy in memory.

If you look carefully at the code above, you will notice that the troublesome value we needed to calculate and store was \_b[ DW]. \_b[ DW] was calculated from data in frame subtask and stored in Stata for subsequent use no matter which frame is current.

It is dataset values that are stored in frames. Programmatic values such as \_b[], r(), e(), and s() are stored in Stata and available across frames.

**Use frames to work with separate datasets simultaneously**

When we say working with datasets simultaneously, we mean datasets that are linked. Linked datasets are an alternative to merged datasets.

You have two datasets. persons.dta contains data on people. uscounties.dta contains data on counties. You want to analyze the people in persons.dta and the counties in which they live. There are issues in combining the two datasets:

1. Some of the people in persons.dta live in the same county.
2. There are counties in uscounties.dta that are irrelevant to your analysis because nobody in persons.dta lives in them.
3. You are not certain that \texttt{uscounties.dta} is complete. There might be some people in \texttt{persons.dta} that live in counties not recorded in \texttt{uscounties.dta}.

4. And beyond that, only some of the variables in \texttt{uscounties.dta} are needed for your analysis.

The frames solution to all of these problems is to link the two datasets. You start by loading \texttt{persons.dta} into one frame and \texttt{uscounties.dta} into another:

\begin{verbatim}
. use persons
. frame create uscounties
. frame uscounties: use uscounties
\end{verbatim}

To link the datasets in the two frames, you type

\begin{verbatim}
. frlink m:1 countyid, frame(uscounties)
\end{verbatim}

This matches the observations in \texttt{persons.dta} to those in \texttt{uscounties.dta} based on equal values of variable \texttt{countyid}. The data are not merged, they are linked. No variables from \texttt{uscounties.dta} are copied to \texttt{persons.dta}, but how the variables would be copied has been worked out.

You copy variables to the person data as you need them, one at a time, or in groups, using the \texttt{frget} command:

\begin{verbatim}
. frget med_income nschools, from(uscounties)
\end{verbatim}

You can perform the desired analysis using \texttt{persons.dta}, the dataset in the current frame:

\begin{verbatim}
. regress income med_income n_schools educ age
\end{verbatim}

\section*{Use frames to record statistics gathered from simulations}

Simulations involve repeating a task—performing a simulation—each step of which produces statistics that are somehow recorded. After that, you analyze the recorded statistics.

The frames solution to the simulation problem is to collect the statistics in another frame. We will name that frame \texttt{results}. You start by creating a new frame and the variables in it to record the statistics, such as \texttt{b1coverage} and \texttt{b2coverage}:

\begin{verbatim}
new frame's name
\end{verbatim}

\begin{verbatim}
. frame create results b1coverage b2coverage
\end{verbatim}

\begin{verbatim}
/ new variables in it
\end{verbatim}

The new frame contains zero observations at this point.

You will next write a do-file to create the values to be stored after each iteration. At the end of each iteration, the do-file will contain the line

\begin{verbatim}
frame's name
\end{verbatim}

\begin{verbatim}
. frame post results (exp_1) (exp_2)
\end{verbatim}

\begin{verbatim}
/ values for b1coverage and b2coverage
\end{verbatim}

The \texttt{frame post} adds an observation to the data in \texttt{results}. \texttt{exp_1} and \texttt{exp_2} are expressions.
When the do-file finishes, the completed set of results will be found in frame \texttt{results}. You will want to save them:

```
.frame results: save filename
```

You will then switch to the frame and begin your analysis of the statistics:

```
.frame change results
.summarize
```

\textbf{Frames make Stata (\texttt{preserve/restore}) faster}

Many programs written in Stata use the commands \texttt{preserve} and \texttt{restore} to temporarily save and later restore the contents of the data in memory. Programs that use \texttt{preserve} and \texttt{restore} now run faster if you are using Stata/MP. They run faster because Stata preserves data by copying them to hidden frames. Those hidden frames are stored in memory. Copying data to frames stored in memory takes a lot less time than copying data to disk.

More correctly, \texttt{preserve} copies data to hidden frames unless memory is in short supply. If it is, \texttt{preserve} resorts to storing them on disk. That is temporary because later, as datasets are restored, memory will again become available and \texttt{preserve} will return to preserving them in hidden frames.

This is all automatic, but you may want to reset the value of \texttt{max\_preservemem}, which controls this behavior. When the amount stored in hidden frames would exceed \texttt{max\_preservemem}, Stata preserves subsequent datasets on disk. Out of the box, \texttt{max\_preservemem} is set to 1 gigabyte. Perhaps you or someone else has already changed that. To find out the current value of \texttt{max\_preservemem}, type

```
.query memory
```

If you want to change \texttt{max\_preservemem} to 2 gigabytes for the duration of the session, type

```
.set max\_preservemem 2g
```

You can set the value up or down. You could set it to \texttt{4g} or \texttt{50m}. You could even set it to \texttt{0}, and then all datasets would be preserved to disk.

If you want to set \texttt{max\_preservemem} to 2 gigabytes permanently, for this session and future Stata sessions, type

```
.set max\_preservemem 2g, permanently
```

\textbf{Other uses will occur to you that we should have listed}

Frames make doing lots of tasks more convenient, and you will find your own uses for them. Frames make code faster too. Manipulating objects stored in memory takes less computer time than manipulating disk files.

\textbf{Learning frames}

Here is a tutorial on using frames. In the tutorial, we will sometimes show you a syntax diagram. For example, we might show you

```
.frame copy framename newframename
```

When we show syntax diagrams in the tutorial, they are not always the full syntax diagrams. \texttt{frame copy}, for instance, also allows a \texttt{replace} option, and we might not only not show it in the syntax diagram but also not even mention it. You can click on the command to see the full syntax.
The current frame

Everything hinges on the current frame. Stata commands use the data in the current frame. When you load a dataset,

```
.sysuse auto
(1978 automobile data)
```
you are loading it into the current frame. Which frame is that? Type frame to discover its identity:

```
.frame
(current frame is default)
```

You can type frame or type pwf, which is a synonym for frame. The letters stand for “print working frame”. We will type frame in this tutorial, but you may prefer to type pwf because it is shorter. Other frame commands also have shorter synonyms. We will mention them as we go along.

We just discovered that the current frame is named default. When Stata is launched, that is what it names the frame it creates for you. You cannot change that, but default is just a name, and you can rename frames if you wish. You can create other frames too. You can create up to 100 of them.

To rename a frame, use the frame rename command:

```
frame rename oldname newname
```

To rename the frame default to genesis, type

```
.frame rename default genesis
.frame
(current frame is genesis)
```

Frames can be renamed whether Stata created them or you did. They can be renamed whether they have data in them or they are empty. Renaming default will not break anything subsequently. Stata commands operate on the current frame, whatever its name.

Creating new frames

Create new frames using the frame create command:

```
frame create newframename
```

We will show you an example in a minute. First, however, if you are going to create a frame with a new name, you need to know how to find out the names of the frames that currently exist. You do that using the frames dir command:

```
frames dir
```

We recall that we renamed our default frame, but we cannot recall the name that we used. So what frames are in memory?

```
.frames dir
genesis 74 x 12; 1978 automobile data
```

There is one frame in memory, named genesis. It contains a dataset that is $74 \times 12$, meaning 74 observations and 12 variables. The dataset has a dataset label “1978 automobile data”, but if it did not, the dataset’s name, auto.dta, would have appeared in its place in frames dir’s output, unless the data had never been saved to disk. In that case, nothing would have appeared where “1978 automobile data” appeared.
Now let’s create a new frame named second:

```
. frame create second
. frame dir
  genesis 74 x 12; 1978 automobile data
  second 0 x 0
```

There are now two frames in memory. The new frame is $0 \times 0$. It is empty.

By the way, `frame create` has a shorter synonym, `mkf`. The letters stand for “make frame”. We could have typed `mkf second` to make the new frame.

**Type frame or frames, it does not matter**

You probably did not notice, but we have used `frames dir` twice so far, but we typed it differently the second time. We typed

```
. frames dir
. frame dir
```

Stata does not care whether you type `frame` or `frames`. This indifference applies to all the `frames/frame` commands.

**Switching frames**

`frame change` (synonym: `cwf` for “change working frame”) switches the identity of the current frame:

```
frame change framename
```

We could make `second` the current frame and switch back to `genesis` again:

```
. frames change second
. count
  0
  . cwf genesis
  . count
    74
```

We used Stata’s `count` command to demonstrate that the current frame really switched. `count` without arguments displays the number of observations.

**Copying frames**

There are two commands for copying frames:

```
frame copy framename newframename
frame put varlist, into(newframename)
frame put if, into(newframename)
```

`frame copy` copies the entire dataset.

`frame put` copies subsets of the dataset.

In either case, the commands create the frame being copied to.
Dropping frames

To drop an existing frame, type

```
frame drop framename
```

Resetting frames

Resetting frames means the following:

1. Drop all the data in all the frames, even if the data have not been saved since they were last saved.
2. Drop (delete) all the frames.
3. Create a new frame named `default`, and make it the current frame.

Each of the following commands resets frames:

```
frames reset
clear frames

clear all
```

`frames reset` and `clear frames` are synonyms.

`clear all` resets the frames and does more. It returns Stata to as close to just-after-launch status as possible.

Frame prefix command

The `frame` prefix command is perhaps the most convenient of the `frame` commands. Its syntax command is

```
frame framename: stata_command
```

The `frame` prefix command 1) changes the current frame to the frame specified, 2) executes `stata_command`, and 3) changes the current frame back to what it was.

For instance, say the current frame is `default` and we have a second frame named `second`. We type

```
.frame second: sysuse census, clear
```

The result would be that frame `second` would contain `census.dta` and the current frame would still be `default`, just as if we had typed

```
.frame change second
.sysuse census, clear
.frame change default
```

Frame prefix has a second feature too. Imagine that in doing the above, we omitted the `clear` option when we use the data. Consider what would have happened if we set about typing the three commands but the data in `second` had changed since they were last saved:

```
.frame change second
.sysuse census
no; dataset in memory has changed since last saved
r(4);
```
What is the current frame? It is \texttt{second}, of course, because we changed to it. Instead of using the two previous commands, we could have used the \texttt{frame} prefix approach. (The current frame is \texttt{default}.)

\begin{verbatim}
. frame second: sysuse census
   no; dataset in memory has changed since last saved
   r(4);
\end{verbatim}

Even though an error occurred, the current frame is still \texttt{default}! To recover from the error, we do not have to change back to the original frame. The \texttt{frame} prefix command did that for us.

\texttt{frame} prefix has another syntax when you have more than one command to be executed:

\begin{verbatim}
frame framename  {
   stata_command
   stata_command
   .
   .
}
\end{verbatim}

This syntax is especially useful in programs.

### Linking frames

When we say linking, we mean linking as shown in the earlier example when we had separate datasets on people and counties and combined them in a merged-data kind of way. Linking can do a lot more than we showed you.

In [D] \texttt{frlink}, we show you how to create a nested linkage to link students (one dataset) to the schools they attend (a second dataset) and to the counties (a third dataset) in which their schools are located. We show you an example of linking a generational dataset with itself, so that adult children are linked to their parents and grandparents, a total of six simultaneous linkages!

Linkages are created by using the \texttt{frlink} command. Its simplest syntaxes are

\begin{verbatim}
frlink m:1 varlist, frame(framename)
frlink 1:1 varlist, frame(framename)
\end{verbatim}

These syntaxes create an \texttt{m:1} or \texttt{1:1} link between the current frame and \texttt{framename} based on observations having equal values of \texttt{varlist}.

Once a link is created, you can use the \texttt{frget} command to copy the appropriate values of variables from \texttt{framename} to the current frame. Its syntaxes are

\begin{verbatim}
frget varlist, from(linkagename)
frget newvar = varname, from(linkagename)
\end{verbatim}

You can use the \texttt{frval()} function in expressions to access appropriate observations of variables in the linked data. Its syntax is

\begin{verbatim}
... frval(linkagename, varname) ...
\end{verbatim}
Ignore the `_frval()` function

While we are on the subject of the `frval()` function, we should warn you. Also available in [FN] Programming functions is `_frval()`. Ignore it. `frval()` is better. `_frval()` is for use by programmers.

Posting new observations to frames

We used posting to perform simulations in an example earlier. That is one use of it. More generally, posting solves problems that require transferring data or values from one frame to a new observation in another.

First, you prepare the other frame to receive the data. `frame create`, which we already discussed, has a syntax for doing this. We showed you its first syntax, which is

```
frame create newframename
```

The second syntax is

```
frame create newframename newvarlist
```

This syntax creates the new frame and creates in it a zero-observation dataset of the new variables specified. `newvarlist` really is a new varlist, and that means that you can specify variables types and variable names. You could type

```
.frame create results strL(rngstate) double(b1coverage b2coverage)
```

Alternatively, you can use `frame create`'s first syntax to create the frame, use `frame change` to switch to it, and create the zero-observation dataset yourself. Then, you can switch back to what was the current frame.

```
.frame post adds observations to the second frame. Its syntax is
```

```
.frame post framename (exp) (exp) ... (exp)
```

The expressions are in the same order as the variables in the second frame.

Programming with frames

Below we discuss writing Stata programs that deal with multiple frames.

If you are not interested in writing such programs, stop reading.

What follows is not a tutorial. What follows are numbered lists detailing everything you need to know to write programs that use more than the current frame. That program could implement a command that does something with frames specified by users. Or it could do something that, as far as users are concerned, uses only the current frame and hidden from them is that your program uses frames to accomplish certain internal tasks.

We also want to emphasize there still exists a place for programs written in Stata that do not use frames at all. Perhaps most programs are like that.
Ado-programming with frames

1. **tempnames.**

   Frames with names created by `tempname` are automatically dropped (deleted) when the program generating the temporary name ends.

   If the program you write is to create a new frame for the user, give the frame a `tempname` in your program, and, at the end, use `frame rename` to change its name. This way, if an error occurs, the frame the program may have been in the midst of creating will be dropped automatically.

2. **Current frame.**

   Stata provides the name of the current frame in `creturn` result `c(frame)`. You can obtain the name of the current frame by coding
   
   ```
   local curframe = c(frame)
   ```

   Programs that use frames invariably change frames during their execution. Programs need to ensure the appropriate frame is the current one at the time the program exits. This includes when the program is successful and when it exits with error.

   The successful case is easy enough to handle. At the point your program exits, set the current frame appropriately. In general, the current frame should be the same as the current frame was when the program started.

   Error cases can be more difficult. Who knows when the user will press break or when the bug buried in your code will bite? The code could be doing literally anything. Even so, your program needs to ensure that the current frame is set appropriately. There is a style of programming that does this.

   Case 1: You are writing new command `foo`. `foo` uses frames but in all cases is to leave the current frame the same as it was initially. The code reads as follows:
   
   ```
   program foo
   version ...
   local curframe = c(frame)
   frame `curframe' {
      foo_cmd '0'
   }
   end
   ```

   Write `foo_cmd` as you usually would. As you write `foo_cmd`, you can ignore the current-frame problem. You can use `frame change` freely in `foo_cmd` and its subroutines. No matter what happens, error or success, the program will end with the current frame unchanged.

   Case 2: You are writing new command `foo`. If `foo` is successful, the new frame will change. The code reads as follows:
   
   ```
   program foo
   version ...
   local curframe = c(frame)
   frame `curframe' {
      foo_cmd '0'
   }
   frame change `s(frame)'
   end
   ```

   Write `foo_cmd` as you usually would. If execution is successful, however, `foo_cmd` must `sreturn` in `s(frame)` the name of the frame that is to be the current frame. As with case 1, you can use `frame change` freely in `foo_cmd` and all of its subroutines.
3. **preserve** and **restore**.

For end users, using frames is sometimes a better alternative to using **preserve** and **restore**. Programmers should not, however, interpret that as **preserve** and **restore** are out of date and not to be used in frame programming. **preserve** and **restore** in programming have the same valid use they have always had.

Before frames existed in Stata, a single program could have at most one active **preserve** in it. Active means not canceled by **restore** or **restore**, not. A program could **preserve**, later **restore** or **restore**, not, and then **preserve** again. It would be odd but allowed.

Nowadays, a single program can have up to one active **preserve** for each frame. If a program deals with frames ‘one’ and ‘two’ and it is necessary, it can **preserve** both of them. **preserve** preserves the current frame. To preserve frames ‘one’ and ‘two’, code,

```
frame 'one': preserve
frame 'two': preserve
```

When frames are automatically restored at the end of the program, both frames will be restored.

If you wish to restore frame ‘one’ early and cancel its automatic restoration when the program ends, code

```
frame 'one': restore
```

If you instead wish to restore frame ‘one’ now and still have it restored when the program ends, code

```
frame 'one': restore, preserve
```

If you instead wish simply to cancel the restoration of frame ‘one’ when the program ends, code

```
frame 'one': restore, not
```

In all three cases, frame ‘two’ will still be restored when the program ends.

Any uncanceled automatic restorations when the program ends will re-create any frames that have been dropped (deleted). Automatic restoration does not change the identity of the current frame.

Mata programming with frames

1. **st_frame**(*) functions.

   Mata provides a suite of frame-related functions. They can change frames, create frames, drop frames, etc.


   Calls to **st_data**() and its associated functions return the data from the current frame. If you want data from other frames, change to the other frame first using **st_framecurrent**().

3. **st_view**() and **st_sview**() functions.

   Views are views onto the frame that was current at the time the view was created by **st_view**() or **st_sview**(), and they remain that after creation even when the identity of the current frame changes. If X is a view onto frame default, it remains a view onto frame default even if the current frame changes.
Views are how data can be copied between frames. Create a view onto the data in one frame. Create another view onto the data in the other. Use one view to update the other.

Reference


Also see

[D] frames — Data frames
[D] frget — Copy variables from linked frame
[D] frlink — Link frames
[FN] Programming functions
[M-5] st_frame*() — Data frame manipulation
frames — Data frames

Description

This entry provides a quick reference to each of the individual commands and functions related to data frames.

If you are new to data frames in Stata, please start by reading [D] frames intro.
Data frames are discussed in detail in [D] frames intro.
There is also a set of Mata functions to work with frames. See [M-5] st_frame*.()

Menu

Data > Frames Manager

Syntax

frame and frames are synonyms. Below, we will use one or the other depending on which one is more natural given the context.

Display name of current (working) frame
frame pwf
frame
pwf

Display names of all frames in memory
frames dir

Create new, empty frame
frame create newframename

Create new frame with specified variables for use with frame post
frame create newframename newvarlist

Change identity of current (working) frame
frame change framename

Execute command on data in specified frame

\begin{verbatim}
frame framenname: stata_command

frame framenname {
    commands to execute in context of framenname
}
\end{verbatim}

Make a copy of a frame

\begin{verbatim}
frame copy frame_from frame_to [, replace]
\end{verbatim}

Copy subset of variables or observations to a new frame

\begin{verbatim}
frame put
\end{verbatim}

Add new observation to frame

\begin{verbatim}
frame post framenname (exp) (exp) ... (exp)
\end{verbatim}

Drop (eliminate) frame that is not the current frame

\begin{verbatim}
frame drop framenname
\end{verbatim}

Rename existing frame (which can be the current frame)

\begin{verbatim}
frame rename oldframenname newframenname
\end{verbatim}

Reestablish initial state of having a single, empty frame named default

\begin{verbatim}
frames reset
\end{verbatim}

Link frames

\begin{verbatim}
frlink
\end{verbatim}

Get variables from linked frame

\begin{verbatim}
frget
\end{verbatim}

Functions to access variables in another frame

\begin{verbatim}
frval(linkvar, varname)
-frval(framename, varname, i)
\end{verbatim}

Also see

[D] frames intro — Introduction to frames
[M-5] st_frame*() — Data frame manipulation
Description

frame change makes the named frame current. This means that any commands you issue after frame change will run on the data in that frame.

cwf (change working frame) is a synonym for frame change.

Menu

Data > Frames Manager

Syntax

frame change framename

cwf framename

Remarks and examples

frame change makes the named frame current, or active. After you change to a frame, any commands you execute work with the data in that frame.

Another way to work with the data in another frame is the frame prefix command. See [D] frame prefix.

Example 1

Let's assume we have several frames in memory, including our current frame named default. We see this by typing frames dir:

. frames dir
  cars  74 x 12; 1978 automobile data
default  50 x 13; 1980 Census data by state
work    28534 x 21; National Longitudinal Survey of Young Women, 14-24 years old in 1968

Our next project uses the 1978 automobile data in the cars frame. To change to this frame, we type

. frame change cars

308
We can now work with the data in this frame. For instance, we can describe the data by typing

. describe

Contains data from https://www.stata-press.com/data/r17/auto.dta
Observations: 74 1978 automobile data
Variables: 12 13 Apr 2020 17:45
(_dta has notes)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage</th>
<th>Display</th>
<th>Value</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str18</td>
<td>%-18s</td>
<td></td>
<td>Make and model</td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%8.0gc</td>
<td></td>
<td>Price</td>
</tr>
<tr>
<td>mpg</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>rep78</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Repair record 1978</td>
</tr>
<tr>
<td>headroom</td>
<td>float</td>
<td>%6.1f</td>
<td></td>
<td>Headroom (in.)</td>
</tr>
<tr>
<td>trunk</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Trunk space (cu. ft.)</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td></td>
<td>Weight (lbs.)</td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Length (in.)</td>
</tr>
<tr>
<td>turn</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Turn circle (ft.)</td>
</tr>
<tr>
<td>displacement</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Displacement (cu. in.)</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>float</td>
<td>%6.2f</td>
<td></td>
<td>Gear ratio</td>
</tr>
<tr>
<td>foreign</td>
<td>byte</td>
<td>%8.0g</td>
<td>origin</td>
<td>Car origin</td>
</tr>
</tbody>
</table>

Sorted by: foreign

At any time, we can change back to the default frame by typing

. frame change default

Also see

[D] frames intro — Introduction to frames
[D] frame prefix — The frame prefix command
frame copy — Make a copy of a frame

Description

frame copy copies an existing frame to a frame with a new name or to an existing frame, replacing its contents. All data and metadata from frame_from are copied.

Quick start

Copy the default frame to a frame named fr1
    frame copy default fr1

Copy frame fr1 to existing frame fr2, replacing the data
    frame copy fr1 fr2, replace

Menu

Data > Frames Manager

Syntax

frame copy frame_from frame_to [, replace]

Option

replace specifies that frame_to be replaced if it already exists.

Remarks and examples

frame_from must be an existing frame. It may be the current frame. frame_to may be the name of a new frame or an existing frame. If it is an existing frame, replace must be specified.

In a programming context within a do-file or an ado-file, if you obtain a temporary name and copy a frame to that name, the frame will automatically be removed upon conclusion of the do-file or program.

Example 1

Let’s assume we have a frame named default in memory. We want to copy this frame to a new frame named counties. To do this, we type
    . frame copy default counties
Later, we decide that we need to copy a frame named uscounties to our existing frame named counties, replacing it

```
.frame copy uscounties counties, replace
```

When programming, we might want to copy a frame to a temporary name. To copy a frame named counties to a temporary name, we could type the following:

```
.tempname newframe
.frame copy counties 'newframe'
```

Also see

[D] frames intro — Introduction to frames
[D] frame put — Copy selected variables or observations to a new frame
[D] frame rename — Rename existing frame
frame create — Create a new frame

Description

frame create creates a new, empty frame.

mkf (make frame) is a synonym for frame create.

frame create with a newvarlist creates a new frame with the specified variables. This syntax is most often used in combination with frame post for posting results in a new frame, see [P] frame post.

Menu

Data > Frames Manager

Syntax

Create new, empty frame

frame create newframename

mkf newframename

Create new frame with specified variables

frame create newframename newvarlist (see [P] frame post)

Remarks and examples

frame create creates a new, empty frame. After creation, you might use frame change to switch to that frame, or you might use the frame prefix with use or import to load data for analysis in that frame.

Example 1

To create a new frame named cars, type

.frame create cars

We can now load our 1978 automobile data into new the new frame:

.frame cars: use https://www.stata-press.com/data/r17/auto.dta

Here we loaded data from the web. More often, we will load data from our computer. If auto.dta was saved in our current working directory, we could have typed

.frame cars: use auto.dta
Also see

[D] frames intro — Introduction to frames

[D] frames — Data frames

[P] frame post — Post results to dataset in another frame
frame drop — Drop frame from memory

Description

frame drop eliminates from memory the specified frame, including any data that are in that frame.

Menu

Data > Frames Manager

Syntax

frame drop framename

Remarks and examples

frame drop eliminates, or removes from memory, the specified frame. Any data in the frame are dropped when the frame is dropped. The specified frame must exist and cannot be the current frame.

To eliminate all frames from memory, including the current frame, use frames reset. See [D] frames reset.

Example 1

To drop a frame named cars, type

    . frame drop cars

Also see

[D] frames intro — Introduction to frames
[D] frames — Data frames
[D] frames reset — Drop all frames from memory
Description

The `frame` prefix allows you to execute one or more Stata commands in another frame, leaving the current frame unchanged.

Quick start

Describe the data in frame `fr1`

```
frame fr1: describe
```

Execute a series of commands in frame `fr2`

```
frame fr2 {
    use mydata
    summarize
    codebook
}
```

Syntax

```
frame framename: stata_command

frame framename {
    commands to execute in context of framename
}
```

Remarks and examples

Remarks are presented under the following headings:

- *Example of interactive use*
- *Example of use in programs*

Example of interactive use

You have data in two frames. In your current frame you have data containing detailed information on sales for your company across four regions. A colleague just sent you an email with a summary dataset named `sales.dta`, which is supposed to contain the total sales for each region. You want to make sure the summary dataset was created from the same base sales information as the detailed dataset.
In your current dataset, you know from `summarize` that the total sales for the South region were $532,399 and the total cost of the goods sold was $330,499. You check that the dataset you just received matches these totals:

```
.frame create summary
.frame summary: use sales
.frame summary: list if region=="South"
```

The `frame` prefix command allowed you to load a dataset in frame `summary` and run a command on that data without affecting anything in your current frame.

Example of use in programs

The `frame` prefix can be used for one-liners, such as above, or it can be used to execute a whole series of commands on the data in another frame. The nice thing in either case is that no matter what happens when those commands are executed, whether they complete successfully or exit with error, the current frame will come back to what it was before you called the `frame` prefix command. In programs, this means that you do not have to hold on to the current frame name and change back to it after working in another frame.

You are writing a program that takes a subset of the current data, performs some manipulations on that subset, and then graphs the result. The required manipulations would damage the original dataset. One way to do this would be to

1. create a temporary frame:
   ```
tempname tmpframe

2. put a subset of data into it:
   ```
.frame put if ..., into('tmpframe')

3. perform the needed manipulations and graph the result:
   ```
.frame 'tmpframe' {
   some commands which manipulate the data
   graph twoway ...
}
```

At the end of this block of code, any commands that appear next will work against the original frame, not `tmpframe`. You could add a line to drop `tmpframe`, but there is no need. Because it has a temporary name, the frame and the data in it will automatically be dropped when your program or do-file completes.

An alternative workflow for the above would be to first `preserve` your data, then manipulate them in place and obtain your graph. You could then `restore` the original data. Whether you should use the `frame` prefix approach or the `preserve` and `restore` approach is up to you. The `frame` approach is often faster, but if your dataset in memory is extremely large, you may not want to make another entire copy of it in memory, even temporarily, and thus, the second approach may be better in such a case.

Also see

[D] frames intro — Introduction to frames
[D] frames — Data frames
Title

frame put — Copy selected variables or observations to a new frame

Description

frame put copies a subset of variables or observations from the current frame to the specified frame. It works much like Stata’s keep command (see [D] drop), except that the data in the current frame are left unchanged, while the selected variables or observations are copied to a new frame.

Quick start

Put variables v1, v2, and v3 from the current frame into new frame fr1
   frame put v1 v2 v3, into(fr1)

Put all variables whose name begins with v into new frame fr2
   frame put v*, into(fr2)

Put all observations where v1 is not missing into new frame fr3
   frame put if !missing(v1), into(fr3)

Put the first observation from each cluster identified by cvar into new frame fr4
   by cvar: frame put if _n==1, into(fr4)

Menu

Data > Frames Manager

Syntax

Copy selected variables from the current frame to a new frame
   frame put varlist, into(newframename)

Copy observations that satisfy specified condition from the current frame to a new frame
   frame put [varlist] if, into(newframename)

Copy a range of observations from the current frame to a new frame
   frame put [varlist] in [if], into(newframename)

by is allowed with the second syntax of frame put; see [D] by.
Remarks and examples

There are three main workflows for operating on a subset of data you already have in memory. One is to make use of Stata’s if and in qualifiers with your commands to restrict the observations to be used. Another is to use preserve to make a temporary copy of the data in memory, then use keep and drop to make a subset of those data for analysis, and then to use restore to bring the original data back. Finally, you can leave the data in memory unchanged and use frame put to place a subset of the data in another frame for analysis. That frame can then be dropped, saved, or left in memory for further analysis.

frame put copies all variable and value labels, characteristics, and notes for any variables copied to the new frame.

Example 1

To demonstrate frame put, we start with data from the 1980 U.S. Census.

```
. use https://www.stata-press.com/data/r17/census
  (1980 Census data by state)
. describe
Contains data from https://www.stata-press.com/data/r17/census.dta
Observations:    50  1980 Census data by state
Variables:       13  6 Apr 2020 15:43
  Variable Storage Display Value Variable label
       name  type format label
state    str14 %-14s State
state2   str2  %-2s  Two-letter state abbreviation
region   int  %-8.0g   cenreg Census region
pop      long  %12.0g Population
poplt5   long  %12.0gc Pop, < 5 year
pop5_17  long  %12.0gc Pop, 5 to 17 years
pop18p   long  %12.0gc Pop, 18 and older
pop65p   long  %12.0gc Pop, 65 and older
popurban long  %12.0gc Urban population
medage   float %9.2f Median age
defath   long  %12.0g  Number of deaths
marriage long  %12.0g  Number of marriages
divorce  long  %12.0g  Number of divorces
```

Sorted by:
We put data from several variables for all states with a population greater than 5,000,000 into new frame `pop5`.

```
.frame put state region pop* medage death if pop > 5000000, into(pop5)
.frame pop5: describe
```

Contains data

<table>
<thead>
<tr>
<th>Observations:</th>
<th>14</th>
<th>1980 Census data by state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables:</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>str14</td>
<td>%-14s</td>
<td>State</td>
</tr>
<tr>
<td>region</td>
<td>int</td>
<td>%-8.0g</td>
<td>cenreg</td>
</tr>
<tr>
<td>pop</td>
<td>long</td>
<td>%12.0gc</td>
<td>Population</td>
</tr>
<tr>
<td>poplt5</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, &lt; 5 year</td>
</tr>
<tr>
<td>pop5_17</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 5 to 17 years</td>
</tr>
<tr>
<td>pop18p</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 18 and older</td>
</tr>
<tr>
<td>pop65p</td>
<td>long</td>
<td>%12.0gc</td>
<td>Pop, 65 and older</td>
</tr>
<tr>
<td>popurban</td>
<td>long</td>
<td>%12.0gc</td>
<td>Urban population</td>
</tr>
<tr>
<td>medage</td>
<td>float</td>
<td>%9.2f</td>
<td>Median age</td>
</tr>
<tr>
<td>death</td>
<td>long</td>
<td>%12.0gc</td>
<td>Number of deaths</td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.

Also see

[D] `frames intro` — Introduction to frames
[D] `frames` — Data frames
[D] `drop` — Drop variables or observations
[D] `frame copy` — Make a copy of a frame
[P] `frame post` — Post results to dataset in another frame
frame pwf — Display name of current (working) frame

Description

frame pwf displays the name of the current frame, also known as the working frame. frame by itself and pwf (print working frame) by itself are synonyms for frame pwf.

Menu

Data > Frames Manager

Syntax

frame pwf
frame
pwf

collect is allowed with frame pwf; see [U] 11.1.10 Prefix commands.

Remarks and examples

You can type any of frame pwf, frame, or pwf to see what the current (working) frame is.

. sysuse auto
   (1978 automobile data)
. frame pwf
   (current frame is default)
. frame create cars
. frame change cars
. pwf
   (current frame is cars)

Stored results

frame pwf stores the following in r():

Macros

r(currentframe) name of current (working) frame
Also see

[D] frames intro — Introduction to frames

[D] frames — Data frames
frame rename — Rename existing frame

Description

frame rename changes the name of an existing frame. You can even rename the current frame.

Menu

Data > Frames Manager

Syntax

frame rename oldframename newframename

Remarks and examples

oldframename must be an existing frame. It may be the current frame. newframename must not be an existing frame.

Example 1

Let’s assume we have several frames in memory, including a frame named default. We see this by typing frames dir:

```
. frames dir
  cars  74 x 12; 1978 automobile data
  default 50 x 13; 1980 Census data by state
  work 28534 x 21; National Longitudinal Survey of Young Women, 14-24 years old in 1968
```

We want to rename the default frame to a new frame named census:

```
. frame rename default census
```

We also want to rename the existing frame cars to automobiles:

```
. frame rename cars automobiles
```

We can then check the changes with frames dir:

```
. frames dir
  automobiles  74 x 12; 1978 automobile data
  census 50 x 13; 1980 Census data by state
  work 28534 x 21; National Longitudinal Survey of Young Women, 14-24 years old in 1968
```
Also see

[D] frames intro — Introduction to frames
[D] frames — Data frames
[D] frame copy — Make a copy of a frame
frames dir — Display names of all frames in memory

Description
frames dir lists all frames in memory, along with the dimensions of the data, the label of the data in each (if any), and an indicator of whether the data in the frame have changed since last saved.

Menu
Data > Frames Manager

Syntax
frames dir

collect is allowed; see [U] 11.1.10 Prefix commands.

Remarks and examples
frames dir shows you at a glance information about all frames in memory.

The first column shows an asterisk if the data in a given frame have changed since they were last saved. If you try to exit Stata and there are unsaved data in one or more frames, you will receive an error warning you. You can type frames dir to see frames with unsaved data.

The third column shows the number of observations and variables along with the data label, if any, for each frame. If there is not a data label, the dataset filename, if there is one, will be displayed.

Example 1
We have been working with data in multiple frames. We now want to see all the frames currently in memory. To do this, we type

```
    . frames dir
    * afewcars  74  x  3; Subset of auto.dta
     default   74  x  12; 1978 automobile data
    * work     3142 x 10; National Longitudinal Survey of Young Women, 14-24 years old in 1968
```

Note: Frames marked with * contain unsaved data.

We are reminded of the names and contents of the three frames in memory. We also see that the data in frames afewcars and work have changed, but those changes have not been saved.
Stored results

`frames dir` stores the following in `r()`:

Macros

- `r(frames)` names of frames in memory
- `r(changed)` 1 or 0 for each frame in memory: 1 means the data in the frame have changed since last save; 0 means they have not changed

Also see

[D] `frames intro` — Introduction to frames  
[D] `frames` — Data frames  
[D] `save` — Save Stata dataset
frames reset — Drop all frames from memory

Description

frames reset eliminates from memory all frames, including any data in them. It restores Stata to its initial state of having a single, empty frame named default. clear frames is a synonym for frames reset.

Menu

Data > Frames Manager

Syntax

frames reset

clear frames

Remarks and examples

frames reset eliminates, or removes from memory, all frames. It then creates a single, empty frame named default. This is the same as Stata’s initial state when it first starts.

To drop a single frame, use frame drop. See [D] frame drop.

To drop results, programs, matrices, etc. in addition to frames, use the clear command. See [D] clear.

Example 1

We have numerous frames in memory:

. frames dir
   cars  74 x 12; 1978 automobile data
   default  50 x 13; 1980 Census data by state
   work 28534 x 21; National Longitudinal Survey of Young Women, 14-24 years of age in 1968
   (output omitted)

We want to drop all the frames. We do this by typing

. frames reset

We now have the empty frame named default.

. frames dir
   default  0 x 0
Also see

[D] frames intro — Introduction to frames
[D] frames — Data frames
[D] frame drop — Drop frame from memory
[D] clear — Clear memory
frget — Copy variables from linked frame

Description

frget copies variables and their associated metadata from the data in the linked frame to the data in the current frame. Copy means copying the relevant observations from the linked frame to the appropriate observations in the current frame.

See [D] frames intro if you do not know what a frame is.

Quick start

Obtain variables v1, v2, and v3 from another frame linked to by linkage lnk
    frget v1 v2 v3, from(lnk)

Obtain variables v4 and v5 via linkage lnk, naming them newv4 and newv5 in the current frame
    frget newv4=v4 newv5=v5, from(lnk)

Obtain all variables via linkage lnk, prefixing them with 1_
    frget *, from(lnk) prefix(1_)

Obtain all variables via linkage lnk, excluding those matching pattern ind*
    frget *, from(lnk) exclude(ind*)

Syntax

    frget varlist, from(linkname) [ rename_options ]  \(1\)

    frget newvar = varname, from(linkname)  \(2\)

linkname is the name of a linkvar in the current frame that was created by frlink; see [D] frlink.

rename_options                Description

  prefix(string)               prefix new variable names with string
  suffix(string)               suffix new variable names with string
  exclude(varlist)             exclude specified variables

collect is allowed; see [U] 11.1.10 Prefix commands.

Syntax 1 copies the variable names specified by varlist from the frame linked by linkname to the current frame.

Syntax 2 copies varname from the frame linked by linkname to newvar in the current frame.

    Copy means copy and clone. Display formats, variable labels, value labels, notes, and characteristics are also copied.

In syntax 2, newvar=varname may be repeated. For example,

    . frget edinc=income hval=homevalue, from(counties)
frget — Copy variables from linked frame

Options

from(linkname) specifies the identity of the linked frame from which variables are copied. Linkages to frames are created by the frlink command. Linkages are usually named for the frame to which they link. Linkage counties links to frame counties, and so you specify from(counties). If linkage c links to frame counties, you specify from(c). from() is required.

prefix(string) specifies a string to be prefixed to the names of the new variables created in the current frame. Say that you type

   . frget inc*, from(counties)

   to request that variables income and income_family be copied to the current frame. If variable income already exists in the current frame, the command would issue an error message to that effect and copy neither variable. To copy the two variables, you could type

   . frget inc*, from(counties) prefix(c_)

   Then the variables would be copied to variables named c_income and c_income_family.

suffix(string) works like prefix(string), the difference being that the string is suffixed rather than prefixed to the variable names. Both options may be specified if you wish.

exclude(varlist) specifies variables that are not to be copied. An example of the option is

   frget *, from(counties) exclude(emp*)

   All variables except variables starting with emp would be copied.

   More correctly, all variables except emp*, _, and the match variables would be copied because frget always omits the underscore and match variables. See the explanation below.

Remarks and examples

Remarks are presented under the following headings:

Overview
Everything you need to know about frget

Overview

You have data on people and data on counties. You loaded the datasets and created a linkage named uscounties by typing

   . use people
   . frame create uscounties
   . frame uscounties: use uscounties
   . frlink m:1 countyid, frame(uscounties)

See example 1 in [D] frlink for details.

   Among the variables in uscounties.dta is median_income. You could copy the variable to the person data in the current frame by typing either of the following:

   . frget median_income, from(uscounties)
   . frget medinc = median_income, from(uscounties)

   The first command names the copy median_income in the current frame. The second names it medinc.
Everything you need to know about frget

Here is everything you need to know in outline form:

1. What it means to copy a linked variable
2. frget can copy variables one at a time
3. frget allows variable names to be abbreviated
4. frget can bring over groups of variables
5. frget copies all the variables specified, or none of them
6. frget ignores repeated variables
7. How to get all the variables 1: frget *
8. How to get all the variables 2: frget *, prefix()
9. How to create new variables
10. frget copies and clones variables

We make two assumptions in what follows:

A1. The current frame contains data on people. A frame named uscounties contains data on counties. That is, we assume

   . use people
   . frame create uscounties
   . frame uscounties: use uscounties

A2. The frames are linked on the match variable countyid, which appears in both datasets. The linkage between the frames is named uscounties, the same name as the frame being linked. That is, we assume

   . frlink m:1 countyid, frame(uscounties)

1. What it means to copy a linked variable

   When you type

   . frget median_income, from(uscounties)

   frget copies variable median_income from frame uscounties to the current frame. Well, we say it copies the variable, but the process is more complicated than that. frget copies the relevant observations of median_income from frame uscounties to the appropriate observations in the current frame. In the process, frget duplicates some observations and ignores others.

   If the person in observation 1 lives in county 401, then the median income recorded for county 401 in the uscounties frame is copied to observation 1 in the current frame.

   If the people in observations 2, 33, and 65 in the current frame reside in county 207, then the median income recorded for county 207 is duplicated in observations 2, 33, and 65 of the current frame.

   If the person in observation 3 lives in county 599 and there is no county 599 in the uscounties frame, then missing value . or "" is stored in observation 3.

   A copy of a variable from a linked frame is a copy of the relevant observations of the variable to the appropriate observations in the current frame when relevant observations exist.

2. frget can copy variables one at a time

   To copy variable median_income from frame uscounties to the current frame, type

   . frget median_income, from(uscounties)
To instead copy `median_income` to a new variable named `medinc` in the current frame, type

```
.frget medinc=median_income, from(uscounties)
```

3. **frget** allows variable names to be abbreviated

   *frget* allows abbreviations if you have not `set varabbrev off`. If `median_income` is the only variable beginning with `median` in the linked frame, you can type

   ```
   . frget median, from(uscounties)
   ```

   Variable `median_income` will be copied, and the new variable in the current frame will be named `median_income`.

   When using *frget*'s `newvar=varname` syntax, you can abbreviate the variable being copied that appears to the right of the equals sign:

   ```
   . frget medinc=median, from(uscounties)
   ```

4. **frget** can bring over groups of variables

   *frget* allows you to specify a `varlist`. Even though you type *frget* in the current frame, the `varlist` is interpreted in the linked frame. You can type

   ```
   . frget emp*, from(uscounties)
   . frget emp* median_income, from(uscounties)
   . frget emp* median, from(uscounties)
   . frget emp* m*, from(uscounties)
   . frget *, from(uscounties)
   ```

   When you specify a `varlist`, *frget* automatically omits the match variable or variables and any variables starting with an underscore (`_`). First, we will tell you why, and then, we will tell you a workaround.

   We start with a match variable. The match variable(s) in our example is match variable `countyid`. The variable has the same name in both frames. Pretend for a moment that *frget* did not exclude match variables. Then, if you tried to copy `countyid`, that would be an error because *frget* will not overwrite existing variables. That seems reasonable until you realize that it would also mean that *frget* would issue an error if you typed

   ```
   . frget c*, from(uscounties)
   ```

   or even if you typed

   ```
   . frget *, from(uscounties)
   ```

   *frget* would issue errors because `c*` and `*` would include `countyid`, which, being the match variable, already exists in the current frame. *frget* automatically omits match variables so that you can type *frget* `c*` and *frget* `*` and get all the other variables.

   *frget* omits `_*` variables because they tend to be Stata system variables that are valid only in the dataset in which they appear. You do not want them.

   What if you need to get one of these variables? Use the `newvar=varname` syntax. Type, for instance,

   ```
   . frget _myvar=_myvar, frame(uscounties)
   ```

   Automatic omission is not applied to this syntax.
5. `frget` copies all the variables specified, or none of them.
   `frget` will not overwrite existing variables. If just one variable in the specified list already exists in the current frame, `frget` copies none of the variables. It issues an error.

   ```
   . frget emp* m*, from(uscounties)
   variable mvalues already exists
   r(110);
   ```

   If you want all the `m*` variables except `mvalues`, use the `exclude()` option:

   ```
   . frget emp* m*, from(uscounties) exclude(mvalues)
   ```

   If you also want `mvalues` copied to `mvals` in the current frame, type

   ```
   . frget mvals=mvalues, from(uscounties)
   ```

6. `frget` ignores repeated variables
   It is not an error to type

   ```
   . frget employment employment, from(uscounties)
   ```

   We specified `employment` twice, but `frget` ignores that and copies the variable once. This is convenient because variables can be inadvertently repeated, as in

   ```
   . frget m* employment-larea, from(uscounties)
   ```

   Although you cannot see it, variable `mds` is repeated in the example. `m*` contains `mds`, and so does `employment-larea` because `mds` is among the variables stored between them.

   When variables are repeated using the `newvar=varname` syntax, `frget` does not ignore repetition. It copies the variables you specify to each of the new variables that you specify:

   ```
   . frget medinc=income inc=income, from(uscounties)
   ```

7. How to get all the variables 1: `frget *`
   To get all the variables, try typing

   ```
   . frget *, from(uscounties)
   ```

   This sometimes works. Other times it does not because some of the variables in `uscounties` already exist in the current frame. When it does not work, `frget` lists the variable names that exist in both frames and, even better, stores them in `r(dups)`. Thus, if you are willing to exclude those variables, you can type

   ```
   . frget *, from(uscounties) exclude('r(dups)')
   ```

8. How to get all the variables 2: `frget *`, `prefix()`
   Another way to get all the variables is to type

   ```
   . frget *, from(uscounties) prefix(c_)
   ```

   This brings in all the variables under their original names but prefixed with `c_`. The variable `mvalues` in the linked frame, for instance, is copied to `c_mvalues`.

   Another advantage of this approach is how easily you can drop the copies from the data should you desire to do so. Type

   ```
   . drop c_*
   ```

   You can choose your own prefix. If you prefer suffixing them, type

   ```
   . frget *, from(uscounties) suffix(_c)
   ```
This names the copies `mvalues_c`, etc. These names are more like the originals, at least if you use tab completion for typing them. Type the first characters of the original name and press tab. And if you wish, you can later drop the suffixed variables just as easily as prefixed ones. Type

```
    . drop *_c
```

9. How to create new variables

Assume that the `uscounties` frame contains variables `total_income` and `population`. You need `avg_income` in the current frame.

One solution would be

```
    . frget total_income population, from(uscounties)
    . generate avg_income = total_income/population
```

Another solution would be to use the `frval()` function to make the calculation directly:

```
    . generate avg_income = frval(uscounties, total_income)/frval(uscounties, population)
```

Here, however, is perhaps the best solution:

```
    . frame uscounties: generate avg_income = total_income/population
    . frget avg_income, from(uscounties)
```

It is not often that one has the opportunity to save computer time and memory. The gist of this approach is to create county-level variables in the `uscounties` frame and then use `frget` to get the ones you need.

10. `frget` copies and clones variables

When `frget` copies variables, it also copies their display formats, variable labels, value labels, notes, and characteristics.

The new variables are not just copies. They are clones.

### Stored results

`frget` stores the following in `r()`:

Scalars

- `r(k)` number of variables copied from linked frame

Macros

- `r(newlist)` new variables in the current frame
- `r(srclist)` variables copied from linked frame
- `r(excluded)` variables not copied from linked frame
- `r(dups)` variables already present in the current frame
- `r(notfound)` variables not found in the linked frame

`r(dups)` is present only if `frget` exits with an error message because a prospective new variable name already exists in the current frame.

`r(notfound)` is present only for syntax 2 when `frget` exits with an error message because a `varname` is not found in the linked frame.
Also see

[D] frlink — Link frames
[D] frames intro — Introduction to frames
[D] merge — Merge datasets
frlink — Link frames

Description

frlink creates and helps manage links between datasets in different frames. A link allows the variables in one frame to be accessed by another. See [D] frames intro if you do not know what a frame is.

Quick start

Create 1-to-1 linkage to frame fr2 and match on variable `matchvar`
```
frlink 1:1 matchvar, frame(fr2)
```

Create many-to-1 linkage to frame `fr3`, matching variables `v1` and `v2` in the current frame to variables `x1` and `x2` in frame `fr3`, naming the linkage `lnk`
```
frlink m:1 v1 v2, frame(fr3 x1 x2) generate(lnk)
```

List names of linkages in current frame
```
frlink dir
```

Show details for linkage `lnk`
```
frlink describe lnk
```

Attempt to re-create linkage `lnk` after data have changed
```
frlink rebuild lnk
```

Eliminate linkage `lnk`
```
drop lnk
```
Syntax

Create linkage between current frame and another

\[ \text{frlink } \{1:1 \mid m:1\} \text{ varlist}_1, \text{ frame(frame}_2 \ [\text{varlist}_2]) \ [\text{generate(linkvar}_1)] \]

List names of existing linkages

\text{frlink dir}

List details about existing linkage, and verify it is still valid

\text{frlink describe linkvar}_2

Re-create existing linkage when data have changed or frames are renamed

\text{frlink rebuild linkvar}_2 \ [\text{, frame(frame}_3)]

Drop existing linkage (dropping the variable eliminates the linkage)

\text{drop linkvar}_2

1:1 and m:1 indicate how observations are to be matched.

\text{varlist}_1 contains the match variables in the current frame, which we will call frame 1.

\text{linkvar}_1 is the name to be given to the new variable that \text{frlink} creates. The variable is added to the dataset in frame 1. The variable contains all the information needed to link the frames.

You specify the name for \text{linkvar}_1 using the \text{generate(linkvar}_1) option, or you let \text{frlink} name it for you. If \text{frlink()} chooses the name, the variable is given the same name as \text{frame}_2.

\text{linkvar}_2 is the name of an existing link variable.

\text{collect} is allowed with \text{frlink dir} and \text{frlink rebuild}; see [U] 11.1.10 \text{ Prefix commands}.

Options

Options are presented under the following headings:

Options for \text{frlink 1:1 and frlink m:1}  
Options for \text{frlink rebuild}

Options for \text{frlink 1:1 and frlink m:1}

\text{frame(frame}_2 \ [\text{varlist}_2]) specifies the name of the frame, \text{frame}_2, to which a linkage is created and optionally the names of variables in \text{varlist}_2 on which to match. If \text{varlist}_2 is not specified, the match variables are assumed to have the same names in both frames. \text{frame}() is required.

To create a link to a frame named \text{counties}, you can type

\text{. frlink m:1 countyid, frame(counties)}
This example omits specification of `varlist2`, and it works when the match variable `countyid` has the same name in both frames. If the variable were named `cntycode`, however, in the other frame, you type

```
.frlink m:1 countyid, frame(counties cntycode)
```

The rule for matching observations is thus that `countyid` in the current frame equals `cntycode` in the other frame.

You can specify multiple match variables when necessary. For example, you want to match on county names in U.S. data. County names repeat across the states, so you match on the combined county and state names by typing

```
.frlink m:1 countynname statename, frame(counties)
```

If the match variables had different names in frame `counties`, such as `county` and `state`, you type

```
.frlink m:1 countynname statename, frame(counties county state)
```

`generate(linkvar1)` specifies the name of the new variable that will contain all the information needed to link the frames. This variable is added to the dataset in frame 1. This option is rarely used.

If this option is not specified, the link variable will then be named the same as the frame name specified in the `frame()` option.

### Options for frlink rebuild

`frame(frame3)` specifies a frame name that differs from the existing linkage. `frame3` is the new name of a frame linked by `linkvar2`.

For instance, yesterday, you created a linkage named `george` to the data in the frame named `george` by typing

```
.frlink m:1 countynname statename, frame(george)
```

Today, you loaded the linked data into a frame named `counties`. To rebuild the linkage so that linkage `george` links to the data in frame `counties`, type

```
.frlink rebuild george, frame(counties)
```

If you also wish to rename the linkage to be `counties`, type

```
.rename george counties
```

Then you would have a linkage named `counties` to the data in the frame named `counties`.

### Remarks and examples

Remarks are presented under the following headings:

- **Overview of the frlink command**
- **Everything you need to know about linkages**
- **Example 1: A typical m:1 linkage**
- **How link variables work**
- **Advanced examples**
- **Example 2: A complex m:1 linkage**
- **Example 3: A 1:1 linkage, a simple solution to a hard problem**
Overview of the \texttt{frlink} command

\texttt{frlink 1:1} and \texttt{frlink m:1} create linkages between the current frame and another frame you specify. This adds a new variable to the current frame, known as the link variable. You can use the \texttt{frget} command to copy variables from the linked frame to the current frame and use the \texttt{frval()} function to use the other frame's variables in expressions.

Linkages are said to be named, but the name is in fact the name of the link variable that \texttt{frlink} creates.

\texttt{frlink dir} lists the names of existing linkages.

\texttt{frlink describe linkvar} displays details about the specified linkage. It also checks the validity of the link variable and, if there are problems, tells you how to fix it.

\texttt{frlink rebuild linkvar} re-creates the specified \texttt{linkvar}. If \texttt{linkvar} is invalid, \texttt{frlink rebuild} will fix it.

Type drop \texttt{linkvar} to delete linkages.

Everything you need to know about linkages

Here is everything you need to know in outline form:

1. A linkage connects one frame to another. Here are the advantages.
   1.1 The \texttt{frval()} function.
   1.2 The \texttt{frget} command.
2. The \texttt{frlink} command creates linkages.
3. Linkages are named.
4. A linkage is variable added to the data.
5. Drop the link variable, remove the link.
6. Do not modify the contents of the link variable.
7. Linkages are formed based on equality of the match variables.
8. You can specify more than one match variable.
9. Match variables can be named differently in the two frames.
10. Match type: One-to-one or many-to-one matching.
11. Linking can result in unmatched observations.
12. Linkages are directional.
13. How to create nested linkages.
14. Saving and using linked frames.
15. Do's and don'ts.

What follows will turn you into an expert.

1. A linkage connects one frame to another. Here are the advantages.

   Create a linkage and you can access the variables in another frame using the \texttt{frval()} function and the \texttt{frget} command.
1.1 The `frval()` function. You can type

```
. generate rel_income = income / frval(counties, median_income)
```

`frval(counties, median_income)` returns the value of the `median_income` variable in frame `counties`. If the current frame contained data on people and the county frame contained data on counties (linked to with link variable `counties` in the current frame), the above would produce person income divided by the median income of the county in which he or she resides. See `frval()` in [FN] Programming functions.

1.2 The `frget` command. You can type

```
(1) . frget median_income, from(counties)
(2) . frget medinc = median_income, from(counties)
(3) . frget median_income pop, from(counties)
(4) . frget median_income pop attr*, from(counties)
(5) . frget median_income pop attr*, from(counties) prefix(c_)
```

and more ...

(1) copies `median_income` from frame `counties` into the data in the current frame.

(2) does the same but names the variable `medinc`.

(3) copies two variables.

(4) copies lots of variables.

(5) copies lots of variables and renames them to start with `c_`.

This is only a smattering of what `frget` can do. See [D] frget.

2. The `frlink` command creates linkages.

`frlink` creates a linkage from the current frame to the frame you specify.

```
. frlink ..., frame(counties)
```

3. Linkages are named.

The command

```
. frlink ..., frame(counties)
```

creates a linkage named `counties` to the frame named `counties`.

You can specify option `generate()` to give the linkage a different name. To create a linkage named `c` to the frame `counties`, type

```
. frlink ..., frame(counties) generate(c)
```

4. A linkage is a variable added to the data.

The entire physical manifestation of a linkage is the addition of a single variable to the dataset in the current frame. Typing

```
. frlink ..., frame(counties)
```

adds new variable `counties` to the dataset in the current frame.

```
. frlink ..., frame(counties) generate(c)
```

adds new variable `c` to the dataset in the current frame.

The added variable is known as the “link variable”, or `linkvar`.
5. Drop the link variable, remove the link.
   Because linkages are just a variable, if you drop the variable, you remove the link.
   
   . drop counties
   . drop c

6. Do not modify the contents of the link variable.
   If you modify the link variable’s contents, you invalidate the linkage. If you are lucky, the
   next time you use the \texttt{frget} command or the \texttt{frval()} function, they will detect the problem
   and issue an error. If not, they will simply produce incorrect results.
   
   . replace counties = ... // Do not do this
   . replace c = ... // Do not do this

   If you accidentally modify the link variable’s contents, use \texttt{frlink rebuild} to repair it.
   
   . frlink rebuild counties
   . frlink rebuild c

7. Linkages are formed based on equality of match variables.
   To construct a link to frame \texttt{counties}, type
   
   . frlink ..., frame(counties)

   The complete command would have the dots filled in. Part of what needs to appear in place
   of the dots are the match variables. A more complete version of the command is
   
   . frlink ... countyid, frame(counties)

   We specified one match variable, \texttt{countyid}.

   Linkages are formed by matching observations in the current frame to observations in the
   other frame when their match variables are equal.

   In the example, the match variables are \texttt{countyid} in the current frame and \texttt{countyid} in
   the county frame. Observations are matched when the \texttt{countyid} variables are equal.

   Let’s unravel that. The data in the current frame are on people. \texttt{countyid} in the current
   frame records the county in which each person resides.

   Meanwhile, the data in the county frame contains information on counties, such as a county’s
   median income. Variable \texttt{countyid} in this frame records the county each observation
   describes.

   Observations in the two frames are matched when the county in which a person resides
   equals the county being described. Once we have formed the linkage by typing
   
   . frlink ... countyid, frame(counties)

   if we then type
   
   . generate rel_income = income / frval(counties, median_income)

   we obtain the ratio of each person’s income to the median income in the county in which
   he or she resides.

8. You can specify more than one match variable.
   We just considered the case of one match variable—\texttt{countyid}—in each of the frames:
   
   . frlink ... countyid, frame(counties)
Let's imagine that instead of containing `countyid`, the datasets contain `countyname`. Substituting `countyname` for `countyid` might be insufficient to form the desired linkage:

```
. frlink ... countyname, frame(counties)
```

County names in the United States are repeated across states. Monroe County, for instance, exists in Florida, Mississippi, Texas, and other states. To link the frames, we need to match on both county and state names:

```
. frlink ... countyname statename, frame(counties)
```

Because county and state names, taken together, uniquely identify the locations, the order in which we specify them is irrelevant:

```
. frlink ... statename countyname, frame(counties)
```

9. Match variables can be named differently in the two frames.

When we type

```
. frlink ... countyname statename, frame(counties)
```

we are stating the variables `countyname` and `statename` appear in both frames. If the names are different in the two frames, specify the names used in the current frame following the `frlink` command, and specify the names used in the other frame in the `frame()` option, after the frame’s name:

```
. frlink ... countyname statename, frame(counties cnty usstate)
```

countyname and statename are the variable names used in the current frame. The variables corresponding to them in frame counties are named cnty and usstate.

10. Match type: One-to-one or many-to-one matching.

Consider the linkage created by

```
. frlink ... countyid, frame(counties)
```

The current frame contains data on persons, and the other frame—counties—contains data on counties.

All that is needed to turn the above into a complete command is to replace the dots with a match type, which can be `1:1` or `m:1`. In this case, the match type should be `m:1`, and the full command is

```
. frlink m:1 countyid, frame(counties)
```

`m:1` stands for many-to-one matching. `m:1` means that is okay if more than one observation in the current frame matches the same observation in the other frame. We specify `m:1` because it is possible that multiple people in the current frame reside in the same county. If five people live in county 207, all five will match to the observation in frame counties that describes county 207.

The alternative `1:1` means that at most one observation in the current frame can match an observation in the other frame. Specifying `1:1` would be appropriate for matching person data in the current frame with more data on him or her in the other frame. If persons were to be matched on `personid` and if the other frame were named `person2`, we type

```
. frlink 1:1 personid, frame(person2)
```

Matched would be persons in the current frame who also appeared in the second frame.
If you think about it, 1:1 is a special case of m:1. 1:1 means at most one observation matches. m:1 means one or more observations match. This means that, if

```
.frlink 1:1 personid, frame(morepersons)
```

forms the linkage you want, so will

```
.frlink m:1 personid, frame(morepersons)
```

So why specify 1:1? We specify 1:1 so that frlink can issue an error message if the result is not 1:1. When matching people’s data to more data on the same people, if two people in the first frame matched the same observation in the second, that means

P1. there is an error in the first dataset: the same person appears more than once in it; or

P2. there is an error in variable personid in the first dataset: the personid variable contains the wrong value; or

P3. we are not thinking clearly and should have specified m:1 instead of 1:1.

You specify 1:1 so that the software can flag situations where the reality is different from your expectations. Then you fix your data or your thinking.

11. Linking can result in unmatched observations.

Imagine that you have successfully executed

```
.frlink m:1 countyid, frame(counties)
```

The result will be that each observation in the current frame will be matched or unmatched. Observations in the current frame are matched when the values of countyid are found in frame counties. The remaining observations, if any, are unmatched. Unmatched observations are not an error; they are a characteristic and perhaps a shortcoming of you datasets.

frlink tells you how many unmatched observations there are when you create the linkage. Function frval() will subsequently return missing values for the unmatched observations. If you type

```
.g generate relative_income = income/frval(linkvar, median_income)
```

variable relative_income would be missing (.) for the unmatched observations, the same as if unmatched observations were matched but contained median_income==.

frget behaves similarly. It sets the unmatched observations equal to missing in the copied variable.

```
.frget median_income, from(counties)
```

In addition, the link variable in the current frame contains missing values for the unmatched observations. This is useful. How many observations in the current frame are unmatched? If you do not remember, type

```
.count if counties==.
```

You can look at the data for the unmatched observations.

```
.browse if counties==.
```

You can analyze the unmatched data.

```
.summarize if counties==.
```
If observations will be useful to you only when they are matched with county data, you can keep just the matched data by typing

```
. keep if counties!=.
```

12. **Linkages are directional.**

We say that we link the current frame to another frame, but it’s really the other way around. Data flow to the current frame from the other frame. If you have created the linkage

```
. frlink m:1 countyid, frame(counties)
```

then you can access data in frame *counties* from the current frame, but you cannot access data in the current frame from frame *counties*.

13. **How to create nested linkages.**

Consider separate frames containing data on students, the schools they attend, and the counties in which the schools are located. Here is the setup:

- **Current frame:** *students.dta* containing variables for each student’s ID, the ID of the schools he or she attends, and student characteristics.
- **Frame *schools:* *schools.dta* containing each school’s ID, the ID of the counties in which the schools are located, and school characteristics.
- **Frame *counties:* *us_counties.dta* containing each county’s ID and county characteristics.

Here is how you load the datasets into the frames:

```
. frame create schools
. frame create counties
. use students
. frame schools: use schools
. frame counties: use us_counties
```

Here is how you link the frames:

```
. frlink m:1 schoolid, frame(schools)
. frget countyid, from(schools)
. frlink m:1 countyid, frame(counties)
```

The first command links students with the schools they attend.

The second command copies variable *countyid* from frame *schools* to the current frame.

The third command links students with the counties in which their schools are located.

The command that copied *countyid* into the current frame was necessary so that the students in the current frame could be linked to the county frame.

Said generically, if you have data in frames *A*, *B*, and *C*, you link frame *A* to *B* and link frame *A* to *C* to access all the data from *A*.

Said negatively, linkages are not transitive. Linking frame *A* to *B* and *B* to *C* is not sufficient to allow frame *A* to access all the data.

14. **Saving and using linked frames.**

You have created students-linked-to-county data:

```
. use students
. frame create counties
. frame counties: use us_counties
. frlink m:1 countyid, frame(counties)
```
To save the datasets so that you can use them later, you need only type

```
.save students, replace
```

It is necessary to save `students.dta` because it has a new variable in it, namely, the linkage variable `counties`. It is not necessary to save `us_counties.dta` because it has not changed.

That said, you might still wish to save both files:

```
.save students, replace
.frame counties: save us_counties, replace
```

The data in frame `counties` were not changed, but the sort order of the data changed. Linking sorts the linked-to frame on its match variables. We recommend you save both datasets.

To later load the data, you type

```
.use students
.frame create counties
.frame counties: use us_counties
```

You might want to put these lines in a do-file. You could call it `usestudents.do`. Then, whenever you wanted to load the data, all you need to do is type

```
.do usestudents
```

15. Do’s and don’ts.

We start with the don’ts. There are only three:

Do not modify the contents of the link variable,

... but if you do, use `frlink rebuild` to fix it.

Do not rename the match variables in either frame,

... but if you do, drop the link variable, and use `frlink m:1` or `1:1` to link the frames again.

Do not drop the match variables from either frame,

... and if you do, we cannot help you.

Everything else is a do, but they come in two flavors. The first is `do` without qualifications. The second is also a do, but do it only if you follow it by typing `frlink rebuild`.

Here are the do’s without qualifications:

Do drop the link variable. That’s how you eliminate the link.

Do rename the link variable.

Do drop observations in the current frame.

Do add new variables in either frame.

Do modify or rename variables in either frame, with the exception of the link and the match variables.
And here are the do’s with qualification, which is always the same: Type `frlink rebuild` afterward.

Do rebuild after adding observations in either or both frames.
Do rebuild after dropping observations in the linked frame.
Do rebuild after modifying the contents of the match variables in either or both frames.

And remember a rule that always applies:

It is always safe to type `frlink rebuild`.
If there is no problem, it will do nothing.
If there is a problem, it will fix it unless it cannot, ...
... then it explains why and do nothing to your data.

You are now an expert on linked frames.

**Example 1: A typical m:1 linkage**

File `persons.dta` contains data on people. Among its variables is `countyid`, containing the county code where each person resides.

File `txcounty.dta` contains data on Texas counties. Among its variables is `countyid`, the county code for the county that each observation describes.

Here is how we load and link the datasets:

```
  . use https://www.stata-press.com/data/r17/persons
  . frame create txcounty
  . frame txcounty: use https://www.stata-press.com/data/r17/txcounty
      (Median income in Texas counties)
  . frlink m:1 countyid, frame(txcounty)
      (all observations in frame default matched)
```

Linkages are for situations where you want to analyze the data in the current frame using variables from both frames.

Below, we create new variable `relative_income` in the current frame equal to `income` (in the current frame) divided by `median_income` (from the county frame):

```
  . generate relative_income = income / frval(txcounty, median_income)
  . summarize relative_income

      Variable |       Obs  Mean    Std. dev.   Min    Max
-------------|----------|---------|-----------------|-------|-------|
       relative_i-e |    20   .5501545    .1090887    .352133    .7038001
```

If we wanted to use `median_income` from the county frame in a linear regression, we would use the `frget` command to add `median_income` to the current frame’s data:

```
  . frget median_income, from(txcounty)
  . regress income ... median_income ...
```
We will not do that because `persons.dta` contains fictional values and is not worth the bother. But realize what would be possible if these datasets were real and contained more variables:

Get a variable:

```
frget median_income, from(txcounty)
```

Get a variable, but change its name:

```
frget medinc = median_income, from(txcounty)
```

Get a lot of variables:

```
frget median* nbus-pop, from(txcounty)
```

Get a lot of variables, but change their names to begin with `c_`:

```
frget median* nbus-pop, prefix(c_) from(txcounty)
```

See \[D\] `frget`.

### How link variables work

`frlink` performs two actions when it creates a link:

1. It adds the link variable to the dataset in the current frame.
2. It sorts the dataset in the other frame by its match variables.

In the example above, this means that

1. `frlink` adds variable `txcounty` to the data in the current frame.
2. `frlink` sorts the data in frame `txcounty` by `countyid`. (It literally executes `frame txcounty: sort countyid`.)

Look at variable `txcounty` in the first observations of `persons.dta` in the current frame:

```
. list in 1/5

<table>
<thead>
<tr>
<th>personid</th>
<th>countyid</th>
<th>income</th>
<th>txcounty</th>
<th>relati-e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>30818</td>
<td>5</td>
<td>.7038001</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>30752</td>
<td>3</td>
<td>.4225046</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>29673</td>
<td>2</td>
<td>.5230381</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>32115</td>
<td>3</td>
<td>.441231</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>31189</td>
<td>2</td>
<td>.5497603</td>
</tr>
</tbody>
</table>
```

Each observation of variable `txcounty` contains the observation number in frame `txcounty` that matches the current observation. The above list says that

- obs. 5 of frame `txcounty` matches obs. 1 of the current frame
- obs. 3 of frame `txcounty` matches obs. 2 of the current frame
- obs. 2 of frame `txcounty` matches obs. 3 of the current frame
- obs. 3 of frame `txcounty` matches obs. 4 of the current frame
- obs. 2 of frame `txcounty` matches obs. 5 of the current frame

... assuming the data in frame `txcounty` are sorted on `countyid`

Frame `txcounty` is the other frame. It is the other frame that must be sorted, not the data in the current frame.
Even so, the assumption is iffy. It is true after \texttt{frlink} creates the linkage because \texttt{frlink} itself sorts the data. And \texttt{frget} and \texttt{frval()} check the sort order before using the other frame’s data so that accidents do not happen.

The only way things can go wrong are 1) if you change the contents of the link variable \texttt{txcounty} or 2) you drop or modify the match variable \texttt{countyid}. So do not do that.

\textbf{Advanced examples}

Example 1 showed you how linkages are usually used. We linked person data to county data. We could show you another example that links student data to school data and student data to county data, but it amounts to nothing more than example 1, done twice.

We have two more examples to show you, but we admit that they are advanced and abstruse.

The first is an example in which linkage shines, but the solution is seldom useful beyond the particular example shown.

The second concerns 1:1 linkages. If 1:1 is appropriate for your problem, you probably want to merge the datasets, not link them. You probably want to use \texttt{merge}, not \texttt{frlink}. On occasion, however, a situation arises where linkage is a better solution. We show you one and provide guidelines on how to identify other such situations.

\textbf{Example 2: A complex m:1 linkage}

We have a dataset on families and the file is named, naturally enough, \texttt{family.dta}. The dataset contains information on variables of interest, as all datasets do, but that is not what makes this dataset interesting, so the variables are simply named \texttt{x1}, \texttt{x2}, ..., \texttt{x5}. What makes this dataset interesting is that it contains observations on related adult people. It contains adult children, parents, and grandparents.

Such data are notoriously difficult to process and analyze.

In the dataset, every person is identified by a person ID, called a “pid”. The data also contain the variables \texttt{pid\_m} and \texttt{pid\_f}, which are the pids for the person’s mother and father, if they too are in the data. The oldest generation in the data has \texttt{pid\_m==}. and \texttt{pid\_f==}.

One person in the data is person number 14982. Here are the values of ID variables for 14982:

\begin{verbatim}
 . list pid* if pid==14982
 8. 14982 695966 933335
\end{verbatim}

Variables \texttt{pid\_m} and \texttt{pid\_f} are the IDs of 14982’s mother and father. The mother is 695966 and the father, 933335.

Here are the recorded ID variables for 695966, 14982’s mother:

\begin{verbatim}
 . list pid* if pid==695966
 431. 695966 186484 238126
\end{verbatim}

14982’s maternal grandmother is 186484 and maternal grandfather, 238126.
Let’s stay with the maternal side of the family. Here are the ID variables for 186484, 14982’s maternal grandmother:

```
. list pid* if pid==186484

     pid   pid_m   pid_f
  +-------------+--------+--------+
 100.  186484   .   .
```

The grandmother’s variables have missing values for her mother’s and father’s ID, so we cannot continue back further. Nonetheless, there are other people in this dataset just like 14982, people on whom we have their data, their parents’ data, and their parents’ parents’ data.

`frlink` can link the data so that we have access to all of them. To do that, we will create six linkages, named

<table>
<thead>
<tr>
<th>linkage name</th>
<th>meaning linkage to</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>father</td>
</tr>
<tr>
<td>m</td>
<td>mother</td>
</tr>
<tr>
<td>mm</td>
<td>mother’s mother</td>
</tr>
<tr>
<td>mf</td>
<td>mother’s father</td>
</tr>
<tr>
<td>fm</td>
<td>father’s mother</td>
</tr>
<tr>
<td>ff</td>
<td>father’s father</td>
</tr>
</tbody>
</table>

Once we have these six linkages, we will be able to access variables for the person, his or her parents, and their parents. We will be able to do that using the `frval()` function or the `fret` command.

If we wanted to access `x1` using function `frval()`, we would do so with the following:

<table>
<thead>
<tr>
<th>value of <code>x1</code> desired</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>own value</td>
<td><code>x1</code></td>
</tr>
<tr>
<td>mother’s value</td>
<td><code>frval(m, x1)</code></td>
</tr>
<tr>
<td>father’s value</td>
<td><code>frval(f, x1)</code></td>
</tr>
<tr>
<td>mother’s mother’s value</td>
<td><code>frval(mm, x1)</code></td>
</tr>
<tr>
<td>mother’s father’s value</td>
<td><code>frval(mf, x1)</code></td>
</tr>
<tr>
<td>father’s mother’s value</td>
<td><code>frval(fm, x1)</code></td>
</tr>
<tr>
<td>father’s father’s value</td>
<td><code>frval(ff, x1)</code></td>
</tr>
</tbody>
</table>
If we wanted to copy all five variables of interest to the current frame, prefixed by their relationship, we would do so with the following:

<table>
<thead>
<tr>
<th>value of x1-x5 desired</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>own value</td>
<td>x1</td>
</tr>
<tr>
<td>mother’s variables</td>
<td>frget x1-x5, from(m) prefix(m)</td>
</tr>
<tr>
<td>father’s variables</td>
<td>frget x1-x5, from(f) prefix(f)</td>
</tr>
<tr>
<td>mother’s mother’s variables</td>
<td>frget x1-x5, from(mm) prefix(mm)</td>
</tr>
<tr>
<td>mother’s father’s variables</td>
<td>frget x1-x5, from(mf) prefix(mf)</td>
</tr>
<tr>
<td>father’s mother’s variables</td>
<td>frget x1-x5, from(fm) prefix(fm)</td>
</tr>
<tr>
<td>father’s father’s variables</td>
<td>frget x1-x5, from(ff) prefix(ff)</td>
</tr>
</tbody>
</table>

If we combined all 5 variables of interest from all 7 sources, we would have a total of 35 variables of interest. We could form that dataset by typing just six commands. Before we can do any of this, we must build the linkages.

To build them, we start in the usual way. We load the data of interest into the current frame and load into the other frame the data we want to link:

```
. clear all
. use https://www.stata-press.com/data/r17/family
   (Fictional family linkage data)
. frame create family
. frame family: use https://www.stata-press.com/data/r17/family // yes, the same data
   (Fictional family linkage data)
```

We are in fact going to link `family.dta` to itself, but `frlink` requires that linkages be made from the current frame to the other frame. Nonetheless, we will be able to create all six linkages to that single frame.

To create the first two linkages, we type

```
. frlink m:1 pid_m, frame(family pid) generate(m)
   (355 observations in frame default unmatched)
. frlink m:1 pid_f, frame(family pid) generate(f)
   (355 observations in frame default unmatched)
```

Because we are linking people to people, your natural inclination might be that the matching needs to be 1:1. That was our inclination too, but when we tried, `frlink` complained that the data were m:1 and refused. It took us a minute to realize why. Some of the people in the data have the same mother or father.

We have shown you the commands to build the first two linkages. Four remain to be built. What is different about these four is that the current frame does not contain the necessary match variable. Think about forming the mm linkage, which is the maternal grandmother of a person in the current frame. We need a variable containing the ID of the current person’s mother’s mother or `frval(m, pid_m)`. We could call the variable `pid_mm`, and construct it and the related match variables by typing
. generate pid_mm = frval(m, pid_m)  
(539 missing values generated)  
. generate pid_mf = frval(m, pid_f)  
(539 missing values generated)  
. generate pid_fm = frval(f, pid_m)  
(539 missing values generated)  
. generate pid_ff = frval(f, pid_f)  
(539 missing values generated)

Alternatively, we could have obtained them by using the `frget` command:

```
frget pid_mm = pid_m, from(m)  
frget pid_mf = pid_f, from(m)  
frget pid_fm = pid_m, from(f)  
frget pid_ff = pid_f, from(f)
```

It does not matter which we use.

Once we have the match variables, we can form the linkages:

```
. frlink m:1 pid_mm, frame(family pid) generate(mm)  
(539 observations in frame default unmatched)  
. frlink m:1 pid_mf, frame(family pid) generate(mf)  
(539 observations in frame default unmatched)  
. frlink m:1 pid_fm, frame(family pid) generate(fm)  
(539 observations in frame default unmatched)  
. frlink m:1 pid_ff, frame(family pid) generate(ff)  
(539 observations in frame default unmatched)
```

At this point, we are basically done. We are interested, however, in the sample of people for whom data on their parents and grandparents are available. We can drop the other people from the data in the current frame.

```
. drop if pid_m ==. | pid_f ==.  
(355 observations deleted)  
. drop if pid_mm==. | pid_mf==.  
(184 observations deleted)  
. drop if pid_fm==. | pid_ff==.  
(0 observations deleted)  
. count // number of observations remaining  
    100
```

We now have our data ready for analysis.

What are the chances that an even 100 people would be left? They would be nil if this were real data. We manufactured these data, however, so there is no reason to continue to analyze variables x1 through x5. They contain fictional values, and random.

**Example 3: A 1:1 linkage, a simple solution to a hard problem**

Most 1:1 cases are better handled by `merge`. Here is an exception.

You are working with hospital patient data, file `discharge1.dta`. The file contains variable `patientid` among other variables, and you receive additional data on the same patients, file `discharge2.dta`. Loading the two datasets into separate frames and linking them is easy to do.

```
. use https://www.stata-press.com/data/r17/discharge1, clear  
. frame create discharge2  
. frame discharge2: use https://www.stata-press.com/data/r17/discharge2  
. frlink 1:1 patientid, frame(discharge2)
```
But should we be doing this at all? Would it not be better to merge \texttt{discharge1.dta} with \texttt{discharge2.dta}? It usually would be. It would be if you received the following note from George:

Kathy: Here are new data on the 1,980 patients. The data contain the five variables that were previously omitted. – George.

\texttt{merge} will allow you to add these five new variables. Use it.

The note you received from George, however, reads

Kathy: Here are the data on the 1,980 patients. You’re not going to believe this, but even though they said there are five values that needed to be updated, they sent all the data again! Verify their claim, and tell me which variables they updated. – George.

This is a case for linking because you will not have to rename the 19 variables so that you can verify their claim. The link solution of handling George’s request is easier:

\begin{verbatim}
. use https://www.stata-press.com/data/r17/discharge1, clear
(Fictional WA hospital discharges)
. frame create discharge2
. frame discharge2: use https://www.stata-press.com/data/r17/discharge2
(Fictional WA hospital discharges)
. frlink 1:1 patientid, frame(discharge2)
(all observations in frame \texttt{default} matched)
. foreach v of varlist patientid-proc3code {
  2. quietly count if `v' != frval(discharge2, `v', discharge2)
  3. if (r(N)!=0) {
  4.    display "'`v': " r(N) " value(s) changed"
  5.  }
  6. }
sex: 1 value(s) changed
los: 1 value(s) changed
billed: 1 value(s) changed
diag1code: 1 value(s) changed
diag2code: 1 value(s) changed
\end{verbatim}

It turns out that the updated data are just as it was represented to be.

These data had two features that made them a candidate for linking rather than merging:

1. The sample of interest was the observations in the original data, the data in the current frame, and
2. lots of variables in the two datasets had the same names, and we were interested in both sets of values.

Let’s now think about other examples. Only some 1:1 problems will have feature 1. 1:1 matches in which you will subsequently analyze the merged data—\texttt{merge==3} in \texttt{merge} speak—will all have feature 1.

Feature 2 arises less often. In the example, the new data updated the old. Linkages make comparing values easier when the names are the same. Linkages in general make it easier when variable names are the same, even when there is no reason to compare them. Imagine that both datasets contain a variable called \texttt{income}, but they are different measures of income. In the combined result, you want them both, so you need to rename one of them. Now imagine that there are hundreds of variables and a handful share the same names across datasets even though they contain different concepts of whatever is being measured. Linkages make renaming them easy.
First, link the data:

```
. frlink personid, frame(newdata)
```

Then, try to copy all the variables:

```
. frget *, from(newdata)
```

The command will either work or tell you the variables that have the same name in both frames. Imagine that `frget` lists `income` and six other variables. You want to copy `income`, so you rename the variable:

```
. frame newdata: rename income farmincome
```

Now try again:

```
. frget *, from(newdata)
```

Of course the command does nothing but repeat the six variables that still have the same names in both frames. You review the list one last time and decide that you still do not care about those six variables. Then you type

```
. frget *, from(newdata) exclude('r(dups)')
```

This time it works! When variables have the same name, in addition to listing them, `frget` saves their names in `r(dups)`. That is why we typed `frget *, from(newdata)` when we knew we had not yet resolved all the duplicate names. We wanted `frget` to set `r(dups)` so that we could next tell `frget` to copy all the variables, except `exclude('r(dups)')`.

Now that we have gotten the variables of interest, we break the link:

```
. drop newdata
. frame drop newdata
```

The data in memory are now the same data that we could have coaxed `merge` into producing had we done everything right.

### Stored results

`frlink m:1` and `frlink 1:1` store the following in `r()`:

- Scalars
  - `r(unmatched)` # of observations in the current frame unable to be matched

`frlink dir` stores the following in `r()`:

- Scalars
  - `r(n_vars)` # of link variables
- Macros
  - `r(vars)` space-separated list of link-variable names

`frlink describe` stores nothing in `r()`.

`frlink rebuild` stores the following in `r()`:

- Scalars
  - `r(unmatched)` # of observations in the current frame unable to be matched
Also see

[D] \texttt{frget} — Copy variables from linked frame

[D] \texttt{frames intro} — Introduction to frames

[D] \texttt{merge} — Merge datasets
**generate** — Create or change contents of variable

---

**Description**

generate creates a new variable. The values of the variable are specified by `=exp`.

If no `type` is specified, the new variable type is determined by the type of result returned by `=exp`. A float variable (or a double, according to `set type`) is created if the result is numeric, and a string variable is created if the result is a string. In the latter case, if the string variable contains values greater than 2,045 characters or contains values with a binary 0 (\0), a strL variable is created. Otherwise, a str# variable is created, where # is the smallest string that will hold the result.

If a `type` is specified, the result returned by `=exp` must be a string or numeric according to whether `type` is string or numeric. If `str` is specified, a strL or a str# variable is created using the same rules as above.

See [D] egen for extensions to `generate`.

**replace** changes the contents of an existing variable. Because `replace` alters data, the command cannot be abbreviated.

**set type** specifies the default storage type assigned to new variables (such as those created by `generate`) when the storage type is not explicitly specified.

---

**Quick start**

Create numeric variable `newv1` equal to `v1 + 2`

```
    generate newv1 = v1 + 2
```

As above, but use type byte and label the values of `newv1` with value label `mylabel`

```
    generate byte newv1:mylabel = v1 + 2
```

String variable `newv2` equal to “my text”

```
    generate newv2 = "my text"
```

Variable `newv3` equal to the observation number

```
    generate newv3 = _n
```

Replace `newv3` with observation number within each value of `catvar`

```
    bycat var: replace newv3 = _n
```

Binary indicator for first observation within each value of `catvar` after sorting on `v2`

```
    bysort catvar (v2): generate byte first = _n==1
```

As above, but for last observation

```
    bysort catvar (v2): generate byte last = _n==_N
```

Combined datetime variable `newv4` from `%td formatted date and `%tc formatted time

```
    generate double newv4 = cofd(date) + time
```
Menu

**generate**
Data > Create or change data > Create new variable

**replace**
Data > Create or change data > Change contents of variable

Syntax

*Create new variable*

```
**generate** [**type**] **newvar** [**:lbname**] =**exp** [**if**] [**in**] [ , **before**(varname) | **after**(varname) ]
```

*Replace contents of existing variable*

```
**replace** **oldvar** =**exp** [**if**] [**in**] [ , **nopromote** ]
```

*Specify default storage type assigned to new variables*

```
**set** **type** {**float** | **double**} [ , **permanently** ]
```

where **type** is one of byte | int | long | float | double | str | str1 | str2 | ... | str2045.

See **Description** below for an explanation of str. For the other types, see [U] 12 Data.

by is allowed with **generate** and **replace**; see [D] by.

Options

**before**(varname) or **after**(varname) may be used with **generate** to place the newly generated variable in a specific position within the dataset. These options are primarily used by the Data Editor and are of limited use in other contexts. A more popular alternative for most users is **order** (see [D] order). **nopromote** prevents **replace** from promoting the variable type to accommodate the change. For instance, consider a variable stored as an integer type (byte, int, or long), and assume that you **replace** some values with nonintegers. By default, **replace** changes the variable type to a floating point (float or double) and thus correctly stores the changed values. Similarly, **replace** promotes byte and int variables to longer integers (int and long) if the replacement value is an integer but is too large in absolute value for the current storage type. **replace** promotes strings to longer strings. **nopromote** prevents **replace** from doing this; instead, the replacement values are truncated to fit into the current storage type. **permanently** specifies that, in addition to making the change right now, the new limit be remembered and become the default setting when you invoke Stata.
Remarks and examples

Remarks are presented under the following headings:

- `generate and replace`
- `set type`
- `Video examples`

**generate and replace**

`generate` and `replace` are used to create new variables and to modify the contents of existing variables, respectively. You can do anything with `replace` that you can do with `generate`. The only difference between the commands is that `replace` requires that the variable already exist, whereas `generate` requires that the variable be new. Because Stata is an interactive system, we force a distinction between replacing existing values and generating new ones so that you do not accidentally replace valuable data while thinking that you are creating a new piece of information.

Detailed descriptions of expressions are given in [U] 13 Functions and expressions.

Also see [D] `edit`.

**Example 1**

We have a dataset containing the variable `age2`, which we have previously defined as `age^2` (that is, \( age^2 \)). We have changed some of the `age` data and now want to correct `age2` to reflect the new values:

```
. use https://www.stata-press.com/data/r17/genxmpl1
   (Wages of women)
. generate age2=age^2
  variable age2 already defined
  r(110);
```

When we attempt to re-generate `age2`, Stata refuses, telling us that `age2` is already defined. We could drop `age2` and then re-generate it, or we could use the `replace` command:

```
. replace age2=age^2
   (204 real changes made)
```

When we use `replace`, we are informed of the number of actual changes made to the dataset.

You can explicitly specify the storage type of the new variable being created by putting the `type`, such as `byte`, `int`, `long`, `float`, `double`, or `str8`, in front of the variable name. For example, you could type `generate double revenue = qty * price`. Not specifying a type is equivalent to specifying `float` if the variable is numeric, or, more correctly, it is equivalent to specifying the default type set by the `set type` command; see below. If the variable is alphanumeric, not specifying a type is equivalent to specifying `str#`, where # is the length of the largest string in the variable.

You may also specify a value label to be associated with the new variable by including `":lblname"` after the variable name. This is seldom done because you can always associate the value label later by using the `label` values command; see [U] 12.6.3 Value labels.
Example 2

Among the variables in our dataset is \textit{name}, which contains the first and last name of each person. We wish to create a new variable called \textit{lastname}, which we will then use to sort the data. \textit{name} is a string variable.

\begin{verbatim}
. use https://www.stata-press.com/data/r17/genxmpl2, clear
. list name

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Johanna Roman</td>
</tr>
<tr>
<td>2</td>
<td>Dawn Mikulin</td>
</tr>
<tr>
<td>3</td>
<td>Malinda Vela</td>
</tr>
<tr>
<td>4</td>
<td>Kevin Crow</td>
</tr>
<tr>
<td>5</td>
<td>Zachary Bimslager</td>
</tr>
</tbody>
</table>

. generate lastname=word(name,2)
. describe
Contains data from https://www.stata-press.com/data/r17/genxmpl2.dta
Observations: 5  Variables: 2  18 Jan 2020 12:24

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage Type</th>
<th>Display Format</th>
<th>Variable Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>str17</td>
<td>%17s</td>
<td></td>
</tr>
<tr>
<td>lastname</td>
<td>str9</td>
<td>%9s</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.
\end{verbatim}

Stata is smart. Even though we did not specify the storage type in our \texttt{generate} statement, Stata knew to create a str9 \textit{lastname} variable, because the longest last name is Bimslager, which has nine characters.

Example 3

We wish to create a new variable, \textit{age2}, that represents the variable \textit{age} squared. We realize that because \textit{age} is an integer, \textit{age2} will also be an integer and will certainly be less than 32,740. We therefore decide to store \textit{age2} as an \texttt{int} to conserve memory:

\begin{verbatim}
. use https://www.stata-press.com/data/r17/genxmpl3, clear
. generate int age2=age^2
(9 missing values generated)
\end{verbatim}

Preceding \textit{age2} with \texttt{int} told Stata that the variable was to be stored as an \texttt{int}. After creating the new variable, Stata informed us that nine missing values were generated. \texttt{generate} informs us whenever it produces missing values.

See \textbf{[U] 13 Functions and expressions} and \textbf{[U] 26 Working with categorical data and factor variables} for more information and examples. Also see \textbf{[D] recode} for a convenient way to recode categorical variables.
Technical note

If you specify the `if` or `in` qualifier, the `=exp` is evaluated only for those observations that meet the specified condition or are in the specified range (or both, if both `if` and `in` are specified). The other observations of the new variable are set to missing:

```
. use https://www.stata-press.com/data/r17/genxmpl3, clear
. generate int age2=age^2 if age>30
   (290 missing values generated)
```

Example 4

`replace` can be used to change just one value, as well as to make sweeping changes to our data. For instance, say that we enter data on the first five odd and even positive integers and then discover that we made a mistake:

```
. use https://www.stata-press.com/data/r17/genxmpl4, clear
. list
   +-------+-------+
   | odd   | even  |
   |-------+-------|
   | 1     | 2     |
   | 3     | 4     |
   | -8    | 6     |
   | 7     | 8     |
   | 9     | 10    |
   +-------+-------+
```

The third observation is wrong; the value of `odd` should be 5, not −8. We can use `replace` to correct the mistake:

```
. replace odd=5 in 3
   (1 real change made)
```

We could also have corrected the mistake by typing `replace odd=5 if odd==-8`.

set type

When you create a new numeric variable and do not specify the storage type for it, say, by typing `generate y=x+2`, the new variable is made a `float` if you have not previously issued the `set type` command. If earlier in your session you typed `set type double`, the new numeric variable would be made a `double`.

Video examples

- How to create a new variable that is calculated from other variables
- How to identify and replace unusual data values
generate — Create or change contents of variable

References


Also see

[D] *compress* — Compress data in memory
[D] *corr2data* — Create dataset with specified correlation structure
[D] *drawnorm* — Draw sample from multivariate normal distribution
[D] *dyngen* — Dynamically generate new values of variables
[D] *edit* — Browse or edit data with Data Editor
[D] *egen* — Extensions to generate
[D] *encode* — Encode string into numeric and vice versa
[D] *label* — Manipulate labels
[D] *recode* — Recode categorical variables
[D] *rename* — Rename variable
[U] *12 Data*
[U] *13 Functions and expressions*
gsort — Ascending and descending sort

Description

gsort arranges observations to be in ascending or descending order of the specified variables and so differs from sort in that sort produces ascending-order arrangements only; see [D] sort.

Each varname can be numeric or a string.

The observations are placed in ascending order of varname if + or nothing is typed in front of the name and are placed in descending order if – is typed.

Quick start

Sort dataset in memory by ascending values of v1, equivalent to sort

gsort v1

Sort dataset in memory by descending values of v1

gsort -v1

Sort dataset by ascending values of v1 and descending values of v2

gsort v1 -v2

Create newv for use in subsequent by operations

gsort v1 -v2, generate(newv)

Place missing values of descending-order v2 at the top of the dataset instead of the end

gsort v1 -v2, mfirst

Menu

Data > Sort
gsort — Ascending and descending sort

Syntax

```
gsort [+|−] varname [ [+|−] varname ... ] [ , generate(newvar) mfirst ]
```

Options

- `generate(newvar)` creates `newvar` containing 1, 2, 3, ... for each group denoted by the ordered data. This is useful when using the ordering in a subsequent by operation; see [U] 11.5 by varlist: construct and examples below.
- `mfirst` specifies that missing values be placed first in descending orderings rather than last.

Remarks and examples

`gsort` is almost a plug-compatible replacement for `sort`, except that you cannot specify a general `varlist` with `gsort`. For instance, `sort alpha-gamma` means to sort the data in ascending order of `alpha`, within equal values of `alpha`; sort on the next variable in the dataset (presumably `beta`), within equal values of `alpha` and `beta`; etc. `gsort alpha-gamma` would be interpreted as `gsort alpha -gamma`, meaning to sort the data in ascending order of `alpha` and, within equal values of `alpha`, in descending order of `gamma`.

Example 1

The difference in `varlist` interpretation aside, `gsort` can be used in place of `sort`. To list the 10 lowest-priced cars in the data, we might type

```
.use https://www.stata-press.com/data/r17/auto
.gsorder price
.list make price in 1/10
```

or, if we prefer,

```
.gsorder +price
.list make price in 1/10
```

To list the 10 highest-priced cars in the data, we could type

```
.gsorder -price
.list make price in 1/10
```

`gsort` can also be used with string variables. To list all the makes in reverse alphabetical order, we might type

```
.gsorder -make
.list make
```

Example 2

`gsort` can be used with multiple variables. Given a dataset on hospital patients with multiple observations per patient, typing

```
.use https://www.stata-press.com/data/r17/bp3
.gsorder id time
.list id time bp
```
lists each patient’s blood pressures in the order the measurements were taken. If we typed

```
grsort id -time
list id time bp
```

then each patient’s blood pressures would be listed in reverse time order.

Technical note

Say that we wished to attach to each patient’s records the lowest and highest blood pressures observed during the hospital stay. The easier way to achieve this result is with `egen`’s `min()` and `max()` functions:

```
. egen lo_bp = min(bp), by(id)
. egen hi_bp = max(bp), by(id)
```

See [D] `egen`. Here is how we could do it with `grsort`:

```
. use https://www.stata-press.com/data/r17/bp3, clear
. grsort id bp
. by id: generate lo_bp = bp[1]
. grsort id -bp
. by id: generate hi_bp = bp[1]
. list, sepby(id)
```

This works, even in the presence of missing values of `bp`, because such missing values are placed last within arrangements, regardless of the direction of the sort.

Technical note

Assume that we have a dataset containing `x` for which we wish to obtain the forward and reverse cumulatives. The forward cumulative is defined as $F(X) = \text{the fraction of observations such that } x \leq X$. Again let’s ignore the easier way to obtain the forward cumulative, which would be to use Stata’s `cumul` command,

```
. set obs 100
. generate x = rnormal()
. cumul x, gen(cum)
```

(see [R] `cumul`). Eschewing `cumul`, we could type

```
. sort x
. by x: generate cum = _N if _n==1
. replace cum = sum(cum)
. replace cum = cum/cum[_N]
```

That is, we first place the data in ascending order of `x`; we used `sort` but could have used `gsort`. Next, for each observed value of `x`, we generated `cum` containing the number of observations that take on that value (you can think of this as the discrete density). We summed the density, obtaining the distribution, and finally normalized it to sum to 1.
The reverse cumulative $G(X)$ is defined as the fraction of data such that $x \geq X$. To obtain this, we could try simply reversing the sort:

```stata
. gsort -x
. by x: generate rcum = _N if _n==1
. replace rcum = sum(rcum)
. replace rcum = rcum/rcum[_N]
```

This would work, except for one detail: Stata will complain that the data are not sorted in the second line. Stata complains because it does not understand descending sorts (gsort is an ado-file). To remedy this problem, gsort’s generate() option will create a new grouping variable that is in ascending order (thus satisfying Stata’s narrow definition) and that is, in terms of the groups it defines, identical to that of the true sort variables:

```stata
. gsort -x, gen(revx)
. by revx: generate rcum = _N if _n==1
. replace rcum = sum(rcum)
. replace rcum = rcum/rcum[_N]
```

Also see

[D] sort — Sort data
hexdump — Display hexadecimal report on file

**Description**

hexdump displays a hexadecimal dump of a file or, optionally, a report analyzing the dump.

**Syntax**

```
hexdump filename [ , options ]
```

**options**

<table>
<thead>
<tr>
<th>option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyze</td>
<td>display a report on the dump rather than the dump itself</td>
</tr>
<tr>
<td>tabulate</td>
<td>display a full tabulation of the ASCII and extended ASCII characters in the analyze report</td>
</tr>
<tr>
<td>noextended</td>
<td>do not display printable extended ASCII characters</td>
</tr>
<tr>
<td>results</td>
<td>store results containing the frequency with which each character code was observed; programmer’s option</td>
</tr>
<tr>
<td>from(#)</td>
<td>dump or analyze first byte of the file; default is to start at first byte, from(0)</td>
</tr>
<tr>
<td>to(#)</td>
<td>dump or analyze last byte of the file; default is to continue to the end of the file</td>
</tr>
</tbody>
</table>

**Options**

- **analyze** specifies that a report on the dump, rather than the dump itself, be presented.
- **tabulate** specifies in the analyze report that a full tabulation of the ASCII and extended ASCII characters also be presented.
- **noextended** specifies that hexdump not display printable extended ASCII characters, characters in the range 161–254 or, equivalently, 0xa1–0xfe. (hexdump does not display characters 128–160 and 255.)
- **results** is for programmers. It specifies that, in addition to other stored results, hexdump store \( r(c0), r(c1), \ldots, r(c255) \), containing the frequency with which each character code was observed.
- **from(#)** specifies the first byte of the file to be dumped or analyzed. The default is to start at the first byte of the file, from(0).
- **to(#)** specifies the last byte of the file to be dumped or analyzed. The default is to continue to the end of the file.
Remarks and examples

hexdump is useful when you are having difficulty reading a file with `infix`, `infix`, or `import delimited`. Sometimes, the reason for the difficulty is that the file does not contain what you think it contains, or that it does contain the format you have been told, and looking at the file in text mode is either not possible or not revealing enough.

Pretend that we have the file `myfile.raw` containing

- Datsun 210 4589 35 5 1
- VW Scirocco 6850 25 4 1
- Merc. Bobcat 3829 22 4 0
- Buick Regal 5189 20 3 0
- VW Diesel 5397 41 5 1
- Pont. Phoenix 4424 19 . 0
- Merc. Zephyr 3291 20 3 0
- Olds Starfire 4195 24 1 0
- BMW 320i 9735 25 4 1

We will use `myfile.raw` with `hexdump` to produce output that looks like the following:

```
$ hexdump myfile.raw

Address  Hex representation  Character representation
         0 1 2 3 4 5 6 7 8 9 a b c d e f
0        4461 7473 756e 2032 3130 2020 2020 2034  Datsun 210  4
10       3538 3920 2033 3520 2035 2020 310a 5667  589 35 5 1.VW
20       2053 6369 726f 6363 6f20 2020 2036 3835  Scirocco  685
30       3020 2032 3520 2034 2020 310a 4d65 7263  0 25 4 1.Merc
40       2e20 426f 6263 6f20 2020 2033 3832 3920  . Bobcat  3829
50       2032 3220 2034 2020 300a 4275 6963 6b20  22 4 0.Buick
60       5265 6761 6c20 2020 2035 3138 3920 2032  Regal  5189 2
70       3020 2033 2020 300a 5657 2044 6965 7365  0 3 0.VW Diese
80       6c20 2020 2020 2035 3339 3720 2034 3120  1 5397 41
90       2035 2020 310a 506f 6e74 2e20 2068 6f65  5 1.Pont. Phoe
a0       6e69 7820 2034 3432 3420 2031 3920 202e  nix 4424 19 .
b0       2020 300a 4d65 7263 2e20 5a65 7068 7972  0.Merc. Zephyr

```
hexdump can also produce output that looks like the following:

. hexdump myfile.raw, analyze

<table>
<thead>
<tr>
<th>Line-end characters</th>
<th>Line length (tab=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r\n (Windows)</td>
<td>0 minimum</td>
</tr>
<tr>
<td>\r by itself (Mac)</td>
<td>0 maximum</td>
</tr>
<tr>
<td>\n by itself (Unix)</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space/separater characters</th>
<th>Number of lines</th>
<th>EOL at EOF?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[blank]</td>
<td>99</td>
<td>yes</td>
</tr>
<tr>
<td>[tab]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>[comma] (,)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control characters</th>
<th>Line 1</th>
<th>29</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTL excl. \r, \n, \t</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>DEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended (128-159,255)</td>
<td>0</td>
<td>29</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII printable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A-Z</td>
<td>20</td>
</tr>
<tr>
<td>a-z</td>
<td>61</td>
</tr>
<tr>
<td>0-9</td>
<td>77</td>
</tr>
<tr>
<td>Special (!@#$ etc.)</td>
<td>4</td>
</tr>
<tr>
<td>Extended (160-254)</td>
<td>0</td>
</tr>
</tbody>
</table>

| Total | 270 |

Observed were:
\n blank . 0 1 2 3 4 5 6 7 8 9 B D M O P R S V W Z a b c d e f g h i k l n o p r s t u x y

Of the two forms of output, the second is often the more useful because it summarizes the file, and the length of the summary is not a function of the length of the file. Here is the summary for a file that is just over 4 MB long:

. hexdump bigfile.raw, analyze

<table>
<thead>
<tr>
<th>Line-end characters</th>
<th>Line length (tab=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r\n (Windows)</td>
<td>147,456 minimum</td>
</tr>
<tr>
<td>\r by itself (Mac)</td>
<td>0 maximum</td>
</tr>
<tr>
<td>\n by itself (Unix)</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space/separater characters</th>
<th>Number of lines</th>
<th>EOL at EOF?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[blank]</td>
<td>1,622,039</td>
<td>yes</td>
</tr>
<tr>
<td>[tab]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>[comma] (,)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control characters</th>
<th>Line 1</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTL excl. \r, \n, \t</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>DEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended (128-159,255)</td>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII printable</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A-Z</td>
<td>327,684</td>
</tr>
<tr>
<td>a-z</td>
<td>999,436</td>
</tr>
<tr>
<td>0-9</td>
<td>1,261,587</td>
</tr>
<tr>
<td>Special (!@#$ etc.)</td>
<td>65,536</td>
</tr>
<tr>
<td>Extended (160-254)</td>
<td>0</td>
</tr>
</tbody>
</table>

| Total | 4,571,196 |

Observed were:
\n blank . 0 1 2 3 4 5 6 7 8 9 B D M O P R S V W Z a b c d e f g h i k l n o p r s t u x y
Here is the same file but with a subtle problem:

```
> hexdump badfile.raw, analyze

<table>
<thead>
<tr>
<th>Line-end characters</th>
<th>Line length (tab=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\r\n (Windows)</td>
<td>147,456 minimum</td>
</tr>
<tr>
<td>\r by itself (Mac)</td>
<td>0 maximum</td>
</tr>
<tr>
<td>\n by itself (Unix)</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space/separator characters</th>
<th>Number of lines</th>
<th>EDL at EOF?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[blank]</td>
<td>1,622,016</td>
<td>yes</td>
</tr>
<tr>
<td>[tab]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>[comma] (,)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Control characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary 0</td>
</tr>
<tr>
<td>CTL excl. \r, \n, \t</td>
</tr>
<tr>
<td>DEL</td>
</tr>
<tr>
<td>Extended (128-159,255)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASCII printable</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-Z</td>
</tr>
<tr>
<td>a-z</td>
</tr>
<tr>
<td>0-9</td>
</tr>
<tr>
<td>Special (!@#$ etc.)</td>
</tr>
<tr>
<td>Extended (160-254)</td>
</tr>
</tbody>
</table>

Total 4,571,196

Observed were:
\0 ^C ^D ^G \n \r ^U blank & . 0 1 2 3 4 5 6 7 8 9 B D E M O P R S U V W
Z a b c d e f g h i k l n o p r s t u v x y } ~ E^A E^C E^I E^M E^P
ö é ö 255
```

In the above, the line length varies between 30 and 90 (we were told that each line would be 30 characters long). Also the file contains what hexdump, analyze labeled control characters. Finally, hexdump, analyze declared the file to be BINARY rather than ASCII.

We created the second file by removing two valid lines from bigfile.raw (60 characters) and substituting 60 characters of binary junk. We would defy you to find the problem without using hexdump, analyze. You would succeed, but only after much work. Remember, this file has 147,456 lines, and only two of them are bad. If you print 1,000 lines at random from the file, your chances of listing the bad part are only 0.013472. To have a 50% chance of finding the bad lines, you would have to list 52,000 lines, which is to say, review about 945 pages of output. On those 945 pages, each line would need to be drawn at random. More likely, you would list lines in groups, and that would greatly reduce your chances of encountering the bad lines.

The situation is not as dire as we make it out to be because, were you to read badfile.raw by using infile, it would complain, and here it would tell you exactly where it was complaining. Still, at that point you might wonder whether the problem was with how you were using infile or with the data. Moreover, our 60 bytes of binary junk experiment corresponds to transmission error. If the problem were instead that the person who constructed the file constructed two of the lines differently, infile might not complain, but later you would notice some odd values in your data (because obviously you would review the summary statistics, right?). Here hexdump, analyze might be the only way you could prove to yourself and others that the raw data need to be reconstructed.
## Technical note

In the full hexadecimal dump,

```
$ hexdump myfile.raw
```

<table>
<thead>
<tr>
<th>address</th>
<th>hex representation</th>
<th>character</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>04461 7473 756e 2032 3130 2020 2020 2034</td>
<td>Datsun 210 4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>03538 3920 2033 3520 2035 2020 310d 0a56</td>
<td>58 35 5 1..V</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>05720 5363 6972 6f63 636f 2020 2020 3638</td>
<td>W Scirocco 68</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>03530 2020 3235 2020 3420 2031 0d0a 4d65</td>
<td>50 25 4 1..Me</td>
<td></td>
</tr>
</tbody>
</table>

(addresses (listed on the left) are listed in hexadecimal. Above, 10 means decimal 16, 20 means decimal 32, and so on. Sixteen characters are listed across each line.

In some other dump, you might see something like

```
$ hexdump myfile2.raw
```

<table>
<thead>
<tr>
<th>address</th>
<th>hex representation</th>
<th>character</th>
<th>representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>04461 7473 756e 2032 3130 2020 2020 2034</td>
<td>Datsun 210 4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>03538 3920 2033 3520 2035 2020 3120 2020</td>
<td>58 35 5 1</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>02020 2020 2020 0a56 5720 5363 6972 6f63</td>
<td>.VW Sciroc</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>0636f 2020 2020 3638 3530 203235 2020 2020</td>
<td>co 6850 25</td>
<td></td>
</tr>
</tbody>
</table>

The * in the address field indicates that the previous line is repeated until we get to hexadecimal address 160 (decimal 352).
Stored results

`hexdump`, analyze and `hexdump`, results store the following in `r()`:

Scalars

- `r(Windows)` number of `\r\n`
- `r(Mac)` number of `\r` by itself
- `r(Unix)` number of `\n` by itself
- `r(blank)` number of blanks
- `r(tab)` number of tab characters
- `r(comma)` number of comma (,) characters
- `r(ctl)` number of binary 0s; A–Z, excluding `\r`, `\n`, `\t`; DELs; and 128–159, 255
- `r(uc)` number of A–Z
- `r(lc)` number of a–z
- `r(digit)` number of 0–9
- `r(special)` number of printable special characters (!@#, etc.)
- `r(extended)` number of printable extended characters (160–254)
- `r(filesize)` number of characters
- `r(lmin)` minimum line length
- `r(lmax)` maximum line length
- `r(lnum)` number of lines
- `r(eoleof)` 1 if EOL at EOF, 0 otherwise
- `r(l1)` length of 1st line
- `r(l2)` length of 2nd line
- `r(l3)` length of 3rd line
- `r(l4)` length of 4th line
- `r(l5)` length of 5th line
- `r(c0)` number of binary 0s (results only)
- `r(c1)` number of binary 1s (^A) (results only)
- `r(c2)` number of binary 2s (^B) (results only)
- `r(c255)` number of binary 255s (results only)

Macros

- `r(format)` ASCII, EXTENDED ASCII, or BINARY

Also see

[D] filefilter — Convert ASCII or binary patterns in a file
[D] type — Display contents of a file
icd — Introduction to ICD commands

Description

This entry provides a brief introduction to the basic concepts of the International Classification of Diseases (ICD). If you are not familiar with ICD terminology, we recommend that you read this entry before proceeding to the individual command entries.

This entry also provides an overview of the format of the codes from each coding system that Stata’s `icd` commands support. Stata supports 9th revision codes (ICD-9) and 10th revision codes (ICD-10). For ICD-9, Stata uses codes from the United States’s Clinical Modification, the ICD-9-CM. For ICD-10, Stata uses the World Health Organization’s (WHO’s) codes for international morbidity and mortality reporting and the United States’s Clinical Modification (ICD-10-CM) and Procedure Coding System (ICD-10-PCS). We encourage you to read this entry to ensure that you choose the correct command and that your data are properly formatted for using the `icd` suite of commands.

Finally, this entry provides information about using the `icd` commands with multiple diagnosis or procedure codes at one time. None of the commands accepts a varlist, so we illustrate methods for working with multiple codes.

If you are familiar with ICD coding and the `icd` commands in Stata, you may want to skip to the command-specific entries for details about syntax and examples.

Commands for ICD-9 codes

- `icd9` ICD-9-CM diagnosis codes
- `icd9p` ICD-9-CM procedure codes

Commands for ICD-10 codes

- `icd10` ICD-10 diagnosis codes
- `icd10cm` ICD-10-CM diagnosis codes
- `icd10pcs` ICD-10-PCS procedure codes

Remarks and examples

Remarks are presented under the following headings:

- Introduction to ICD coding
- Terminology
- Diagnosis codes
- Procedure codes
- Working with multiple codes
Introduction to ICD coding

The `icd` commands in Stata work with four different diagnosis and procedure coding systems: ICD-9-CM, ICD-10, ICD-10-CM, and ICD-10-PCS.

The International Classification of Diseases (ICD) coding system was developed by and is copyrighted by the World Health Organization (WHO). The ICD coding system is used for standardized mortality reporting and, by many countries, for reporting of morbidity and coding of diagnoses during healthcare encounters. Since 1999, the ICD system has been under its 10th revision, ICD-10 (World Health Organization 2011). These codes provide information only about diagnoses, not about procedures.

The United States and some other countries have also developed country-specific coding systems that are extensions of WHO’s system. These systems are used for coding information about healthcare encounters. In the United States, the coding system is referred to as the International Classification of Diseases, Clinical Modification. These codes are maintained and distributed by the National Center for Health Statistics (NCHS) at the U.S. Centers for Disease Control and Prevention (CDC) and by the Centers for Medicare and Medicaid Services (CMS).

Terminology

The `icd9` and `icd10` entries assume knowledge of common terminology used in the ICD-9-CM documentation from the NCHS or CMS or in the ICD-10 revision manuals from WHO. The following brief definitions are provided as a reference.

**edition.** The ICD-9-CM and ICD-10 each have editions, which represent major periodic changes. ICD-9-CM is currently in its sixth edition (National Center for Health Statistics 2011). ICD-10 is currently in its fifth edition (World Health Organization 2011).

**version.** In the ICD-9-CM coding system, the version number is a sequential number assigned by CMS that is updated each Federal Fiscal Year when new codes are released. The last version was 32, which was published on October 1, 2014. In ICD-10-CM/PCS, the version corresponds to the Federal Fiscal Year.

**update.** In the ICD-10 coding system, an update may occur each year. The update is not issued with a number but may be identified by the year in which it occurred.

**category code.** A category code is the portion of the ICD code that precedes the period. It may represent a single disease or a group of related diseases or conditions.

**valid code.** A valid code is one that may be used for reporting in the current version of the ICD-10-CM/PCS or current update to the ICD-10 edition. What constitutes a valid code changes over time.

**defined code.** A defined code is any code that is currently valid, was valid at a previous time, or has meaning as a grouping of codes. See [D] icd9, [D] icd9p, [D] icd10, [D] icd10cm, and [D] icd10pcs for information about how the individual commands treat defined codes.

**Diagnosis codes**

Let’s begin with the diagnostic codes processed by `icd9`. An ICD-9-CM diagnosis code may have one of two formats. Most use the format

\[ \{0-9,V\}\{0-9\}\{0-9\}\[ . ]\{0-9\}[0-9]\]

while E-codes have the format

\[ E\{0-9\}\{0-9\}\{0-9\}\[ . ]\{0-9\}\]

where braces, \{\}, indicate required items and brackets, [ ], indicate optional items.
ICD-9-CM codes begin with a digit from 0 to 9, the letter V, or the letter E. E-codes are always followed by three digits and may have another digit in the fifth place. All other codes are followed by two digits and may have up to two more digits.

The format of an ICD-10 diagnosis code is

\[
\{ \text{A–T, V–Z} \} \{0–9\} \{0–9\} [ . ] [0–9]
\]

Each ICD-10 code begins with a single letter followed by two digits. It may have an additional third digit after the period.

ICD-10-CM diagnosis codes have up to seven characters; otherwise, the format is like that for ICD-10 codes. Each ICD-10-CM code begins with a single letter followed by a digit. However, ICD-10-CM permits the third character to be a digit, the letter A, or the letter B. This forms the category code. The fourth and fifth characters may be used to make up any potential subcategory code. For certain diagnoses, there exist only three-, four- or five-character codes, so the diagnosis code and (sub)category code are equivalent.

Finally, the sixth and seventh characters provide additional detail. A peculiarity of the ICD-10-CM coding system is that it is not strictly hierarchical. The letter X is used as a placeholder if a subcategory has not been defined at a particular level. For example, the code J09 indicates influenza due to an identified virus. There is no subcategory for J09, so the fourth character is an X, and additional detail about complications is provided in the fifth character.

Codes in ICD-10-CM may have up to four more alpha-numeric characters after the period. Only codes with the finest level of detail under a category code are considered valid.

Diagnosis codes must be stored in a string variable (see [D] Data types). For codes from either revision, the period separating the category code from the other digits is treated as implied if it is not present.

Technical note

There are defined five- and six-character ICD-10 codes. However, these codes are not part of the standard four-character system codified by WHO for international morbidity and mortality reporting and are not considered valid by icd10. See [D] icd10 for additional details about these codes and options for using icd10 with them.

Technical note

ICD-10 codes U00–U49 are reserved for use by WHO for provisional assignment of new diseases. Codes U50–U99 may be used for research to identify subjects with specific conditions under study for which there is no defined ICD-10 code (World Health Organization 2011).

If you are working in one of these specialized cases, see the technical note in Creating new variables under Remarks and examples of [D] icd10.
Procedure codes

The ICD-9-CM coding system also includes procedure codes. The format of ICD-9-CM procedure codes is

$$\{0-9\} \{0-9\} [.] [0-9] [0-9]$$

The general format of an ICD-10-PCS procedure code is a three-character category code followed by four alpha-numeric characters after an (implied) period. The full codes are always seven characters long and may be any combination of letters and numbers.

Procedure codes must be stored in a string variable.

Working with multiple codes

Oftentimes, multiple diagnoses or procedures are recorded for each observation. None of the \texttt{icd} commands accepts a varlist, but you can still work with multiple diagnosis or multiple procedure records. To use the \texttt{icd} commands with more than one diagnosis or procedure variable at a time, you must either first \texttt{reshape} your data or use a loop; see \texttt{[D] reshape} and \texttt{[P] forvalues}.

Example 1: Summarizing information from multiple variables

In example 1 of \texttt{[D] icd9}, we add a variable indicating whether each diagnosis code was invalid or undefined. Here we use the same extract from the National Hospital Discharge Survey (NHDS).

It is often more useful to add a single variable that summarizes the results from several diagnosis or procedure variables. For example, we may wish to add a variable indicating whether a particular diagnosis code or range of codes appeared in any field. Summary variables can be created from the results of the check subcommand with option \texttt{generate()} or the \texttt{generate} subcommand with option \texttt{range()} or option \texttt{category()}.

Suppose that we want a single variable that contains the number of improperly formatted or undefined codes that each discharge had. To illustrate, we use the \texttt{nhds2010} dataset, keeping the variables for discharge identifier (\texttt{recid}), patient age, and patient sex, as well as the three diagnosis variables. We list the first ten observations below.

```stata
. use https://www.stata-press.com/data/r17/nhds2010
(Adult same-day discharges, 2010)
. keep recid age sex dx1 dx2 dx3
. list in 1/10, noobs
```

<table>
<thead>
<tr>
<th>age</th>
<th>sex</th>
<th>dx1</th>
<th>dx2</th>
<th>dx3</th>
<th>recid</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>Female</td>
<td>4414</td>
<td>99811</td>
<td>14275</td>
<td>84</td>
</tr>
<tr>
<td>23</td>
<td>Male</td>
<td>25013</td>
<td>3572</td>
<td>-2506</td>
<td>105</td>
</tr>
<tr>
<td>63</td>
<td>Male</td>
<td>51909</td>
<td>1489</td>
<td>-V146</td>
<td>255</td>
</tr>
<tr>
<td>43</td>
<td>Female</td>
<td>9678</td>
<td>E8528</td>
<td>8</td>
<td>651</td>
</tr>
<tr>
<td>25</td>
<td>Female</td>
<td>V271</td>
<td>64421</td>
<td>16564</td>
<td>696</td>
</tr>
<tr>
<td>57</td>
<td>Female</td>
<td>5409</td>
<td>V1582</td>
<td>2V106</td>
<td>779</td>
</tr>
<tr>
<td>61</td>
<td>Female</td>
<td>27651</td>
<td>V1087</td>
<td>7V436</td>
<td>814</td>
</tr>
<tr>
<td>60</td>
<td>Male</td>
<td>9951</td>
<td>462</td>
<td>-2724</td>
<td>826</td>
</tr>
<tr>
<td>22</td>
<td>Male</td>
<td>42789</td>
<td>5409</td>
<td>-2780</td>
<td>833</td>
</tr>
<tr>
<td>49</td>
<td>Male</td>
<td>5770</td>
<td>29181</td>
<td>14255</td>
<td>863</td>
</tr>
</tbody>
</table>
The data are in wide form, so we specify `reshape long` with stub `dx` because our diagnosis codes are in `dx1`, `dx2`, and `dx3`. The observation identifier, `recid`, is specified in `i()`. `reshape` creates the new variable `dxnum` for us.

```
. reshape long dx, i(recid) j(dxnum)  
( j = 1 2 3 )
```

<table>
<thead>
<tr>
<th>Data</th>
<th>Wide</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>2,210</td>
<td>6,630</td>
</tr>
<tr>
<td>Number of variables</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>j variable (3 values)</td>
<td></td>
<td><code>dxnum</code></td>
</tr>
<tr>
<td>xij variables:</td>
<td></td>
<td><code>dx1 dx2 dx3</code> -&gt; <code>dx</code></td>
</tr>
</tbody>
</table>

The output shows that `dxnum` has 3 values, so we know that all three diagnosis variables were recognized by `reshape`.

```
. list in 1/9, sepby(recid) noobs
```

<table>
<thead>
<tr>
<th>recid</th>
<th>dxnum</th>
<th>age</th>
<th>sex</th>
<th>dx</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td>1</td>
<td>85</td>
<td>Female</td>
<td>4414</td>
</tr>
<tr>
<td>84</td>
<td>2</td>
<td>85</td>
<td>Female</td>
<td>99811</td>
</tr>
<tr>
<td>84</td>
<td>3</td>
<td>85</td>
<td>Female</td>
<td>14275</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>23</td>
<td>Male</td>
<td>25013</td>
</tr>
<tr>
<td>105</td>
<td>2</td>
<td>23</td>
<td>Male</td>
<td>3572</td>
</tr>
<tr>
<td>105</td>
<td>3</td>
<td>23</td>
<td>Male</td>
<td>-2506</td>
</tr>
<tr>
<td>255</td>
<td>1</td>
<td>63</td>
<td>Male</td>
<td>51909</td>
</tr>
<tr>
<td>255</td>
<td>2</td>
<td>63</td>
<td>Male</td>
<td>1489</td>
</tr>
<tr>
<td>255</td>
<td>3</td>
<td>63</td>
<td>Male</td>
<td>-V146</td>
</tr>
</tbody>
</table>

Notice that our data on `recid`, `age`, and `sex` are retained and duplicated for each new observation.

If you are working with a large dataset, you may wish to drop variables other than a merge key and your diagnosis (or procedure) variables to conserve space and speed up `reshape`.

After we `reshape`, we create `prob` using `icd9 check`, an indicator for whether there was a problem with a given diagnosis code. We then use `egen` to create `anyprob`, the total number of codes that had a problem within each `recid`. See `[D] egen` for information about summary functions.

```
. icd9 check dx, generate(prob)  
(dx contains 358 missing values)
```

```
dx contains invalid codes: 
1. Invalid placement of period 0
2. Too many periods 0
3. Code too short 177
4. Code too long 0
5. Invalid 1st char (not 0-9, E, or V) 875
6. Invalid 2nd char (not 0-9) 128
7. Invalid 3rd char (not 0-9) 0
8. Invalid 4th char (not 0-9) 0
9. Invalid 5th char (not 0-9) 36
10. Code not defined 778
```

Total 1,994

```
. generate anyprob=prob>0
. by recid, sort: egen numprobs=total(anyprob)
```
Before we reshape, we drop prob and anyprob because they are specific to diagnosis variables. By construction, numprobs is constant within recid, so we do not specify it when we reshape.

```stata
. drop prob anyprob
. reshape wide dx, i(recid) j(dxnum) (j = 1 2 3)
```

Data     Long ->  Wide

Number of observations  6,630 -> 2,210
Number of variables 6 -> 7
j variable (3 values) dxnum -> (dropped)
xij variables:

```
dx -> dx1 dx2 dx3
```

The three diagnosis variables are restored to the dataset. We have added a single variable showing the total number of codes with problems for each record.

Example 2: Adding multiple variables from ICD codes

Now suppose that rather than creating a summary variable flagging any problem as we did in example 1, we want a new variable for each diagnosis variable indicating whether there is a coding problem. In example 1 of [D] icd9, we icd9 check each diagnosis variable separately, which requires us to type the command three times. While this is not burdensome for 3 variables, the full NHDS includes 14 diagnosis variables, for which we almost certainly would not want to type separate commands.

The easiest way to accomplish this is with a loop. We use forvalues because our codes all end in a number.
. use https://www.stata-press.com/data/r17/nhds2010, clear
(Adult same-day discharges, 2010)
. forvalues i=1/3 {
    2. icd9 check dx`i', generate(dx`i'_prob)
    3. }
(dx1 contains defined ICD-9-CM codes; no missing values)
(dx2 contains defined ICD-9-CM codes; 179 missing values)
(dx3 contains 179 missing values)

dx3 contains invalid codes:
   1. Invalid placement of period 0
   2. Too many periods 0
   3. Code too short 177
   4. Code too long 0
   5. Invalid 1st char (not 0-9, E, or V) 875
   6. Invalid 2nd char (not 0-9) 128
   7. Invalid 3rd char (not 0-9) 0
   8. Invalid 4th char (not 0-9) 0
   9. Invalid 5th char (not 0-9) 36
  10. Code not defined 778

    Total                1,994

This is exactly what we obtain in example 1 of [D] icd9.

If our variables had not been numbered sequentially, we could have either renamed them or used foreach; see [P] foreach.

The methods shown above will work for any of the icd9, icd9p, icd10, icd10cm, or icd10pcs data management commands.

References


Also see

[D] icd9 — ICD-9-CM diagnosis codes
[D] icd9p — ICD-9-CM procedure codes
[D] icd10 — ICD-10 diagnosis codes
[D] icd10cm — ICD-10-CM diagnosis codes
[D] icd10pcs — ICD-10-PCS procedure codes
**icd9 — ICD-9-CM diagnosis codes**

### Description

icd9 is a suite of commands for working with ICD-9-CM diagnosis codes from the 16th version (effective October 1998) to the 32nd version. To see the current version of the ICD-9-CM diagnosis codes and any changes that have been applied, type icd9 query.

icd9 check, icd9 clean, and icd9 generate are data management commands. icd9 check verifies that a variable contains defined ICD-9-CM diagnosis codes and provides a summary of any problems encountered. icd9 clean standardizes the format of the codes. icd9 generate can create a binary indicator variable for whether the code is in a specified set of codes, a variable containing a corresponding higher-level code, or a variable containing the description of the code.

icd9 lookup and icd9 search are interactive utilities. icd9 lookup displays descriptions of the codes specified on the command line. icd9 search looks for relevant ICD-9-CM diagnosis codes from keywords given on the command line.

### Quick start

Determine whether ICD-9-CM diagnosis codes in diag1 are invalid, and store reasons in invalid

```
icd9 check diag1, generate(invalid)
```

Standardize display of codes in diag2 to remove all periods, and align codes by padding with spaces

```
icd9 clean diag2, pad
```

Create descr3 as the diagnosis code prepended to short description of diagnosis code in diag3

```
icd9 generate descr3 = diag3, description long
```

Create diabetes as an indicator for a diabetes diagnosis in diag4 using ICD-9-CM codes 250.xx

```
icd9 generate diabetes = diag4, range(25000/25093)
```

Look up descriptions for ICD-9-CM diagnosis codes E827.0 to E828.9

```
icd9 lookup E8270/E8289
```

### Menu

Data > ICD codes > ICD-9
Syntax

Verify that variable contains defined codes

\[ \text{icd9 check } \text{varname} \ [if] \ [in] \ [, \ \text{any list generate(newvar)}] \]

Clean variable and verify format of codes

\[ \text{icd9 clean varname} \ [if] \ [in] \ [, \ \text{dots pad}] \]

Generate new variable from existing variable

\[ \text{icd9 generate newvar = varname} \ [if] \ [in], \ \text{category} \]
\[ \text{icd9 generate newvar = varname} \ [if] \ [in], \ \text{description} \ [long \ end] \]
\[ \text{icd9 generate newvar = varname} \ [if] \ [in], \ \text{range(codelist)} \]

Display code descriptions

\[ \text{icd9 lookup codelist} \]

Search for codes from descriptions

\[ \text{icd9 search} \ ["text[" ["text[" ["text"] ... [", or] \]

Display ICD-9 code source

\[ \text{icd9 query} \]

codelist is

\[ \text{icd9code} \] (the particular code)
\[ \text{icd9code*} \] (all codes starting with)
\[ \text{icd9code/icd9code} \] (the code range)

or any combination of the above, such as 001* 018/019 E* 018.02. icd9codes must be typed with leading 0s. For example, type 001; typing 1 will result in an error.

collect is allowed with icd9 check, icd9 clean, and icd9 lookup; see [U] 11.1.10 Prefix commands.
Options

Options are presented under the following headings:

- Options for icd9 check
- Options for icd9 clean
- Options for icd9 generate
- Option for icd9 search

Options for icd9 check

any tells icd9 check to verify that the codes fit the format of ICD-9-CM diagnosis codes but not to check whether the codes are defined.

list specifies that icd9 check list the observation number, the invalid or undefined ICD-9-CM diagnosis code, and the reason the code is invalid or whether it is an undefined code.

generate(newvar) specifies that icd9 check create a new variable containing, for each observation, 0 if the observation contains a defined code or is missing. Otherwise, it contains a number from 1 to 10. The positive numbers indicate the kind of problem and correspond to the listing produced by icd9 check.

Options for icd9 clean

dots specifies that the period be included in the final format. If dots is not specified, then all periods are removed.

pad specifies that icd9 clean pad the codes with spaces, front and back, to make the (implied) dots align vertically in listings. Specifying pad makes the resulting codes look better when used with most other Stata commands.

Options for icd9 generate

category, description, and range(codelist) specify the contents of the new variable that icd9 generate is to create. You do not need to icd9 clean varname before using icd9 generate; it will accept any supported format or combination of formats.

category creates a new variable that contains ICD-9-CM diagnosis category codes. The resulting variable may be used with the other icd9 subcommands. For diagnosis codes, the category code is the first three characters, except for E-codes, when it is the first four characters.

description creates newvar containing descriptions of the ICD-9-CM diagnosis codes.

long is for use with description. It specifies that the code be prepended to the text describing the code.

end modifies long (specifying end implies long) and places the code at the end of the string.

range(codelist) creates a new indicator variable equal to 1 when the ICD-9-CM diagnosis code is in the range specified, equal to 0 when the ICD-9-CM diagnosis code is not in the range, and equal to missing when varname is missing.
Option for icd9 search

or specifies that ICD-9-CM diagnosis codes be searched for descriptions that contain any word specified with icd9 search. The default is to list only descriptions that contain all the words specified.

Remarks and examples

Remarks are presented under the following headings:

- Using icd9 and icd9p
- Verifying and cleaning variables
- Interactive utilities
- Creating new variables

If you have not yet read Introduction to ICD coding in [D] icd, please do so before using the icd9 commands.

Using icd9 and icd9p

The ICD-9-CM coding system includes diagnosis and procedure codes. Some examples of diagnosis codes are 552.3 (Diaphragmatic hernia with obstruction) and E871.0 (Foreign object left in body during surgical operation). Some example of procedure codes are 01.2 (Craniotomy and craniectomy) and 55.23 (Closed renal biopsy).

Many datasets record (and some people write) codes without the period; for example, diagnosis code 550.1 may appear as 5501. The icd9 commands understand both ways of recording codes. The commands are also insensitive to codes recorded with or without leading and trailing blanks. For E-codes and V-codes, the icd9 commands are case insensitive. All the following codes are acceptable formats.

```
   diagnosis   procedure
    001       27.62
    001.       72
     00581   32.6
      552.3  97.11
       E800.2   872
      e8002   5523
       v82.2   08.51
```

Important note: What constitutes a valid code changes between versions. For the rest of this entry, a defined code is any code that is currently valid, was valid at some point since version 16 (V16, effective 1 October 1998), or has meaning as a grouping of codes. The list of valid codes and their associated descriptions is from the U.S. Centers for Medicare and Medicaid Services (CMS). These codes are jointly maintained and distributed by the U.S. Centers for Disease Control and Prevention’s National Center for Health Statistics and by CMS (Centers for Disease Control and Prevention 2013).

In icd9, descriptions that end with an asterisk (*) are used to denote codes that are invalid for medical coding purposes but are defined as a category code or a subcategory code that has been further subdivided. For example, diagnosis code 001 (Cholera) is invalid without a fourth digit but is defined as a category code, so its description appears as cholera*. CMS does not distribute short descriptions of category and subcategory codes that are defined but not valid for coding. To ensure that Stata reports that these codes are defined, we added them to the dataset icd9 uses with a description of *.
Codes that were valid in the past, but no longer are, have descriptions that end with a hash mark (#). For example, the diagnosis code 645.01 was deleted between V16 and V18. It remains a defined code, and its description appears as prolonged preg-delivered#.

To view the current version of ICD-9-CM diagnosis codes in Stata, its source, and a log of changes that have been made to the list of ICD-9-CM codes since the icd9 commands were implemented, type

```
.icd9 query
```

**ICD9 Diagnostic Code Mapping Data for use with Stata, History**

(output omitted)

**V32**

Dataset obtained 26aug2014 from

<http://www.cms.gov/Medicare/Coding/ICD9ProviderDiagnosticCodes/enum codes.html>, by selecting the 'Version 32...' file. Can be gotten directly via

<http://www.cms.gov/Medicare/Coding/ICD9ProviderDiagnosticCodes/Downloads/ICD-9-CM-v32-master-descriptions.zip>. After unzipping, the useful file name is "CMS32_DESC_SHORT_DX.txt (there are other files we did not use)."

09oct2014: V32 put into Stata distribution

BETWEEN V31 and V32: There were no additional codes.

BETWEEN V31 and V32: 0 codes were deleted.

BETWEEN V31 and V32: There were no description changes.

(output omitted)

Throughout the remainder of this entry, we use nhds2010.dta, an extract of adult same-day discharges from the 2010 National Hospital Discharge Survey (NHDS). Below we `describe` the data and `list` the first five observations for the diagnosis and procedure code variables.

```
. use https://www.stata-press.com/data/r17/nhds2010
(Adult same-day discharges, 2010)
. describe
```

Contains data from https://www.stata-press.com/data/r17/nhds2010.dta

<table>
<thead>
<tr>
<th>Observations:</th>
<th>2,210</th>
<th>Adult same-day discharges, 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables:</td>
<td>15</td>
<td>30 Jan 2020 15:03 (_dta has notes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ageu</td>
<td>byte</td>
<td>%8.0g</td>
<td>ageu</td>
<td>Units for age</td>
</tr>
<tr>
<td>age</td>
<td>byte</td>
<td>%8.0g</td>
<td>Age</td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td>byte</td>
<td>%8.0g</td>
<td>sex</td>
<td>Sex</td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td>race</td>
<td>Race</td>
</tr>
<tr>
<td>month</td>
<td>byte</td>
<td>%8.0g</td>
<td>Discharge month</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>byte</td>
<td>%8.0g</td>
<td>Discharge status</td>
<td></td>
</tr>
<tr>
<td>region</td>
<td>byte</td>
<td>%8.0g</td>
<td>Region</td>
<td></td>
</tr>
<tr>
<td>atype</td>
<td>byte</td>
<td>%8.0g</td>
<td>atype</td>
<td>Type of admission</td>
</tr>
<tr>
<td>dx1</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 1</td>
<td></td>
</tr>
<tr>
<td>dx2</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 2</td>
<td></td>
</tr>
<tr>
<td>dx3</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 3 (imported incorrectly)</td>
<td></td>
</tr>
<tr>
<td>dx3corr</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 3 (corrected)</td>
<td></td>
</tr>
<tr>
<td>pr1</td>
<td>str4</td>
<td>%9s</td>
<td>Procedure 1</td>
<td></td>
</tr>
<tr>
<td>wgt</td>
<td>int</td>
<td>%12.0g</td>
<td>Frequency weight</td>
<td></td>
</tr>
<tr>
<td>recid</td>
<td>float</td>
<td>%9.0g</td>
<td>Order of record (raw data)</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: recid
Verifying and cleaning variables

icd9 check verifies that varname contains defined ICD-9-CM codes and, if not, provides a full report on the problems. It is a good idea to begin with this command and fix any potential problems before proceeding to other icd9 commands. However, the check subcommand is also useful for tracking down problems when any of the other icd9 commands tell you that the “variable does not contain ICD-9 codes”.

icd9 clean modifies the variable to ensure consistency and to make subsequent output look better. This is not strictly necessary because all icd9 commands work equally well with cleaned or uncleaned codes. icd9 clean also can be used to verify that the codes in a variable conform with the ICD-9-CM diagnosis format, without checking to see whether the codes are defined.

Example 1: Checking the validity of a variable

We noticed when we listed our data that dx3 appears to be padded with dashes instead of spaces. As a preemptive step, we replace the dashes with spaces by using the subinstr() function because the icd9 commands ignore spaces.

```
. replace dx3=subinstr(dx3,"-", " ",.)
(1,009 real changes made)
```

Now that we have replaced the characters we know will be a problem, we can icd9 check the diagnosis variables. We add the generate() option so that we can identify any observations with invalid codes.

```
. icd9 check dx1, generate(prob1)
(dx1 contains defined ICD-9-CM codes; no missing values)
```

```
. icd9 check dx2, generate(prob2)
(dx2 contains defined ICD-9-CM codes; 179 missing values)
```
We see that all codes in dx1 are valid and all discharges have a primary diagnosis recorded. Likewise, all codes in dx2 are defined, and we see that 179 observations did not have a second diagnosis.

However, icd9 check reports that 1,000 of the 2,210 observations on dx3 have some sort of problem: 79 codes are too short, 128 have an invalid second character, and 793 are undefined. After some investigation, we discover that when we imported the data, we started reading from the wrong position in the file. Hereafter, we use the correctly imported variable, dx3corr.

Rather than typing the icd9 check command once for each variable, we could have checked all three simultaneously. See Working with multiple codes in [D] icd.

Example 2: Standardizing the format of codes

If we plan to do any reporting with these codes later, we may want to make them more readable. Suppose we want to report the primary diagnosis and procedure for each discharge. We can use icd9 clean with the dots and pad options to add the period between the category code and any subsequent digits and to align the periods.

Using icd9 clean with undefined codes will not result in an error message. So if you are using codes from a country other than the United States, the clean subcommand can still be used to standardize the format of your codes and check for correct placement of the period.

Interactive utilities

icd9 search looks for relevant ICD-9-CM diagnosis codes from the description given on the command line, and icd9 lookup lists the descriptions of codes given on the command line. The two commands complement each other.
Example 3: Finding diagnosis codes

Suppose that we want to identify the observations for which the primary diagnosis is congestive heart failure (CHF). As part of a quick exploratory analysis, we can use `icd9 search` to find ICD-9-CM codes that we may want to use to define our study population. We use the terms “heart failure” and “chf”. We enclose “heart failure” in quotation marks and use the `or` option so that `icd9 search` looks for either term.

```
. icd9 search "heart failure" chf, or
5 matches found:
  398.91 rheumatic heart failure
  428 heart failure*
  428.0 chf nos
  428.1 left heart failure
  428.9 heart failure nos
```

Because the descriptions are abbreviated, we are concerned that some of the 428 codes may be left out. So we use `icd9 lookup` to list a range of codes.

```
. icd9 lookup 428*
19 matches found:
  428 heart failure*
  428.0 chf nos
  428.1 left heart failure
  428.2 *
  428.20 systolic hrt failure nos
  428.21 ac systolic hrt failure
  428.22 chr systolic hrt failure
  428.23 ac on chr syst hrt fail
  428.3 *
  428.30 diastolic hrt failure nos
  428.31 ac diastolic hrt failure
  428.32 chr diastolic hrt fail
  428.33 ac on chr diast hrt fail
  428.4 *
  428.40 syst/diast hrt fail nos
  428.41 ac syst/diastol hrt fail
  428.42 chr syst/diast1 hrt fail
  428.43 ac/chr syst/dia hrt fail
  428.9 heart failure nos
```

The same result could be found by typing

```
. icd9 lookup 428/4289
```

if we knew that 428.9 was the last code in the 428 category.

Creating new variables

`icd9 generate` produces new variables based on existing variables containing (cleaned or uncleaned) ICD-9-CM diagnosis codes. `icd9 generate, category` creates `newvar` containing the category code that corresponds to the code in the existing variable. `icd9 generate, description` creates `newvar` containing the abbreviated textual description of the ICD-9-CM diagnosis code. `icd9 generate, range()` produces numeric `newvar` containing 1 if `varname` records an ICD-9-CM diagnosis code in the range listed and containing 0 otherwise.
Example 4: Creating an indicator variable

We review the list of codes we found in example 3 and decide that we will use 398.91 and all of the 428 codes in our definition of a CHF diagnosis. Now we can use `icd9 generate` with the `range()` option to create an indicator variable.

```
    . icd9 generate chf = dx1, range(398.91 428*)
    . tabulate chf [fweight=wgt]
```

<table>
<thead>
<tr>
<th>chf</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>563,048</td>
<td>97.88</td>
<td>97.88</td>
</tr>
<tr>
<td>1</td>
<td>12,192</td>
<td>2.12</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Total 575,240 100.00

After tabulating the results, we see that about 2.1% of all same-day discharges were for CHF in 2010.

Technical note

The dataset that supports `icd9` includes all codes that were added or deleted between V16 and the last version (V32). However, the descriptions were updated with each new version. If you are using `icd9 generate` with option `description` for codes from a version other than 32, please review the `icd9 query` log for any changes to descriptions between the version you are using and version 32.

Example 5: Combining commands for reporting

The `icd9 generate` commands are useful in isolation, but their real power comes when they are combined. For example, suppose that we want to make a graph showing the number of discharges in each diagnosis category for ICD-9-CM chapter 4, “Diseases of Blood and Blood Forming Organs”. We could use several `generate` commands and string functions, but `icd9 generate` greatly reduces our work.

First, we extract the category code from the detailed diagnosis code. Then, because the `icd9` commands work equally well with complete codes or category codes, we can use `icd9 generate` with the `range(280/289)` option to create an indicator variable for whether the discharge had a primary diagnosis in chapter 4.

```
    . icd9 generate dx1cat = dx1, category
    . icd9 generate ch4 = dx1cat, range(280/289)
```

Next, we create a variable with the descriptions of the category codes in chapter 4.

```
    . icd9 generate ch4des = dx1cat if ch4==1, description long
```

Finally, we use `graph hbar` to make a horizontal bar graph showing the frequencies of same-day discharges by diagnosis category.
. graph hbar (count) [fweight=wgt], over(ch4des) ytitle(Discharges)
> title(Diseases of Blood and Blood Forming Organs, span)
> subtitle(Same-day Discharges (2010), span)

See [G-2] graph bar for information about customizing the graph above. For more information about graphing results, see [G-2] graph.

Stored results

icd9 check stores the following in r():

Scalars

r(e#)            number of errors of type #
r(ESUM)          total number of errors

icd9 clean stores the following in r():

Scalars

r(N)             number of changes

icd9 lookup stores the following in r():

Scalars

r(N)             number of codes found

References


Also see

[D] icd  — Introduction to ICD commands
[D] icd9p  — ICD-9-CM procedure codes
[D] icd10cm  — ICD-10-CM diagnosis codes
icd9p is a suite of commands for working with ICD-9-CM procedure codes from the 16th version (effective October 1998) to the 32nd version. To see the current version of the ICD-9-CM procedure codes and any changes that have been applied, type icd9p query.

icd9p check, icd9p clean, and icd9p generate are data management commands. icd9p check verifies that a variable contains defined ICD-9-CM procedure codes and provides a summary of any problems encountered. icd9p clean standardizes the format of the codes. icd9p generate can create a binary indicator variable for whether the code is in a specified set of codes, a variable containing a corresponding higher-level code, or a variable containing the description of the code.

icd9p lookup and icd9p search are interactive utilities. icd9p lookup displays descriptions of the codes specified on the command line. icd9p search looks for relevant ICD-9-CM procedure codes from keywords given on the command line.

Quick start

Determine whether ICD-9-CM procedure codes in proc1 are invalid, and store reasons in invalid
icd9p check proc1, generate(invalid)

Standardize display of codes in proc2 to remove all periods
icd9p clean proc2

Create descr3 as the procedure code prepended to short description of procedure code in proc3
icd9p generate descr3 = proc3, description long

Create eye as an indicator for eye surgery in proc4 using ICD-9-CM procedure codes 16.1 through 16.99
icd9p generate eye = proc4, range(16*)

Look up descriptions for ICD-9-CM procedure codes 25.0 through 25.4 and 25.9 through 25.99
icd9p lookup 25.0/25.4 25.9*

Menu

Data > ICD codes > ICD-9
Syntax

Verify that variable contains defined codes

\[ \text{icd9p check } \text{varname [if] [in] [, any list generate(newvar)]} \]

Clean variable and verify format of codes

\[ \text{icd9p clean } \text{varname [if] [in] [, dots pad]} \]

Generate new variable from existing variable

\[ \text{icd9p generate newvar = varname [if] [in], category} \]
\[ \text{icd9p generate newvar = varname [if] [in], description [long end]} \]
\[ \text{icd9p generate newvar = varname [if] [in], range(codelist)} \]

Display code descriptions

\[ \text{icd9p lookup codelist} \]

Search for codes from descriptions

\[ \text{icd9p search ["text[" ["text[" ... [, or]} \]

Display ICD-9 code source

\[ \text{icd9p query} \]

codelist is

\[ \text{icd9code} \quad (\text{the particular code}) \]
\[ \text{icd9code*} \quad (\text{all codes starting with}) \]
\[ \text{icd9code/icd9code} \quad (\text{the code range}) \]

or any combination of the above, such as 50.21 37.7* 88.71/88.79. icd9codes must be typed with leading 0s. For example, type 01; typing 1 will result in an error.

collect is allowed with icd9p check, icd9p clean, and icd9p lookup; see [U] 11.1.10 Prefix commands.
Options

Options are presented under the following headings:

- Options for icd9p check
- Options for icd9p clean
- Options for icd9p generate
- Option for icd9p search

Options for icd9p check

- `any` tells `icd9p check` to verify that the codes fit the format of ICD-9-CM procedure codes but not to check whether the codes are defined.
- `list` specifies that `icd9p check` list the observation number, the invalid or undefined ICD-9-CM procedure code, and the reason the code is invalid or whether it is an undefined code.
- `generate(newvar)` specifies that `icd9p check` create a new variable containing, for each observation, 0 if the observation contains a defined code or is missing. Otherwise, it contains a number from 1 to 10. The positive numbers indicate the kind of problem and correspond to the listing produced by `icd9p check`.

Options for icd9p clean

- `dots` specifies that `icd9p clean` include the period in the final format. If `dots` is not specified, then all periods are removed.
- `pad` specifies that `icd9p clean` pad the codes with spaces, front and back, to make the (implied) dots align vertically in listings. Specifying `pad` makes the resulting codes look better when used with most other Stata commands.

Options for icd9p generate

- `category`, `description`, and `range(codelist)` specify the contents of the new variable that `icd9p generate` is to create. You do not need to `icd9p clean varname` before using `icd9p generate`; it will accept any supported format or combination of formats.
- `category` creates a new variable that contains ICD-9-CM procedure category codes. The resulting variable may be used with the other `icd9p` subcommands. For procedure codes, the category code is the first two characters.
- `description` creates `newvar` containing descriptions of the ICD-9-CM procedure codes.
  - `long` is for use with `description`. It specifies that the code be prepended to the text describing the code.
  - `end` modifies `long` (specifying `end` implies `long`) and places the code at the end of the string.
- `range(codelist)` creates a new indicator variable equal to 1 when the ICD-9-CM procedure code is in the range specified, equal to 0 when the ICD-9-CM procedure code is not in the range, and equal to missing when `varname` is missing.
Option for icd9p search

or specifies that ICD-9-CM procedure codes be searched for descriptions that contain any word specified with icd9p search. The default is to list only descriptions that contain all the words specified.

Remarks and examples

Remarks are presented under the following headings:

- Verifying and cleaning variables
- Interactive utilities
- Creating new variables

If you have not yet read Introduction to ICD coding in [D] icd, please do so before using the icd9p commands. Please also see Using icd9 and icd9p in [D] icd9 for information about Stata’s implementation of the ICD-9 coding system.

Throughout the remainder of this entry, we use nhds2010.dta, an extract of adult same-day discharges from the 2010 National Hospital Discharge Survey (NHDS). Below we describe the data.

```
. use https://www.stata-press.com/data/r17/nhds2010
(Adult same-day discharges, 2010)
. describe
Contains data from https://www.stata-press.com/data/r17/nhds2010.dta
Observations: 2,210 Adult same-day discharges, 2010
Variables: 15 30 Jan 2020 15:03
(_dta has notes)
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ageu</td>
<td>byte</td>
<td>%8.0g</td>
<td>ageu</td>
<td>Units for age</td>
</tr>
<tr>
<td>age</td>
<td>byte</td>
<td>%8.0g</td>
<td>Age</td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td>byte</td>
<td>%8.0g</td>
<td>sex</td>
<td>Sex</td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td>race</td>
<td>Race</td>
</tr>
<tr>
<td>month</td>
<td>byte</td>
<td>%8.0g</td>
<td>Discharge month</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>byte</td>
<td>%8.0g</td>
<td>status</td>
<td>Discharge status</td>
</tr>
<tr>
<td>region</td>
<td>byte</td>
<td>%8.0g</td>
<td>region</td>
<td>Region</td>
</tr>
<tr>
<td>atype</td>
<td>byte</td>
<td>%8.0g</td>
<td>atype</td>
<td>Type of admission</td>
</tr>
<tr>
<td>dx1</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 1</td>
<td></td>
</tr>
<tr>
<td>dx2</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 2</td>
<td></td>
</tr>
<tr>
<td>dx3</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 3 (imported incorrectly)</td>
<td></td>
</tr>
<tr>
<td>dx3corr</td>
<td>str5</td>
<td>%9s</td>
<td>Diagnosis 3 (corrected)</td>
<td></td>
</tr>
<tr>
<td>pr1</td>
<td>str4</td>
<td>%9s</td>
<td>Procedure 1</td>
<td></td>
</tr>
<tr>
<td>wgt</td>
<td>int</td>
<td>%12.0g</td>
<td>Frequency weight</td>
<td></td>
</tr>
<tr>
<td>recid</td>
<td>float</td>
<td>%9.0g</td>
<td>Order of record (raw data)</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: recid

Verifying and cleaning variables

icd9p check verifies that varname contains defined ICD-9-CM procedure codes and, if not, provides a full report on the problems. It is a good idea to begin with this command and fix any potential problems before proceeding to other icd9p commands. However, the check subcommand is also useful for tracking down problems when any of the other icd9p commands tell you that the “variable does not contain ICD-9 codes”.

\[
\text{\textbf{icd9p — ICD-9-CM procedure codes 391}}
\]
icd9p clean modifies the variable to ensure consistency and to make subsequent output look better. This is not strictly necessary because all icd9p commands work equally well with cleaned or uncleaned codes. icd9p clean also can be used to verify that the codes in a variable conform with the ICD-9-CM procedure format, without checking to see whether the codes are defined.

Example 1: Standardizing the format of codes

If we plan to do any reporting with the codes in our data, we may want to make them more readable. Suppose we want to report the primary procedure for each discharge. We can use icd9p clean with the dots option to add the period between the category code and any subsequent digits.

```
. icd9p clean pr1, dots pad
  (821 changes made)
. list recid pr1 in 1/5
```

```
  recid  pr1
  1.     84  38.34
  2.     105
  3.     255
  4.     651
  5.     696  73.59
```

Using icd9p clean with undefined codes will not result in an error message. So if you are using codes from a country other than the United States, the clean subcommand can still be used to standardize the format of your codes and check for correct placement of the period.

Interactive utilities

icd9p search looks for relevant ICD-9-CM procedure codes from the description given on the command line, and icd9p lookup lists the descriptions of codes given on the command line. The two commands complement each other.

Example 2: Finding procedure code descriptions

If we wanted to find the corresponding abbreviated description for procedure code 38.34, we would type

```
. icd9p lookup 38.34
1 match found:
  38.34 aorta resection & anast
```

If you are curious, the cryptic result translates into resection with anastomosis of the aorta.

To find a list of other procedure codes for resection with anastomosis and their descriptions, we could type icd9p lookup 38.3*. Or if we were interested in finding codes for procedures on the aorta, we could type

```
. icd9p search aorta
(output omitted)
```
Creating new variables

*icd9p generate* produces new variables based on existing variables containing (cleaned or uncleaned) ICD-9-CM procedure codes. *icd9p generate, category* creates *newvar* containing the category code that corresponds to the code in the existing variable. *icd9p generate, description* creates *newvar* containing the abbreviated textual description of the ICD-9-CM procedure code. *icd9p generate, range()* produces numeric *newvar* containing 1 if *varname* records an ICD-9-CM procedure code in the range listed and containing 0 otherwise.

Example 3: Adding descriptions to codes

In example 4 of [D] *icd9*, we created an indicator variable for whether a patient had congestive heart failure (CHF). We may want to know what procedures were performed for patients with CHF. We check the procedure codes in *pr1* and then generate a new variable with their descriptions. We include the *long* option so that we can see the ICD-9-CM procedure code as well.

```
. icd9p check pr1
(pr1 contains defined ICD-9-CM procedure codes; 1389 missing values)
. icd9p generate pr1descr = pr1, description long
. tabulate pr1descr [fweight=wgt] if chf==1, missing sort
```

<table>
<thead>
<tr>
<th>label for pr1</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.22 left heart cardiac cath</td>
<td>7,185</td>
<td>58.93</td>
<td>58.93</td>
</tr>
<tr>
<td>92.05 c-vasc scan/isotop funct</td>
<td>1,906</td>
<td>15.63</td>
<td>74.57</td>
</tr>
<tr>
<td>88.72 dx ultrasound-heart</td>
<td>1,027</td>
<td>8.42</td>
<td>82.99</td>
</tr>
<tr>
<td>03.31 spinal tap</td>
<td>776</td>
<td>6.36</td>
<td>89.35</td>
</tr>
<tr>
<td>39.95 hemodialysis</td>
<td>498</td>
<td>4.08</td>
<td>93.44</td>
</tr>
<tr>
<td>34.91 thoracentesis</td>
<td>388</td>
<td>3.18</td>
<td>96.62</td>
</tr>
<tr>
<td>99.60 cardiopulm resuscita nos</td>
<td>138</td>
<td>1.13</td>
<td>97.75</td>
</tr>
<tr>
<td>37.94 implt/repl carddefib tot</td>
<td>110</td>
<td>0.90</td>
<td>99.57</td>
</tr>
<tr>
<td>89.44 cardiac stress test nec</td>
<td>52</td>
<td>0.43</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Total | 12,192 | 100.00 |

We see that the majority of same-day discharges (58.9%) did not involve any procedure. When a procedure was performed, the most common was left heart cardiac catheterization (15.6%).

Technical note

The dataset that supports *icd9p* includes all codes that were added or deleted between V16 and the last version (V32). However, the descriptions were updated with each new version. If you are using *icd9p generate* with option *description* for codes from a version other than 32, please review the *icd9p query* log for any changes to descriptions between the version you are using and version 32.
Stored results

`icd9p check` stores the following in `r()`:

Scalars
- `r(e#)` number of errors of type `#`
- `r(esum)` total number of errors

`icd9p clean` stores the following in `r()`:

Scalars
- `r(N)` number of changes

`icd9p lookup` stores the following in `r()`:

Scalars
- `r(N)` number of codes found

References


Also see

[D] `icd` — Introduction to ICD commands
[D] `icd9` — ICD-9-CM diagnosis codes
[D] `icd10pcs` — ICD-10-PCS procedure codes
Description

icd10 is a suite of commands for working with the World Health Organization’s (WHO’s) ICD-10 diagnosis codes from the second edition (2003) to the fifth edition (2016). To see the current version of the ICD-10 diagnosis codes and any changes that have been applied, type icd10 query.

icd10 check, icd10 clean, and icd10 generate are data management commands. icd10 check verifies that a variable contains defined ICD-10 diagnosis codes and provides a summary of any problems encountered. icd10 clean standardizes the format of the codes. icd10 generate can create a binary indicator variable for whether the code is in a specified set of codes, a variable containing a corresponding higher-level code, or a variable containing the description of the code.

icd10 lookup and icd10 search are interactive utilities. icd10 lookup displays descriptions of the codes specified on the command line. icd10 search looks for relevant ICD-10 diagnosis codes from keywords given on the command line.

Quick start

Determine whether ICD-10 diagnosis codes in diag1 are invalid, and store reasons in invalid
icd10 check diag1, generate(invalid)

Standardize display of codes in diag2 to add a period and left-align codes
icd10 clean diag2, replace

Generate descr3 as descriptions of the diagnosis codes in diag3
icd10 generate descr3 = diag3, description

Generate binary indicator for malignant or benign neoplasm, as indicated by an ICD-10 code beginning with C or D in diag4
icd10 generate cancer = diag4, range(C* D*)

Look up current descriptions for ICD-10 diagnosis codes W70 through W79
icd10 lookup W70/W79

Look up codes where the description contains the words “delivery” or “birth”
icd10 search delivery birth, or
Syntax

Verify that variable contains defined codes

```
icd10 check varname [if] [in] [, checkopts]
```

Clean variable and verify format of codes

```
icd10 clean varname [if] [in], {generate(newvar) | replace} [cleanopts]
```

Generate new variable from existing variable

```
icd10 generate newvar = varname [if] [in], {category | short} [check]
icd10 generate newvar = varname [if] [in], description [genopts]
icd10 generate newvar = varname [if] [in], range(codelist) [check]
```

Display code descriptions

```
icd10 lookup codelist [, version(#)]
```

Search for codes from descriptions

```
icd10 search "text" [["text"] ...] [, searchopts]
```

Display ICD-10 version

```
icd10 query
```

codelist is one of the following:

- `icd10code` (the particular code)
- `icd10code*` (all codes starting with)
- `icd10code/icd10code` (the code range)

or any combination of the above, such as `A27.0 G40* Y60/Y69.9`.

<table>
<thead>
<tr>
<th>checkopts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmtonly</td>
<td>check only format of the codes</td>
</tr>
<tr>
<td>summary</td>
<td>frequency of each invalid or undefined code</td>
</tr>
<tr>
<td>list</td>
<td>list observations with invalid or undefined ICD-10 codes</td>
</tr>
<tr>
<td>generate(newvar)</td>
<td>create new variable marking invalid codes</td>
</tr>
<tr>
<td>version(#)</td>
<td>year to check codes against; default is version(2016)</td>
</tr>
</tbody>
</table>
**cleanopts**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>generate(newvar)</em></td>
<td>create new variable containing cleaned codes</td>
</tr>
<tr>
<td><em>replace</em></td>
<td>replace existing codes with the cleaned codes</td>
</tr>
<tr>
<td>check</td>
<td>check that variable contains ICD-10 codes before cleaning</td>
</tr>
<tr>
<td>nodots</td>
<td>format codes without a period</td>
</tr>
<tr>
<td>pad</td>
<td>add space to the right of three-character codes</td>
</tr>
</tbody>
</table>

*Either generate() or replace is required.*

**genopts**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addcode(begin</td>
<td>end)</td>
</tr>
<tr>
<td>pad</td>
<td>add spaces to the right of the code; must specify addcode(begin)</td>
</tr>
<tr>
<td>nodots</td>
<td>format codes without a period; must specify addcode()</td>
</tr>
<tr>
<td>check</td>
<td>check that variable contains ICD-10 codes before generating new variable</td>
</tr>
<tr>
<td>version(#)</td>
<td>select description from year #; default is version(2016)</td>
</tr>
</tbody>
</table>

**searchopts**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>match any keyword</td>
</tr>
<tr>
<td>matchcase</td>
<td>match case of keywords</td>
</tr>
<tr>
<td>version(#)</td>
<td>search description from year #; default is all</td>
</tr>
</tbody>
</table>

collect is allowed with icd10 check and icd10 clean; see [U] 11.1.10 Prefix commands.

**Options**

Options are presented under the following headings:

- Options for icd10 check
- Options for icd10 clean
- Options for icd10 generate
- Option for icd10 lookup
- Options for icd10 search

Warning: The option descriptions are brief and use jargon. Please read Introduction to ICD coding in [D] icd before using the icd10 command.

**Options for icd10 check**

fmtonly tells icd10 check to verify that the codes fit the format of ICD-10 diagnosis codes but not to check whether the codes are defined.

summary specifies that icd10 check should report the frequency of each invalid or undefined code that was found in the data. Codes are displayed in descending order by frequency. summary may not be combined with list.

list specifies that icd10 check list the observation number, the invalid or undefined ICD-10 diagnosis code, and the reason the code is invalid or whether it is an undefined code. list may not be combined with summary.
**generate(newvar)** specifies that icd10 check create a new variable containing, for each observation, 0 if the observation contains a defined code. Otherwise, it contains a number from 1 to 8 if the code is invalid, 99 if the code is undefined, or missing if the code is missing. The positive numbers indicate the kind of problem and correspond to the listing produced by icd10 check.

**version(#)** specifies the version of the codes that icd10 check should reference. # may be any value between 2003, which is the second edition of ICD-10 without any updates applied, and 2016, which is the fifth edition of ICD-10. The appropriate value of # should be determined from the data source. The default is the current year.

**Options for icd10 clean**

**generate(newvar)** and **replace** specify how the formatted values of **varname** are to be handled. You must specify either **generate()** or **replace**.

**generate()** specifies that the cleaned values be placed in the new variable specified in **newvar**.

**replace** specifies that the existing values of **varname** be replaced with the formatted values.

**check** specifies that icd10 clean should first check that **varname** contains codes that fit the format of ICD-10 diagnosis codes. Specifying the **check** option will slow down icd10 clean.

**nodots** specifies that the period be removed in the final format.

**pad** specifies that spaces be added to the end of the codes to make the (implied) dots align vertically in listings. The default is to left-align codes without adding spaces.

**Options for icd10 generate**

**category, short, description, and range(codelist)** specify the contents of the new variable that icd10 generate is to create. You do not need to icd10 clean varname before using icd10 generate; it will accept any supported format or combination of formats.

**category** and **short** generate a new variable that also contains ICD-10 diagnosis codes. The resulting variable may be used with the other icd10 subcommands.

**category** specifies to extract the three-character category code from the ICD-10 diagnosis code.

**short** is designed for users who have data with greater specificity than the standard four-character ICD-10 codes. **short** will reduce five- and six-character codes to their first four characters. Three- and four-character codes are left as they are.

**description** creates **newvar** containing descriptions of the ICD-10 diagnosis codes.

**range(codelist)** creates a new indicator variable equal to 1 when the ICD-10 diagnosis code is in the range specified, equal to 0 when the ICD-10 diagnosis code is not in the range, and equal to missing when **varname** is missing.

**addcode(begin | end)** specifies that the code should be included with the text describing the code. Specifying **addcode(begin)** will prepend the code to the text. Specifying **addcode(end)** will append the code to the text.

**pad** specifies that the code that is to be added to the description should be padded spaces to the right of the code so that the start of description text is aligned for all codes. **pad** may be specified only with addcode(begin).

**nodots** specifies that the code that is added to the description should be formatted without a period. **nodots** may be specified only if addcode() is also specified.
check specifies that icd10 generate should first check that varname contains codes that fit the format of ICD-10 diagnosis codes. Specifying the check option will slow down the generate subcommand.

version(#) specifies the version of the codes that icd10 generate should reference. # may be any value between 2003, which is the second edition of ICD-10 without any updates applied, and 2016, which is the fifth edition of ICD-10. The appropriate value of # should be determined from the data source. The default is the current year.

Option for icd10 lookup

version(#) specifies the version of the codes that icd10 lookup should reference. # may be any value between 2003, which is the second edition of ICD-10 without any updates applied, and 2016, which is the fifth edition of ICD-10. The appropriate value of # should be determined from the data source. The default is the current year.

Options for icd10 search

or specifies that ICD-10 diagnosis codes be searched for descriptions that contain any word specified with icd10 search. The default is to list only descriptions that contain all the words specified. matchcase specifies that icd10 search should match the case of the keywords given on the command line. The default is to perform a case-insensitive search.

version(#) specifies the version of the codes that icd10 search should reference. # may be any value between 2003, which is the second edition of ICD-10 without any updates applied, and 2016, which is the fifth edition of ICD-10. By default, descriptions for all versions are searched, meaning that codes that changed descriptions and that have descriptions in multiple versions that contain the search terms will be duplicated. To ensure a list of unique code values, specify the version number.

Remarks and examples

Remarks are presented under the following headings:

Introduction
Managing datasets with ICD-10 codes
Creating new variables

If you have not yet read Introduction to ICD coding in [D] icd, please do so before using the icd10 commands.

Introduction

The general format of an ICD-10 diagnosis code is

\{A–Z\}{0–9}\{0–9\}[ .][0–9]

The code begins with a single letter followed by two digits. It may have an additional third digit after the period.
For example, in the ICD-10 coding system, E11.0 (Type 2 diabetes mellitus: With coma) and C56 (Malignant neoplasm of ovary) are diagnosis codes, although some datasets record (and some people write) E110 rather than E11.0. The icd10 commands understand both ways of recording codes. The commands are also insensitive to codes recorded with or without leading and trailing blanks and are case insensitive.

All the following are acceptable formats to record codes in Stata.

N94.0
M32
K12
F102
x40

The list of defined codes and their associated descriptions is provided under license from the World Health Organization (WHO); see [R] Copyright ICD-10. To view the current license and a log of changes that WHO has made to the list of ICD-10 codes since the icd10 commands were implemented in Stata, type

    . icd10 query

**ICD-10 Version and Change Log**

**License agreement**


**Edition 2016**

The ICD-10 data were obtained from WHO on 05feb2015. All updates scheduled for implementation through 01jan2016 have been applied. This was verified using the Cumulative Official Updates to ICD-10 which may be found at http://www.who.int/classifications/icd/icd10updates/en/index.html and then clicking the "Official WHO Updates combined 1996-2014 Volume 3" link.

Between 2015 and 2016:

13 codes added, 4 codes deleted, 0 code descriptions changed.

(output omitted)

**Technical note**

Codes can have up to two more digits to form five- and six-character codes. Supplemental subdivisions of ICD-10 codes may occur at the fifth and sixth characters. These supplemental subdivisions are primarily used to indicate anatomical site and additional information about the diagnosis, for example, whether a fracture was open or closed (World Health Organization 2011). However, these codes are not part of the standard four-character system codified by WHO for international morbidity and mortality reporting and are not considered valid by icd10.

If your data contain these longer codes, you can use icd10 generate with option short to shorten your codes to the relevant four-character subcategory code. Any existing three- and four-character codes in the data are left as they were originally.
Managing datasets with ICD-10 codes

The icd10 suite of commands has three data management commands. icd10 check verifies that the ICD-10 codes in `varname` are valid. icd10 clean standardizes the format of ICD-10 codes in `varname`. And icd10 generate produces a new variable from an existing variable containing ICD-10 codes. It will create a variable containing the associated category code, a description of the code, or a binary indicator for whether the code is in a specified set of codes.

Example 1: Checking the validity of a variable

Although not necessary, a good place to start is with icd10 check. The commands in the icd10 suite will return an error message if the codes in your data are not valid. Running icd10 check is a good way to avoid error messages later.

`australia10.dta` contains total deaths in 2010 for males and females from Australia, taken from WHO’s mortality data. Below we list the first 10 observations.

```
use https://www.stata-press.com/data/r17/australia10
(listing Australian mortality data, 2010)
list in 1/10, sepby(cause) noobs
```

<table>
<thead>
<tr>
<th>cause</th>
<th>sex</th>
<th>deaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>A020</td>
<td>Male</td>
<td>1</td>
</tr>
<tr>
<td>A020</td>
<td>Female</td>
<td>4</td>
</tr>
<tr>
<td>A021</td>
<td>Male</td>
<td>3</td>
</tr>
<tr>
<td>A021</td>
<td>Female</td>
<td>1</td>
</tr>
<tr>
<td>A047</td>
<td>Male</td>
<td>16</td>
</tr>
<tr>
<td>A047</td>
<td>Female</td>
<td>25</td>
</tr>
<tr>
<td>A048</td>
<td>Female</td>
<td>4</td>
</tr>
<tr>
<td>A049</td>
<td>Male</td>
<td>1</td>
</tr>
<tr>
<td>A049</td>
<td>Female</td>
<td>1</td>
</tr>
<tr>
<td>A063</td>
<td>Male</td>
<td>1</td>
</tr>
</tbody>
</table>

We will specify the generate() option to create a new variable called `prob` that will indicate that the code in `cause` is valid (`prob = 0`) or will indicate a value of 1 through 8 for the reason the code is not valid. icd10 check also creates a value of 99, which indicates that the code is not defined but otherwise conforms to the formatting requirements for ICD-10 codes.
. icd10 check cause, generate(prob)
(cause contains no missing values)

cause contains undefined codes:
1. Invalid placement of period 0
2. Too many periods 0
3. Code too short 0
4. Code too long 0
5. Invalid 1st char (not A-Z) 0
6. Invalid 2nd char (not 0-9) 0
7. Invalid 3rd char (not 0-9) 0
8. Invalid 4th char (not 0-9) 0
77. Valid only for previous versions 6
88. Valid only for later versions 0
99. Code not defined 0

Total 6

icd10 check reports that there are six observations with undefined codes. In this case, this is because we failed to specify that the data were reported using the ICD-10 codes from 2010.

. drop prob
. icd10 check cause, generate(prob) year(2010)
(cause contains defined codes; no missing values)

We see now that there are no errors in our dataset.

Example 2: Standardizing the format of codes

If we plan to do any reporting with these codes later, we may want to make them more readable, so we use icd10 clean. This command will automatically add a dot after the third character and change the display format of the diagnosis variable so that it is left aligned. We specify replace so that the standardized codes are placed in the existing cause variable.

When we listed our data before, they were sorted by cause of death and showed very few deaths assigned to the first several codes. It might be more interesting to see the most frequent causes of death. So before we list the data this time, we sort them in descending order with gsort.
. icd10 clean cause, replace
variable cause was str4 now str5
(2,921 real changes made)
. gsort -deaths
. list cause sex deaths in 1/10, sepby(cause)

<table>
<thead>
<tr>
<th>cause</th>
<th>sex</th>
<th>deaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>I21.9</td>
<td>Male</td>
<td>5,057</td>
</tr>
<tr>
<td>I21.9</td>
<td>Female</td>
<td>4,885</td>
</tr>
<tr>
<td>C34.9</td>
<td>Male</td>
<td>4,859</td>
</tr>
<tr>
<td>I25.9</td>
<td>Male</td>
<td>3,805</td>
</tr>
<tr>
<td>I25.9</td>
<td>Female</td>
<td>3,636</td>
</tr>
<tr>
<td>F03</td>
<td>Female</td>
<td>3,517</td>
</tr>
<tr>
<td>C61</td>
<td>Male</td>
<td>3,236</td>
</tr>
<tr>
<td>I64</td>
<td>Female</td>
<td>3,204</td>
</tr>
<tr>
<td>C34.9</td>
<td>Female</td>
<td>3,130</td>
</tr>
<tr>
<td>C50.9</td>
<td>Female</td>
<td>2,842</td>
</tr>
</tbody>
</table>

Now it is clear that we have a mix of three- and four-character codes.

Example 3: Looking up a single code

In example 2, we see that the highest number of reported deaths for men and women is for code I21.9. If we were curious about what this code is, we could type

. icd10 lookup I21.9
   I21.9 Acute myocardial infarction, unspecified

and we would see that these are deaths from acute myocardial infarction, commonly known as heart attacks. Because the icd10 commands are case insensitive and do not care whether we use the dot, we could have typed i21.9, I219, or i219, and Stata would have returned the same results.

Creating new variables

We now proceed to create new variables for later use.

Example 4: Creating an indicator variable

Suppose that after watching several high-action nature shows on television, we now believe that death due to shark attack is common in Australia. It did not show up in our top-ten list above, but we would like to see how many deaths we have in our data. We can look up the code using WHO’s interactive web utility (http://apps.who.int/classifications/icd10/browse/2010/en/) and then use icd10 generate with the range() option to create an indicator for whether death occurred by shark bite (shark).
Reality was not nearly as exciting as television—there was only one death with a code relating to shark bite in Australia in 2010.

If we wanted to study something less sensational, we could expand the `icd10rangelist` to a more complex list of codes. For example, perhaps we want to study the number of deaths from myocardial infarction (MI) and complications that occurred afterward. We might pick codes I21.0 through I21.9, I22.0 through I22.9, and I23.0 through I23.8. We could create the variable `mi` by typing

```
. icd10 generate mi=cause, range(I210/I219 I220/I229 I230/I238)
```

```
. tabulate mi [fweight=deaths]
```

<table>
<thead>
<tr>
<th>mi</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>133,522</td>
<td>93.06</td>
<td>93.06</td>
</tr>
<tr>
<td>1</td>
<td>9,951</td>
<td>6.94</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>143,473</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

We see that 9,951 deaths were from MI or complications thereof, which equates to about 6.9% of all deaths in Australia in 2010. It appears that hearts are far more dangerous than sharks.

Technical note

WHO reserves codes in categories U00 through U49 for the provisional assignment of new diseases and designates codes U50 through U99 for research purposes (World Health Organization 2011).

In general, codes in categories U50 through U99 are treated as undefined. This means that you do not need to take any special steps as long as your codes fit within the accepted four-character format. However, if you wish to exclude U codes from the commands, you can use the `if` qualifier.

With the exception of `icd10 generate` with the `description` option, the `icd10` commands will continue to work as normal with undefined U codes. As a rule, `icd10 generate` with the `description` option will return missing values for codes U50 through U99. Note that some of these codes, however, are defined and considered valid by `icd10` because WHO has distributed descriptions for them. For these codes, `icd10 generate` with option `description` will return results. The affected codes vary by year.
Stored results

`icd10 check` stores the following in \( r() \):

Scalars
\[
\begin{align*}
\text{\( r(e\#) \)} & \quad \text{number of errors of type \#} \\
\text{\( r(esum) \)} & \quad \text{total number of errors} \\
\text{\( r(miss) \)} & \quad \text{number of missing values} \\
\text{\( r(N) \)} & \quad \text{number of nonmissing values}
\end{align*}
\]

`icd10 clean` stores the following in \( r() \):

Scalars
\[
\begin{align*}
\text{\( r(N) \)} & \quad \text{number of changes}
\end{align*}
\]

`icd10 lookup` and `icd10 search` store the following in \( r() \):

Scalars
\[
\begin{align*}
\text{\( r(N\_codes) \)} & \quad \text{number of codes found}
\end{align*}
\]

Acknowledgments

We thank the World Health Organization for making ICD-10 codes available to Stata users. See [R] Copyright ICD-10 for allowed usage.

We thank Joe Canner of the Johns Hopkins University School of Medicine, who wrote `mycd10` and `mycd10p`, which provide many utilities for ICD-10 diagnosis and procedure codes. The commands rely on a user-supplied ICD-10 lookup dataset for diagnosis codes and ICD-10-PCS codes from the U.S. Centers for Medicare and Medicaid Services for procedure codes.

References


Also see

[D] icd — Introduction to ICD commands
[D] icd10cm — ICD-10-CM diagnosis codes
**Description**

icd10cm is a suite of commands for working with ICD-10-CM diagnosis codes from U.S. federal fiscal year 2016 to the present. To see the current version of the ICD-10-CM diagnosis codes and any changes that have been applied, type icd10cm query.

icd10cm check, icd10cm clean, and icd10cm generate are data management commands. icd10cm check verifies that a variable contains defined ICD-10-CM diagnosis codes and provides a summary of any problems encountered. icd10cm clean standardizes the format of the codes. icd10cm generate can create a binary indicator variable for whether the code is in a specified set of codes, a variable containing a corresponding higher-level code, or a variable containing the description of the code.

icd10cm lookup and icd10cm search are interactive utilities. icd10cm lookup displays descriptions of the codes specified on the command line. icd10cm search looks for relevant ICD-10-CM diagnosis codes from keywords given on the command line.

**Quick start**

Determine whether ICD-10-CM diagnosis codes in diag1 are invalid, and store reasons in invalid

```
icd10cm check diag1, generate(invalid)
```

Standardize display of codes in diag2 to add a period and left-align codes

```
icd10cm clean diag2, replace
```

Generate descr3 as the diagnosis code prepended to the short description of diagnosis code in diag3

```
icd10cm generate descr3 = diag3, description addcode(begin)
```

Generate mhypertn as an indicator for a maternal hypertension diagnosis in diag4 using ICD-10-CM codes O16.1 through O16.5 or O16.9

```
icd10cm generate mhypertn = diag4, range(O161/O165 O169)
```

Look up descriptions for ICD-10-CM diagnosis codes T46.1X1, T46.1X1A, T46.1X1D, and T46.1X1S

```
icd10cm lookup T46.1X1*
```

Look up codes where the description contains the words “delivery” or “birth”

```
icd10cm search delivery birth, or
```

**Menu**

Data > ICD codes > ICD-10-CM
### Syntax

**Verify that variable contains defined codes**

```plaintext
icd10cm check varname [if] [in] [, checkopts]
```

**Clean variable and verify format of codes**

```plaintext
icd10cm clean varname [if] [in], {generate(newvar) | replace} [cleanopts]
```

**Generate new variable from existing variable**

```plaintext
icd10cm generate newvar = varname [if] [in], category [check]
icd10cm generate newvar = varname [if] [in], description [genopts]
icd10cm generate newvar = varname [if] [in], range(codelist) [check]
```

**Display code descriptions**

```plaintext
icd10cm lookup codelist [, version(#)]
```

**Search for codes from descriptions**

```plaintext
icd10cm search ["]text["] [["]text["] ...] [, searchopts]
```

**Display ICD-10-CM version**

```plaintext
icd10cm query
```

codelist is one of the following:

- `icd10code` (the particular code)
- `icd10code*` (all codes starting with)
- `icd10code/icd10code` (the code range)

or any combination of the above, such as `A27.0 G40* Y60/Y69.9`.

### checkopts

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fmtonly</code></td>
</tr>
<tr>
<td><code>summary</code></td>
</tr>
<tr>
<td><code>list</code></td>
</tr>
<tr>
<td><code>generate(newvar)</code></td>
</tr>
<tr>
<td><code>version(#)</code></td>
</tr>
</tbody>
</table>
cleanopts

<table>
<thead>
<tr>
<th>Description</th>
<th>cleanopts</th>
</tr>
</thead>
<tbody>
<tr>
<td>create new variable containing cleaned codes</td>
<td><em>generate(newvar)</em></td>
</tr>
<tr>
<td>replace existing codes with the cleaned codes</td>
<td><em>replace</em></td>
</tr>
<tr>
<td>check that variable contains ICD-10-CM codes before cleaning</td>
<td>check</td>
</tr>
<tr>
<td>format codes without a period</td>
<td>nodots</td>
</tr>
<tr>
<td>add space to the right of codes shorter than seven characters</td>
<td>pad</td>
</tr>
</tbody>
</table>

*Either generate() or replace is required.*

genopts

<table>
<thead>
<tr>
<th>Description</th>
<th>genopts</th>
</tr>
</thead>
<tbody>
<tr>
<td>add code to the beginning or end of the description</td>
<td>addcode(begin</td>
</tr>
<tr>
<td>add spaces to the right of the code; must specify addcode(begin)</td>
<td>pad</td>
</tr>
<tr>
<td>format codes without a period; must specify addcode()</td>
<td>nodots</td>
</tr>
<tr>
<td>check that variable contains ICD-10-CM codes before generating new variable</td>
<td>check</td>
</tr>
<tr>
<td>use long description rather than short</td>
<td>long</td>
</tr>
<tr>
<td>select description from fiscal year #; default is the current year</td>
<td>version(#)</td>
</tr>
</tbody>
</table>

searchopts

<table>
<thead>
<tr>
<th>Description</th>
<th>searchopts</th>
</tr>
</thead>
<tbody>
<tr>
<td>match any keyword</td>
<td>or</td>
</tr>
<tr>
<td>match case of keywords</td>
<td>matchcase</td>
</tr>
<tr>
<td>search description from fiscal year #; default is all</td>
<td>version(#)</td>
</tr>
</tbody>
</table>

collect is allowed with icd10cm check and icd10cm clean; see [U] 11.1.10 Prefix commands.

Options

Options are presented under the following headings:

Options for icd10cm check
Options for icd10cm clean
Options for icd10cm generate
Option for icd10cm lookup
Options for icd10cm search

Options for icd10cm check

fmonly tells icd10cm check to verify that the codes fit the format of ICD-10-CM diagnosis codes but not to check whether the codes are defined.

summary specifies that icd10cm check should report the frequency of each invalid or undefined code that was found in the data. Codes are displayed in descending order by frequency. summary may not be combined with list.

list specifies that icd10cm check list the observation number, the invalid or undefined ICD-10-CM diagnosis code, and the reason the code is invalid or whether it is an undefined code. list may not be combined with summary.
generate(newvar) specifies that icd10cm check create a new variable containing, for each observation, 0 if the observation contains a defined code. Otherwise, it contains a number from 1 to 11 if the code is invalid, 77 if the code is valid only for a previous version, 88 if the code is valid only for a later version, 99 if the code is undefined, or missing if varname is missing. The positive numbers indicate the kind of problem and correspond to the listing produced by icd10cm check.

version(#) specifies the version of the codes that icd10cm check should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. icd10cm supports all years after the United States officially adopted ICD-10-CM. The appropriate value of # should be determined from the data source. The default is the current year.

Warning: The default value of version() will change over time so that the most recent codes are used. Using the default value rather than specifying a specific version may change results after a new version of the codes is introduced.

Options for icd10cm clean

generate(newvar) and replace specify how the formatted values of varname are to be handled. You must specify either generate() or replace.

generate() specifies that the cleaned values be placed in the new variable specified in newvar.

replace specifies that the existing values of varname be replaced with the formatted values.

check specifies that icd10cm clean should first check that varname contains codes that fit the format of ICD-10-CM diagnosis codes. Specifying the check option will slow down icd10cm clean.

nodots specifies that the period be removed in the final format.

pad specifies that spaces be added to the end of the codes to make the (implied) dots align vertically in listings. The default is to left-align codes without adding spaces.

Options for icd10cm generate

category, description, and range(codelist) specify the contents of the new variable that icd10cm generate is to create. You do not need to icd10cm clean varname before using icd10cm generate; it will accept any supported format or combination of formats.

category specifies to extract the three-character category code from the ICD-10-CM diagnosis code. The resulting variable may be used with the other icd10cm subcommands.

description creates newvar containing descriptions of the ICD-10-CM diagnosis codes.

range(codelist) creates a new indicator variable equal to 1 when the ICD-10-CM diagnosis code is in the range specified, equal to 0 when the ICD-10-CM diagnosis code is not in the range, and equal to missing when varname is missing.

addcode(begin|end) specifies that the code should be included with the text describing the code. Specifying addcode(begin) will prepend the code to the text. Specifying addcode(end) will append the code to the text.

pad specifies that the code that is to be added to the description should be padded spaces to the right of the code so that the start of description text is aligned for all codes. pad may be specified only with addcode(begin).
nodots specifies that the code that is added to the description should be formatted without a period. nodots may be specified only if addcode() is also specified.

check specifies that icd10cm generate should first check that varname contains codes that fit the format of ICD-10-CM diagnosis codes. Specifying the check option will slow down the generate subcommand.

long specifies that the long description of the code be used rather than the short (abbreviated) description.

version(#) specifies the version of the codes that icd10cm generate should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. icd10cm supports all years after the United States officially adopted ICD-10-CM. The appropriate value of # should be determined from the data source. The default is the current year.

Warning: The default value of version() will change over time so that the most recent codes are used. Using the default value rather than specifying a specific version may change results after a new version of the codes is introduced.

Option for icd10cm lookup

version(#) specifies the version of the codes that icd10cm lookup should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. icd10cm supports all years after the United States officially adopted ICD-10-CM. The appropriate value of # should be determined from the data source. The default is the current year.

Warning: The default value of version() will change over time so that the most recent codes are used. Using the default value rather than specifying a specific version may change results after a new version of the codes is introduced.

Options for icd10cm search

or specifies that ICD-10-CM diagnosis codes be searched for descriptions that contain any word specified with icd10cm search. The default is to list only descriptions that contain all the words specified.

matchcase specifies that icd10cm search should match the case of the keywords given on the command line. The default is to perform a case-insensitive search.

version(#) specifies the version of the codes that icd10cm search should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. icd10cm supports all years after the United States officially adopted ICD-10-CM.

By default, descriptions for all versions are searched, meaning that codes that changed descriptions and that have descriptions in multiple versions that contain the search terms will be duplicated. To ensure a list of unique code values, specify the version number.
Remarks and examples

Remarks are presented under the following headings:

- Introduction
- Managing datasets with ICD-10-CM codes
- Interactive utilities

If you have not yet read Introduction to ICD coding in [D] icd, please do so before using the icd10cm commands.

Introduction

The general format of an ICD-10-CM diagnosis code is a three-character category code followed by up to four characters after an (implied) period. The first character is always a letter and the second character is always a number, but the remaining characters may be any combination of letters and numbers.

Some examples of ICD-10-CM diagnosis codes are B69 (cysticercosis) and W20.0XXA (struck by falling object in cave-in, initial encounter). Many datasets record (and some people write) codes without the period; for example, the code I74.3 may appear as I743. The icd10cm commands understand both ways of recording codes. The commands are also insensitive to codes recorded with or without leading and trailing blanks and are case insensitive.

All the following are acceptable formats to record codes in Stata:

```
T37.0X3A
A25.1
C52
a80.0
z8261
```

Important note: What constitutes a valid code changes between versions. For the rest of this entry, a defined code is any code that is currently valid, was valid at some point since the ICD-10-CM coding system was introduced, or has a meaning as a grouping of codes. The list of valid codes and their associated descriptions is from the U.S. Centers for Disease Control and Prevention’s National Center for Health Statistics (Centers for Disease Control and Prevention 2013). The ICD-10-CM is a licensed adaptation of the ICD-10, which is copyrighted by the World Health Organization (WHO); see [R] Copyright ICD-10.

To view the current version of the ICD-10-CM diagnosis codes in Stata, its source, and a log of changes that have been made to the list of ICD-10-CM diagnosis codes since the icd10cm commands were implemented, type

```
    . icd10cm query
```

**ICD-10-CM Diagnosis Code Version and Change Log**

**Note**
The ICD-10 coding system is copyrighted by the World Health Organization. The ICD-10-CM is the WHO’s authorized adaptation for use in the United States. It is maintained by the National Center for Health Statistics (NCHS), at the Center for Disease Control and Prevention. Stata obtains the ICD-10-CM data from the NCHS website. See copyright icd10 for the ICD-10 copyright notification.

(output omitted)
Managing datasets with ICD-10-CM codes

The `icd10cm` suite of commands has three data management commands. `icd10cm check` verifies that the ICD-10-CM diagnosis codes in `varname` are valid. `icd10cm clean` standardizes the format of ICD-10-CM diagnosis codes in `varname`. And `icd10cm generate` produces a new variable from an existing variable containing ICD-10-CM diagnosis codes.

Examples in this section use `hosp2015.dta`, a fictional sample of inpatient hospital discharges in Washington State from July 2015 to December 2015. The data were simulated based on the Comprehensive Hospital Abstract Reporting System (CHARS); see https://www.doh.wa.gov/DataandStatisticalReports/HealthcareinWashington/HospitalandPatientData/HospitalDischargeDataCHARS. Examples analyzing the procedure codes for this dataset may be found in [D] `icd10pcs`.

```stata
(Fictional WA hospital discharges)
. describe
Contains data from https://www.stata-press.com/data/r17/hosp2015.dta
Observations: 3,935 Fictional WA hospital discharges
Variables: 18 6 Apr 2020 13:10

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>hospid</td>
<td>str5</td>
<td>%9s</td>
<td>Hospital ID</td>
</tr>
<tr>
<td>age</td>
<td>byte</td>
<td>%11.0g</td>
<td>Age (years)</td>
</tr>
<tr>
<td>sex</td>
<td>byte</td>
<td>%8.0g</td>
<td>Sex</td>
</tr>
<tr>
<td>ins</td>
<td>byte</td>
<td>%9.0g</td>
<td>Insurance type</td>
</tr>
<tr>
<td>los</td>
<td>byte</td>
<td>%19.0g</td>
<td>Length of stay (days)</td>
</tr>
<tr>
<td>atype</td>
<td>byte</td>
<td>%9.0g</td>
<td>Admtype</td>
</tr>
<tr>
<td>asource</td>
<td>byte</td>
<td>%18.0g</td>
<td>Admission source</td>
</tr>
<tr>
<td>aday</td>
<td>byte</td>
<td>%8.0g</td>
<td>Day admission</td>
</tr>
<tr>
<td>dmonth</td>
<td>int</td>
<td>%tm</td>
<td>Discharge month</td>
</tr>
<tr>
<td>dstatus</td>
<td>byte</td>
<td>%22.0g</td>
<td>Discharge status</td>
</tr>
<tr>
<td>died</td>
<td>byte</td>
<td>%8.0g</td>
<td>Patient died (1=yes)</td>
</tr>
<tr>
<td>diag1</td>
<td>str7</td>
<td>%9s</td>
<td>Diagnosis 1</td>
</tr>
<tr>
<td>diag2</td>
<td>str7</td>
<td>%9s</td>
<td>Diagnosis 2</td>
</tr>
<tr>
<td>diag3</td>
<td>str7</td>
<td>%9s</td>
<td>Diagnosis 3</td>
</tr>
<tr>
<td>proc1</td>
<td>str7</td>
<td>%9s</td>
<td>Procedure 1</td>
</tr>
<tr>
<td>proc2</td>
<td>str7</td>
<td>%9s</td>
<td>Procedure 2</td>
</tr>
<tr>
<td>proc3</td>
<td>str7</td>
<td>%9s</td>
<td>Procedure 3</td>
</tr>
<tr>
<td>billed</td>
<td>float</td>
<td>%8.2fc</td>
<td>Amount billed ($1,000s)</td>
</tr>
</tbody>
</table>
```

Although not necessary, it is a good idea to begin with `icd10cm check` and fix any potential problems before proceeding to other `icd10cm` commands. By default, it verifies that `varname` contains defined ICD-10-CM diagnosis codes and, if not, tabulates the type of problems encountered.

> Example 1: Checking the validity of a variable

We want to verify that the primary diagnosis code (`diag1`) contains only valid ICD-10-CM diagnosis codes. Because any discharges that use ICD-10-CM diagnosis codes in our data will be from October 1, 2015 to December 31, 2015, we use `version(2016)` to specify the FFY-2016 version of ICD-10-CM. If there are invalid or undefined codes in our data, we want to see what the codes are, their frequency, and the reason they were not valid, so we add the `summary` option.
. icd10cm check diag1, version(2016) summary
(diag1 contains no missing values)
diag1 contains invalid codes:

1. Invalid placement of period 0
2. Too many periods 0
3. Code too short 0
4. Code too long 0
5. Invalid 1st char (not A-Z) 1,916
6. Invalid 2nd char (not 0-9) 0
7. Invalid 3rd char (not 0-9 A or B) 0
8. Invalid 4th char (not 0-9 or A-Z) 0
9. Invalid 5th char (not 0-9 or A-Z) 0
10. Invalid 6th char (not 0-9 or A-Z) 0
11. Invalid 7th char (not 0-9 or A-Z) 0
77. Valid only for previous versions 0
88. Valid only for later versions 0
99. Code not defined 32

Total 1,948

Summary of invalid and undefined codes

<table>
<thead>
<tr>
<th>diag1</th>
<th>Count</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0389</td>
<td>91</td>
<td>Invalid 1st char</td>
</tr>
<tr>
<td>65421</td>
<td>57</td>
<td>Invalid 1st char</td>
</tr>
<tr>
<td>64511</td>
<td>45</td>
<td>Invalid 1st char</td>
</tr>
<tr>
<td>71536</td>
<td>33</td>
<td>Invalid 1st char</td>
</tr>
<tr>
<td>66411</td>
<td>31</td>
<td>Invalid 1st char</td>
</tr>
</tbody>
</table>

(output omitted)

| 4940   | 1     | Invalid 1st char               |
| 4270   | 1     | Invalid 1st char               |
| 1570   | 1     | Invalid 1st char               |
| 53550  | 1     | Invalid 1st char               |
| 64413  | 1     | Invalid 1st char               |

It looks like the records with problems used ICD-9-CM codes instead of ICD-10-CM codes. We could confirm our suspicion by using icd9 check or icd9 lookup to see whether the codes are defined in the ICD-9-CM coding system.

Because our data span the date the U.S. switched to ICD-10-CM (October 1, 2015), we create an indicator for whether the record should use ICD-10-CM based on the date of discharge (dmonth). We then run icd10cm check again for only these records.

. generate use10 = (dmonth>=tm(2015m10))
. icd10cm check diag1 if use10==1, version(2016)
(diag1 contains defined codes; no missing values)

All the problems in diag1 are before the switch, so we proceed without concern about our data.

In the generate command above, we used the tm() function, which lets us easily provide date values to Stata in string form; see [D] Datetime for more information about working with dates.

If we wanted to check codes in more than one diagnosis variable, we could use a foreach loop or reshape our data; see Working with multiple codes in [D] icd. Also, additional options for icd10cm check help you identify the source of any errors. For example, you can obtain a list of observations that have invalid codes. See Options for icd10cm check.
icd10cm clean formats the variable to ensure consistency and to make subsequent output from other commands such as list and tabulate look better. icd10cm clean also can be used to verify that the codes in a variable conform to the ICD-10-CM format, without checking to see whether the codes are defined.

Example 2: Creating a variable with standardized codes

We would like to find the frequency of each primary diagnosis in our dataset. We can use tabulate with the sort option to see the most common primary diagnoses first.

So that the codes in diag1 are more readable in the tabulate output, we first use icd10cm clean. This adds a period after the three-character category code. We specify the pad option to make sure our codes align and store the result in the new variable pdx.

```
icd10cm clean diag1 if use10==1, pad generate(pdx)
(1,955 missing values generated)
.tabulate pdx, sort
```

<table>
<thead>
<tr>
<th>pdx</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A41.9</td>
<td>105</td>
<td>5.30</td>
<td>5.30</td>
</tr>
<tr>
<td>048.0</td>
<td>40</td>
<td>2.02</td>
<td>7.32</td>
</tr>
<tr>
<td>121.4</td>
<td>37</td>
<td>1.87</td>
<td>9.19</td>
</tr>
<tr>
<td>070.1</td>
<td>36</td>
<td>1.82</td>
<td>11.01</td>
</tr>
<tr>
<td>M17.11</td>
<td>33</td>
<td>1.67</td>
<td>12.68</td>
</tr>
<tr>
<td>034.21</td>
<td>28</td>
<td>1.41</td>
<td>14.09</td>
</tr>
<tr>
<td>J96.01</td>
<td>21</td>
<td>1.06</td>
<td>15.15</td>
</tr>
<tr>
<td>M16.11</td>
<td>21</td>
<td>1.06</td>
<td>16.21</td>
</tr>
<tr>
<td>J18.9</td>
<td>20</td>
<td>1.01</td>
<td>17.22</td>
</tr>
<tr>
<td>070.0</td>
<td>20</td>
<td>1.01</td>
<td>18.23</td>
</tr>
</tbody>
</table>

(output omitted)

Notice that we used if with the use10 variable we created in example 1 to restrict icd10cm clean to just those diagnosis codes where the ICD-10-CM coding system should have been applied.

Aside from validating values of codes, the icd10cm command is primarily used to create inputs for other Stata commands. For example, in example 5 of [D] icd9, we show how to graph the frequency of category codes with descriptions, and in example 3 of [D] icd10pcs, we calculate average billed amounts over different procedures.

Example 3: Creating a variable indicating diagnosis

In example 2, we found that the most common primary diagnosis code in our data is A41.9, a code for a type of sepsis (a complication of infection).

Suppose we are interested in differences in length of stay (los) for discharges with and without a primary diagnosis of sepsis. We can use icd10cm generate with the range() option to search records for other diagnosis codes starting with A40, A41, and A42, which also indicate a sepsis diagnosis.

```
icd10cm generate sepsis=diag1 if use10==1, range(A40* A41* A42*)
```

An informal way to examine differences is to plot the average length of stay for discharges with and without a sepsis diagnosis. We first label the values of our sepsis variable so that it displays nicely in the graph.
. label define sepsis 0 "No sepsis" 1 "Sepsis"
. label values sepsis sepsis
. graph hbar los, over(sepsis) ytitle("Average length of stay (days)"

More formally, we could include the new `sepsis` indicator as a factor variable in a regression model.

Interactive utilities

`icd10cm lookup` and `icd10cm search` are interactive tools. You can use them without having any ICD-10-CM diagnosis data in memory.

`icd10cm lookup` lists the descriptions of codes given on the command line, and `icd10cm search` looks for relevant ICD-10-CM diagnosis codes from the specified keywords. The two commands complement each other.
Example 4: Finding diagnosis codes from descriptions

In example 3, we specified codes for sepsis as any code starting with A40, A41, or A42. Suppose we want to look for other relevant codes. We can search the descriptions of the ICD-10-CM codes to locate codes of interest.

```
.icd10cm search sepsis, version(2016)
   A02.1   Salmonella sepsis
   A22.7   Anthrax sepsis
   A26.7   Erysipelothrix sepsis
   A32.7   Listerial sepsis
(output omitted)
```

Note that icd10cm search is case insensitive. If you want icd10cm search to respect the case of the search terms you type, specify the matchcase option.

Using icd10cm lookup is similar to icd10pcs lookup. See example 4 in [D] icd10pcs.

Stored results

icd10cm check stores the following in r():

Scalars
   r(e#)  number of errors of type #
   r(esum) total number of errors
   r(miss) number of missing values
   r(N)   number of nonmissing values

icd10cm clean stores the following in r():

Scalars
   r(N)   number of changes

icd10cm lookup and icd10cm search store the following in r():

Scalars
   r(N_codes) number of codes found

Acknowledgments

We thank the Washington State Department of Health’s Center for Health Statistics for providing us with access to its 2015 Comprehensive Hospital Abstract Reporting System (CHARS) inpatient dataset. The hosp2015 dataset used here was partially simulated based on information from the 2015 limited use CHARS. We also thank Jeanne M. Sears of the University of Washington for bringing the CHARS to our attention.

We thank Joe Canner of the Johns Hopkins University School of Medicine, who wrote mycd10 and mycd10p, which provide many utilities for ICD-10 diagnosis and procedure codes. The commands rely on a user-supplied ICD-10 lookup dataset for diagnosis codes and ICD-10-PCS codes from the U.S. Centers for Medicare and Medicaid Services for procedure codes.
Reference


Also see

[D] icd — Introduction to ICD commands
[D] icd9 — ICD-9-CM diagnosis codes
[D] icd10 — ICD-10 diagnosis codes
[D] icd10pcs — ICD-10-PCS procedure codes
icd10pcs is a suite of commands for working with ICD-10-PCS procedure codes from U.S. federal fiscal year 2016 to the present. To see the current version of the ICD-10-PCS procedure codes and any changes that have been applied, type icd10pcs query.

icd10pcs check, icd10pcs clean, and icd10pcs generate are data management commands. icd10pcs check verifies that a variable contains defined ICD-10-PCS procedure codes and provides a summary of any problems encountered. icd10pcs clean standardizes the format of the codes. icd10pcs generate can create a binary indicator variable for whether the code is in a specified set of codes, a variable containing a corresponding higher-level code, or a variable containing the description of the code.

icd10pcs lookup and icd10pcs search are interactive utilities. icd10pcs lookup displays descriptions of the codes specified on the command line. icd10pcs search looks for relevant ICD-10-PCS procedure codes from keywords given on the command line.

Quick start

Determine whether ICD-10-PCS procedure codes in proc1 are invalid, and store reasons in invalid

```
icd10pcs check proc1, generate(invalid)
```

Standardize display of codes in proc2 to add a period and left-align codes

```
icd10pcs clean proc2, replace
```

Check that the codes in proc3 conform to ICD-10-PCS formatting rules, and if so, create main as the corresponding three-character category code

```
icd10pcs generate main = proc3, category check
```

Generate descr4 as the current short description of procedure code in proc4

```
icd10pcs generate descr4 = proc4, description
```

Look up current descriptions for procedure codes 081.23J4 through 081.Y3Z3

```
icd10pcs lookup 081.23J4/081.Y3Z3
```

Look up codes where the description from FFY-2016 contains the word “foot”

```
icd10pcs search foot, version(2016)
```

Menu

Data > ICD codes > ICD-10-PCS
Syntax

Verify that variable contains defined codes

```
icd10pcs check varname [ if ] [ in ] [, checkopts ]
```

Clean variable and verify format of codes

```
icd10pcs clean varname [ if ] [ in ] , { generate(newvar) | replace } [ cleanopts ]
```

Generate new variable from existing variable

```
icd10pcs generate newvar = varname [ if ] [ in ] , category [ check ]
icd10pcs generate newvar = varname [ if ] [ in ] , description [ genopts ]
icd10pcs generate newvar = varname [ if ] [ in ] , range(codelist) [ check ]
```

Display code descriptions

```
icd10pcs lookup codelist [, version(#) ]
```

Search for codes from descriptions

```
icd10pcs search [ " ] text [ " ] [ [ " ] text [ " ] ... ] [, searchopts ]
```

Display ICD-10-PCS version

```
icd10pcs query
```

codelist is one of the following:

- `icd10code` (the particular code)
- `icd10code*` (all codes starting with)
- `icd10code/icd10code` (the code range)

or any combination of the above, such as `041.E09P 2W3* BQ2L/BQ2LZZZ`.

<table>
<thead>
<tr>
<th>checkopts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmtonly</td>
<td>check only format of the codes</td>
</tr>
<tr>
<td>summary</td>
<td>frequency of each invalid or undefined code</td>
</tr>
<tr>
<td>list</td>
<td>list observations with invalid or undefined ICD-10-PCS codes</td>
</tr>
<tr>
<td>generate(newvar)</td>
<td>create new variable marking invalid codes</td>
</tr>
<tr>
<td>version(#)</td>
<td>fiscal year to check codes against; default is the current year</td>
</tr>
</tbody>
</table>
cleanopts

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>gen</strong></td>
<td>create new variable containing cleaned codes</td>
</tr>
<tr>
<td><strong>replace</strong></td>
<td>replace existing codes with the cleaned codes</td>
</tr>
<tr>
<td>check</td>
<td>check that variable contains ICD-10-PCS codes before cleaning</td>
</tr>
<tr>
<td>nodots</td>
<td>format codes without a period</td>
</tr>
</tbody>
</table>

Either `gen` or `replace` is required.

genopts

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addcode(begin</td>
<td>end)</td>
</tr>
<tr>
<td>nodots</td>
<td>format codes without a period; must specify addcode()</td>
</tr>
<tr>
<td>check</td>
<td>check that variable contains ICD-10-PCS codes before generating new variable</td>
</tr>
<tr>
<td>long</td>
<td>use long description rather than short</td>
</tr>
<tr>
<td>version(#)</td>
<td>select description from fiscal year #; default is the current year</td>
</tr>
</tbody>
</table>

searchopts

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>match any keyword</td>
</tr>
<tr>
<td>matchcase</td>
<td>match case of keywords</td>
</tr>
<tr>
<td>version(#)</td>
<td>search description from fiscal year #; default is all</td>
</tr>
</tbody>
</table>

collect is allowed with `icd10pcs check` and `icd10pcs clean`; see [U] 11.1.10 Prefix commands.

Options

Options are presented under the following headings:

- Options for `icd10pcs check`
- Options for `icd10pcs clean`
- Options for `icd10pcs generate`
- Option for `icd10pcs lookup`
- Options for `icd10pcs search`

Options for `icd10pcs check`

- **fmtonly** tells `icd10pcs check` to verify that the codes fit the format of ICD-10-PCS procedure codes but not to check whether the codes are defined.
- **summary** specifies that `icd10pcs check` should report the frequency of each invalid or undefined code that was found in the data. Codes are displayed in descending order by frequency. `summary` may not be combined with `list`.
- **list** specifies that `icd10pcs check` list the observation number, the invalid or undefined ICD-10-PCS procedure code, and the reason the code is invalid or whether it is an undefined code. `list` may not be combined with `summary`. 
**generate(newvar)** specifies that icd10pcs check create a new variable containing, for each observation, 0 if the observation contains a defined code. Otherwise, it contains a number from 1 to 11 if the code is invalid, 77 if the code is valid only for a previous version, 88 if the code is valid only for a later version, 99 if the code is undefined, or missing if the code is missing. The positive numbers indicate the kind of problem and correspond to the listing produced by icd10pcs check.

**version(#)** specifies the version of the codes that icd10pcs check should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. icd10pcs supports all years after the United States officially adopted ICD-10-PCS. The appropriate value of # should be determined from the data source. The default is the current year.

Warning: The default value of version() will change over time so that the most recent codes are used. Using the default value rather than specifying a specific version may change results after a new version of the codes is introduced.

### Options for icd10pcs clean

- **generate(newvar)** and **replace** specify how the formatted values of varname are to be handled. You must specify either generate() or replace.
  - **generate()** specifies that the cleaned values be placed in the new variable specified in newvar.
  - **replace** specifies that the existing values of varname be replaced with the formatted values.

- **check** specifies that icd10pcs clean should first check that varname contains codes that fit the format of ICD-10-PCS procedure codes. Specifying the check option will slow down icd10pcs clean.

- **nodots** specifies that the period be removed in the final format.

### Options for icd10pcs generate

- **category**, **description**, and **range(codelist)** specify the contents of the new variable that icd10pcs generate is to create. You do not need to icd10pcs clean varname before using icd10pcs generate; it will accept any supported format or combination of formats.

- **category** specifies to extract the three-character category code from the ICD-10-PCS procedure code. The resulting variable may be used with the other icd10pcs subcommands.

- **description** creates newvar containing descriptions of the ICD-10-PCS procedure codes.

- **range(codelist)** creates a new indicator variable equal to 1 when the ICD-10-PCS procedure code is in the range specified, equal to 0 when the ICD-10-PCS procedure code is not in the range, and equal to missing when varname is missing.

- **addcode(begin|end)** specifies that the code should be included with the text describing the code. Specifying addcode(begin) will prepend the code to the text. Specifying addcode(end) will append the code to the text.

- **nodots** specifies that the code that is added to the description should be formatted without a period. nodots may be specified only if addcode() is also specified.

- **check** specifies that icd10pcs generate should first check that varname contains codes that fit the format of ICD-10-PCS procedure codes. Specifying the check option will slow down the generate subcommand.
long specifies that the long description of the code be used rather than the short (abbreviated) description.

`version(#)` specifies the version of the codes that `icd10pcs generate` should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. `icd10pcs` supports all years after the United States officially adopted ICD-10-PCS. The appropriate value of # should be determined from the data source. The default is the current year.

Warning: The default value of `version()` will change over time so that the most recent codes are used. Using the default value rather than specifying a specific version may change results after a new version of the codes is introduced.

**Option for icd10pcs lookup**

`version(#)` specifies the version of the codes that `icd10pcs lookup` should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. `icd10pcs` supports all years after the United States officially adopted ICD-10-PCS. The appropriate value of # should be determined from the data source. The default is the current year.

Warning: The default value of `version()` will change over time so that the most recent codes are used. Using the default value rather than specifying a specific version may change results after a new version of the codes is introduced.

**Options for icd10pcs search**

`or` specifies that ICD-10-PCS procedure codes be searched for descriptions that contain any word specified with `icd10pcs search`. The default is to list only descriptions that contain all the words specified.

`matchcase` specifies that `icd10pcs search` should match the case of the keywords given on the command line. The default is to perform a case-insensitive search.

`version(#)` specifies the version of the codes that `icd10pcs search` should reference. # indicates the federal fiscal year for the codes. For example, use 2016 for federal fiscal year 2016 (FFY-2016), which is October 1, 2015 to September 30, 2016. `icd10pcs` supports all years after the United States officially adopted ICD-10-PCS.

By default, descriptions for all versions are searched, meaning that codes that changed descriptions and that have descriptions in multiple versions that contain the search terms will be duplicated. To ensure a list of unique code values, specify the version number.

**Remarks and examples**

Remarks are presented under the following headings:

- Introduction
- Managing datasets with ICD-10-PCS codes
- Interactive utilities

If you have not yet read *Introduction to ICD coding* in [D] icd, please do so before using the `icd10pcs` commands.
Introduction

The general format of an ICD-10-PCS procedure code is a three-character category code followed by four alpha-numeric characters after an (implied) period. The full codes are always seven characters long and may be any combination of letters and numbers.

Some examples of ICD-10-PCS procedure codes are 081 (Eye, Bypass) and 0GT.D0ZZ (Resection of Aortic Body, Open Approach). Many datasets record (and some people write) codes without the period; for example, the code 090.KXZZ may appear as 090KXZZ. The icd10pcs commands understand both ways of recording codes. The commands are also insensitive to codes recorded with or without leading and trailing blanks and are case insensitive.

All the following are acceptable formats to record codes in Stata:

- 03R
- 0jj
- 00f53zz
- 0TL.C0ZZ
- 091

Important note: What constitutes a valid code changes between versions. For the rest of this entry, a defined code is any code that is currently valid, was valid at some point since the ICD-10-CM/PCS coding system was introduced, or has a meaning as a grouping of codes. The list of valid codes and their associated descriptions is from the U.S. Centers for Medicare and Medicaid Services (CMS).

To view the current version of the ICD-10-PCS procedure codes in Stata, its source, and a log of changes that have been made to the list of ICD-10-PCS procedure codes since the icd10pcs commands were implemented, type .

```
  . icd10pcs query
  ICD-10-PCS Procedure Code Version and Change Log
  Note
    Stata obtains the ICD-10-PCS dataset from the Centers for Medicare and Medicaid Services website.
  (output omitted)
```

Managing datasets with ICD-10-PCS codes

The icd10pcs suite of commands has three data management commands. icd10pcs check verifies that the ICD-10-PCS procedure codes in varname are valid. icd10pcs clean standardizes the format of ICD-10-PCS procedure codes in varname. And icd10pcs generate produces a new variable from an existing variable containing ICD-10-PCS procedure codes.

Examples in this section use hosp2015.dta, a fictional sample of inpatient hospital discharges in Washington state from July 2015 to December 2015. The data were simulated based on the Comprehensive Hospital Abstract Reporting System (CHARS); see https://www.doh.wa.gov/DataandStatisticalReports/HealthcareinWashington/HospitalandPatientData/HospitalDischargeDataCHARS. Examples analyzing the diagnosis codes for this dataset can be found in [D] icd10cm.

```
  (Fictional WA hospital discharges)
```

icd10pcs check is the primary subcommand for validating ICD-10-PCS procedure codes. However, if you just want to verify that the codes conform to the formatting rules for ICD-10-PCS procedure, you can use the check option with icd10pcs clean or icd10pcs generate.
Example 1: Checking for valid code values

You use `icd10pcs check` just like you do `icd10cm check`. Because the data are from federal fiscal year 2016, we specify `version(2016)`.

In example 1 of [D] `icd10cm`, we found that we needed to account for the date of the admission when we used the `icd10cm` commands. The same is true of the `icd10pcs` commands because the two systems were implemented simultaneously. We preemptively exclude records before October 2015 here.

```
. drop if dmonth < tm(2015m10)
   (1,955 observations deleted)
. icd10pcs check proc1, version(2016)
   (proc1 contains defined codes; 594 missing values)
```

We find that there are no errors in the coding of the `proc1` variable and that 594 records in our dataset did not have any procedure at all.

If we wanted to check codes in more than one procedure variable, we could use a `foreach` loop or `reshape` our data; see Working with multiple codes in [D] `icd`. With large datasets, it is generally faster to use a loop.

It is a good idea to begin with `icd10pcs check` and fix any potential problems before proceeding to other `icd10pcs` commands. The `icd10pcs check` command with the `generate()` or `list` option is also useful for tracking down problems when any of the other `icd10pcs` commands tell you that the variable “contains invalid codes”.

`icd10pcs clean` formats the variable to ensure consistency and to make subsequent output from other commands such as `list` and `tabulate` look better. `icd10pcs clean` also can be used to verify that the codes in a variable conform to the ICD-10-CM format, without checking to see whether the codes are defined.

Example 2: Cleaning an existing variable

We standardize all the ICD-10-PCS procedure codes in `proc1` to include a period after the third character. We specify the `replace` option rather than the `generate()` option so that the values in `proc1` are replaced with their formatted values.

```
   . icd10pcs clean proc1, replace
   variable proc1 was str7 now str8
   (1,980 real changes made)
```

`icd10pcs clean` reports that 1,980 values were replaced. If we wanted to standardize to a format without the period, we could have specified the `nodots` option.

Aside from validating values of codes, the `icd10pcs` command is primarily used to create inputs for other Stata commands. For example, in example 5 of [D] `icd9`, we show how to graph the frequency of category codes with descriptions, and in example 3 of [D] `icd10cm`, we show how to graph summary statistics by diagnosis.
Example 3: Creating an indicator for common procedures

If we use `tabulate` on the primary procedure code (proc1) the same way we did for the primary diagnosis in example 2, we find that the three most frequent primary procedure codes in our data are 10E0XZZ, 10D00Z1, and 0SRC0J9. Suppose we want to know the average billed amount (billed) for all admissions that had one of these procedure codes in the primary procedure field.

Our first step is to create an indicator for whether one of these codes is present in proc1. Then, we summarize billed over the three top values of proc1 by using `tabulate`; see [R] `tabulate, summarize()`.

```stata
dc10pcs generate top3 = proc1, range(10E0XZZ 10D00Z1 0SRC0J9)
dc10pcs tabulate proc1 if top3==1, summarize(billed) freq means
```

<table>
<thead>
<tr>
<th>Procedure 1</th>
<th>Summary of Amount billed ($1,000s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>0SR.C0J9</td>
<td>60.62</td>
</tr>
<tr>
<td>10D.00Z1</td>
<td>27.55</td>
</tr>
<tr>
<td>10E.0XZZ</td>
<td>14.05</td>
</tr>
<tr>
<td>Total</td>
<td>24.00</td>
</tr>
</tbody>
</table>

We find that the highest average billed amount for the top three codes is for ICD-10-PCS procedure code 0SR.C0J9. There are 40 discharges in our dataset with this code as their principal procedure, and their average billed amount is about $60,620.

Interactive utilities

`dc10pcs lookup` and `dc10pcs search` are interactive tools. You can use them without having any ICD-10-PCS procedure data in memory.

`dc10pcs lookup` lists the descriptions of codes given on the command line, and `dc10pcs search` looks for relevant ICD-10-PCS procedure codes from the specified keywords. The two commands complement each other.

Example 4: Finding procedure code descriptions

Suppose we wanted to find the short descriptions of the most frequent codes in our dataset. We can supply `dc10pcs lookup` with the same list of codes we used in example 3.

```stata
dc10pcs lookup 10E0XZZ 10D00Z1 0SRC0J9, version(2016)
```

OSR.C0J9 Replace of R Knee Jt with Synth Sub, Cement, Open Approach
10D.00Z1 Extraction of POC, Low Cervical, Open Approach
10E.0XZZ Delivery of Products of Conception, External Approach

We see, for example, that ICD-10-PCS procedure code 0SR.C0J9 is for a type of knee replacement surgery.

Using `dc10pcs search` is similar to using `dc10cm search`. See example 4 in [D] `dc10cm`.
Stored results

```
icd10pcs check stores the following in r():
Scalars
  r(e#)     number of errors of type #
  r(ESUM)   total number of errors
  r(miss)   number of missing values
  r(N)      number of nonmissing values
```

```
icd10pcs clean stores the following in r():
Scalars
  r(N)      number of changes
```

```
icd10pcs lookup and icd10pcs search store the following in r():
Scalars
  r(N_codes) number of codes found
```

Acknowledgments

We thank the Washington State Department of Health’s Center for Health Statistics for providing us with access to its 2015 Comprehensive Hospital Abstract Reporting System (CHARS) inpatient dataset. The hosp2015 dataset used here was partially simulated based on information from the 2015 limited use CHARS. We also thank Jeanne M. Sears of the University of Washington for bringing the CHARS to our attention.

We thank Joe Canner of the Johns Hopkins University School of Medicine, who wrote mycd10 and mycd10p, which provide many utilities for ICD-10 diagnosis and procedure codes. The commands rely on a user-supplied ICD-10 lookup dataset for diagnosis codes and ICD-10-PCS codes from the U.S. Centers for Medicare and Medicaid Services for procedure codes.

Also see

[D] icd — Introduction to ICD commands
[D] icd9p — ICD-9-CM procedure codes
[D] icd10cm — ICD-10-CM diagnosis codes
import — Overview of importing data into Stata

Description

This entry provides a quick reference for determining which method to use for reading non-Stata data into memory. See [U] 22 Entering and importing data for more details.

Remarks and examples

Remarks are presented under the following headings:

Summary of the different methods

- import excel
- import delimited
- jdbc
- odbc
- infile (free format)—infile without a dictionary
- infix (fixed format)
- infile (fixed format)—infile with a dictionary
- import sas
- import sasxport5 and import sasxport8
- import spss
- import fred
- import haver (Windows only)
- import dbase
- spshape2dta

Examples

Video example

Summary of the different methods

import excel

- import excel reads worksheets from Microsoft Excel (.xls and .xlsx) files.
- Entire worksheets can be read, or custom cell ranges can be read.
- See [D] import excel.

import delimited

- import delimited reads text-delimited files.
- The data can be tab-separated or comma-separated. A custom delimiter may also be specified.
- An observation must be on only one line.
- The first line in the file can optionally contain the names of the variables.
- See [D] import delimited.
jdbc

- Java Database Connectivity (JDBC) is an application programming interface for the programming language Java. The `jdbc` command allows you to connect to, load data from, insert data into, and execute queries on a database using JDBC.
- See [D] `jdbc`.

odbc

- ODBC, an acronym for Open DataBase Connectivity, is a standard for exchanging data between programs. Stata supports the ODBC standard for importing data via the `odbc` command and can read from any ODBC data source on your computer.
- See [D] `odbc`.

infile (free format)—infile without a dictionary

- The data can be space-separated, tab-separated, or comma-separated.
- Strings with embedded spaces or commas must be enclosed in quotes (even if tab- or comma-separated).
- An observation can be on more than one line, or there can even be multiple observations per line.
- See [D] `infile` (free format).

infix (fixed format)

- The data must be in fixed-column format.
- An observation can be on more than one line.
- `infix` has simpler syntax than `infile` (fixed format).
- See [D] `infix` (fixed format).

infix (fixed format)—infile with a dictionary

- The data may be in fixed-column format.
- An observation can be on more than one line.
- ASCII or EBCDIC data can be read.
- `infix` (fixed format) has the most capabilities for reading data.
- See [D] `infile` (fixed format).

import sas

- `import sas` reads Version 7 SAS (.sas7bdat) files.
- `import sas` will also read value-label information from a .sas7bcat file.
- See [D] `import sas`.
import sasxport5 and import sasxport8
  o import sasxport5 reads SAS XPORT Version 5 Transport format files.
  o import sasxport5 will also read value-label information from a *formats.xpf* XPORT file.
  o import sasxport8 reads SAS XPORT Version 8 Transport format files.
  o See [D] import sasxport5 and [D] import sasxport8.

import spss
  o import spss reads IBM SPSS Statistics (*sav* and *zsav*) files.
  o See [D] import spss.

import fred
  o import fred reads Federal Reserve Economic Data.
  o To use import fred, you must have a valid API key obtained from the St. Louis Federal Reserve.
  o See [D] import fred.

import haver (Windows only)
  o import haver reads Haver Analytics (http://www.haver.com/) database files.
  o See [D] import haver.

import dbase
  o import dbase reads a version III or version IV dBase (*dbf*) file.
  o See [D] import dbase.

spshape2dta
  o spshape2dta translates the *dbf* and *shp* files of a shapefile into two Stata datasets.
  o See [SP] spshape2dta.

Examples

➤ Example 1: Tab-separated data

```
begin example1.raw
  1 0 1 John Smith m
  0 0 1 Paul Lin m
  0 1 0 Jan Doe f
  0 0 . Julie McDonald f
```

```
end example1.raw
```
contains tab-separated data. The type command with the showtabs option shows the tabs:

```
. type example1.raw, showtabs
1<T>0<T>0<T>1<T>John Smith<T>m
0<T>0<T>0<T>1<T>Paul Lin<T>m
0<T>1<T>0<T>0<T>Jan Doe<T>f
0<T>0<T>0<T>Julie McDonald<T>f
```

It could be read in by

```
. import delimited a b c name gender using example1
```

### Example 2: Comma-separated data

```
begin example2.raw
a,b,c,name,gender
1,0,1,John Smith,m
0,0,1,Paul Lin,m
0,1,0,Jan Doe,f
0,0,.Julie McDonald,f
end example2.raw
```

could be read in by

```
. import delimited using example2
```

### Example 3: Tab-separated data with double-quoted strings

```
begin example3.raw
1 0 1 "John Smith" m
0 0 1 "Paul Lin" m
0 1 0 "Jan Doe" f
0 0 . "Julie McDonald" f
end example3.raw
```

contains tab-separated data with strings in double quotes.

```
. type example3.raw, showtabs
1<T>0<T>1<T>"John Smith"<T>m
0<T>0<T>1<T>"Paul Lin"<T>m
0<T>1<T>0<T>"Jan Doe"<T>f
0<T>0<T>0<T>."Julie McDonald"<T>f
```

It could be read in by

```
. infile byte (a b c) str15 name str1 gender using example3
```

or

```
. import delimited a b c name gender using example3
```

or

```
. infile using dict3
```
where the dictionary `dict3.dct` contains

```plaintext
begin dict3.dct
    infile dictionary using example3 {
        byte  a
        byte  b
        byte  c
        str15 name
        str1  gender
    }
end dict3.dct
```

**Example 4: Space-separated data with double-quoted strings**

```plaintext
begin example4.raw
    1 0 1 "John Smith" m
    0 0 1 "Paul Lin" m
    0 1 0 "Jan Doe" f
    0 0 . "Julie McDonald" f
end example4.raw
```

could be read in by

```
    . infile byte (a b c) str15 name str1 gender using example4
```

or

```
    . infile using dict4
```

where the dictionary `dict4.dct` contains

```plaintext
begin dict4.dct
    infile dictionary using example4 {
        byte  a
        byte  b
        byte  c
        str15 name
        str1  gender
    }
end dict4.dct
```

**Example 5: Fixed-column format**

```plaintext
begin example5.raw
    101mJohn Smith
    001mPaul Lin
    010fJan Doe
    00 fJulie McDonald
end example5.raw
```

could be read in by

```
    . infix a 1 b 2 c 3 str gender 4 str name 5-19 using example5
```

or

```
    . infix using dict5a
```
where dict5a.dct contains

```
begin dict5a.dct
infix dictionary using example5 {
    a 1
    b 2
    c 3
    str gender 4
    str name 5-19
}
end dict5a.dct
```

or

```
.infile using dict5b
```

where dict5b.dct contains

```
begin dict5b.dct
infile dictionary using example5 {
    byte a %1f
    byte b %1f
    byte c %1f
    str1 gender %1s
    str15 name %15s
}
end dict5b.dct
```

Example 6: Fixed-column format with headings

```
line 1 : a heading
There are a total of 4 lines of heading.
The next line contains a useful heading:
--------|--------|--------|--------|--------|
  1 0 1 m John Smith
  0 0 1 m Paul Lin
  0 1 0 f Jan Doe
  0 0 f Julie McDonald
end example6.raw
```

could be read in by

```
.infile using dict6a
```

where dict6a.dct contains

```
infile dictionary using example6 {
    _firstline(5)
    byte a
    byte b
    _column(17) byte c %1f
    _column(33) str1 gender
    _column(33) str15 name %15s
}
end dict6a.dct
```

or could be read in by

```
.infix 5 first a 1 b 9 c 17 str gender 25 str name 33-46 using example6
```
or could be read in by

```
.infix using dict6b
```

where `dict6b.dct` contains

```
begin dict6b.dct
infix dictionary using example6 {
  5 first
    a  1
    b  9
    c 17
    str gender 25
    str name 33-46
}
end dict6b.dct
```

Example 7: Fixed-column format with observations spanning multiple lines

```
begin example7.raw
  a b c gender name
  1 0 1 m
  John Smith
  0 0 1 m
  Paul Lin
  0 1 0 f
  Jan Doe
  0 0 f
  Julie McDonald
end example7.raw
```

could be read in by

```
.infile using dict7a
```

where `dict7a.dct` contains

```
infile dictionary using example7 {
  _firstline(2)
    byte a
    byte b
    byte c
  _line(2)
    str1 gender
  _line(3)
    str15 name %15s
}
```

or, if we wanted to include variable labels,

```
.infile using dict7b
```
where dict7b.dct contains

```
begin dict7b.dct
infile dictionary using example7 {
    _firstline(2)
        byte a "Question 1"
        byte b "Question 2"
        byte c "Question 3"
    _line(2)
        str1 gender "Gender of subject"
    _line(3)
        str15 name %15s
}
end dict7b.dct
```

infix could also read these data,

```
    . infix 2 first 3 lines a 1 b 3 c 5 str gender 2:1 str name 3:1-15 using example7
```

or the data could be read in by

```
    . infix using dict7c
```

where dict7c.dct contains

```
begin dict7c.dct
infix dictionary using example7 {
    2 first
        a 1
        b 3
        c 5
    str gender 2:1
    str name 3:1-15
}
end dict7c.dct
```

or the data could be read in by

```
    . infix using dict7d
```

where dict7d.dct contains

```
begin dict7d.dct
infix dictionary using example7 {
    2 first
        a 1
        b 3
        c 5
    /
    str gender 1
    /
    str name 1-15
}
end dict7d.dct
```
Video example

Copy/paste data from Excel into Stata

References


Also see

[D] edit — Browse or edit data with Data Editor

[D] export — Overview of exporting data from Stata

[D] input — Enter data from keyboard

[U] 22 Entering and importing data
import dBASE — Import and export dBase files

Description

import dBASE reads into memory a version III or version IV dBase (.dbf) file. export dBASE exports data in memory to a version IV dBase (.dbf) file.

Stata has other commands for importing data. If you are not sure that import dBASE will do what you are looking for, see [D] import and [U] 22 Entering and importing data.

Quick start

Load the contents of the dBase file called mydata.dbf

import dBASE mydata

Write data in memory to a version IV dBASE file called mydata.dbf

export dBASE mydata

As above, but export only variables v1 and v2

export dBASE v1 v2 using mydata

Menu

import dBASE

File > Import > dBase (*.dbf)

export dBASE

File > Export > dBase (*.dbf)
import dbase — Import and export dBase files

Syntax

Load a dBase file

import dbase [using] filename [, clear case(preserve|lower|upper)]

Save data in memory to a dBase file

export dbase [using] filename [if] [in] [, datafmt replace]

Save subset of variables in memory to a dBase file

export dbase [varlist] using filename [if] [in] [, datafmt replace]

If filename is specified without an extension, .dbf is assumed for both import dbase and export dbase. If filename contains embedded spaces, enclose it in double quotes.

collect is allowed with import dbase; see [U] 11.1.10 Prefix commands.

Options for import dbase

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

case(preserve|lower|upper) specifies the case of the variable names after import. The default is case(preserve).

Options for export dbase

datafmt specifies that all variables be exported using their display format. For example, the number 1000 with a display format of %7.2f would export as 1000.00, not 1000. The default is to use the raw, unformatted value when exporting.

replace specifies that filename be replaced if it already exists.

Remarks

import dbase reads into memory a version III or version IV dBase (.dbf) file. If the dBase file is not version III or IV, import dbase will issue an error. dBase files are often paired with shapefiles for storing geometric location data. To import a shapefile, see [SP] spshape2dta.

export dbase exports data in memory to a version IV dBase (.dbf) file. dBase version IV has several file limitations when exporting.

1. Unicode is not supported.

2. Data cannot be more than 2 GB in size.

3. Data in memory must be less than 1,000,000,000 observations.

4. Data in memory must have less than 255 variables.

5. Variable names cannot exceed 10 characters in length.
6. Maximum string variable length is 255 characters.
7. Data width must be less than 4,000.

If your data in memory exceed any of these limits, `export dbase` will issue an error when trying to export the data.

To demonstrate the use of `import dbase` and `export dbase`, we will first load `autornd.dta` and export it as a dBase file named `auto.dbf`.

```
use https://www.stata-press.com/data/r17/autornd
(1978 automobile data)
export dbase auto.dbf
file auto.dbf saved
```

To import the data back into Stata, we need only to specify the filename. `import dbase` assumes an extension of `.dbf`.

```
import dbase auto, clear
(3 vars, 74 obs)
```

We could verify that our data loaded correctly by using `list` or `browse`.

### Stored results

`import dbase` stores the following in `r()`:

 Scalars

- `r(N)` number of observations imported
- `r(k)` number of variables imported

### Also see

[D] export — Overview of exporting data from Stata
[D] import — Overview of importing data into Stata
[SP] spshape2dta — Translate shapefile to Stata format
import delimited — Import and export delimited text data

Description

import delimited reads into memory a text file in which there is one observation per line and the values are separated by commas, tabs, or some other delimiter. The two most common types of text data to import are comma-separated values (.csv) text files and tab-separated text files, often .txt files. Similarly, export delimited writes Stata’s data to a text file.

Stata has other commands for importing data. If you are not sure that import delimited will do what you are looking for, see [D] import and [U] 22 Entering and importing data.

Quick start

Load comma-delimited mydata.csv with the variable names on the first row

\begin{verbatim}
import delimited mydata
\end{verbatim}

As above, but with variable names in row 5 and an ignorable header in the first 4 rows

\begin{verbatim}
import delimited mydata, varnames(5)
\end{verbatim}

Load only columns 2 to 300 and the first 1,000 rows with variable names in row 1

\begin{verbatim}
import delimited mydata, colrange(2:300) rowrange(:1000)
\end{verbatim}

Load tab-delimited data from mydata.txt

\begin{verbatim}
import delimited mydata.txt, delimiters(tab)
\end{verbatim}

Load semicolon-delimited data from mydata.txt

\begin{verbatim}
import delimited mydata.txt, delimiters(";")
\end{verbatim}

Force columns 2 to 6 to be read as string to preserve leading zeros

\begin{verbatim}
import delimited mydata, stringcols(2/6)
\end{verbatim}

Load comma-delimited mydata2.csv without variable names in row 1 and with two variables to be named v1 and v2

\begin{verbatim}
import delimited v1 v2 using mydata
\end{verbatim}

Export data in memory to mydata.csv

\begin{verbatim}
export delimited mydata
\end{verbatim}

As above, but export only v1 and v2

\begin{verbatim}
export delimited v1 v2 using mydata
\end{verbatim}

As above, but output numeric values for variables with value labels

\begin{verbatim}
export delimited v1 v2 using mydata, nolabel
\end{verbatim}
Menu

import delimited
File > Import > Text data (delimited, *.csv, ...)

export delimited
File > Export > Text data (delimited, *.csv, ...)

Syntax

Load a delimited text file

import delimited [using] filename [, import_delimited_options]

Rename specified variables from a delimited text file

import delimited extvarlist using filename [, import_delimited_options]

Save data in memory to a delimited text file

export delimited [using] filename [if] [in] [, export_delimited_options]

Save subset of variables in memory to a delimited text file

export delimited [varlist] using filename [if] [in] [, export_delimited_options]

If filename is specified without an extension, .csv is assumed for both import delimited and export delimited. If filename contains embedded spaces, enclose it in double quotes.

extvarlist specifies variable names of imported columns.
import delimited — Import and export delimited text data 441

delimiters("chars", collapse|asstring))

Description

treat row # of data as variable names or the data do not have variable names

preserve the case or read variable names as lowercase (the default) or uppercase

import all floating-point data as floats

import all floating-point data as doubles

specify the encoding of the text file being imported

remove or keep double quotes in data

number of rows of data allowed inside a quoted string when bindquote(strict) is specified

row range of data to load

column range of data to load

specify the locale to use for interpreting numbers in the text file being imported

character to use for the decimal separator when parsing numbers

character to use for the grouping separator when parsing numbers

force specified columns to be numeric

force specified columns to be string

replace data in memory

favor storing string variables as str# rather than strL

collect is allowed with import delimited; see [U] 11.1.10 Prefix commands.
favorstrfixed does not appear in the dialog box.

export delimited_options

Main

delimiter("char"|tab)
delimiters("chars", collapse|asstring)

use char as delimiter

force specified columns to be numeric

output numeric values (not labels) of labeled variables

force specified columns to be string

replace data in memory

always enclose strings in double quotes

overwrite existing filename

Options for import delimited

delimiters("chars", collapse|asstring)) allows you to specify other separation characters. For instance, if values in the file are separated by a semicolon, specify delimiters(";"). By default, import delimited will check if the file is delimited by tabs or commas based on
the first line of data. Specify delimiters("\t") to use a tab character, or specify delimiters("whitespace") to use whitespace as a delimiter.

collapse forces import delimited to treat multiple consecutive delimiters as just one delimiter.

asstring forces import delimited to treat chars as one delimiter. By default, each character in chars is treated as an individual delimiter.

varnames(#) | nonames) specifies where or whether variable names are in the data. By default, import delimited tries to determine whether the file includes variable names. import delimited translates the names in the file to valid Stata variable names. The original names from the file are stored unmodified as variable labels.

varnames(#) specifies that the variable names are in row # of the data; any data before row # should not be imported.

varnames(nonames) specifies that the variable names are not in the data.

case(preserve | lower | upper) specifies the case of the variable names after import. The default is case(lowercase).

asfloat imports floating-point data as type float. The default storage type of the imported variables is determined by set type.

asdouble imports floating-point data as type double. The default storage type of the imported variables is determined by set type.

ing encoding(encoding) specifies the encoding of the text file to be read. If encoding() is not specified, the file will be scanned to try to automatically determine the correct encoding. import delimited uses encodings available in Java, a list of which can be found at https://www.oracle.com/java/technologies/javase/jdk11-supported-locales.html.

Option charset() is a synonym for encoding().

stripquotes(yes | no | default) tells import delimited how to handle double quotes. yes causes all double quotes to be stripped. no leaves double quotes in the data unchanged. default automatically strips quotes that can be identified as binding quotes. default also will identify two adjacent double quotes as a single double quote because some software encodes double quotes that way.

bindquotes(loose | strict | nobind) specifies how import delimited handles double quotes in data. Specifying loose (the default) tells import delimited that it must have a matching open and closed double quote on the same line of data. strict tells import delimited that once it finds one double quote on a line of data, it should keep searching through the data for the matching double quote even if that double quote is on another line. Specifying nobind tells import delimited to ignore double quotes for binding.

maxquotedrows(# | unlimited) specifies the number of rows allowed inside a quoted string when parsing the file to import. The default is maxquotedrows(20). If this option is specified without bindquote(strict), then maxquotedrows() will be ignored.

Option maxquotedrows(0) is a synonym for maxquotedrows(unlimited).

rowrange([start][:end]) specifies a range of rows within the data to load. start and end are integer row numbers.

colrange([start][:end]) specifies a range of variables within the data to load. start and end are integer column numbers.

parislocale(locale) specifies the locale to use for interpreting numbers in the text file being imported. This option invokes an alternative parsing method and can result in slightly different
behavior than not specifying this option. The default is to not use a locale when parsing numbers where the behavior is to treat . as the decimal separator. A list of available locales can be found at https://www.oracle.com/technetwork/java/javase/java8locales-2095355.html.

decimalseparator(character) specifies the character to use for interpreting the decimal separator when parsing numbers. This option implicitly invokes option parselocale() with your system’s default locale. parselocale(locale) can be specified to override the default system locale.

groupeparator(character) specifies the character to use for interpreting the grouping separator when parsing numbers. This option implicitly invokes option parselocale() with your system’s default locale. parselocale(locale) can be specified to override the default system locale.

numericcols(numlist | _all) forces the data type of the column numbers in numlist to be numeric. Specifying _all will import all data as numeric.

stringcols(numlist | _all) forces the data type of the column numbers in numlist to be string. Specifying _all will import all data as strings.

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

The following option is available with import delimited but is not shown in the dialog box:

favorstrfixed forces import delimited to favor storing strings as a str#.

By default, import delimited will attempt to save space by importing string data as a strL if doing so will save space. The favorstrfixed option prevents the space-saving calculation from occurring, causing strings to be stored as a str# unless the string is larger than a str# can hold. In that case, strL must be used. See [R] Limits for details about the maximum size of a str#.

Options for export delimited

delimiter("char" | tab) allows you to specify other separation characters. For instance, if you want the values in the file to be separated by a semicolon, specify delimiter(";"). The default delimiter is a comma.

delimiter(tab) specifies that a tab character be used as the delimiter.

novarnames specifies that variable names not be written in the first line of the file; the file is to contain data values only.

nolabel specifies that the numeric values of labeled variables be written into the file rather than the label associated with each value.

datafmt specifies that all variables be exported using their display format. For example, the number 1000 with a display format of %4.2f would export as 1000.00, not 1000. The default is to use the raw, unformatted value when exporting.

quote specifies that string variables always be enclosed in double quotes. The default is to only double quote strings that contain spaces or the delimiter.

replace specifies that filename be replaced if it already exists.
Remarks and examples

Remarks are presented under the following headings:

- Introduction
- Importing a text file
  - Using other delimiters
  - Specifying variable types
- Exporting to a text file
- Video example

Introduction

`import delimited` reads into memory a text file in which there is one observation per line and the values are separated by commas, tabs, or some other delimiter. The two most common types of text data to import are comma-separated values (.csv) text files and tab-separated text files, often .txt files. `import delimited` will automatically detect either a comma or a tab as the delimiter.

Similarly, `export delimited` writes Stata data to a text file. By default, `export delimited` uses a comma as the delimiter, but you may specify another delimiter.

Imported string data containing ASCII or UTF-8 will always display correctly in the Data Editor and Results window. Imported string data containing extended ASCII may not display correctly unless you specify the character encoding using the `encoding()` option to convert the extended ASCII to UTF-8.

Exported text files are UTF-8 encoded.

If you are not sure that `import delimited` will do what you are looking for, see [D] `import` and [U] 22 Entering and importing data for information about Stata’s other commands for importing data.

Importing a text file

Suppose we have a .csv data file such as the following `auto.csv`, which contains variable names and data for different cars.

```
. copy https://www.stata.com/examples/auto.csv auto.csv
. type auto.csv
  make, price, mpg, rep78, foreign
  "AMC Concord", 4099, 22, 3, "Domestic"
  "AMC Pacer", 4749, 17, 3, "Domestic"
  "AMC Spirit", 3799, 22, 3, "Domestic"
  "Buick Century", 4816, 20, 3, "Domestic"
  "Buick Electra", 7827, 15, 4, "Domestic"
  "Buick LeSabre", 5788, 18, 3, "Domestic"
  "Buick Opel", 4453, 26, 3, "Domestic"
  "Buick Regal", 5189, 20, 3, "Domestic"
  "Buick Riviera", 10372, 16, 3, "Domestic"
  "Buick Skylark", 4082, 19, 3, "Domestic"
```

We would like to import these data into Stata for subsequent analysis.

Example 1: Importing all data

To import the complete dataset, we need to specify only the filename. `import delimited` assumes an extension of .csv. If our data were stored in a .txt file instead, we would need to specify the file extension. Here we enclose `auto` in double quotes (" "). We do this to remind you to use quotes for filenames with spaces, but it is not necessary here.
We can verify that our data loaded correctly by using `list` or `browse`.

```
. list

<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>mpg</th>
<th>rep78</th>
<th>foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC Concord</td>
<td>4099</td>
<td>22</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>AMC Pacer</td>
<td>4749</td>
<td>17</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>AMC Spirit</td>
<td>3799</td>
<td>22</td>
<td>.</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Century</td>
<td>4816</td>
<td>20</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Electra</td>
<td>7827</td>
<td>15</td>
<td>4</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick LeSabre</td>
<td>5788</td>
<td>18</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Opel</td>
<td>4453</td>
<td>26</td>
<td>.</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Regal</td>
<td>5189</td>
<td>20</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Riviera</td>
<td>10372</td>
<td>16</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Skylark</td>
<td>4082</td>
<td>19</td>
<td>3</td>
<td>Domestic</td>
</tr>
</tbody>
</table>
```

Notice that `import delimited` automatically assigned the variable names such as `make` and `price` based on the first row of the data. If the variable names were located on, for example, line 3, we would have specified `varnames(3)`, and `import delimited` would have ignored the first two rows. If our file did not contain any variable names, we would have specified `varnames(nonames)`.

**Example 2: Importing a subset of the data**

`import delimited` also allows you to import a subset of the text data by using the `rowrange()` and `colrange()` options. Use `rowrange()` to specify which observations you want to import and `colrange()` to specify which variables you want to import.

Suppose that we want only cars that were manufactured by AMC. We can use the `drop` command to drop the cars manufactured by Buick after we import the data. If we know the rows in which AMC cars are located, we can also restrict our import to just those rows. Because `foreign` is constant, we also want to skip the last column.

To import rows 1 through 3 of the data in `auto.csv`, we need to specify `rowrange(2:4)` because the first row of the file contains the variable names. To import the first four columns, we need to also specify `colrange(1:4)`.}

```
. clear
. import delimited "auto", rowrange(2:4) colrange(1:4)
  (encoding automatically selected: ISO-8859-1)
  (4 vars, 3 obs)
. list

<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>mpg</th>
<th>rep78</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC Concord</td>
<td>4099</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>AMC Pacer</td>
<td>4749</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>AMC Spirit</td>
<td>3799</td>
<td>22</td>
<td>.</td>
</tr>
</tbody>
</table>
```
import delimited still used the first line of the file to obtain the variable names even though we did not start our rowrange() specification with 1. rowrange() controls only which rows are read as data to be imported into Stata.

Using other delimiters

Many delimited files use commas or tabs; other common delimiters are semicolons and whitespace. import delimited detects commas and tabs by default but can handle other characters. Suppose that you had the auto.txt file, which contains the following data.

<table>
<thead>
<tr>
<th>Make</th>
<th>Year</th>
<th>Miles</th>
<th>Size</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;AMC Concord&quot;</td>
<td>4000</td>
<td>22</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;AMC Pacer&quot;</td>
<td>4700</td>
<td>17</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;AMC Spirit&quot;</td>
<td>3700</td>
<td>22</td>
<td>NA</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick Century&quot;</td>
<td>4816</td>
<td>20</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick Electra&quot;</td>
<td>7827</td>
<td>15</td>
<td>4</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick LeSabre&quot;</td>
<td>5788</td>
<td>18</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick Opel&quot;</td>
<td>4453</td>
<td>26</td>
<td>NA</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick Regal&quot;</td>
<td>5189</td>
<td>20</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick Riviera&quot;</td>
<td>10372</td>
<td>16</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
<tr>
<td>&quot;Buick Skylark&quot;</td>
<td>4082</td>
<td>19</td>
<td>3</td>
<td>&quot;Domestic&quot;</td>
</tr>
</tbody>
</table>

These data are whitespace delimited. If you use import delimited without any options, you will not get the results you expect.

```stata
.clear
.import delimited "auto.txt"
(encoding automatically selected: ISO-8859-1)
(1 var, 10 obs)
```

When import delimited tries to read data that have no tabs or commas, it is fooled into thinking that the data contain just one variable.

Example 3: Changing the delimiter

We can use the delimiters() option to import the data correctly. delimiters(" ") tells import delimited to use spaces (" ") as the delimiter. Adding the collapse suboption will treat multiple consecutive space delimiters as one delimiter.

```stata
.clear
.import delimited "auto.txt", delimiters(" ", collapse)
(encoding automatically selected: ISO-8859-1)
(5 vars, 10 obs)
.describe
Contains data
Observations: 10
Variables: 5
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>str13</td>
<td>%13s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v3</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v4</td>
<td>str2</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v5</td>
<td>str8</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.
The data that were imported now contain the correct number of variables and observations.

Because `import delimited` did not find variable names in the first row of `auto.txt`, Stata assigned default names of `v#` to the imported variables. If we wanted to specify our own names, we could have instead submitted

```
. clear
. import delimited make price mpg rep78 foreign using auto.txt,
> delimiters(" ", collapse)
(5 vars, 10 obs)
```

### Specifying variable types

The data in a file may contain a combination of string and numeric variables. `import delimited` will generally determine the correct data type for each variable. However, you may want to force a different data type by using the `numericcols()` or `stringcols()` option. For example, string values may be used to indicate missing values in a numeric variable, or you may want to import numeric values as strings to preserve leading zeros.

Another common case where you want to control the import type is when your data contain identifiers or other large numeric values. In this case, you should specify the `asdouble` option to avoid introducing duplicate values or losing values after the import.

**Example 4: Specify the storage type**

Continuing with example 3, we know that the fourth variable, `rep78`, should be a numeric variable. But it was imported as a string because the value `NA` was used for missing values.

```
. list

   +----------+----------+----+--------+----------+
   | make     | price    | mpg | rep78  | foreign  |
   +----------+----------+----+--------+----------+
   | AMC Concord | 4099     | 22 | 3      | Domestic |
   | AMC Pacer  | 4749     | 17 | 3      | Domestic |
   | AMC Spirit | 3799     | 22 | NA     | Domestic |
   | Buick Century | 4816   | 20 | 3      | Domestic |
   | Buick LeSabre | 5788   | 18 | 3      | Domestic |
   +----------+----------+----+--------+----------+
   | Buick Opel | 4453     | 26 | NA     | Domestic |
   | Buick Regal | 5189     | 20 | 3      | Domestic |
   | Buick Riviera | 10372  | 16 | 3      | Domestic |
   | Buick Skylark | 4082   | 19 | 3      | Domestic |
   +----------+----------+----+--------+----------+
```
To force `rep78` to have a numeric storage type, we can use the `numericcols(4)` option.

```
. clear
. import delimited make price mpg rep78 foreign using "auto.txt",
   delimiters(" ", collapse) numericcols(4)
(5 vars, 10 obs)
. describe
Contains data
   Observations: 10
   Variables: 5

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str13</td>
<td>%13s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>str8</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Sorted by:
   Note: Dataset has changed since last saved.

```
. list
<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>mpg</th>
<th>rep78</th>
<th>foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC Concord</td>
<td>4099</td>
<td>22</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>AMC Pacer</td>
<td>4749</td>
<td>17</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>AMC Spirit</td>
<td>3799</td>
<td>22</td>
<td>.</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Century</td>
<td>4816</td>
<td>20</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick LeSabre</td>
<td>7827</td>
<td>15</td>
<td>4</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Opel</td>
<td>4453</td>
<td>26</td>
<td>.</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Regal</td>
<td>5189</td>
<td>20</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Riviera</td>
<td>10372</td>
<td>16</td>
<td>3</td>
<td>Domestic</td>
</tr>
<tr>
<td>Buick Skylark</td>
<td>4082</td>
<td>19</td>
<td>3</td>
<td>Domestic</td>
</tr>
</tbody>
</table>
```

`rep78` is now stored as an `int` variable, and the `NA` values are replaced by `.`, the system missing value for numeric variables.

### Exporting to a text file

`export delimited` creates text files from the Stata dataset in memory. A comma-separated `.csv` file is created by default, but you can change the delimiter by specifying the `delimiter()` option and the file extension by specifying it with the `filename`. 
Example 5: Export all data

We want to export the data from example 4 to myauto.csv. We can use the type command to see the contents of the file.

```bash
. export delimited "myauto"
file myauto.csv saved
. type "myauto.csv"
make,price,mpg,rep78,foreign
AMC Concord,4099,22,3,Domestic
AMC Pacer,4749,17,3,Domestic
AMC Spirit,3799,22,,Domestic
Buick Century,4816,20,3,Domestic
Buick Electra,7827,15,4,Domestic
Buick LeSabre,5788,18,3,Domestic
Buick Opel,4453,26,,Domestic
Buick Regal,5189,20,3,Domestic
Buick Riviera,10372,16,3,Domestic
Buick Skylark,4082,19,3,Domestic
```

Example 6: Export a subset of the data

You can also export a subset of the data in memory by typing a variable list, specifying an if condition, specifying a range with an in condition, or a combination of the three. For example, here we export only the first 5 observations of the make, mpg, and rep78 variables.

```bash
. export delimited make mpg rep78 in 1/5 using "myauto", replace
file myauto.csv saved
```

If you open myauto.csv, you will see that only the 5 observations shown in example 5 appear in the file. We specified the replace option because we previously exported data to myauto.csv. If we had not specified replace, we would have received an error message.

Video example

Importing delimited data

Stored results

import delimited stores the following in r():

Scalars

- r(N) number of observations imported
- r(k) number of variables imported

Macros

- r(delimiter) delimiters used when importing the file
- r(encoding) encoding used when importing the file
Also see

[D] `export' — Overview of exporting data from Stata

[D] `import' — Overview of importing data into Stata
import excel — Import and export Excel files

Description

import excel loads an Excel file, also known as a workbook, into Stata. import excel filename, describe lists available sheets and ranges of an Excel file. export excel saves data in memory to an Excel file. Excel 1997/2003 (.xls) files and Excel 2007/2010 (.xlsx) files can be imported, exported, and described using import excel, export excel, and import excel, describe.

import excel and export excel are supported on Windows, Mac, and Linux.

import excel and export excel look at the file extension, .xls or .xlsx, to determine which Excel format to read or write.

For performance, import excel imposes a size limit of 40 MB for Excel 2007/2010 (.xlsx) files. Be warned that importing large .xlsx files can severely affect your machine’s performance.

import excel auto first looks for auto.xls and then looks for auto.xlsx if auto.xls is not found in the current directory.

The default file extension for export excel is .xls if a file extension is not specified.

Quick start

Check the contents of Excel file mydata.xls before importing
import excel mydata, describe

As above, but for mydata.xlsx
import excel mydata.xlsx, describe

Load data from mydata.xls
import excel mydata

As above, but load data from cells A1:G10 of mysheet
import excel mydata, cellrange(A1:G10) sheet(mysheet)

Read first row as lowercase variable names
import excel mydata, firstrow case(lower)

Import only v1 and v2
import excel v1 v2 using mydata

Save data in memory to mydata.xls
export excel mydata

As above, but export variables v1, v2, and v3
export excel v1 v2 v3 using mydata

451
Menu

import excel

File > Import > Excel spreadsheet (*.xls;*.xlsx)

export excel

File > Export > Data to Excel spreadsheet (*.xls;*.xlsx)

Syntax

Load an Excel file

\texttt{import excel [using] filename [, import_excel_options]}

Load subset of variables from an Excel file

\texttt{import excel extvarlist using filename [, import_excel_options]}

Describe contents of an Excel file

\texttt{import excel [using] filename, describe}

Save data in memory to an Excel file

\texttt{export excel [using] filename [if] [in] [, export_excel_options]}

Save subset of variables in memory to an Excel file

\texttt{export excel [varlist] using filename [if] [in] [, export_excel_options]}

\textit{import_excel_options}

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sheet(&quot;sheetname&quot;)</td>
</tr>
<tr>
<td>cellrange([start]:end)</td>
</tr>
<tr>
<td>firstrow</td>
</tr>
<tr>
<td>case(preserve</td>
</tr>
<tr>
<td>allstring[&quot;format&quot;]</td>
</tr>
<tr>
<td>clear</td>
</tr>
<tr>
<td>locale(&quot;locale&quot;)</td>
</tr>
</tbody>
</table>

\texttt{allstring("format") and locale()} do not appear in the dialog box.
### export_excel_options

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>save to Excel worksheet</td>
</tr>
<tr>
<td>start (upper-left) cell in Excel to begin saving to</td>
</tr>
<tr>
<td>save variable names or variable labels to first row</td>
</tr>
<tr>
<td>export values instead of value labels</td>
</tr>
<tr>
<td>when writing data, preserve the cell style and format of existing worksheet</td>
</tr>
<tr>
<td>overwrite Excel file</td>
</tr>
</tbody>
</table>

**Main**

- `sheet("sheetname"[, modify|replace])`
- `cell(start)`
- `firstrow(variables|varlabels)`
- `nolabel`
- `keepcellfmt`
- `replace`

**Advanced**

- `datestring("datetime_format")`
- `missing("repval")`
- `locale("locale")`

`collect` is allowed with `import excel`; see [U] 11.1.10 Prefix commands.

`locale()` does not appear in the dialog box.

**extvarlist** specifies variable names of imported columns. An extvarlist is one or more of any of the following:

```
varname
varname=columnname
```


Example: `import excel make=A mpg=B price=D using auto.xlsx, clear` imports columns A, B, and D from the Excel file `auto.xlsx`. Column C and any columns after D are skipped.

### Options for import excel

- `sheet("sheetname")` imports the worksheet named `sheetname` in the workbook. The default is to import the first worksheet.
- `cellrange([start][:end])` specifies a range of cells within the worksheet to load. `start` and `end` are specified using standard Excel cell notation, for example, A1, BC2000, and C23.
- `firstrow` specifies that the first row of data in the Excel worksheet consists of variable names. This option cannot be used with `extvarlist`. `firstrow` uses the first row of the cell range for variable names if `cellrange()` is specified. `import excel` translates the names in the first row to valid Stata variable names. The original names in the first row are stored unmodified as variable labels.
- `case(preserve|lower|upper)` specifies the case of the variable names read when using the `firstrow` option. The default is `case(preserve)`, meaning to preserve the variable name case. Only the ASCII letters in names are changed to lowercase or uppercase. Unicode characters beyond ASCII range are not changed.
- `allstring[("format")]]` forces `import excel` to import all Excel data as string data. You can specify the numeric display format used to convert the numeric data to string using the optional argument `format`. See [D] `format`.

`clear` clears data in memory before loading data from the Excel workbook.
The following option is available with import excel but is not shown in the dialog box:

locale("locale") specifies the locale used by the workbook. You might need this option when working with extended ASCII character sets. This option has no effect on Microsoft Windows. The default locale is UTF-8.

### Options for export excel

#### Main

- **sheet("sheetname"[, modify | replace])** saves to the worksheet named *sheetname*. If there is no worksheet named *sheetname* in the workbook, a new sheet named *sheetname* is created. If this option is not specified, the first worksheet of the workbook is used. If *sheetname* does exist in the workbook, you can either modify or replace the worksheet.

  modify exports data to the worksheet without changing the cells outside the exported range. This option cannot be specified with replace, nor when overwriting the Excel workbook.

  replace clears the worksheet before the data are exported to it. replace cannot be specified with modify, nor when overwriting the Excel workbook.

- **cell(start)** specifies the start (upper-left) cell in the Excel worksheet to begin saving to. By default, export excel saves starting in the first row and first column of the worksheet.

- **firstrow(variables | varlabels)** specifies that the variable names or the variable labels be saved in the first row in the Excel worksheet. The variable name is used if there is no variable label for a given variable.

- **nolabel** exports the underlying numeric values instead of the value labels.

- **keepcellfmt** specifies that, when writing data, export excel should preserve the existing worksheet’s cell style and format. By default, export excel does not preserve a cell’s style or format.

- **replace** overwrites an existing Excel workbook. replace cannot be specified when modifying or replacing a given worksheet: export excel ..., sheet("", modify) or export excel ... sheet(""", replace).

#### Advanced

- **datestring("datetime_format")** exports all datetime variables as strings formatted by *datetime_format*. See [D] Datetime display formats.

- **missing("repval")** exports missing values as *repval*. *repval* can be either string or numeric. Without specifying this option, export excel exports the missing values as empty cells.

The following option is available with export excel but is not shown in the dialog box:

locale("locale") specifies the locale used by the workbook. You might need this option when working with extended ASCII character sets. The default locale is UTF-8.

### Remarks and examples

To demonstrate the use of import excel and export excel, we will first load auto.dta and export it as an Excel file named auto.xls:

```
use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
.export excel auto, firstrow(variables)
file auto.xls saved
```
Now we can import from the auto.xls file we just created, telling Stata to clear the current data from memory and to treat the first row of the worksheet in the Excel file as variable names:

```
. import excel auto.xls, firstrow clear
(12 vars, 74 obs)
. describe
Contains data
Observations: 74
Variables: 12
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%17s</td>
<td>make</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%10.0gc</td>
<td>price</td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%10.0g</td>
<td>mpg</td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%10.0g</td>
<td>rep78</td>
<td></td>
</tr>
<tr>
<td>headroom</td>
<td>double</td>
<td>%10.0g</td>
<td>headroom</td>
<td></td>
</tr>
<tr>
<td>trunk</td>
<td>byte</td>
<td>%10.0g</td>
<td>trunk</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%10.0gc</td>
<td>weight</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%10.0g</td>
<td>length</td>
<td></td>
</tr>
<tr>
<td>turn</td>
<td>byte</td>
<td>%10.0g</td>
<td>turn</td>
<td></td>
</tr>
<tr>
<td>displacement</td>
<td>int</td>
<td>%10.0g</td>
<td>displacement</td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>double</td>
<td>%14.2f</td>
<td>gear_ratio</td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>str8</td>
<td>%9s</td>
<td>foreign</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.

We can also import a subrange of the cells in the Excel file:

```
. import excel auto.xls, cellrange(:D70) firstrow clear
(4 vars, 69 obs)
. describe
Contains data
Observations: 69
Variables: 4
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%17s</td>
<td>make</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%10.0gc</td>
<td>price</td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%10.0g</td>
<td>mpg</td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%10.0g</td>
<td>rep78</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.

Both .xls and .xlsx files are supported by import excel and export excel. If a file extension is not specified with export excel, .xls is assumed, because this format is more common and is compatible with more applications that also can read from Excel files. To save the data in memory as a .xlsx file, specify the extension:

```
. use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
. export excel auto.xlsx
file auto.xlsx saved
```
To export a subset of variables and overwrite the existing `auto.xls` Excel file, specify a variable list and the `replace` option:

```plaintext
.expexcel make mpg weight using auto, replace
    file `auto.xls` saved
```

For additional examples illustrating `import excel` and `export excel`, see Mitchell (2020, chap. 2–3).

### Technical note: Excel data size limits

For an Excel `.xls`-type workbook, the worksheet size limits are 65,536 rows by 256 columns. The string size limit is 255 characters.

For an Excel `.xlsx`-type workbook, the worksheet size limits are 1,048,576 rows by 16,384 columns. The string size limit is 32,767 characters.

### Technical note: Dates and times

Excel has two different date systems, the “1900 Date System” and the “1904 Date System”. Excel stores a date and time as an integer representing the number of days since a start date plus a fraction of a 24-hour day.

In the 1900 Date System, the start date is 00Jan1900; in the 1904 Date System, the start date is 01Jan1904. In the 1900 Date System, there is another artificial date, 29feb1900, besides 00Jan1900. `import excel` translates 29feb1900 to 28feb1900 and 00Jan1900 to 31dec1899.

See *Converting Excel dates* in [D] *Datetime values from other software* for a discussion of the relationship between Stata datetimes and Excel datetimes.

### Technical note: Mixed data types

Because Excel’s data type is cell based, `import excel` may encounter a column of cells with mixed data types. In such a case, the following rules are used to determine the variable type in Stata of the imported column:

- If the column contains at least one cell with nonnumerical text, the entire column is imported as a string variable.
- If an all-numerical column contains at least one cell formatted as a date or time, the entire column is imported as a Stata date or datetime variable. `import excel` imports the column as a Stata date if all date cells in Excel are dates only; otherwise, a datetime is used.

### Video example

*Import Excel data into Stata*
Stored results

Import Excel

`import excel filename, describe` stores the following in `r()`:

Scalars

- `r(N_worksheet)` number of worksheets in the Excel workbook

Macros

- `r(worksheet_#)` name of worksheet # in the Excel workbook
- `r(range_#)` available cell range for worksheet # in the Excel workbook

References


Also see

[D] Datetime — Date and time values and variables
[D] export — Overview of exporting data from Stata
[D] import — Overview of importing data into Stata
[M-5] _docx*( ) — Generate Office Open XML (.docx) file
[M-5] xl( ) — Excel file I/O class
[RPT] putexcel — Export results to an Excel file
**import fred — Import data from Federal Reserve Economic Data**

**Description**

`import fred` imports data from the Federal Reserve Economic Data (FRED) into Stata. `import fred` supports data on FRED as well as historical vintage data on Archival FRED (ALFRED). `freddescribe` and `fredsearch` provide tools to describe series in the database and to search FRED for data based on keywords and tags.

**Quick start**

Before running any of the commands below, you will need to obtain a FRED key and set it using `set fredkey`.

- Import series `code1` and `code2` from FRED
  ```stata
  import fred code1 code2
  ```

- Import vintage series `code1` and `code2` as available on September 15, 2008, and September 15, 2009, from FRED
  ```stata
  ```

- Display metadata describing series `code1` and `code2`
  ```stata
  freddescribe code1 code2
  ```

- Search FRED for series matching keywords “investment” and “share” and tagged with “pwt” and “usa”
  ```stata
  fredsearch investment share, tags(pwt usa)
  ```

**Menu**

File > Import > Federal Reserve Economic Data (FRED)
Syntax

Set FRED key

set fredkey key [, permanently]

Import FRED data

import fred series_list [, options]

or

import fred, serieslist(filename) [options]

Describe series

freddescribe series_list [, detail realtime(start end)]

Search series

fredsearch keyword_list [, search_options]

key is a valid API key, which is provided by the St. Louis Federal Reserve and may be obtained from https://research.stlouisfed.org/docs/api/api_key.html.

series_list is a list of FRED codes, for example, FEDFUNDS.

keyword_list is a list of keywords.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*serieslist(filename)</td>
<td>specify series IDs using a file</td>
</tr>
<tr>
<td>daterange(start end)</td>
<td>restrict to only observations within specified date range</td>
</tr>
<tr>
<td>aggregate(frequency[, method])</td>
<td>specify the aggregation level and aggregation type</td>
</tr>
<tr>
<td>realtime(start end)</td>
<td>import historical vintages between specified dates</td>
</tr>
<tr>
<td>vintage(datespec)</td>
<td>import historical data by vintage dates</td>
</tr>
<tr>
<td>nrobs</td>
<td>import only new and revised observations</td>
</tr>
<tr>
<td>initial</td>
<td>import only first value for each observation in a series</td>
</tr>
<tr>
<td>long</td>
<td>import data in long format</td>
</tr>
<tr>
<td>nosumm</td>
<td>suppress summary table</td>
</tr>
<tr>
<td>clear</td>
<td>clear data in memory before importing FRED series</td>
</tr>
</tbody>
</table>

*serieslist() is required if series_list is not specified.
collect is allowed with fredsearch; see [U] 11.1.10 Prefix commands.
clear does not appear in the dialog box.

If start and end are provided as dates, they must be daily dates using notation of the form 31Jan2016, 2016-01-31, 2016/01/31, or 01/31/2016.

datespec may be

- date
- date1 date2 ... daten
- _all

a daily date
a list of daily dates
all available dates
search_options | Description
---|---
idonly | require keywords to appear in series IDs only
tags(tag_list) | search by tag_list
taglist | list tags present in current search results
sort(sortby[, sortorder]) | list matched series in order specified by sortby
detail | list full metainformation for each search result
saving(filename[, replace]) | save series information to filename.dta

saving() does not appear in the dialog box.

## Options

Options are presented under the following headings:

- Option for set fredkey
- Options for import fred
- Options for freddescribe
- Options for fredsearch

### Option for set fredkey

`permanently` specifies that, in addition to setting the key for the current Stata session, the key be remembered and become the default key when you invoke Stata.

### Options for import fred

- `serieslist(filename)` allows you to import the series specified in `filename`. The series file must contain a variable called `seriesid` that contains the IDs of the series you wish to import. `serieslist()` is required if `series_list` is not specified.

- `daterange(start end)` specifies that only observations between the `start` date and `end` date should be imported. `start` and `end` must be specified as either a daily date or a missing value (.). Use `daterange(. end)` to import all observations from the first available through `end`. Use `daterange(start .)` to import from `start` through the most recently available date.

- `aggregate(frequency[, method])` specifies that the data should be imported at a lower frequency than the series’ native frequency along with an optional method of aggregation.

  - `frequency` may be `daily`, `weekly`, `biweekly`, `monthly`, `quarterly`, `semiannual`, `annual`, `weekly ending friday`, `weekly ending thursday`, `weekly ending wednesday`, `weekly ending tuesday`, `weekly ending monday`, `weekly ending sunday`, `weekly ending saturday`, `biweekly ending wednesday`, or `biweekly ending monday`.

  - `method` may be `avg` (the within-period average), `sum` (the within-period sum), or `eop` (the end-of-period value). The default is `avg`.

- `realtime(start end)` specifies a real-time period between which all vintages for each series are imported. The vintage available on `start` is imported, as are all vintages released between `start` and `end`. Either of `start` or `end` may be replaced by a missing value (.). If `start` is a missing value, then all vintages from the first available up through `end` are imported. If `end` is a missing value, then all vintages from `start` up through the most recent available are imported. `realtime()` may not be combined with `vintage()`.
vintage((datespec)) imports historical vintage data according to datespec. datespec may either be a list of daily dates or _all. When datespec is a list of dates, the specified series are imported as they were available on the dates in datespec. When datespec is _all, all vintages of the specified series are imported. vintage() may not be combined with realtime().

nrobs specifies that only observations that are new or revised in each vintage be imported. Old and unrevised observations are imported as the missing value .u.

initial specifies that only the first value for each observation of the series be imported. This option may not be combined with nrobs.

long specifies that each series be imported in long format.

nosummary suppresses the summary table.

The following option is available with import fred but is not shown in the dialog box:

clear specifies that the data in memory should be replaced with the imported FRED data.

Options for freddescribe

detail displays full metainformation available about series_list.

realtime(start end) provides historical vintage information about series_list during the real-time period specified by start and end. Either start or end may be replaced by a missing value (.). If start is a missing value, then all vintages from the first available up through end are described. If end is a missing value, then all vintages from start up through the most recent available are described.

Options for fredsearch

idonly specifies that the keywords in keyword_list be found in series IDs rather than elsewhere in the metadata.

tags(tag_list) searches for series that have all the tags specified in tag_list. The complete list of available tags is provided by FRED. Tags form a space-separated list. Tags are case-sensitive and all FRED tags are in lowercase.

taglist lists all the tags present in the current search results.

sort(sortby[ , sortorder ]) lists the search results in the order specified by sortby.

When searching series, sortby may be popularity, id, title, lastupdated, frequency, obsstart, obsend, units, or seasonaladj. By default, popularity is used.

When searching with the taglist option, sortby may be name or series_count. name means the tag name, and series_count is the count of series associated with the tag in the search results. By default, series_count is used.

You can optionally change the order of the search results from descending (descending) to ascending (ascending) order. The default order when searching by popularity, lastupdated, or series_count is descending; otherwise, the default sort order is ascending.

detail lists full metainformation for each series that appears in the search results.

The following option is available with fredsearch but is not shown in the dialog box:

saving(filename[ , replace ]) saves the search results to a file. The filename may then be specified in the serieslist() option of import fred to import the series located by the search. The optional replace specifies that filename be overwritten if it exists.
Remarks and examples

Remarks are presented under the following headings:

- Introduction and setup
- The FRED interface
- Advanced imports using the import fred command
- Importing historical vintage data
- Searching, saving, and retrieving series information
- Describing series

Introduction and setup

`import fred` imports data from the Federal Reserve Economic Data (FRED) into Stata. FRED is maintained by the Economic Research Division of the Federal Reserve Bank of St. Louis and contains hundreds of thousands of economic and financial time series. FRED includes data from a variety of sources, including the Federal Reserve, the Penn World Table, Eurostat, the World Bank, and U.S. statistical agencies, among others. `import fred` extends `freduse` discussed in Drukker (2006).

Series in FRED are updated and revised over time as new observations are added and as older observations are revised in light of more complete source information. The series are updated on an annual, quarterly, monthly, weekly, or daily basis, depending on the series. Each time a series is updated or revised, a new “vintage” is created. The archived data, or historical vintage data, are data in their unrevised form as they would have been available on a particular date in history. These data are from Archival FRED, or ALFRED. `import fred` can import data from either FRED or ALFRED. FRED data can be imported using the `import fred` command or using the FRED interface. If you are exploring FRED, learning the names of series, or importing series occasionally, we recommend using the FRED interface. If you already know the names of the series that you would like to import or if you repeatedly download series as they are updated, we recommend using the `import fred` command. You may also use the FRED interface to learn series names that you subsequently specify in `import fred` commands. See `The FRED interface` below to learn more about using this tool.

Whether you plan to use the FRED interface or the `import fred` command, you must first have a valid API key. API keys are provided by the St. Louis Federal Reserve and may be obtained from https://research.stlouisfed.org/docs/api/api_key.html. The key will be a 32-character alphanumeric string. You will be prompted to enter this key the first time you open the FRED interface. Alternatively, you can type

```
.set fredkey key, permanently
```

where `key` is your API key.

Example 1: A basic search and import

Suppose we want monthly data on the exchange rate between the U.S. dollar and the Japanese Yen. We can use `fredsearch` to find the name of this series in FRED.

```
.fredsearch us dollar yen exchange rate monthly
```

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Title</th>
<th>Data range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXJpus</td>
<td>Japan / U.S. Foreign Exchange Rate</td>
<td>1971-01-01 to 2021-03-01</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

Total: 1
The output says that EXJPUS is the name that FRED uses for this series. When we performed this search, 2021-03-01 was the last available observation. More data will be available when you type this command, so the endpoint of the data range will be more recent.

Having learned from the output that EXJPUS is the name that FRED uses for this series, we use `import fred` to import it.

```
. import fred EXJPUS
```

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXJPUS</td>
<td>603</td>
<td>1971-01-01 to 2021-03-01</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

The output says that 603 monthly observations on EXJPUS were imported.

To clarify what we imported, we can describe the imported data and list the first five observations.

```
. describe
Contains data
Observations: 603
Variables: 3

<table>
<thead>
<tr>
<th>Variable Storage Display Value</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>type format label</td>
</tr>
<tr>
<td>datestr</td>
<td>str10 %-10s observation date</td>
</tr>
<tr>
<td>daten</td>
<td>%td numeric (daily) date</td>
</tr>
<tr>
<td>EXJPUS</td>
<td>float %9.0g Japan / U.S. Foreign Exchange Rate</td>
</tr>
</tbody>
</table>

Sorted by: datestr

Note: Dataset has changed since last saved.
```

```
. list datestr daten EXJPUS in 1/5
```

<table>
<thead>
<tr>
<th>datestr</th>
<th>daten</th>
<th>EXJPUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1971-01-01</td>
<td>01jan1971</td>
<td>358.02</td>
</tr>
<tr>
<td>1971-02-01</td>
<td>01feb1971</td>
<td>357.545</td>
</tr>
<tr>
<td>1971-03-01</td>
<td>01mar1971</td>
<td>357.5187</td>
</tr>
<tr>
<td>1971-04-01</td>
<td>01apr1971</td>
<td>357.5032</td>
</tr>
<tr>
<td>1971-05-01</td>
<td>01may1971</td>
<td>357.413</td>
</tr>
</tbody>
</table>

Each series in FRED is paired with a string variable that records the daily date for each observation. `import fred` imports this daily date variable as the string variable `datestr`, and it creates `daten`, which is a Stata datetime variable that encodes the date in `datestr`. EXJPUS contains the observations on the FRED series EXJPUS.

Each series has metadata associated with it that is stored in the characteristics and may be viewed with the `char list` command. We now list out the metadata on EXJPUS.
The names of FRED series are not predictable. The FRED interface makes it easy to find series, to import series, and to explore the thousands of series by keyword searches or by browsing by category, release type, source, or release date.

Selecting

File > Import > Federal Reserve Economic Data (FRED)

from the menu opens the FRED interface.
In the top left-hand corner, the drop-down menu defaults to Search FRED, which searches for series by keywords that appear in those series’ metadata. From this menu, we can also select Browse by category, Browse by release, Browse by source, and Search by release date.

Browse by category finds series by browsing through FRED defined categories, such as Production & Business Activity.

Browse by release finds series by browsing through FRED defined release types, such as the BEA Regions Employment and Unemployment and the Consumer Price Index.

Browse by source finds series by browsing through sources, such as the Bank of England, the US Bureau of the Census, and the University of Pennsylvania.

Search by release date finds regularly released series that were updated in a specified date range.

Example 2: Finding and importing series with the FRED interface

Suppose we want to import series measuring the real gross domestic product (GDP) in the U.S. and the interbank overnight interest rate controlled by the U.S. Federal Reserve, known as the Federal Funds Rate. We can use a keyword search and a then browse by category to find and select them for import.

After selecting

File > Import > Federal Reserve Economic Data (FRED)
to open the control panel, we type real gross domestic product us in the Keywords field and click on the Search button, which produces
Clicking on GDPC1 and then on the Add button adds GDPC1 to list of series to import.
Now, we want to add the Federal Funds Rate series. We select Browse by category from the drop-down menu in the top left-hand corner.
We double-click on **Money, Banking, & Finance** to get a list of subcategories.
Next, we double-click on Interest Rates to get a list of interest-rate categories. Scrolling down, we find FRB Rates – discount, fed funds, primary credit.
We double-click on FRB Rates - discount, fed funds, primary credit to produce a list of interest-rate series. We click on FEDFUNDS and then on the Add button to add it the list of series to be imported.

Clicking on import brings up a dialog box that allows us to restrict the imported observations.
We click **OK** to import all available observations.

The output from the command issued by the control panel was

```
.import fred GDPC1 FEDFUNDS
```

**Summary**

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDPC1</td>
<td>297</td>
<td>1947-01-01 to 2021-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>FEDFUNDS</td>
<td>801</td>
<td>1954-07-01 to 2021-03-01</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

# of series imported: 2

- highest frequency: Monthly
- lowest frequency: Quarterly

The number of observations and the date ranges will differ when you follow these same steps using the FRED interface, because more data have been made available.
Example 3: Refining a search using tags

Suppose that we want to find and import data on the median income in each U.S. state and the District of Columbia for each available year. After opening the control panel, typing median household income in the Keywords box, and clicking on the Search button, we see...
This keyword search finds thousands more series than the 51 we want. To filter the found series by the tag state, we expand the Geography Types category, click on state, and then click on the Add to filters button, which produces

There are still too many series. To filter the series by the tag real, we expand the Concepts category, click on real, and then click on the Add to filters button, which produces the desired 51 series.
After selecting the 51 series, we add them to the import list by clicking on the Add button. We could now import them by clicking on the Import button.

### Advanced imports using the import fred command

FRED data users commonly import series of different frequencies.

#### Example 4: Importing series with different frequencies

Suppose we wish to import current data on U.S. real GDP, the price level, and the interest rate. These data are stored in FRED with the series IDs “GDPC1”, “GDPDEF”, and “FEDFUNDS”, so we supply those names to import fred.

```
.import fred GDPC1 GDPDEF FEDFUNDS
```

**Summary**

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDPC1</td>
<td>297</td>
<td>1947-01-01 to 2021-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GDPDEF</td>
<td>297</td>
<td>1947-01-01 to 2021-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>FEDFUNDS</td>
<td>801</td>
<td>1954-07-01 to 2021-03-01</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

# of series imported: 3  
highest frequency: Monthly  
lowest frequency: Quarterly

FEDFUNDS is a monthly series, while GDPC1 and GDPDEF are quarterly series. To further illustrate, we list the observations on each variable from 1959 using the list command.
FRED provides all series in daily date format, and each observation is recorded as existing on the first day of the period. For example, a monthly series records the observation in 1959 January as existing on 01Jan1959; a quarterly series records the observation in 1959 Q1 as existing on 01Jan1959. When importing series of different frequencies, the lower-frequency series will appear to contain gaps; these gaps are filled with missing values.

Example 5: Importing series at a desired frequency

Continuing with example 4, at times you may wish to import a high-frequency series at a particular lower frequency. This is accomplished with the `aggregate()` option. There are three aggregation methods available: you may take the within-period average, the sum, or the end-of-period value. The default is to take the within-period average.

```stata
.import fred GDPC1 GDPDEF FEDFUNDS, aggregate(quarterly) clear
```

Summary

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDPC1</td>
<td>297</td>
<td>1947-01-01 to 2021-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GDPDEF</td>
<td>297</td>
<td>1947-01-01 to 2021-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>FEDFUNDS</td>
<td>267</td>
<td>1954-07-01 to 2021-01-01</td>
<td>Quarterly</td>
</tr>
</tbody>
</table>

# of series imported: 3
- highest frequency: Quarterly
- lowest frequency: Quarterly

```stata
.list if year(daten)==1959, separator(4)
```

<table>
<thead>
<tr>
<th>datestr</th>
<th>daten</th>
<th>GDPC1</th>
<th>GDPDEF</th>
<th>FEDFUNDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>01jan1959</td>
<td>3121.936</td>
<td>16.347</td>
<td>2.57</td>
</tr>
<tr>
<td>50</td>
<td>01apr1959</td>
<td>3192.38</td>
<td>16.372</td>
<td>3.08</td>
</tr>
<tr>
<td>51</td>
<td>01jul1959</td>
<td>3194.653</td>
<td>16.435</td>
<td>3.58</td>
</tr>
<tr>
<td>52</td>
<td>01oct1959</td>
<td>3203.759</td>
<td>16.499</td>
<td>3.99</td>
</tr>
</tbody>
</table>
The monthly series FEDFUNDS has been reduced to quarterly frequency. The value of FEDFUNDS for the first quarter of 1959, 2.57, is the average of its values for the three months in that quarter. The date variable daten now stores the first date of each quarter.

Example 6: Importing a subset of observations

The daterange() option causes import fred to restrict importing of data to only observations within the specified beginning and ending dates. daterange() takes two arguments, both of which must be either daily dates or missing (.). If a missing value is used for the first date, then all observations from the beginning up to the end date are imported. If a missing value is used for the second date, then all observations from the first date through the most current are imported.

Returning to example 4, we may wish to import only data between 1984 and 2005 for GDPC1, GDPDEF, and FEDFUNDS.

```
.import fred GDPC1 GDPDEF FEDFUNDS, daterange(1984-01-15 2005-12-31) clear
```

Summary

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDPC1</td>
<td>88</td>
<td>1984-01-01 to 2005-10-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GDPDEF</td>
<td>88</td>
<td>1984-01-01 to 2005-10-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>FEDFUNDS</td>
<td>264</td>
<td>1984-01-01 to 2005-12-01</td>
<td>Monthly</td>
</tr>
</tbody>
</table>

# of series imported: 3

highest frequency: Monthly
lowest frequency: Quarterly

Note that GDPC1 and GDPDEF now have 88 observations rather than 278; similarly, FEDFUNDS has 264 observations rather than 745.

Importing historical vintage data

In example 1, we imported monthly data on the exchange rate between the U.S. Dollar and the Japanese Yen. The observations on EXJPUS listed in that example were observed end-of-day values. In contrast, the values in many FRED series, like the U.S. real gross domestic product series (GDPC1), are estimates. The values of observed series do not change over time. The values of estimated series change over time because the rules that define them change over time. A set of rules is known as a vintage.

FRED contains the most recent vintage of a given series. At times, you may wish to import prior vintages or to view the series as it would have been seen on a particular date in history. ALFRED contains prior vintages of economic data and allows you to import data as they were seen on a particular date in history. For example, you may import the real GDP series that you would have had access to on October 15, 2008.

By default, import fred imports data from the current vintage. The vintage() and realtime() options allow you to import data from prior vintages. You can request a single date, multiple dates, all vintages between two dates in history, or the complete revision history.
Example 7: Importing vintages by date

We wish to import the gross national product (GNP) series as it would have been available on September 16, 2008 and September 16, 2009, so we specify these dates in the `vintage()` option. We also use the `daterange()` option to import only observations since 2006:

```plaintext
> clear
```

**Summary**

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNPC96_20080916</td>
<td>10</td>
<td>2006-01-01 to 2008-04-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090916</td>
<td>14</td>
<td>2006-01-01 to 2009-04-01</td>
<td>Quarterly</td>
</tr>
</tbody>
</table>

# of series imported: 2  
  highest frequency: Quarterly  
  lowest frequency: Quarterly

```plaintext
.list, separator(4) abbreviate(16)
```

<table>
<thead>
<tr>
<th>datestr</th>
<th>daten</th>
<th>GNPC96_20080916</th>
<th>GNPC96_20090916</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006-01-01</td>
<td>01jan2006</td>
<td>11286.5</td>
<td>12994.2</td>
</tr>
<tr>
<td>2006-04-01</td>
<td>01apr2006</td>
<td>11365.1</td>
<td>13035.4</td>
</tr>
<tr>
<td>2006-07-01</td>
<td>01jul2006</td>
<td>11370.8</td>
<td>13025.1</td>
</tr>
<tr>
<td>2006-10-01</td>
<td>01oct2006</td>
<td>11426.5</td>
<td>13129.5</td>
</tr>
<tr>
<td>2007-01-01</td>
<td>01jan2007</td>
<td>11419.1</td>
<td>13160.5</td>
</tr>
<tr>
<td>2007-04-01</td>
<td>01apr2007</td>
<td>11541.7</td>
<td>13275.9</td>
</tr>
<tr>
<td>2007-07-01</td>
<td>01jul2007</td>
<td>11719.9</td>
<td>13451.5</td>
</tr>
<tr>
<td>2007-10-01</td>
<td>01oct2007</td>
<td>11758.3</td>
<td>13563.3</td>
</tr>
<tr>
<td>2008-01-01</td>
<td>01jan2008</td>
<td>11760.9</td>
<td>13525.4</td>
</tr>
<tr>
<td>2008-04-01</td>
<td>01apr2008</td>
<td>11835.9</td>
<td>13533.7</td>
</tr>
<tr>
<td>2008-07-01</td>
<td>01jul2008</td>
<td>.</td>
<td>13470.7</td>
</tr>
<tr>
<td>2008-10-01</td>
<td>01oct2008</td>
<td>.</td>
<td>13240.5</td>
</tr>
<tr>
<td>2009-01-01</td>
<td>01jan2009</td>
<td>.</td>
<td>13018.1</td>
</tr>
<tr>
<td>2009-04-01</td>
<td>01apr2009</td>
<td>.</td>
<td>12991.6</td>
</tr>
</tbody>
</table>

We specified one series and two vintage dates, so we have imported two series. Each vintage is named with the series requested and the date that it was requested. For example, the series GNPC96_20080916 reports real GNP as it was available on 16 September 2008. Note that the series is appended with the date requested, not the date the vintage was released.

These two vintages of GNPC96 differ dramatically because they are on different scales. The output also illustrates that, as of 16 September 2008, data on GNPC96 were only available through 1 April 2008.

Example 8: Importing vintages by real-time period

You may also wish to obtain the complete vintage history of a series between two dates. For example, we import all the vintages of real GNP from December 2007 through July 2010 by specifying this date range in the `realtime()` option.
import fred GNPC96, realtime(2007-12-01 2010-07-31) clear

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNPC96_20071201</td>
<td>243</td>
<td>1947-01-01 to 2007-07-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20071220</td>
<td>243</td>
<td>1947-01-01 to 2007-07-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20080327</td>
<td>244</td>
<td>1947-01-01 to 2007-10-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20080529</td>
<td>245</td>
<td>1947-01-01 to 2008-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20080626</td>
<td>245</td>
<td>1947-01-01 to 2008-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20080731</td>
<td>245</td>
<td>1947-01-01 to 2008-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20080828</td>
<td>246</td>
<td>1947-01-01 to 2008-04-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20080926</td>
<td>246</td>
<td>1947-01-01 to 2008-04-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20081125</td>
<td>247</td>
<td>1947-01-01 to 2008-07-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20081223</td>
<td>247</td>
<td>1947-01-01 to 2008-07-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090326</td>
<td>248</td>
<td>1947-01-01 to 2008-10-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090529</td>
<td>249</td>
<td>1947-01-01 to 2009-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090625</td>
<td>249</td>
<td>1947-01-01 to 2009-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090731</td>
<td>249</td>
<td>1947-01-01 to 2009-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090817</td>
<td>249</td>
<td>1947-01-01 to 2009-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090827</td>
<td>250</td>
<td>1947-01-01 to 2009-04-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20090930</td>
<td>250</td>
<td>1947-01-01 to 2009-04-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20091124</td>
<td>251</td>
<td>1947-01-01 to 2009-07-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20091222</td>
<td>251</td>
<td>1947-01-01 to 2009-07-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20100326</td>
<td>252</td>
<td>1947-01-01 to 2009-10-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20100527</td>
<td>253</td>
<td>1947-01-01 to 2010-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20100625</td>
<td>253</td>
<td>1947-01-01 to 2010-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20100730</td>
<td>253</td>
<td>1947-01-01 to 2010-01-01</td>
<td>Quarterly</td>
</tr>
<tr>
<td>GNPC96_20100731</td>
<td>253</td>
<td>1947-01-01 to 2010-01-01</td>
<td>Quarterly</td>
</tr>
</tbody>
</table>

# of series imported: 24
highest frequency: Quarterly
lowest frequency: Quarterly

Each series contains the data from a vintage, and each series’ name is appended with the date that the vintage was released.

Different vintages of a series may not be directly comparable. For example, the units of a series may change over time. The different vintages must be converted to a common unit before they are analyzed, and it is crucial that you be aware of the units of the vintages you are analyzing.

Note that there is slightly different behavior depending on whether you specify vintage dates or import all vintages within a real-time period. If you specify a list of dates, then each vintage will be named series_date. On the other hand, if you import every vintage between two dates using the realtime() option, then each vintage will be named series_vintage_date. This behavior follows FRED’s behavior when handling vintages.

Searching, saving, and retrieving series information

fredsearch finds series that match keywords or tags. Around 5,000 tags are supplied by FRED. You can also search by keywords, which will search for the keyword anywhere in the metadata of a series.

You can save the names of the series found by a search to a file and then import these series. The following example uses tags in combination with keywords to import median income per capita for states in the United States.
Example 9: Using the search engine

Suppose we wish to import median income per capita for each state. This requires us to identify 51 series, one for each state and the District of Columbia. The series IDs may follow some pattern, but it is not immediately obvious what those IDs are. We could use the FRED interface, as in example 3, or we could use fredsearch to search for the relevant series, save the IDs to a file, and use that file to load the correct series. This example takes the latter approach.

The fredsearch command invokes the search engine. fredsearch keywords allows you to search for keywords anywhere in the series metadata. The tags() option allows you to filter the search results using some of FRED’s 5,000 designated tags.

```
. fredsearch median household income, tags(state real)
```

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Title</th>
<th>Data range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEHOINUSNYA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSDCA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSTX672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSCA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSNC672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSOH672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSCO672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSFL672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMN672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSWA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMD672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSME672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSPA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSLA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSUT672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSAZ672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSCT672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSWA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSKY672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSIL672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSAR672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSIN672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSVA672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSOK672N</td>
<td>Real Median Household Income</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
</tbody>
</table>

(output omitted)

Total: 51

In the above search command, we searched FRED for all series containing “median”, “household”, and “income” somewhere in their metadata, and restricted the search to series with the tags “state” (for states) and “real” (for inflation-adjusted series). The result is 51 series, one for each state and the District of Columbia.
freedsearch provides information about series but does not import them. We can save the search results to a file, then import all series that matched our search results:

```
.freedsearch median household income, tags(state real) saving(myfile.dta)
(51 series added to myfile.dta)
.import fred, serieslist(myfile.dta) clear
```

Summary

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Nobs</th>
<th>Date range</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEHOINUSNYA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSDCA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSTX672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSCAA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMAA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSNCA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSDHA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSCOA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSFLA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMIA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSNJ672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSWI672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMDA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSMNA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSPA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSAL672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
<tr>
<td>MEHOINUSUTA672N</td>
<td>36</td>
<td>1984-01-01 to 2019-01-01</td>
<td>Annual</td>
</tr>
</tbody>
</table>

(output omitted)

# of series imported: 51
highest frequency: Annual
lowest frequency: Annual

This example showed how to quickly import 51 series for median household income by state. A similar procedure can quickly isolate and import the roughly 200 series that report data on infant mortality by country or the roughly 200 series that report the investment share of GDP by country.

Describing series

freddescribe provides facilities to describe series based on their metadata. freddescribe series list provides a brief summary of series list. The series are only described, not imported.

With the detail option, detailed series metadata are displayed, including the full title of the series, the source agency, the source data release, seasonal adjustment, date range for which observations exist, frequency of observations, units, date and time that the series was last updated, and notes, which contain FRED’s notes about the series. Finally, the full metadata includes a list of all vintage dates associated with the series.

Specifying the realtime(start end) option on freddescribe provides information about a series by a real-time period. This option allows you to see how a series’ units have changed over time. freddescribe will display the series description for each vintage between the specified start and end dates.

freddescribe, realtime(. end) describes all vintages from the first available vintage up to that of end. Similarly, freddescribe, realtime(start .) describes all vintages from start up through the most current vintage available.
Example 10: Describing series

Suppose we wish to know what vintages are available for real GDP, whose FRED series name is GDPC1. We use freddescribe with the detail option to list all the vintages.

```
.freddescribe GDPC1, detail
```

<table>
<thead>
<tr>
<th>Title:</th>
<th>Real Gross Domestic Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source:</td>
<td>U.S. Bureau of Economic Analysis</td>
</tr>
<tr>
<td>Release:</td>
<td>Gross Domestic Product</td>
</tr>
<tr>
<td>Seasonal adjustment:</td>
<td>Seasonally Adjusted Annual Rate</td>
</tr>
<tr>
<td>Date range:</td>
<td>1947-01-01 to 2021-01-01</td>
</tr>
<tr>
<td>Frequency:</td>
<td>Quarterly</td>
</tr>
<tr>
<td>Units:</td>
<td>Billions of Chained 2012 Dollars</td>
</tr>
<tr>
<td>Last updated:</td>
<td>2021-04-29 07:52:02-05</td>
</tr>
<tr>
<td>Notes:</td>
<td>BEA Account Code: A191RX Real gross domestic product i...</td>
</tr>
</tbody>
</table>

(output omitted)

Total: 1

Vintages since 1991 are available for download. If we had not specified detail, only the series name, start and end date, and frequency would have been displayed.
Example 11: Obtaining historical descriptions

Information for real GNP in the United States is contained in FRED series GNPC96. Real GNP is expressed in the units of some base year, and over time the base year changes. In this example, we will examine how the units for GNPC96 have changed over time by requesting a description of all vintages up through December 31, 2015 using the realtime() option.

```
.freddescribe GNPC96, realtime(. 2015-12-31)
```

<table>
<thead>
<tr>
<th>Series ID</th>
<th>Real time</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNPC96</td>
<td>1958-12-21 to 1959-02-18</td>
<td>Billions of 1957 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>1959-02-19 to 1965-08-18</td>
<td>Billions of 1954 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>1965-08-19 to 1976-01-15</td>
<td>Billions of 1958 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>1976-01-16 to 1985-12-19</td>
<td>Billions of 1972 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>1985-12-20 to 1991-12-03</td>
<td>Billions of 1982 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>1991-12-04 to 1996-01-18</td>
<td>Billions of 1987 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>1999-10-29 to 2003-12-09</td>
<td>Billions of Chained 1996 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>2003-12-10 to 2009-07-30</td>
<td>Billions of Chained 2000 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>2009-07-31 to 2013-07-30</td>
<td>Billions of Chained 2005 Dollars</td>
</tr>
<tr>
<td>GNPC96</td>
<td>2013-07-31 to 2015-12-31</td>
<td>Billions of Chained 2009 Dollars</td>
</tr>
</tbody>
</table>

Total: 11
Vintages for this series begin in 1958. A new row signifies a change in units. There are 11 total changes in units in GNPC96. Every vintage of GNPC96 between 2009-07-31 and 2013-07-30, for example, is in the units “Billions of chained 2005 dollars”. Meanwhile, vintages since 2013-07-30 are in units “Billions of chained 2009 dollars”. Real GNP vintages from 2010 and 2014 will not be immediately comparable due to the difference in units; they should be converted into a common unit before analysis.

Additional information by real-time period can be obtained by specifying the detail option. We can inspect the details of vintages since 2008:

```
> freddescribe GNPC96, detail realtime(2007-12-31 2013-01-15)
```

```
GNPC96
Title: Real Gross National Product
Source: U.S. Bureau of Economic Analysis
Release: Gross Domestic Product
Seasonal adjustment: Seasonally Adjusted Annual Rate
Date range: 1947-01-01 to 2009-01-01
Frequency: Quarterly
Units: Billions of Chained 2000 Dollars
Last updated: 2009-06-25 10:47:06-05
Notes: BEA Account Code: A001RX1 A Guide to the National Inco...

GNPC96
Title: Real Gross National Product
Source: U.S. Bureau of Economic Analysis
Release: Gross Domestic Product
Seasonal adjustment: Seasonally Adjusted Annual Rate
Date range: 1947-01-01 to 2012-07-01
Frequency: Quarterly
Units: Billions of Chained 2005 Dollars
Last updated: 2012-12-20 08:17:16-06
Notes: BEA Account Code: A001RX1 A Guide to the National Inco...
2010-08-27 2010-09-30 2010-11-23 2010-12-22 2011-03-25
2012-07-27 2012-08-29 2012-09-27 2012-11-29 2012-12-20
```

Total: 2

The detail option provides much of the same information as it did without realtime(), but now a new detail block is provided for each vintage where the details themselves change. Most of the details remain constant across vintages, but in this example, “Units” and “Date range” are different for each block.

The vintage list is now separated, with each vintage falling into the appropriate describe block. For example, all vintages of GNPC96 in 2010 have metainformation corresponding to the block that describes vintages from 2009-07-31 to 2013-01-15.
Stored results

`fredsearch` stores the following in `r()`:

Scalars

- `r(series_ids)` list of series IDs contained in the search results

References


Also see

[D] `import` — Overview of importing data into Stata
[D] `import delimited` — Import and export delimited text data
[D] `import haver` — Import data from Haver Analytics databases
[D] `odbc` — Load, write, or view data from ODBC sources
[TS] `tsset` — Declare data to be time-series data
Title

**import haver — Import data from Haver Analytics databases**

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>Options for import haver</td>
<td>Options for import haver, describe</td>
</tr>
<tr>
<td>Option for set haverdir</td>
<td>Remarks and examples</td>
<td>Stored results</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>Also see</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Haver Analytics ([http://www.haver.com](http://www.haver.com)) provides economic and financial databases to which you can purchase access. The `import haver` command allows you to use those databases with Stata. The `import haver` command is provided only with Stata for Windows.

`import haver seriesdblist` loads data from one or more Haver databases into Stata’s memory.

`import haver seriesdblist, describe` describes the contents of one or more Haver databases. If a database is specified without a suffix, then the suffix `.dat` is assumed.

**Quick start**

Describe available time span, frequency of measurement, and source of series E for net fixed assets and consumer durables from the Haver Analytics CAPSTOCK database

`import haver E@CAPSTOCK, describe`

Load all available observations for quarterly series ASACX and ASAHS from the US1PLUS database

`import haver (ASACX ASAHS)@US1PLUS`

As above, but restrict data to the first quarter of 2000 through the fourth quarter of 2010

`import haver (ASACX ASAHS)@US1PLUS, fin(2000q1,2010q4)`

**Menu**

File > Import > Haver Analytics database
## Syntax

Load Haver data

```plaintext
import haver `seriesdblist` [, `import_haver_options`
```

Load Haver data using a dataset that is loaded in memory

```plaintext
import haver, frommemory `[import_haver_options]
```

Describe contents of Haver database

```plaintext
import haver `seriesdblist`, describe `[import_haver_describe_options]
```

Specify the directory where the Haver databases are stored

```plaintext
set haverdir "path" [, permanently]
```

### `import_haver_options`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fin(datestring, datestring)</code></td>
<td>load data within specified date range</td>
</tr>
<tr>
<td><code>fwithin(datestring, datestring)</code></td>
<td>same as <code>fin</code>() but exclude the end points of range</td>
</tr>
<tr>
<td><code>tvar(varname)</code></td>
<td>create time variable <code>varname</code></td>
</tr>
<tr>
<td>`case(lower</td>
<td>upper)`</td>
</tr>
<tr>
<td><code>hmissing(misval)</code></td>
<td>record missing values as <code>misval</code></td>
</tr>
<tr>
<td>`aggmethod(strict</td>
<td>relaxed</td>
</tr>
</tbody>
</table>

### `frommemory`

load data using file in memory

### `clear`

clear data in memory before loading Haver database

`frommemory` and `clear` do not appear in the dialog box.

### `import_haver_describe_options`

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*describe</td>
<td>describe contents of <code>seriesdblist</code></td>
</tr>
<tr>
<td>detail</td>
<td>list full series information table for each series</td>
</tr>
<tr>
<td><code>saving(filename [, verbose replace])</code></td>
<td>save series information to <code>filename.dta</code></td>
</tr>
</tbody>
</table>

*`describe` is required.

collect is allowed with `import haver`; see [U] 11.1.10 Prefix commands.

`seriesdblist` is one or more of the following:

- `dbfile`
- `series@dbfile`
- `(series series ...)@dbfile`
where `dbfile` is the name of a Haver Analytics database and `series` contains a Haver Analytics series. Wildcards `?` and `*` are allowed in `series`. `series` and `dbfile` are not case sensitive.

Example: `import haver gdp@usecon`
Import series GDP from the USECON database.

Example: `import haver gdp@usecon c1*@ifs`
Import series GDP from the USECON database, and import any series that starts with `c1` from the IFS database.

Note: You must specify a path to the database if you did not use the `set haverdir` command.

Example: `import haver gdp"C:\data\usecon" c1*"C:\data\ifs"
If you do not specify a path to the database and you did not set `haverdir`, then `import haver` will look in the current working directory for the database.

### Options for import haver

`fin([datestring], [datestring])` specifies the date range of the data to be loaded. `datestring` must adhere to the Stata default for the different frequencies. See [D] Datetime display formats. Examples are `23mar2012` (daily and weekly), `2000m1` (monthly), `2003q4` (quarterly), and `1998` (annual). `fin(1jan1999, 31dec1999)` would mean from and including 1 January 1999 through 31 December 1999. Note that weekly data must be specified as daily data because Haver-week data are conceptually different than Stata-week data.

`fin()` also determines the aggregation frequency. If you want to retrieve data in a frequency that is lower than the one in which the data are stored, specify the dates in option `fin()` accordingly. For example, to retrieve series that are stored in quarterly frequency into an annual dataset, you can type `fin(1980,2010)`.

`fwithin([datestring], [datestring])` functions the same as `fin()` except that the endpoints of the range will be excluded in the loaded data.

`tvar(varname)` specifies the name of the time variable Stata will create. The default is `tvar(time)`. The `tvar()` variable is the name of the variable that you would use to `tsset` the data after loading, although doing so is unnecessary because `import haver` automatically `tssets` the data for you.

`case(lower|upper)` specifies the case of the variable names after import. The default is `case(lower)`.

`hmissing(misval)` specifies which of Stata’s 27 missing values (., .a, ., .z) to record when there are missing values in the Haver database.

Two kinds of missing values occur in Haver databases. The first occurs when nothing is recorded because the data do not span the entire range; these missing values are always stored as . by Stata. The second occurs when Haver has recorded a Haver missing value; by default, these are stored as . by Stata, but you can use `hmissing()` to specify that a different extended missing-value code be used.

`aggmethod(strict|relaxed|force)` specifies a method of temporal aggregation in the presence of missing observations. `aggmethod(strict)` is the default aggregation method.

Most Haver series of higher than annual frequency has an aggregation type that determines how data can be aggregated. The three aggregation types are average (AVG), sum (SUM), and end of period (EOP). Each aggregation method behaves differently for each aggregation type.
An aggregated span is a time period expressed in the original frequency. The goal is to aggregate the data in an aggregated span to a single observation in the (lower) target frequency. For example, 1973m1–1973m3 is an aggregated span for quarterly aggregation to 1973q1.

strict aggregation method:

1) (Average) The aggregated value is the average value if no observation in the aggregated span is missing; otherwise, the aggregated value is missing.

2) (Sum) The aggregated value is the sum if no observation in the aggregated span is missing; otherwise, the aggregated value is missing.

3) (End of period) The aggregated value is the series value in the last period in the aggregated span, be it missing or not.

relaxed aggregation method:

1) (Average) The aggregated value is the average value as long as there is one nonmissing data point in the aggregated span; otherwise, the aggregated value is missing.

2) (Sum) The aggregated value is the sum if no observation in the aggregated span is missing; otherwise, the aggregated value is missing.

3) (End of period) The aggregated value is the last available nonmissing data point in the aggregated span; otherwise, the aggregated value is missing.

force aggregation method:

1) (Average) The aggregated value is the average value as long as there is one nonmissing data point in the aggregated span; otherwise, the aggregated value is missing.

2) (Sum) The aggregated value is the sum if there is at least one nonmissing data point in the aggregated span; otherwise, the aggregated value is missing.

3) (End of period) The aggregated value is the last available nonmissing data point in the aggregated span; otherwise, the aggregated value is missing.

The following options are available with import haver but are not shown in the dialog box:

frommemory specifies that each observation of the dataset in memory specifies the information for a Haver series to be imported. The dataset in memory must contain variables named path, file, and series. The observations in path specify paths to Haver databases, the observations in file specify Haver databases, and the observations in series specify the series to import.

clear clears the data in memory before loading the Haver database.

Options for import haver, describe

describe describes the contents of one or more Haver databases.

detail specifies that a detailed report of all the information available on the variables be presented.

saving(filename [, verbose replace]) saves the series meta-information to a Stata dataset. By default, the series meta-information is not displayed to the Results window, but you can use the verbose option to display it.

saving() saves a Stata dataset that can subsequently be used with the frommemory option.
Option for set haverdir

permanently specifies that in addition to making the change right now, the haverdir setting be remembered and become the default setting when you invoke Stata.

Remarks and examples

Remarks are presented under the following headings:

- Installation
- Setting the path to Haver databases
- Download example Haver databases
- Determining the contents of a Haver database
- Loading a Haver database
- Loading a Haver database from a describe file
- Temporal aggregation
- Daily data
- Weekly data

Installation

Haver Analytics (http://www.haver.com) provides more than 200 economic and financial databases in the form of .dat files to which you can purchase access. The `import haver` command provides easy access to those databases from Stata. `import haver` is provided only with Stata for Windows.

Setting the path to Haver databases

If you want to retrieve data from Haver Analytics databases, you must discover the directory in which the databases are stored. This will most likely be a network location. If you do not know the directory, contact your technical support staff or Haver Analytics (http://www.haver.com). Once you have determined the directory location—for example, H:\haver_files—you can save it by using the command

```
    . set haverdir "H:\haver_files", permanently
```

Using the `permanently` option will preserve the Haver directory information between Stata sessions. Once the Haver directory is set, you can start retrieving data. For example, if you are subscribing to the USECON database, you can type

```
    . import haver gdp@usecon
```

to load the GDP series into Stata. If you did not use `set haverdir`, you would type

```
    . import haver gdp@"H:\haver_files\usecon"
```

The directory path passed to `set haverdir` is saved in the `creturn` value `c(haverdir)`. You can view it by typing

```
    . display "'c(haverdir)'
```
Download example Haver databases

There are three example Haver databases you can download to your working directory. Run the copy commands below to download HAVERD, HAVERW, and HAVERMQA.

```bash
. copy https://www.stata.com/haver/HAVERD.DAT haverd.dat
. copy https://www.stata.com/haver/HAVERD.IDX haverd.idx
. copy https://www.stata.com/haver/HAVERW.DAT havew.dat
. copy https://www.stata.com/haver/HAVERW.IDX havew.idx
. copy https://www.stata.com/haver/HAVERMQA.DAT havermqa.dat
. copy https://www.stata.com/haver/HAVERMQA.IDX havermqa.idx
```

To use these files, you need to make sure your Haver directory is not set:

```bash
. set haverdir ""
```

Determining the contents of a Haver database

import haver `seriesdblist`, describe displays the contents of a Haver database. If no series is specified, then all series are described.

```bash
. import haver haverd, describe
Dataset: haverd
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Time span</th>
<th>Frequency</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>FXTWB</td>
<td>Nominal Broad Trade-Weighted Exchange Value of the US$ (1/97=100)</td>
<td>03jan2005-02mar2012</td>
<td>Daily</td>
<td>FRB</td>
</tr>
<tr>
<td>FXTWM</td>
<td>Nominal Trade-Weighted Exchange Value of the US$ (1/97=100)</td>
<td>03jan2005-02mar2012</td>
<td>Daily</td>
<td>FRB</td>
</tr>
<tr>
<td>FXTWOTP</td>
<td>Nominal Trade-Weighted Exchange Value of the US$ (1/97=100)</td>
<td>03jan2005-02mar2012</td>
<td>Daily</td>
<td>FRB</td>
</tr>
</tbody>
</table>

Summary

- number of series described: 3
- series not found: 0

Above we describe the Haver database `haverd.dat`, which we already have on our computer and in our current directory.

By default, each line of the output corresponds to one Haver series. Specifying `detail` displays more information about each series, and specifying `seriesname@` allows us to restrict the output to the series that interests us:

```bash
. import haver FXTWB@haverd, describe detail
```

<table>
<thead>
<tr>
<th>FXTWB</th>
<th>Nominal Broad Trade-Weighted Exchange Value of the US$ (1/97=100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency: <strong>Daily</strong></td>
<td>Time span: 03Jan2005-02Mar2012</td>
</tr>
<tr>
<td>Number of Observations: <strong>1870</strong></td>
<td>Date Modified: 07Mar2012 11:27:33</td>
</tr>
<tr>
<td>Aggregation Type: <strong>AVG</strong></td>
<td>Decimal Precision: <strong>4</strong></td>
</tr>
<tr>
<td>Difference Type: <strong>0</strong></td>
<td>Magnitude: <strong>0</strong></td>
</tr>
<tr>
<td>Data Type: <strong>INDEX</strong></td>
<td>Group: <strong>R03</strong></td>
</tr>
<tr>
<td>Primary Geography Code: <strong>111</strong></td>
<td>Secondary Geography Code:</td>
</tr>
<tr>
<td>Source: <strong>FRB</strong></td>
<td>Source Description: <strong>Federal Reserve..</strong></td>
</tr>
</tbody>
</table>

Summary

- number of series described: 1
- series not found: 0
You can describe multiple Haver databases with one command:

```
.import haver haverd haverw, describe
(output omitted)
```

To restrict the output to the series that interest us for each database, you could type

```
.import haver (FXTWB FXTWOTP)@haverd FARVSN@haverw, describe
(output omitted)
```

### Loading a Haver database

`import haver seriesdblist` loads Haver databases. If no series is specified, then all series are loaded.

```
.import haver haverd, clear
```

Summary

```
Haver data retrieval: 10 Dec 2020 11:41:18
# of series requested: 3
# of database(s) used: 1 (HAVERD)
All series have been successfully retrieved
```

**Frequency**

```
highest Haver frequency: Daily
lowest Haver frequency: Daily
frequency of Stata dataset: Daily
```

The table produced by `import haver seriesdblist` displays a summary of the loaded data, frequency information about the loaded data, series that could not be loaded because of errors, and notes about the data.

The dataset now contains a time variable and three variables retrieved from the HAVERD database:

```
.describe
```

Contains data

```
Observations: 1,870
Variables: 4
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>double</td>
<td>%td</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fxtwb_haverd</td>
<td>double</td>
<td>%10.0g</td>
<td>Nominal Broad Trade-Weighted Exchange Value of the US$ (1/97=100)</td>
<td></td>
</tr>
<tr>
<td>fxtwm_haverd</td>
<td>double</td>
<td>%10.0g</td>
<td>Nominal Trade-Weighted Exchange Value of US$ vs Major Currencies (3/73=100)</td>
<td></td>
</tr>
<tr>
<td>fxtwotp_haverd</td>
<td>double</td>
<td>%10.0g</td>
<td>Nominal Trade-Weighted Exchange Value of US$ vs OITP (1/97=100)</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: time

Note: Dataset has changed since last saved.
Haver databases include the following meta-information about each variable:

- **HaverDB**: database name
- **Series**: series name
- **DateTimeMod**: date/time the series was last modified
- **Frequency**: frequency of series (from daily to annual) as it is stored in the Haver database
- **Magnitude**: magnitude of the data
- **DecPrecision**: number of decimals to which the variable is recorded
- **DifType**: relevant within Haver software only: if =1, percentage calculations are not allowed
- **AggType**: temporal aggregation type (one of AVG, SUM, EOP)
- **DataType**: type of data (e.g., ratio, index, US$, %)
- **Group**: Haver series group to which the variable belongs
- **Geography1**: primary geography code
- **Geography2**: secondary geography code
- **StartDate**: data start date
- **EndDate**: data end date
- **Source**: Haver code associated with the source for the data
- **SourceDescription**: description of Haver code associated with the source for the data

When a variable is loaded, this meta-information is stored in variable characteristics (see [P] char). Those characteristics can be viewed using `char list`:

```
. char list fxtwb_haverd[]
  fxtwb_haverd[HaverDB]: HAVERD
  fxtwb_haverd[Series]: FXTWB
  fxtwb_haverd[DateTimeMod]: 07mar2012 11:27:33
  fxtwb_haverd[Frequency]: Daily
  fxtwb_haverd[Magnitude]: 0
  fxtwb_haverd[DecPrecision]: 4
  fxtwb_haverd[DifType]: 0
  fxtwb_haverd[AggType]: AVG
  fxtwb_haverd[DataType]: INDEX
  fxtwb_haverd[Group]: R03
  fxtwb_haverd[Geography1]: 111
  fxtwb_haverd[StartDate]: 03jan2005
  fxtwb_haverd[EndDate]: 02mar2012
  fxtwb_haverd[Source]: FRB
  fxtwb_haverd[SourceDescription]:
      Federal Reserve Board
```

You can load multiple Haver databases/series with one command. To load the series FXTWB and FXTWOTP from the HAVERD database and all series that start with V from the HAVERMQA database, you would type

```
. import haver (FXTWB FXTWOTP)@haverd V*@havermqa, clear
```

import haver automatically `tsset` the data for you.

**Loading a Haver database from a describe file**

You often need to search through the series information of a Haver database(s) to see which series you would like to load. You can do this by saving the output of `import haver, describe` to a Stata dataset with the `saving(filename)` option. The dataset created can be used by `import haver, frommemory` to load data from the described Haver database(s). For example, here we search through the series information of database HAVERMQA.
The resulting dataset contains information on the 164 series in HAVERMQA. Suppose that we want to retrieve all monthly series whose description includes the word “Yield”. We need to keep only the observations from our dataset where the frequency variable equals “Monthly” and where the description variable contains “Yield”.

```
. keep if frequency=="Monthly" & strpos(description,"Yield")
   (152 observations deleted)
```

To load the selected series into Stata, we type

```
. import haver, frommemory clear
```

Note: We must clear the described data in memory to load the selected series. If you do not want to lose the changes you made to the description dataset, you must save it before using import haver, frommemory.

### Temporal aggregation

If you request series with different frequencies, the higher frequency data will be aggregated to the lowest frequency. For example, if you request a monthly and a quarterly series, the monthly series will be aggregated. In rare cases, a series cannot be aggregated to a lower frequency and so will not be retrieved. A list of these series will be stored in r(noaggtype).

The options fin() and fwithin() are useful for aggregating series by hand.

### Daily data

Haver’s daily frequency corresponds to Stata’s daily frequency. Haver’s daily data series are business series for which business calendars are useful. See [D] Datetime business calendars for more information on business calendars.
Weekly data

Haver’s weekly data are also retrieved to Stata’s daily frequency. See [D] Datetime business calendars for more information on business calendars.

Stored results

import haver stores the following in r():

Scalars
- r(k_requested): number of series requested
- r(k_noaggtype): number of series dropped because of invalid aggregation type
- r(k_nodisagg): number of series dropped because their frequency is lower than that of the output dataset
- r(k_notindata): number of series dropped because data were out of the date range specified in fwithin() or fin()
- r(k_notfound): number of series not found in the database

Macros
- r(noaggtype): list of series dropped because of invalid aggregation type
- r(nodisagg): list of series dropped because their frequency is lower than that of the output dataset
- r(notindata): list of series dropped because data were out of the date range specified in fwithin() or fin()
- r(notfound): list of series not found in the database

import haver, describe stores the following in r():

Scalars
- r(k_described): number of series described
- r(k_notfound): number of series not found in the database

Macros
- r(notfound): list of series not found in the database

Acknowledgment

import haver was written with the help of Daniel C. Schneider of the Max Planck Institute for Demographic Research, Rostock, Germany.

Also see

[D] import — Overview of importing data into Stata
[D] import delimited — Import and export delimited text data
[D] import fred — Import data from Federal Reserve Economic Data
[D] odbc — Load, write, or view data from ODBC sources
[TS] tsset — Declare data to be time-series data
**import sas — Import SAS files**

**Description**

`import sas` reads into memory a version 7 or higher SAS (.sas7bdat) file. It can also import SAS value labels from a .sas7bcat file. `import sas` can import up to 32,766 variables at one time (up to 2,048 variables in Stata/BE). If your SAS file contains more variables than this, you can break up the SAS file into multiple Stata datasets. You can also import SAS value labels from a .sas7bcat file.

**Quick start**

Import SAS file `myfile.sas7bdat` into Stata

```stata
import sas myfile
```

As above, but replace the data in memory

```stata
import sas myfile, clear
```

As above, but only import variables `x1` and `x2`

```stata
import sas x1 x2 using myfile, clear
```

Import data from SAS file `myfile` and value labels from file `labels.sas7bcat`

```stata
import sas myfile, bcat(labels)
```

**Menu**

File > Import > SAS data (*.sas7bdat)
Syntax

Load a SAS file (*.sas7bdat)

    import sas [using] filename [, options]

Load a subset of a SAS file (*.sas7bdat)

    import sas [namelist] [if] [in] using filename [, options]

If filename is specified without an extension, .sas7bdat is assumed. If filename contains embedded spaces, enclose it in double quotes.

namelist specifies SAS variable names to be imported.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcat(filename_vl)</td>
<td>load value labels defined in filename_vl into memory</td>
</tr>
<tr>
<td>case(lower</td>
<td>upper</td>
</tr>
<tr>
<td>clear</td>
<td>replace data in memory</td>
</tr>
<tr>
<td>encoding(&quot;encoding&quot;)</td>
<td>specify the file encoding; see help encodings</td>
</tr>
</tbody>
</table>

Options

bcat(filename_vl) specifies that the value labels defined in filename_vl be loaded into memory along with the dataset. If filename_vl is specified without an extension, .sas7bcat is assumed. If filename_vl contains embedded spaces, enclose it in double quotes.

SAS does not assign value labels to variables; therefore, you must use the label values command to assign the value labels to specific variables after importing them.

case(lower | upper | preserve) specifies the case of the variable names after import. The default is case(preserve).

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

The following option is available with import sas but is not shown in the dialog box:

encoding("encoding") specifies the encoding of the file. If your file has an incorrect encoding specified in the file header, you can use this option to specify the correct encoding. See help encodings for details.

Remarks and examples

import sas reads into memory version 7 or higher SAS (.sas7bdat) files. If a SAS variable name from the file does not conform to a Stata variable name, a generic v# name will be assigned, and the original variable name will be stored as a characteristic for the variable. If a SAS variable label is too long, it will be truncated to 80 characters. The original variable label will be stored as a variable characteristic. If a SAS data label is too long, it will be truncated to 80 characters, and the original label will be stored as a data characteristic.
Example 1: Importing a SAS file into Stata

We can import SAS files into Stata, either by selecting the entire file or by selecting subsets of the data, with `import sas`. For example, we have the SAS file `auto.sas7bdat`, which contains data on automobiles, and we have value labels for these data stored in `formats.sas7bcat`. Below, we demonstrate how to import these data into Stata. To follow along, download these files to your working directory by typing the `copy` commands below:

```
. copy https://www.stata.com/sampledata/auto.sas7bdat auto.sas7bdat
. copy https://www.stata.com/sampledata/formats.sas7bcat formats.sas7bcat
```

To load the file `auto.sas7bdat` into Stata’s memory, we type

```
. import sas auto.sas7bdat
```

(12 vars, 74 obs)

We can instead import only the variables `make`, `weight`, and `foreign` from `auto.sas7bdat`. We use the `bcat()` option to add the value labels defined in the `formats.sas7bcat` file and the `clear` option to replace the data in memory without saving them.

```
. import sas make weight foreign using auto, bcat(formats) clear
```

(3 vars, 74 obs)

```
. list in 1/5
make    weight    foreign
1.      AMC Concord  2930        0
2.      AMC Pacer    3350        0
3.      AMC Spirit    2640        0
4.    Buick Century  3250        0
5.     Buick Electra  4080        0
```

We list the value labels that we imported using `label list`

```
. label list
ORIGIN:
   0 Domestic
   1 Foreign
```

ORIGIN contains value labels for the variable `foreign`. We need to use the `label values` command to apply this label to `foreign`. Then, we `save` the data with these labels attached.

```
. label values foreign ORIGIN
```

```
. list in 1/5
make    weight    foreign
1.      AMC Concord  2930 Domestic
2.      AMC Pacer    3350 Domestic
3.      AMC Spirit    2640 Domestic
4.    Buick Century  3250 Domestic
5.     Buick Electra  4080 Domestic
```

```
. save myauto
file myauto.dta saved
```
import sas — Import SAS files

**Stored results**

import sas stores the following in `r()`:

Scalars

- `r(N)` number of observations imported
- `r(k)` number of variables imported

**Also see**

[D] import sasxport5 — Import and export data in SAS XPORT Version 5 format

[D] import sasxport8 — Import and export data in SAS XPORT Version 8 format

[D] import — Overview of importing data into Stata
**Description**

`import sasxport5` and `export sasxport5` convert data from and to SAS XPORT Version 5 Transport format. The U.S. Food and Drug Administration uses this SAS XPORT Transport format as the format for datasets submitted with new drug and new device applications (NDAs).

`export sasxport5` saves the data in memory as a SAS XPORT Transport (.xpt) file. If needed, this command also creates `formats.xpf`—an additional XPORT file—containing the value-label definitions. These files can be easily read into SAS.

`import sasxport5` reads into memory data from a SAS XPORT Transport (.xpt) file. When available, this command also reads the value-label definitions stored in `formats.xpf` or `FORMATS.xpf`.

`import sasxport5`, `describe` describes the contents of a SAS XPORT Version 5 Transport file.

**Quick start**

Describe the contents of SAS XPORT Version 5 Transport file `mydata.xpt`

```
import sasxport5 mydata, describe
```

Load the contents of `mydata.xpt` into memory

```
import sasxport5 mydata
```

As above, and ignore the accompanying SAS formats file `formats.xpf`

```
import sasxport5 mydata, novallabels
```

Save data in memory to `mydata.xpt`

```
export sasxport5 mydata
```

As above, but rename variables to meet SAS XPORT restrictions

```
export sasxport5 mydata, rename
```

As above, and do not save value labels

```
export sasxport5 mydata, rename replace vallabfile(none)
```

Save `v1`, `v2`, and `v3` to `mydata.xpt`, where time variable `tvar` is equal to 2010

```
export sasxport5 v1 v2 v3 using mydata if tvar==2010
```
Menu

**import sasxport5**

File > Import > SAS XPORT Version 5 (*.xpt)

**export sasxport5**

File > Export > SAS XPORT Version 5 (*.xpt)

Syntax

_Import SAS XPORT Version 5 Transport file into Stata_

\[
\text{import sasxport5 } \text{filename} \ [, \text{import}\_options]
\]

_Describe contents of SAS XPORT Version 5 Transport file_

\[
\text{import sasxport5 } \text{filename}, \text{describe} \ [\text{member}(\text{mbrname})]
\]

_Export data in memory to a SAS XPORT Version 5 Transport file_

\[
\text{export sasxport5 } \text{filename} \ [\text{if}] \ [\text{in}] \ [, \text{export}\_options]
\]

\[
\text{export sasxport5 } \text{varlist using} \text{filename} \ [\text{if}] \ [\text{in}] \ [, \text{export}\_options]
\]

If \text{filename} is specified without an extension, .xpt is assumed. If \text{filename} contains embedded spaces, enclose it in double quotes.

<table>
<thead>
<tr>
<th>\text{import}_options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>replace data in memory</td>
</tr>
<tr>
<td>novallabels</td>
<td>ignore accompanying \text{formats.xpf} file if it exists</td>
</tr>
<tr>
<td>\text{member}(\text{mbrname})</td>
<td>member to use; seldom used</td>
</tr>
</tbody>
</table>

collect is allowed with import sasxport5; see [U] 11.1.10 Prefix commands.

<table>
<thead>
<tr>
<th>\text{export}_options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename</td>
<td>rename variables and value labels to meet SAS XPORT restrictions</td>
</tr>
<tr>
<td>replace</td>
<td>overwrite files if they already exist</td>
</tr>
<tr>
<td>\text{valabfile}(\text{xpf})</td>
<td>save value labels in \text{formats.xpf}</td>
</tr>
<tr>
<td>\text{valabfile}(\text{sascode})</td>
<td>save value labels in \text{SAS command file}</td>
</tr>
<tr>
<td>\text{valabfile}(both)</td>
<td>save value labels in \text{formats.xpf} and in a \text{SAS command file}</td>
</tr>
<tr>
<td>\text{valabfile}(none)</td>
<td>do not save value labels</td>
</tr>
</tbody>
</table>
Options for import sasxport5

`describe` describes the contents of the SAS XPORT Version 5 Transport file. This option can be combined only with `member()`.

`clear` specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

`novallabels` specifies that value-label definitions stored in `formats.xpf` or `FORMATS.xpf` not be looked for or loaded. By default, if variables are labeled in `filename.xpt`, then `import sasxport5` looks for `formats.xpf` to obtain and load the value-label definitions. If the file is not found, Stata looks for `FORMATS.xpf`. If that file is not found, a warning message is issued.

`import sasxport5` can use only a `formats.xpf` or `FORMATS.xpf` file to obtain value-label definitions. `import sasxport5` cannot understand value-label definitions from a SAS command file.

`member(mbrname)` specifies a member of the `.xpt` file. Although no longer often used, the original XPORT definition allowed multiple datasets to be placed in one file. The `member()` option allows you to read these old files, selecting only specific datasets (members) to be used by `import sasxport5`. You can obtain a list of member names by using `import sasxport5, describe`. By default, only the first member is used, unless `describe` is specified, in which case all members are described. Because it is rare for an XPORT file to have more than one member, this option is seldom used.

Options for export sasxport5

`rename` specifies that `export sasxport5` may rename variables and value labels to attempt to meet the SAS XPORT restrictions, which are that names be no more than eight bytes long and that there be no distinction between uppercase and lowercase letters. Note that `rename` does not remove characters beyond the normal ASCII range, such as most Unicode characters and all extended ASCII characters. SAS may or may not support such characters in variable labels and value labels.

We recommend specifying the `rename` option. If this option is specified, any name violating the restrictions is changed to a different but related name in the file. The name changes are listed. The new names are used only in the file; the names of the variables and value labels in memory remain unchanged.

If `rename` is not specified and one or more names violate the XPORT restrictions, an error message will be issued and no file will be saved. The alternative to the `rename` option is that you can rename variables yourself with the `rename` command:

```
.rename mylongvariablename myname
```

See [D] rename. Renaming value labels yourself is more difficult. The easiest way to rename value labels is to use `label save`, edit the resulting file to change the name, execute the file by using `do`, and reassign the new value label to the appropriate variables by using `label values`:

```
.label save mylongvaluelabel using myfile.do
.doedit myfile.do (change mylongvaluelabel to, say, mlvlab)
do myfile.do
.label values myvar mlvlab
```

See [D] label and [R] do for more information about renaming value labels.
import sasxport5 — Import and export data in SAS XPORT Version 5 format

replace permits export sasxport5 to overwrite existing filename.xpt, formats.xpf, and filename.sas files.

vallabfile(xpf | sascode | both | none) specifies whether and how value labels are to be stored. SAS XPORT Transport files do not really have value labels. Value-label definitions can be preserved in one of two ways:

1. In an additional SAS XPORT Version 5 Transport file whose data contain the value-label definitions
2. In a SAS command file that will create the value labels

export sasxport5 can create either or both of these files.

vallabfile(xpf), the default, specifies that value labels be written into a separate SAS XPORT Transport file named formats.xpf. Thus, export sasxport5 creates two files: filename.xpt, containing the data, and formats.xpf, containing the value labels. No formats.xpf file is created if there are no value labels.

SAS XPORT Transport files do not really have value labels. Value-label definitions can be preserved in one of two ways:

1. In an additional SAS XPORT Version 5 Transport file whose data contain the value-label definitions
2. In a SAS command file that will create the value labels

export sasxport5 can create either or both of these files.

vallabfile(xpf), the default, specifies that value labels be written into a separate SAS XPORT Transport file named formats.xpf. Thus, export sasxport5 creates two files: filename.xpt, containing the data, and formats.xpf, containing the value labels. No formats.xpf file is created if there are no value labels.

SAS users can easily use the resulting .xpt and .xpf XPORT files. See https://www.sas.com/govedu/fda/macro.html, and click on the FDA Submission Standards tab. Then, click on the Processing Data Sets Code tab that appears below the “FDA and SAS Technology” text for SAS-provided macros for reading the XPORT files. The SAS macro fromexp() reads the XPORT files into SAS. The SAS macro toexp() creates XPORT files. When obtaining the macros, remember to save the macros at SAS’s webpage as a plain-text file and to remove the examples at the bottom.

If the SAS macro file is saved as C:\project\macros.mac and the files mydat.xpt and formats.xpf created by export sasxport5 are in C:\project, the following SAS commands would create the corresponding SAS dataset and format library and list the data:

```sas
%include "C:\project\macros.mac";
%fromexp(C:\project, C:\project);
libname library 'C:\project';
data _null_; set library.mydat; put _all_; run;
proc print data = library.mydat;
quit;
```

vallabfile(sascode) specifies that the value labels be written into a SAS command file, filename.sas, containing SAS proc format and related commands. Thus, export sasxport5 creates two files: filename.xpt, containing the data, and filename.sas, containing the value labels. SAS users may wish to edit the resulting filename.sas file to change the “libname datapath” and “libname xptfile xport” lines at the top to correspond to the location that they desire. export sasxport5 sets the location to the current working directory at the time export sasxport5 was issued. No .sas file will be created if there are no value labels.

vallabfile(both) specifies that both the actions described above be taken and that three files be created: filename.xpt, containing the data; formats.xpf, containing the value labels in XPORT format; and filename.sas, containing the value labels in SAS command-file format.

vallabfile(none) specifies that value-label definitions not be saved. Only one file is created: filename.xpt, which contains the data.
Remarks and examples

All users, of course, may use these commands to transfer data between SAS and Stata, but there are limitations in the SAS XPORT Transport format, such as the eight-character limit on the names of variables (specifying `export sasxport5`’s `rename` option works around that). For a complete listing of limitations and issues concerning the SAS XPORT Transport format and an explanation of how `export sasxport5` and `import sasxport5` work around these limitations, see *Technical appendix* below.

Remarks are presented under the following headings:

- Saving XPORT files for transferring to SAS
- Determining the contents of XPORT files received from SAS
- Using XPORT files received from SAS

Saving XPORT files for transferring to SAS

Example 1: Exporting data to XPORT files

To demonstrate, we first load `auto.dta`. To save only variables `make`, `mpg`, and `weight` in `auto_sub.xpt`, we type

```
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. export sasxport5 make mpg weight using auto_sub
  file auto_sub.xpt saved
```

We can save all the variables in the data to `auto.xpt` and save the value labels in `formats.xpf`. We specify the `rename` option to rename variable names and value labels that are too long or are case sensitive.

```
. export sasxport5 auto, rename
  the following variable(s) were renamed in the output file:
    displacement -> DISPLACE
    gear_ratio -> GEAR_RAT
  file auto.xpt saved
  file formats.xpf saved
```

Alternatively, we can save the data in `auto.xpt` and save the value labels to a `formats.xpf` file and in a SAS command file `auto.sas`. We include the `replace` option to allow replacement of the files we created with our previous command.

```
. export sasxport5 auto, rename replace vallabfile(both)
  the following variable(s) were renamed in the output file:
    displacement -> DISPLACE
    gear_ratio -> GEAR_RAT
  file auto.xpt saved
  file auto.sas saved
  file formats.xpf saved
```

If we instead wanted to save the value labels only in the SAS command file, we could have typed

```
. export sasxport5 auto, rename replace vallabfile(sas)
```

If we did not want to save the value labels at all, thus creating only `auto.xpt`, we could have typed

```
. export sasxport5 typed, rename replace vallabfile(none)
```
Determining the contents of XPORT files received from SAS

Example 2: Describing XPORT files

To investigate the contents of the `auto.xpt` file we created above, we can type

```
.import sasxport5 auto, describe
data from auto.xpt, member(auto)
```

<table>
<thead>
<tr>
<th>variable name</th>
<th>variable type</th>
<th>variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str18</td>
<td>Make and model</td>
</tr>
<tr>
<td>price</td>
<td>numeric</td>
<td>Price</td>
</tr>
<tr>
<td>mpg</td>
<td>numeric</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>rep78</td>
<td>numeric</td>
<td>Repair record 1978</td>
</tr>
<tr>
<td>headroom</td>
<td>numeric</td>
<td>Headroom (in.)</td>
</tr>
<tr>
<td>trunk</td>
<td>numeric</td>
<td>Trunk space (cu. ft.)</td>
</tr>
<tr>
<td>weight</td>
<td>numeric</td>
<td>Weight (lbs.)</td>
</tr>
<tr>
<td>length</td>
<td>numeric</td>
<td>Length (in.)</td>
</tr>
<tr>
<td>turn</td>
<td>numeric</td>
<td>Turn circle (ft.)</td>
</tr>
<tr>
<td>displace</td>
<td>numeric</td>
<td>Displacement (cu. in.)</td>
</tr>
<tr>
<td>gear_rat</td>
<td>numeric</td>
<td>Gear ratio</td>
</tr>
<tr>
<td>foreign</td>
<td>numeric</td>
<td>origin</td>
</tr>
</tbody>
</table>

Using XPORT files received from SAS

Example 3: Importing XPORT files

To read data from `auto.xpt` and obtain value labels from `formats.xpf`, we can type

```
.import sasxport5 auto, clear
```

Stored results

`import sasxport5, describe` stores the following in `r()`:

Scalars
- `r(N)` number of observations
- `r(k)` number of variables
- `r(size)` size of data
- `r(n_members)` number of members

Macros
- `r(members)` names of members
Technical appendix

Technical details concerning the SAS XPORT Version 5 Transport format and how `export sasxport5` and `import sasxport5` handle issues regarding the format are presented under the following headings:

A1. Overview of SAS XPORT Transport format
A2. Implications for writing XPORT datasets from Stata
A3. Implications for reading XPORT datasets into Stata

A1. Overview of SAS XPORT Transport format

A SAS XPORT Transport file may contain one or more separate datasets, known as members. It is rare for a SAS XPORT Transport file to contain more than one member. See [https://support.sas.com/techsup/technote/ts140.pdf](https://support.sas.com/techsup/technote/ts140.pdf) for the SAS technical document describing the layout of the SAS XPORT Transport file.

A SAS XPORT dataset (member) is subject to certain restrictions:

1. The dataset may contain only 9,999 variables.
2. The names of the variables and value labels may not be longer than eight characters and are case insensitive; for example, `myvar`, `Myvar`, `MyVar`, and `MYVAR` are all the same name.
3. Variable labels may not be longer than 40 characters.
4. The contents of a variable may be numeric or string:
   a. Numeric variables may be integer or floating but may not be smaller than $5.398 \times 10^{-79}$ or greater than $9.046 \times 10^{74}$, absolutely. Numeric variables may contain missing, which may be `.`, `_`, `.a`, `.b`, `.z`.
   b. String variables may not exceed 200 characters. String variables are recorded in a “padded” format, meaning that, when variables are read, it cannot be determined whether the variable had trailing blanks.
5. Value labels are not written in the XPORT dataset. Suppose that you have variable `sex` in the data with values 0 and 1 and that the values are labeled for gender (0 = male, and 1 = female). When the dataset is written in SAS XPORT Transport format, you can record that the variable label `gender` is associated with the `sex` variable, but you cannot record the association with the value labels male and female.

Value-label definitions are typically stored in a second XPORT dataset or in a text file containing SAS commands. You can use the `vallabfile()` option of `export sasxport5` to produce these datasets or files.
Value labels and formats are recorded in the same position in an XPORT file, meaning that names corresponding to formats used in SAS cannot be used. Thus, value labels may not be named

```
best, binary, comma, commax, d, date, datetime, dateampm, day, dmmmyy,
dollar, dollarx, downname, e, eurdfdd, eurdfde, eurdfdn, eurdfdt, eu-
rdfdbn, eurdfmn, eurdfmy, eurdfwdx, eurdfwxx, float, fract, hex, hmmm,
hour, ib, ibr, ieee, julday, julian, percent, mmingo, mmddyy, mmss, mmyy,
monname, month, mony, negparen, nengo, numx, octal, pd, pdjulg, pdjuli,
pib, pibr, pk, pvalue, qtr, qtrrr, rb, roman, s370ff, s370fib, s370fibu,
s370fpd, s370fpdu, s370fpib, s370frb, s370fzd, s370fzdl, s370fzds,
s370fzdt, s370fzdu, ssn, time, timeampm, tod, weekdate, weekdatx, week-
day, worddate, worddatx, wordf, words, year, yen, xmm, yymmdd, yymon,
yyq, yyqr, z, zd, or any uppercase variation of these.
```

We refer to this as the “Known Reserved Word List” in this documentation. Other words may also be reserved by SAS; the technical documentation for the SAS XPORT Transport format provides no guidelines. This list was created by examining the formats defined in SAS Language Reference: Dictionary, Version 8. If SAS adds new formats, the list will grow.

6. A flaw in the XPORT design can make it impossible, in rare instances, to determine the exact number of observations in a dataset. This problem can occur only if 1) all variables in the dataset are string and 2) the sum of the lengths of all the string variables is less than 80. Actually, the above is the restriction, assuming that the code for reading the dataset is written well. If it is not, the flaw could occur if 1) the last variable or variables in the dataset are string and 2) the sum of the lengths of all variables is less than 80.

To prevent stumbling over this flaw, make sure that the last variable in the dataset is not a string variable. This is always sufficient to avoid the problem.

7. There is no provision for saving the Stata concepts notes and characteristics.

**A2. Implications for writing XPORT datasets from Stata**

Stata datasets for the most part fit well into the SAS XPORT Transport format. With the same numbering scheme as above,

1. Stata refuses to write the dataset if it contains more than 9,999 variables.

2. Stata issues an error message if any variable or label name violates the naming restrictions, or if the rename option is specified, Stata fixes any names that violate the restrictions.

   Whether or not rename is specified, names will be recorded without regard to case: you do not have to name all your variables with all lowercase or all uppercase letters. Stata verifies that ignoring case does not lead to problems, complaining or, if option rename is specified, fixing them.

3. Stata truncates variable labels to 40 characters to fit within the XPORT limit.

4. Stata treats variable contents as follows:

   a. If a numeric variable records a value greater than 9.046e+74 in absolute value, Stata issues an error message. If a variable records a value less than 5.398e–79 in absolute value, 0 is written.
b. If you have string variables longer than 200 characters, Stata issues an error message. Also, if any string variable has trailing blanks, Stata issues an error message. To remove trailing blanks from string variable \( s \), you can type
\[
\text{. replace } s = \text{rtrim}(s)
\]
To remove leading and trailing blanks, type
\[
\text{. replace } s = \text{trim}(s)
\]

5. Value-label names are written in the XPORT dataset. The contents of the value label are not written in the same XPORT dataset. By default, \texttt{formats.xpf}, a second XPORT dataset, is created containing the value-label definitions.

SAS recommends creating a \texttt{formats.xpf} file containing the value-label definitions (what SAS calls format definitions). They have provided SAS macros, making the reading of \texttt{xpt} and \texttt{formats.xpf} files easy. See https://www.sas.com/govedu/fda/macro.html for details.

Alternatively, a SAS command file containing the value-label definitions can be produced. The \texttt{vallabfile()} option of \texttt{export sasxport5} is used to indicate which, if any, of the formats to use for recording the value-label definitions.

If a value-label name matches a name on the Known Reserved Word List, and the \texttt{rename} option is not specified, Stata issues an error message.

If a variable has no value label, the following format information is recorded:

<table>
<thead>
<tr>
<th>Stata format</th>
<th>SAS format</th>
</tr>
</thead>
<tbody>
<tr>
<td>%td...</td>
<td>MMDDYY10.</td>
</tr>
<tr>
<td>%-td...</td>
<td>MMDDYY10.</td>
</tr>
<tr>
<td>%#s</td>
<td>$CHAR#.</td>
</tr>
<tr>
<td>%-#s</td>
<td>$CHAR#.</td>
</tr>
<tr>
<td>% #s</td>
<td>$CHAR#.</td>
</tr>
<tr>
<td>all other</td>
<td>BEST12.</td>
</tr>
</tbody>
</table>

6. If you have a dataset that could provoke the XPORT design flaw, a warning message is issued. Remember, the best way to avoid this flaw is to ensure that the last variable in the dataset is numeric. This is easily done. You could, for instance, type
\[
\text{. generate ignoreme = 0}
\]
\[
\text{. export sasxport ...}
\]

7. Because the XPORT file format does not support notes and characteristics, Stata ignores them when it creates the XPORT file. You may wish to incorporate important notes into the documentation that you provide to the user of your XPORT file.

### A3. Implications for reading XPORT datasets into Stata

Reading SAS XPORT Version 5 Transport format files into Stata is easy, but sometimes there are issues to consider:

1. If there are too many variables, Stata issues an error message. If you are using Stata/MP or Stata/SE, you can increase the maximum number of variables with the \texttt{set maxvar} command; see \texttt{[D] memory}.  


A3. Implications for reading XPORT datasets into Stata

Reading SAS XPORT Version 5 Transport format files into Stata is easy, but sometimes there are issues to consider:

1. If there are too many variables, Stata issues an error message. If you are using Stata/MP or Stata/SE, you can increase the maximum number of variables with the \texttt{set maxvar} command; see \texttt{[D] memory}.
2. The XPORT variable-naming restrictions are more restrictive than those of Stata, so no problems should arise. However, Stata reserves the following names:

_-all, _b, byte, _coef, _cons, double, float, if, in, int, long, _n, _N, _pi, 
_pred, _rc, _skip, str#, strL, using, with

If the XPORT file contains variables with any of these names, Stata issues an error message. Also, the error message

```
.import sasxport5 ...
________ already defined
r(110);
```

indicates that the XPORT file was incorrectly prepared by some other software and that two or more variables share the same name.

3. The XPORT variable-label-length limit is more restrictive than that of Stata, so no problems can arise.

4. Variable contents may cause problems:

a. The range of numeric variables in an XPORT dataset is a subset of that allowed by Stata, so no problems can arise. All variables are brought back as doubles; we recommend that you run `compress` after loading the dataset:

```
.import sasxport5 ...
.compress
```

See [D] `compress`.

Stata has no missing-value code corresponding to `. `. If any value records `. `, then `u` is stored.

b. String variables are brought back as recorded but with all trailing blanks stripped.

5. Value-label names are read directly from the XPORT dataset. Any value-label definitions are obtained from a separate XPORT dataset, if available. If a value-label name matches any in the Known Reserved Word List, no value-label name is recorded, and instead, the variable display format is set to `%9.0g`, `%10.0g`, or `%td`.

The `%td` Stata format is used when the following SAS formats are encountered:

```
DATE, EURDFDN, JULDAY, MONTH, QTRR, YEAR, DAY, EURDFDW, JULIAN, MONYY,
WEEKDATE, YYYM, DDMYY, EURDFMN, MINGUO, NENGO, WEEKDATX, YYMDD, DOW-
NAME, EURDFMY, MMDDYY, PDJULG, WEEKDAY, YMON, EURDFDD, EURDFWDX, MMYY,
PDJULI, WORDDATE, YYQ, EURDFDE, EURDFWXX, MONNAME, QTR, WORDDATX, YYQR
```

If the XPORT file indicates that one or more variables have value labels, `import sasxport5` looks for the value-label definitions in `formats.xpf`, another XPORT file. If it does not find this file, it looks for `FORMATS.xpf`. If this file is not found, `import sasxport5` issues a warning message unless the `novallabels` option is specified.

Stata does not allow value-label ranges or string variables with value labels. If the `.xpt` file or `formats.xpf` file contains any of these, an error message is issued. The `novallabels` option allows you to read the data, ignoring all value labels.

6. If a dataset is read that provokes the all-strings XPORT design flaw, the dataset with the minimum number of possible observations is returned, and a warning message is issued. This duplicates the behavior of SAS.

7. SAS XPORT format does not allow notes or characteristics, so no issues can arise.
import sasxport5 — Import and export data in SAS XPORT Version 5 format

Also see

[D] import sas — Import SAS files
[D] import sasxport8 — Import and export data in SAS XPORT Version 8 format
[D] export — Overview of exporting data from Stata
[D] import — Overview of importing data into Stata
Title

import sasxport8 — Import and export data in SAS XPORT Version 8 format

Description

import sasxport8 and export sasxport8 import and export data from and to SAS XPORT Version 8 Transport format.

To import and export datasets from and to SAS XPORT Version 5 Transport format, see [D] import sasxport5.

Quick start

Load the contents of mydata.v8xpt into memory, replacing the data in memory
import sasxport8 mydata, clear

As above, but read variable names as lowercase
import sasxport8 mydata, clear case(lower)

Save data in memory to mydata.v8xpt, replacing the existing file
export sasxport8 mydata, replace

Save v1 and v2 to mydata.v8xpt, and save their corresponding value labels in a SAS command file, mydata.sas
export sasxport8 v1 v2 using mydata, replace vallabfile

Menu

import sasxport8
File > Import > SAS XPORT Version 8 (*.v8xpt)

export sasxport8
File > Export > SAS XPORT Version 8 (*.v8xpt)
Syntax

Import SAS XPORT Version 8 Transport file into Stata

```
import sasxport8  filename  [, import_options]
```

Export data in memory to a SAS XPORT Version 8 Transport file

```
export sasxport8  filename  [if]  [in]  [, export_options]
export sasxport8  varlist  using  filename  [if]  [in]  [, export_options]
```

If `filename` is specified without an extension, `.v8xpt` is assumed. If `filename` contains embedded spaces, enclose it in double quotes.

<table>
<thead>
<tr>
<th><code>import_options</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`case(lower</td>
<td>upper</td>
</tr>
<tr>
<td><code>clear</code></td>
<td>replace data in memory</td>
</tr>
</tbody>
</table>

collect is allowed with `import sasxport8`; see [U] 11.1.10 Prefix commands.

<table>
<thead>
<tr>
<th><code>export_options</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>replace</code></td>
<td>overwrite files if they already exist</td>
</tr>
<tr>
<td><code>vallabfile</code></td>
<td>save value labels in SAS command file</td>
</tr>
</tbody>
</table>

Options for `import sasxport8`

- `case(lower|upper|preserve)` specifies the case of the variable names after import. The default is `case(preserve)`.
- `clear` specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

Options for `export sasxport8`

- `replace` permits `export sasxport8` to overwrite the existing `filename.v8xpt`.
- `vallabfile` specifies that the value labels be written into a SAS command file, `filename.sas`, containing SAS `proc format` and related commands. Thus, `export sasxport8` creates two files: `filename.v8xpt`, containing the data, and `filename.sas`, containing the value labels. SAS users may wish to edit the resulting `filename.sas` file to change the “libname datapath” and “libname xptfile xport” lines at the top to correspond to the location that they desire. `export sasxport8` sets the location to the current working directory at the time `export sasxport8` was issued. No `.sas` file will be created if there are no value labels.
Remarks and examples

To save the data in memory as a SAS XPORT Version 8 Transport file, type

```
.export sasxport8 filename
```

To read a SAS XPORT Version 8 Transport file into Stata, type

```
.import sasxport8 filename
```

Stata will read into memory the XPORT file `filename.v8xpt` containing the data.

To demonstrate the use of `export sasxport8` and `import sasxport8`, we will first load `auto.dta` and export these data to a SAS V8XPORT named `auto.v8xpt`:

```
.use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
.export sasxport8 auto
file auto.v8xpt saved
```

We can export a subset of the data that includes only the variables `make`, `mpg`, and `weight` to a file named `auto_sub.v8xpt`.

```
.export sasxport8 make mpg weight using auto_sub
file auto_sub.v8xpt saved
```

Now, we import the data from `auto_sub.v8xpt` that we just created.

```
.import sasxport8 auto_sub, clear
(3 vars, 74 obs)
.describe
Contains data
Observations: 74 1978 automobile data
Variables: 3
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%17s</td>
<td>Make and model</td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%10.0g</td>
<td>Mileage (mpg)</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%15.4g</td>
<td>Weight (lbs.)</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.

Stored results

`import sasxport8` stores the following in `r()`:

Scalars

- `r(N)` number of observations imported
- `r(k)` number of variables imported

Also see

[D] import sas — Import SAS files
[D] import sasxport5 — Import and export data in SAS XPORT Version 5 format
[D] export — Overview of exporting data from Stata
[D] import — Overview of importing data into Stata
import spss — Import SPSS files

**Description**

import spss reads into memory a version 16 or higher IBM SPSS Statistics (.sav) file or a version 21 or higher compressed IBM SPSS Statistics (.zsav) file. import spss can import up to 32,766 variables at one time (up to 2,048 in Stata/BE). If your SPSS file contains more variables than this, you can break up the SPSS file into multiple Stata datasets.

**Quick start**

Load the IBM SPSS Statistics file myfile.sav into Stata

```
import spss myfile
```

As above, but replace data in memory

```
import spss myfile, clear
```

Import only variables x1 and x4 from myfile.sav into Stata

```
import spss x1 x4 using myfile
```

Load the compressed IBM SPSS Statistics file compfile.zsav into Stata

```
import spss compfile, zsav
```

As above, but read variable names as lowercase

```
import spss compfile, zsav case(lower)
```

**Menu**

File > Import > SPSS data (*.sav)
Syntax

Load an IBM SPSS Statistics file (*.sav)

    import spss [using] filename [, options]

Load a compressed IBM SPSS Statistics file (*.zsav)

    import spss [using] filename, zsav [, options]

Load a subset of an IBM SPSS Statistics file (*.sav)

    import spss [namelist] [if] [in] using filename [, options]

Load a subset of a compressed IBM SPSS Statistics file (*.zsav)

    import spss [namelist] [if] [in] using filename, zsav [, options]

If filename is specified without an extension, .sav is assumed unless you specify the zsav option, in which case extension .zsav is assumed. If filename contains embedded spaces, enclose it in double quotes.

namelist specifies SPSS variable names to be imported.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>case(lower</td>
<td>upper</td>
</tr>
<tr>
<td>clear</td>
<td>replace data in memory</td>
</tr>
<tr>
<td>encoding(&quot;encoding&quot;)</td>
<td>specify the file encoding; see help encodings</td>
</tr>
</tbody>
</table>

collect is allowed with import spss; see [U] 11.1.10 Prefix commands.
encoding() does not appear in the dialog box.

Options

zsav indicates the file to load is a compressed IBM SPSS Statistics file.

case(lower|upper|preserve) specifies the case of the variable names after import. The default is case(preserve).

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

The following option is available with import spss but is not shown in the dialog box:

encoding("encoding") specifies the encoding of the file. If your file has an incorrect encoding specified in the file header, you can use this option to specify the correct encoding. See help encodings for details.
Remarks and examples

import spss reads into memory a version 16 or higher IBM SPSS Statistics (.sav) file or a version 21 or higher compressed IBM SPSS Statistics (.zsav) file. If an SPSS variable name from the file does not conform to a Stata variable name, a generic v# name will be assigned, and the original variable name will be stored as a characteristic for the variable. If an SPSS variable label is too long, it will be truncated to 80 characters, and the original variable label will be stored as a variable characteristic. All value labels for string variables will be ignored. Value labels for numeric variables will be named label# and attached to the corresponding variable.

Example 1: Importing an SPSS file into Stata

We can import SPSS files into Stata, either by selecting the entire file or by selecting subsets of the data, with import spss. For example, we have the SPSS file auto.sav, which contains data on automobiles. Below, we demonstrate how to import these data into Stata. To follow along, download this file to your working directory by typing the copy command below:

.copy https://www.stata.com/sampledata/auto.sav auto.sav

We first load the entire auto.sav file into Stata by typing

.import spss auto
(12 vars, 74 obs)
.describe
Contains data
Observations: 74
Variables: 12

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%17s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%5.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%2.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%1.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>headroom</td>
<td>double</td>
<td>%3.1f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trunk</td>
<td>byte</td>
<td>%2.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%4.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%3.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>turn</td>
<td>byte</td>
<td>%2.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>displacement</td>
<td>int</td>
<td>%3.0f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>double</td>
<td>%4.2f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>byte</td>
<td>%1.0f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.
We can instead import only variables `make` and `weight` into memory from `auto.sav`. We include the `clear` option to replace the data in memory without saving them.

```
.import spss make weight using auto, clear
(2 vars, 74 obs)
.describe
```

Contains data

- Observations: 74
- Variables: 2

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%17s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%4.0f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

- Note: Dataset has changed since last saved.

### Stored results

`import spss` stores the following in `r()`:

**Scalars**

- `r(N)`: number of observations imported
- `r(k)`: number of variables imported

### Also see

[D] `import` — Overview of importing data into Stata
**Description**

`infile using` reads a dataset that is stored in text form. `infile using` does this by first reading `dfilename`—a “dictionary” that describes the format of the data file—and then reads the file containing the data. The dictionary is a file you create with the Do-file Editor or an editor outside Stata.

Strings containing plain ASCII or UTF-8 are imported correctly. Strings containing extended ASCII will not be imported (that is, displayed) correctly; you can use Stata’s `replace` command with the `ustrfrom()` function to convert extended ASCII to UTF-8. If `ebcdic` is specified, the data will be converted from EBCDIC to ASCII as they are imported. The dictionary in all cases must be ASCII.

If `using filename` is not specified, the data are assumed to begin on the line following the closing brace. If `using filename` is specified, the data are assumed to be located in `filename`.

The data may be in the same file as the dictionary or in another file. `infile` with a dictionary can import both numeric and string data. Individual strings may be up to 100,000 bytes long. Strings longer than 2,045 bytes are imported as `strL`s (see [U] 12.4.8 strL).

Another variation on `infile` omits the intermediate dictionary; see [D] `infile (free format)`). This variation is easier to use but will not read fixed-format files. On the other hand, although `infile` with a dictionary will read free-format files, `infile` without a dictionary is even better at it.

An alternative to `infile using` for reading fixed-format files is `infix`; see [D] `infix (fixed format)`. `infix` provides fewer features than `infile using` but is easier to use.

Stata has other commands for reading data. If you are not certain that `infile using` will do what you are looking for, see [D] `import` and [U] 22 Entering and importing data.

**Quick start**

For dictionary file `mydata.dct` that reads `int`-type `v1` and `str10`-type `v2`

```stata
    dictionary {
        int v1
        str10 v2
    }
```

Import data from `mydata.raw` with instructions for reading the data contained in dictionary file `mydata.dct`

```
infile using mydata.dct, using(mydata.raw)
```

Same as above

```
infile using mydata, using(mydata)
```

As above, but import data from `mydata.txt`

```
infile using mydata, using(mydata.txt)
```
As above, but read only the first 10 observations

```
infile using mydata in 1/10, using(mydata.txt)
```

Read only observations where `catvar` is equal to 4 or 5

```
infile using mydata if catvar==4 | catvar==5, using(mydata.txt)
```

### Menu

File > Import > Text data in fixed format with a dictionary

### Syntax

```
infile using dfilename [if] [in] [ , options ]
```

If `dfilename` is specified without an extension, `.dct` is assumed. If `dfilename` contains embedded spaces, remember to enclose it in double quotes.

#### options

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
</tr>
<tr>
<td><code>using(filename)</code></td>
</tr>
<tr>
<td><code>clear</code></td>
</tr>
<tr>
<td><strong>Options</strong></td>
</tr>
<tr>
<td><code>automatic</code></td>
</tr>
<tr>
<td><code>ebcdic</code></td>
</tr>
</tbody>
</table>

A dictionary is a text file that is created with the Do-file Editor or an editor outside Stata. This file specifies how Stata should read fixed-format data from a text file. The syntax for a dictionary is

```
begin dictionary file
[infile] dictionary [using filename] { 
  * comments may be included freely
  _lrecl(#) _firstlineoffile(#) _lines(#) _line(#) _newline[#] _column(#) _skip[#]
  [type] varname [::lblname] [%infmt] ["variable label"]
}
(your data might appear here)
end dictionary file
```

where `%infmt` is \{ `%[#.#]` | `%f` | `%g` | `%e` | `%s` | `%S` \}
Options

using(filename) specifies the name of a file containing the data. If using() is not specified, the data are assumed to follow the dictionary in dfilename, or if the dictionary specifies the name of some other file, that file is assumed to contain the data. If using(filename) is specified, filename is used to obtain the data, even if the dictionary says otherwise. If filename is specified without an extension, .raw is assumed.

If filename contains embedded spaces, remember to enclose it in double quotes.
clear specifies that it is okay for the new data to replace what is currently in memory. To ensure that you do not lose something important, infile using will refuse to read new data if other data are already in memory. clear allows infile using to replace the data in memory. You can also drop the data yourself by typing drop _all before reading new data.

automatic causes Stata to create value labels from the nonnumeric data it reads. It also automatically widens the display format to fit the longest label.
ebcdic specifies that the data be stored using EBCDIC character encoding rather than the default ASCII encoding and that the data be converted from EBCDIC to ASCII as they are imported.

Dictionary directives

* marks comment lines. Wherever you wish to place a comment, begin the line with a *. Comments can appear many times in the same dictionary.

{lrecl(#)} is used only for reading datasets that do not have end-of-line delimiters (carriage return, line feed, or some combination of these). Such files are often produced by mainframe computers and are either coded in EBCDIC or have been translated from EBCDIC into ASCII. _lrecl() specifies the logical record length. _lrecl() requests that infile act as if a line ends every # bytes.

_lrecl() appears only once, and typically not at all, in a dictionary.

_firstlineoffile(#) (abbreviation _first()) is also rarely specified. It states the line of the file where the data begin. You do not need to specify _first() when the data follow the dictionary; Stata can figure that out for itself. However, you might specify _first() when reading data from another file in which the first line does not contain data because of headers or other markers.

_first() appears only once, and typically not at all, in a dictionary.

_lines(#) states the number of lines per observation in the file. Simple datasets typically have _lines(1). Large datasets often have many lines (sometimes called records) per observation. _lines() is optional, even when there is more than one line per observation because infile can sometimes figure it out for itself. Still, if _lines(1) is not right for your data, it is best to specify the correct number through _lines(#).

_lines() appears only once in a dictionary.

_line(#) tells infile to jump to line # of the observation. _line() is not the same as _lines(). Consider a file with _lines(4), meaning four lines per observation. _line(2) says to jump to the second line of the observation. _line(4) says to jump to the fourth line of the observation. You may jump forward or backward. infile does not care, and there is no inefficiency in going forward to _line(3), reading a few variables, jumping back to _line(1), reading another variable, and jumping forward again to _line(3).
You need not ensure that, at the end of your dictionary, you are on the last line of the observation. \texttt{infile} knows how to get to the next observation because it knows where you are and it knows \_lines(), the total number of lines per observation.

\_line() may appear many times in a dictionary.

\_newline[#] is an alternative to \_line(). \_newline(1), which may be abbreviated \_newline, goes forward one line. \_newline(2) goes forward two lines. We do not recommend using \_newline() because \_line() is better. If you are currently on line 2 of an observation and want to get to line 6, you could type \_newline(4), but your meaning is clearer if you type \_line(6).

\_newline() may appear many times in a dictionary.

\_column(#) jumps to column # (in bytes) of the current line. You may jump forward or backward within a line. \_column() may appear many times in a dictionary.

\_skip[#] jumps forward # columns on the current line. \_skip() is just an alternative to \_column(). \_skip() may appear many times in a dictionary.

\[type\] varname [:lbname] [%infmt] ["variable label"] instructs \texttt{infile} to read a variable. The simplest form of this instruction is the variable name itself: varname.

At all times, \texttt{infile} is on some column of some line of an observation. \texttt{infile} starts on column 1 of line 1, so pretend that is where we are. Given the simplest directive, ‘varname’, \texttt{infile} goes through the following logic:

If the current column is blank, it skips forward until there is a nonblank column (or until the end of the line). If it just skipped all the way to the end of the line, it stores a missing value in varname. If it skipped to a nonblank column, it begins collecting what is there until it comes to a blank column or the end of the line. These are the data for varname. Then it sets the current column to wherever it is.

The logic is a bit more complicated. For instance, when skipping forward to find the data, \texttt{infile} might encounter a quote. If so, it then collects the characters for the data by skipping forward until it finds the matching quote. If you specified a %infmt, then \texttt{infile} skips the skipping-forward step and simply collects the specified number of bytes. If you specified a %S infmt, then \texttt{infile} does not skip leading or trailing blanks. Nevertheless, the general logic is (optionally) skip, collect, and reset.

Remarks and examples

Remarks are presented under the following headings:

\begin{itemize}
  \item \texttt{Introduction}
  \item Reading free-format files
  \item Reading fixed-format files
  \item Numeric formats
  \item String formats
  \item Specifying column and line numbers
  \item Examples of reading fixed-format files
  \item Reading fixed-block files
  \item Reading EBCDIC files
\end{itemize}
Introduction

`infile` using follows a two-step process to read your data. You type something like `infile` using `descript`, and

1. `infile` using reads the file `descript.dct`, which tells `infile` about the format of the data; and

2. `infile` using then reads the data according to the instructions recorded in `descript.dct`.

`descript.dct` (the file could be named anything) is called a dictionary, and `descript.dct` is just a text file that you create with the Do-file Editor or an editor outside Stata.

As for the data, they can be in the same file as the dictionary or in a different file. It does not matter.

Reading free-format files

Another variation of `infile` for reading free-format files is described in `[D] infile (free format)`. We will refer to this variation as `infile` without a dictionary. The distinction between the two variations is in the treatment of line breaks. `infile` without a dictionary does not consider them significant. `infile` with a dictionary does.

A line, also known as a record, physical record, or physical line (as opposed to observations, logical records, or logical lines), is a string of characters followed by the line terminator. If you were to type the file, a line is what would appear on your screen if your screen were infinitely wide. Your screen would have to be infinitely wide so that there would be no possibility that one line could take more than one line of your screen, thus fooling you into thinking that there are multiple lines when there is only one.

A logical line, on the other hand, is a sequence of one or more physical lines that represent one observation of your data. `infile` with a dictionary does not spontaneously go to new physical lines; it goes to a new line only between observations and when you tell it to. `infile` without a dictionary, on the other hand, goes to a new line whenever it needs to, which can be right in the middle of an observation. Thus consider the following little bit of data, which is for three variables:

```
5 4
1 9 3
2
```

How do you interpret these data?

Here is one interpretation: There are 3 observations. The first is 5, 4, and missing. The second is 1, 9, and 3. The third is 2, missing, and missing. That is the interpretation that `infile` with a dictionary makes.

Here is another interpretation: There are 2 observations. The first is 5, 4, and 1. The second is 9, 3, and 2. That is the interpretation that `infile` without a dictionary makes.

Which is right? You would have to ask the person who entered these data. The question is, are the line breaks significant? Do they mean anything? If the line breaks are significant, you use `infile` with a dictionary. If the line breaks are not significant, you use `infile` without a dictionary.

The other distinction between the two `infiles` is that `infile` with a dictionary does not process comma-separated–value format. If your data are comma-separated, tab-separated, or otherwise delimited, see `[D] import delimited` or `[D] infile (free format).
Example 1: A simple dictionary with data

Outside Stata, we have typed into the file `highway.dct` information on the accident rate per million vehicle miles along a stretch of highway, the speed limit on that highway, and the number of access points (on-ramps and off-ramps) per mile. Our file contains

```plaintext
infile dictionary {
    acc_rate    spdlimit    acc_pts
}
4.58 55 4.6
2.86 60 4.4
1.61 . 2.2
3.02 60 4.7
```

This file can be read by typing the commands below. Stata displays the dictionary and reads the data:

```
    . infile using highway
    infile dictionary {
        acc_rate    spdlimit    acc_pts
    }
    (4 observations read)
    . list
```

<table>
<thead>
<tr>
<th>acc_rate</th>
<th>spdlimit</th>
<th>acc_pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.58</td>
<td>55</td>
<td>4.6</td>
</tr>
<tr>
<td>2.86</td>
<td>60</td>
<td>4.4</td>
</tr>
<tr>
<td>1.61</td>
<td>.</td>
<td>2.2</td>
</tr>
<tr>
<td>3.02</td>
<td>60</td>
<td>4.7</td>
</tr>
</tbody>
</table>

Example 2: Specifying variable labels

We can include variable labels in a dictionary so that after we `infile` the data, the data will be fully labeled. We could change `highway.dct` to read

```plaintext
infile dictionary {
    * This is a comment and will be ignored by Stata
    * You might type the source of the data here.
    acc_rate    "Acc. Rate/Million Miles"
    spdlimit    "Speed Limit (mph)"
    acc_pts     "Access Pts/Mile"
}
4.58 55 4.6
2.86 60 4.4
1.61 . 2.2
3.02 60 4.7
```

Now when we type `infile using highway`, Stata not only reads the data but also labels the variables.
Example 3: Specifying variable storage types

We can indicate the variable types in the dictionary. For instance, if we wanted to store acc_rate as a double and spdlimit as a byte, we could change highway.dct to read

```
infile dictionary {
* This is a comment and will be ignored by Stata
* You might type the source of the data here.
  double acc_rate "Acc. Rate/Million Miles"
  byte  spdlimit "Speed Limit (mph)"
  acc_pts "Access Pts/Mile"
}
4.58 55 4.6
2.86 60 4.4
1.61 . 2.2
3.02 60 4.7
```

Because we do not indicate the variable type for acc_pts, it is given the default variable type float (or the type specified by the set type command).

Example 4: Reading string variables

By specifying the types, we can read string variables as well as numeric variables. For instance,

```
infile dictionary {
* data on employees
  str20 name "Name"
  age  "Age"
  int sex "Sex coded 0 male 1 female"
}
"Lisa Gilmore" 25 1
Branton 32 1
'Bill Ross' 27 0
```

The strings can be delimited by single or double quotes, and quotes may be omitted altogether if the string contains no blanks or other special characters.

Example 5: Specifying value labels

You may attach value labels to variables in the dictionary by using the colon notation:

```
infile dictionary {
* data on name, sex, and age
  str16 name "Name"
  sex:sexlbl "Sex"
  int age  "Age"
}
"Arthur Doyle" Male 22
"Mary Hope" Female 37
"Guy Fawkes" Male 48
"Karen Cain" Female 25
```
If you want the value labels to be created automatically, you must specify the automatic option on the `infile` command. These data could be read by typing `infile using emp2, automatic`, assuming the dictionary and data are stored in the file `emp2.dct`.

**Example 6: Separate the dictionary and data files**

The data need not be in the same file as the dictionary. We might leave the highway data in `highway.raw` and write a dictionary called `highway.dct` describing the data:

```plaintext
begin highway.dct, example 4
infile dictionary using highway {
* This dictionary reads the file highway.raw. If the
* file were called highway.txt, the first line would
* read "dictionary using highway.txt"
  acc_rate  "Acc. Rate/Million Miles"
  spdlimit  "Speed Limit (mph)"
  acc_pts   "Access Pts/Mile"
}
end highway.dct, example 4
```

**Example 7: Ignoring the top of a file**

The `firstlineoffile()` directive allows us to ignore lines at the top of the file. Consider the following raw dataset:

```plaintext
begin mydata.raw
The following data were entered by Marsha Martinez. It was checked by
Helen Troy.
id income educ sex age
1024 25000 HS Male 28
1025 27000 C Female 24
end mydata.raw
```

Our dictionary might read

```plaintext
begin mydata.dct
infile dictionary using mydata {
  _first(4)
  int id "Identification Number"
  income "Annual income"
  str2 educ "Highest educ level"
  str6 sex
  byte age
}
end mydata.dct
```
Example 8: Data spread across multiple lines

The \_line() and \_lines() directives tell Stata how to read our data when there are multiple records per observation. We have the following in mydata2.raw:

```
begin mydata2.raw
id income educ sex age
1024 25000 HS Male 28
1025 27000 C Female 24
1035 26000 HS Male 32
1036 25000 C Female 25
end mydata2.raw
```

We can read this with a dictionary mydata2.dct, which we will just let Stata list as it simultaneously reads the data:

```
.infile using mydata2, clear
infile dictionary using mydata2 {
    _first(2) * Begin reading on line 2
    _lines(3) * Each observation takes 3 lines.
    int id "Identification Number" * Since _line is not specified, Stata
    income "Annual income" * assumes that it is 1.
    str2 educ "Highest educ level"
    _line(2) * Go to line 2 of the observation.
    str6 sex * (values for sex are located on line 2)
    _line(3) * Go to line 3 of the observation.
    int age * (values for age are located on line 3)
}
(4 observations read)
.list
```

<table>
<thead>
<tr>
<th>id</th>
<th>income</th>
<th>educ</th>
<th>sex</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1024</td>
<td>25000</td>
<td>HS</td>
<td>Male</td>
</tr>
<tr>
<td>2</td>
<td>1025</td>
<td>27000</td>
<td>C</td>
<td>Female</td>
</tr>
<tr>
<td>3</td>
<td>1035</td>
<td>26000</td>
<td>HS</td>
<td>Male</td>
</tr>
<tr>
<td>4</td>
<td>1036</td>
<td>25000</td>
<td>C</td>
<td>Female</td>
</tr>
</tbody>
</table>

Here is the really good part: we read these variables in order, but that was not necessary. We could just as well have used the dictionary:

```
begin mydata2p.dct
infile dictionary using mydata2 {
    _first(2) * Begin reading on line 2
    _lines(3) * Each observation takes 3 lines.
    _line(1) int id "Identification number"
    income "Annual income" * Since _line is not specified, Stata
    str2 educ "Highest educ level"
    _line(2) int age
    str6 sex * (values for sex are located on line 2)
    _line(3) int age * (values for age are located on line 3)
}
```
We would have obtained the same results just as quickly, the only difference being that our variables in the final dataset would be in the order specified: id, income, educ, age, and sex.

Technical note

You can use `_newline` to specify where breaks occur, if you prefer:

```
infile dictionary {
  acc_rate "Acc. Rate/Million Miles"
  spdlimit "Speed Limit (mph)"
  _newline acc_pts "Access Pts/Mile"
}
```

The line reading `1.61 .` could have been read 1.61 (without the period), and the results would have been unchanged. Because dictionaries do not go to new lines automatically, a missing value is assumed for all values not found in the record.

Reading fixed-format files

Values in formatted data are sometimes packed one against the other with no intervening blanks. For instance, the highway data might appear as

```
begin highway.raw, example 6
4.58 55
4.6
2.86 60
4.4
1.61 .
2.2
3.02 60
4.7
end highway.raw, example 6
```

The first four columns of each record represent the accident rate; the next two columns, the speed limit; and the last three columns, the number of access points per mile.

To read these data, you must specify the `%infmt` in the dictionary. Numeric `%infmts` are denoted by a leading percent sign (%) followed optionally by a string of the form `w` or `w.d`, where `w` and `d` stand for two integers. The first integer, `w`, specifies the width of the format. The second integer, `d`, specifies the number of digits that are to follow the decimal point. `d` must be less than or equal to `w`. Finally, a character denoting the format type (f, g, or e) is appended. For example, `%9.2f` specifies an f format that is nine characters wide and has two digits following the decimal point.
Numeric formats

The f format indicates that infile is to attempt to read the data as a number. When you do not specify the %infmt in the dictionary, infile assumes the %f format. The width, w, being missing means that infile is to attempt to read the data in free format.

As it starts reading each observation, infile reads a record into its buffer and sets a column pointer to 1, indicating that it is currently on the first column. When infile processes a %f format, it moves the column pointer forward through white space. It then collects the characters up to the next occurrence of white space and attempts to interpret those characters as a number. The column pointer is left at the first occurrence of white space following those characters. If the next variable is also free format, the logic repeats.

When you explicitly specify the field width w, as in %wf, infile does not skip leading white space. Instead, it collects the next w characters starting at the column pointer and attempts to interpret the result as a number. The column pointer is left at the old value of the column pointer plus w, that is, on the first character following the specified field.

Example 9: Specifying the width of fields

If the data above were stored in highway.raw, we could create the following dictionary to read the data:

```
begin highway.dct, example 6

infile dictionary using highway {
    acc_rate %4f "Acc. Rate/Million Miles"
    spdlimit %2f "Speed Limit (mph)"
    acc_pts %3f "Access Pts/Mile"
}
```

When we explicitly indicate the field width, infile does not skip intervening characters. The first four columns are used for the variable acc_rate, the next two for spdlimit, and the last three for acc_pts.

Technical note

The d specification in the %w.df indicates the number of implied decimal places in the data. For instance, the string 212 read in a %3.2f format represents the number 2.12. Do not specify d unless your data have elements of this form. The w alone is sufficient to tell infile how to read data in which the decimal point is explicitly indicated.

When you specify d, Stata takes it only as a suggestion. If the decimal point is explicitly indicated in the data, that decimal point always overrides the d specification. Decimal points are also not implied if the data contain an E, e, D, or d, indicating scientific notation.

Fields are right-justified before implying decimal points. Thus ‘2’, ‘2’, and ‘2’ are all read as 0.2 by the %3.1f format.
Technical note

The g and e formats are the same as the f format. You can specify any of these letters interchangeably. The letters g and e are included as a convenience to those familiar with Fortran, in which the e format indicates scientific notation. For example, the number 250 could be indicated as 2.5E+02 or 2.5D+02. Fortran programmers would refer to this as an E7.5 format, and in Stata, this format would be indicated as %7.5e. In Stata, however, you need specify only the field width $w$, so you could read this number by using %7f, %7g, or %7e.

The g format is really a Fortran output format that indicates a freer format than f. In Stata, the two formats are identical.

Throughout this section, you may freely substitute the g or e formats for the f format.

Technical note

Be careful to distinguish between %fmts and %infmts. %fmts are also known as display formats—they describe how a variable is to look when it is displayed; see [U] 12.5 Formats: Controlling how data are displayed. %infmts are also known as input formats—they describe how a variable looks when you input it. For instance, there is an output date format, %td, but there is no corresponding input format. (See [U] 25 Working with dates and times for recommendations on how to read dates.) For the other formats, we have attempted to make the input and output definitions as similar as possible. Thus we include g, e, and f %infmts, even though they all mean the same thing, because g, e, and f are also %fmts.

String formats

The s and S formats are used for reading strings. The syntax is %ws or %wS, where the $w$ is optional. If you do not specify the field width, your strings must either be enclosed in quotes (single or double) or not contain any characters other than letters, numbers, and “_”.

This may surprise you, but the s format can be used for reading numeric variables, and the f format can be used for reading string variables! When you specify the field width, $w$, in the %wf format, all embedded blanks in the field are removed before the result is interpreted. They are not removed by the %ws format.

For instance, the %3f format would read “−2”, “−2 ”, or “−2” as the number −2. The %3s format would not be able to read “−2” as a number, because the sign is separated from the digit, but it could read “−2” or “−2 ”. The %wf format removes blanks; datasets written by some Fortran programs separate the sign from the number.

There are, however, some side effects of this practice. The string “2 2” will be read as 22 by a %3f format. Most Fortran compilers would read this number as 202. The %3s format would issue a warning and store a missing value.

Now consider reading the string “a b” into a string variable. Using a %3s format, Stata will store it as it appears: a b. Using a %3f format, however, it will be stored as ab—the middle blank will be removed.

%ws is a special case of %ws. A string read with %ws will have leading and trailing blanks removed, but a string read with %wS will not have them removed.

Examples using the %s format are provided below, after we discuss specifying column and line numbers.
Specifying column and line numbers

(Column) jumps to the specified column. For instance, the documentation of some dataset indicates that the variable age is recorded as a two-digit number in column 47. You could read this by coding

```
-column(47) age %2f
```

After typing this, you are now at column 49, so if immediately following age there were a one-digit number recording sex as 0 or 1, you could code

```
-column(47) age %2f
sex %1f
```

or, if you wanted to be explicit about it, you could instead code

```
-column(47) age %2f
-column(49) sex %1f
```

It makes no difference. If at column 50 there were a one-digit code for race and you wanted to read it but skip reading the sex code, you could code

```
-column(47) age %2f
-column(50) race %1f
```

You could equivalently skip forward using _skip():

```
-column(47) age %2f
(skip(1) race %1f
```

One advantage of (Column) over _skip is that it lets you jump forward or backward in a record. If you wanted to read race and then age, you could code

```
-column(50) race %1f
-column(47) age %2f
```

If the data you are reading have multiple lines per observation (sometimes said as multiple records per observation), you can tell infile how many lines per record there are by using _lines():

```
_lines(4)
```

_lines() appears only once in a dictionary. Good style says that it should be placed near the top of the dictionary, but Stata does not care.

When you want to go to a particular line, include the _line() directive. In our example, let’s assume that race, sex, and age are recorded on the second line of each observation:

```
_lines(4)
-line(2)
-column(47) age %2f
-column(50) race %1f
```

Let’s assume that id is recorded on line 1.

```
_lines(4)
-line(1)
-column(1) id %4f
-line(2)
-column(47) age %2f
-column(50) race %1f
```
(infile (fixed format) — Import text data in fixed format with a dictionary)

_line() works like _column() in that you can jump forward or backward, so these data could just as well be read by:

```plaintext
_line(4)
_line(2)
  _column(47) age %2f
  _column(50) race %1f
_line(1)
  _column(1) id %4f
```

Remember that this dataset has four lines per observation, and yet we have never referred to line(3) or line(4). That is okay. Also, at the end of our dictionary, we are on line 1, not line 4. That is okay, too. infile will still get to the next observation correctly.

Technical note

Another way to move between records is _newline(). _newline() is to _line() as _skip() is to _column(), which is to say, _newline() can only go forward. There is one difference: _skip() has its uses, whereas _newline() is useful only for backward capability with older versions of Stata.

(skip()) has its uses because sometimes we think in columns and sometimes we think in widths. Some data documentation might include the sentence, “At column 54 are recorded the answers to the 25 questions, with one column allotted to each.” If we want to read the answers to questions 1 and 5, it would indeed be natural to code

```plaintext
  _column(54) q1 %1f
  _skip(3)
    q5 %1f
```

Nobody has ever read data documentation with the statement, “Demographics are recorded on record 2, and two records after that are the income values.” The documentation would instead say, “Record 2 contains the demographic information and record 4, income.” The _newline() way of thinking is based on what is convenient for the computer, which does, after all, have to move past a certain number of records. That, however, is no reason for making you think that way.

Before that thought occurred to us, Stata users specified _newline() to go forward a number of records. They still can, so their old dictionaries will work. When you use _newline() and do not specify _lines(), you must move past the correct number of records so that, at the end of the dictionary, you are on the last record. In this mode, when Stata reexecutes the dictionary to process the next observation, it goes forward one record.

Examples of reading fixed-format files

Example 10: A file with two lines per observation

In this example, each observation occupies two lines. The first 2 observations in the dataset are

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Dunbar</td>
<td>10001</td>
<td>101 North 42nd Street</td>
</tr>
<tr>
<td>Sam K. Newey Jr.</td>
<td>10002</td>
<td>15663 Roustabout Boulevard</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The first observation tells us that the name of the respondent is John Dunbar; that his ID is 10001; that his address is 101 North 42nd Street; and that his answers to questions 1–10 were yes, no, yes, no, yes, yes, yes, yes, yes, and yes.

The second observation tells us that the name of the respondent is Sam K. Newey Jr.; that his ID is 10002; that his address is 15663 Roustabout Boulevard; and that his answers to questions 1–10 were no, yes, no, yes, no, no, no, no, no, and no.

To see the layout within the file, we can temporarily add two rulers to show the appropriate columns:

```
John Dunbar 10001 101 North 42nd Street
1010111111
Sam K. Newey Jr. 10002 15663 Roustabout Boulevard
0101000000
```

Each observation in the data appears in two physical lines within our text file. We had to check in our editor to be sure that there really were new-line characters (for example, “hard returns”) after the address. This is important because some programs will wrap output for you so that one line may appear as many lines. The two seemingly identical files will differ in that one has a hard return and the other has a soft return added only for display purposes.

In our data, the name occupies columns 1–32; a person identifier occupies columns 33–37; and the address occupies columns 40–80. Our worksheet revealed that the widest address ended in column 80.

The text file containing these data is called `fname.txt`. Our dictionary file looks like this:

```plaintext
begin fname.dct
infile dictionary using fname.txt {

* Example reading in data where observations extend across more
* than one line. The next line tells infile there are 2 lines/obs:
*
_lines(2)
*
_column(33)   long   id   %5f   "Person id"
_skip(2)      str50   addr  %41s  "Address"
_column(1)    byte   q1   %1f   "Question 1"
    byte   q2   %1f   "Question 2"
    byte   q3   %1f   "Question 3"
    byte   q4   %1f   "Question 4"
    byte   q5   %1f   "Question 5"
    byte   q6   %1f   "Question 6"
    byte   q7   %1f   "Question 7"
    byte   q8   %1f   "Question 8"
    byte   q9   %1f   "Question 9"
    byte   q10  %1f  "Question 10"

} end fname.dct
```

Up to five pieces of information may be supplied in the dictionary for each variable: the location of the data, the storage type of the variable, the name of the variable, the input format, and the variable label.

Thus the `str50` line says that the first variable is to be given a storage type of `str50`, called `name`, and is to have the variable label “Name of respondent”. The `%32s` is the input format, which
tells Stata how to read the data. The s tells Stata not to remove any embedded blanks; the 32 tells Stata to go across 32 columns when reading the data.

The next line says that the second variable is to be assigned a storage type of long, named id, and be labeled “Person id”. Stata should start reading the information for this variable in column 33. The f tells Stata to remove any embedded blanks, and the 5 says to read across five columns.

The third variable is to be given a storage type of str50, called addr, and be labeled “Address”. The _skip(2) directs Stata to skip two columns before beginning to read the data for this variable, and the %41s instructs Stata to read across 41 columns and not to remove embedded blanks.

_line(2) instructs Stata to go to line 2 of the observation.

The remainder of the data is 0/1 coded, indicating the answers to the questions. It would be convenient if we could use a shorthand to specify this portion of the dictionary, but we must supply explicit directives.

Technical note

In the preceding example, there were two pieces of information about location: where the data begin for each variable (the _column(), _skip(), _line()) and how many columns the data span (the %32s, %5f, %41s, %1f). In our dictionary, some of this information was redundant. After reading name, Stata had finished with 32 columns of information. Unless instructed otherwise, Stata would proceed to the next column—column 33—to begin reading information about id. The _column(33) was unnecessary.

The _skip(2) was necessary, however. Stata had read 37 columns of information and was ready to look at column 38. Although the address information does not begin until column 40, columns 38 and 39 contain blanks. Because these are leading blanks instead of embedded blanks, Stata would just ignore them without any trouble. The problem is with the %41s. If Stata begins reading the address information from column 38 and reads 41 columns, Stata would stop reading in column 78 (78 − 41 + 1 = 38), but the widest address ends in column 80. We could have omitted the _skip(2) if we had specified an input format of %43s.

The _line(2) was necessary, although we could have read the second line by coding _newline instead.

The _column(1) could have been omitted. After the _line(), Stata begins in column 1.

See the next example for a dataset in which both pieces of location information are required.
Example 11: Manipulating the column pointer

The following file contains six variables in a variety of formats. In the dictionary, we read the variables fifth and sixth out of order by forcing the column pointer.

```
infile dictionary {
    first  %3f
    double second  %2.1f
    third  %6f
    _skip(2) str4 fourth  %4s
    _column(21) sixth %4.1f
    _column(18) fifth %2f
}
```

```
1.2125.7e+252abcd 1 .232
1.3135.7 52efgh2 5
1.41457 52abcd 3 100.
1.5155.7D+252efgh04 1.7
16 16 .57 52abcd 5 1.71
```

Assuming that the above is stored in a file called `example.dct`, we can `infile` and `list` it by typing

```
.infile using example
.infile dictionary {
    first  %3f
    double second  %2.1f
    third  %6f
    _skip(2) str4 fourth  %4s
    _column(21) sixth %4.1f
    _column(18) fifth %2f
}
```

(5 observations read)

```
.list
```

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
<th>third</th>
<th>fourth</th>
<th>sixth</th>
<th>fifth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>1.2</td>
<td>570</td>
<td>abcd</td>
<td>.232</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>1.3</td>
<td>5.7</td>
<td>efgh</td>
<td>.5</td>
<td>2</td>
</tr>
<tr>
<td>1.4</td>
<td>1.4</td>
<td>57</td>
<td>abcd</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>1.5</td>
<td>570</td>
<td>efgh</td>
<td>1.7</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>1.6</td>
<td>.57</td>
<td>abcd</td>
<td>1.71</td>
<td>5</td>
</tr>
</tbody>
</table>

Reading fixed-block files

Technical note

The `_lrec1(#)` directive is used for reading datasets that do not have end-of-line delimiters (carriage return, line feed, or some combination of these). Such datasets are typical of IBM mainframes, where they are known as fixed block, or FB. The abbreviation LRECL is IBM mainframe jargon for logical record length.
In a fixed-block dataset, each # characters are to be interpreted as a record. For instance, consider the data

1 21
2 42
3 63

In fixed-block format, these data might be recorded as

```
begin mydata.ibm
1 21 2 42 3 63
end mydata.ibm
```

and you would be told, on the side, that the LRECL is 4. If you then pass along that information to `infile`, it can read the data:

```
infile dictionary using mydata.ibm {
    _lrecl(4)
    int id
    int age
}
```

When you do not specify the `_lrecl(#)` directive, `infile` assumes that each line ends with the standard text EOL delimiter (which can be a line feed, a carriage return, a line feed followed by a carriage return, or a carriage return followed by a line feed). When you specify `_lrecl(#)`, `infile` reads the data in blocks of # characters and then acts as if that is a line.

A common mistake in processing fixed-block datasets is to use an incorrect LRECL value, such as 160 when it is really 80. To understand what can happen, pretend that you thought the LRECL in your data was 6 rather than 4. Taking the characters in groups of 6, the data appear as

```
1 21 2
42 3
63
```

Stata cannot verify that you have specified the correct LRECL, so if the data appear incorrect, verify that you have the correct number.

The maximum LRECL `infile` allows is 524,275.

---

**Reading EBCDIC files**

In the previous section, we discussed the `_lrecl(#)` directive that is often necessary for files that originated on mainframes and do not have end-of-line delimiters.

Such files sometimes are not even plain-text files. Sometimes, these files have an alternate character encoding known as extended binary coded decimal interchange code (EBCDIC). The EBCDIC encoding was created in the 1960s by IBM for its mainframes.

Because EBCDIC is a different character encoding, we cannot even show you a printed example; it would be unreadable. Nevertheless, Stata can convert EBCDIC files to ASCII (see [D] `filefilter`) and can read data from EBCDIC files.
If you have a data file encoded with EBCDIC, you undoubtedly also have a description of it from which you can create a dictionary that includes the LRECL of the file (EBCDIC files do not typically have end-of-line delimiters) and the character positions of the fields in the file. You create a dictionary for an EBCDIC file just as you would for a plain-text file, using the Do-file Editor or another text editor, and being sure to use the \_lrecl() directive in the dictionary to specify the LRECL. You then simply specify the ebcdic option for infile, and Stata will convert the characters in the file from EBCDIC to ASCII on the fly:

```
.infile using mydict, ebcdic
```

Also see

[D] infile (free format) — Import unformatted text data
[D] infix (fixed format) — Import text data in fixed format
[D] export — Overview of exporting data from Stata
[D] import — Overview of importing data into Stata
[U] 22 Entering and importing data
Title

infile (free format) — Import unformatted text data

Description

infile reads into memory from a disk a dataset that is not in Stata format.

Here we discuss using infile to read free-format data, meaning datasets in which Stata does not need to know the formatting information. Another variation on infile allows reading fixed-format data; see [D] infile (fixed format). Yet another alternative is import delimited, which is easier to use if your data are tab- or comma-separated and contain 1 observation per line. Stata has other commands for reading data, too. If you are not certain that infile will do what you are looking for, see [D] import and [U] 22 Entering and importing data.

After the data are read into Stata, they can be saved in a Stata-format dataset; see [D] save.

Quick start

Import unformatted text data from mydata1.raw, and name the imported float variables v1, v2, and v3

    infile v1 v2 v3 using mydata1

As above, but skip 1 variable in the original file between v1 and v2

    infile v1 _skip(1) v2 v3 using mydata1

As above, and indicate that v1 is a byte variable, v2 is a string variable of length 30, and v3 is a double variable

    infile byte v1 _skip(1) str30 v2 double v3 using mydata1

Also read v4 as a double

    infile byte v1 _skip(1) str30 v2 double(v3 v4) using mydata1

Import unformatted text data from mydata2.raw where 74 observations on v1, v2, and v3 are stored in rows instead of columns

    infile v1 v2 v3 using mydata2, byvariable(74)

As above, but import mydata2.csv

    infile v1 v2 v3 using mydata2.csv, byvariable(74)

Menu

File > Import > Unformatted text data
Syntax

```
infile varlist [ _skip[#] ] [ varlist [ _skip[#] ... ] ] using filename [ if ] [ in ]
[ , options ]
```

If `filename` is specified without an extension, `.raw` is assumed. If `filename` contains embedded spaces, remember to enclose it in double quotes.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>clear replace data in memory</td>
</tr>
<tr>
<td>Options</td>
<td>automatic create value labels from nonnumeric data</td>
</tr>
<tr>
<td></td>
<td>byvariable(#) organize external file by variables; # is number of observations</td>
</tr>
</tbody>
</table>

Options

`clear` specifies that it is okay for the new data to replace the data that are currently in memory. To ensure that you do not lose something important, `infile` will refuse to read new data if data are already in memory. `clear` allows `infile` to replace the data in memory. You can also drop the data yourself by typing `drop _all` before reading new data.

`automatic` causes Stata to create value labels from the nonnumeric data it reads. It also automatically widens the display format to fit the longest label.

`byvariable(#)` specifies that the external data file is organized by variables rather than by observations.

All the observations on the first variable appear, followed by all the observations on the second variable, and so on. Time-series datasets sometimes come in this format.

Remarks and examples

This section describes `infile` features for reading data in free or comma-separated–value format. Remarks are presented under the following headings:

- Reading free-format data
- Reading comma-separated data
- Specifying variable types
- Reading string variables
- Skipping variables
- Skipping observations
- Reading time-series data
Reading free-format data

In free format, data are separated by one or more white-space characters—blanks, tabs, or new lines (carriage return, line feed, or carriage-return/line feed combinations). Thus one observation may span any number of lines.

Numeric missing values are indicated by single periods (".").

Example 1

In the file highway.raw, we have information on the accident rate per million vehicle miles along a stretch of highway, the speed limit on that highway, and the number of access points (on-ramps and off-ramps) per mile. Our file contains

```
<table>
<thead>
<tr>
<th>acc_rate</th>
<th>spdlimit</th>
<th>acc_pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.58</td>
<td>55</td>
<td>4.6</td>
</tr>
<tr>
<td>2.86</td>
<td>60</td>
<td>4.4</td>
</tr>
<tr>
<td>1.61</td>
<td>.</td>
<td>2.2</td>
</tr>
<tr>
<td>3.02</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

We can read these data by typing

```
. infile acc_rate spdlimit acc_pts using highway
(4 observations read)
. list
  acc_rate   spdlimit   acc_pts
  1.  4.58   55       4.6
  2.  2.86   60       4.4
  3.  1.61   .        2.2
  4.  3.02   60       4.7
```

The spacing of the numbers in the original file is irrelevant.

Technical note

Missing values need not be indicated by one period. The third observation on the speed limit is missing in example 1. The raw data file indicates this by recording one period. Let’s assume, instead, that the missing value was indicated by the word unknown. Thus the raw data file appears as

```
<table>
<thead>
<tr>
<th>acc_rate</th>
<th>spdlimit</th>
<th>acc_pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.58</td>
<td>55</td>
<td>4.6</td>
</tr>
<tr>
<td>2.86</td>
<td>60</td>
<td>4.4</td>
</tr>
<tr>
<td>1.61</td>
<td>unknown</td>
<td>2.2</td>
</tr>
<tr>
<td>3.02</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Here is the result of infiling these data:

```
. infile acc_rate spdlimit acc_pts using highway
'unknown' cannot be read as a number for spdlimit[3]
(4 observations read)
```
infile warned us that it could not read the word unknown, stored a missing, and then continued to read the rest of the dataset. Thus aside from the warning message, results are unchanged.

Because not all packages indicate missing data in the same way, this feature can be useful when reading data. Whenever infile sees something that it does not understand, it warns you, records a missing, and continues. If, on the other hand, the missing values were recorded not as unknown but as, say, 99, Stata would have had no difficulty reading the number, but it would also have stored 99 rather than missing. To convert such coded missing values to true missing values, see [D] mvencode.

Reading comma-separated data

In comma-separated–value format, data are separated by commas. You may mix comma-separated–value and free formats. Missing values are indicated either by single periods or by multiple commas that serve as placeholders, or both. As with free format, 1 observation may span any number of input lines.

Example 2

We can modify the format of highway.raw used in example 1 without affecting infile’s ability to read it. The dataset can be read with the same command, and the results would be the same if the file instead contained

```
4.58, 55 4.6
2.86, 60, 4.4
1.61,, 2.2
3.02, 60
4.7
```

Specifying variable types

The variable names you type after the word infile are new variables. The syntax for a new variable is

```
[type] new_varname[label_name]
```

A full discussion of this syntax can be found in [U] 11.4 varname and varlists. As a quick review, new variables are, by default, of type float. This default can be overridden by preceding the variable name with a storage type (byte, int, long, float, double, or str#) or by using the set type command. A list of variables placed in parentheses will be given the same type. For example,

```
double(first_var second_var ... last_var)
```

causes first_var second_var ... last_var to all be of type double.

There is also a shorthand syntax for variable names with numeric suffixes. The varlist var1-var4 is equivalent to specifying var1 var2 var3 var4.
Example 3

In the highway example, we could `infile` the data `acc_rate`, `spdlimit`, and `acc_pts` and force the variable `spdlimit` to be of type int by typing

```
.infile acc_rate int spdlimit acc_pts using highway, clear
(4 observations read)
```

We could force all variables to be of type double by typing

```
.infile double(acc_rate spdlimit acc_pts) using highway, clear
(4 observations read)
```

We could call the three variables `v1`, `v2`, and `v3` and make them all of type double by typing

```
.infile double(v1-v3) using highway, clear
(4 observations read)
```

Reading string variables

By explicitly specifying the types, you can read string variables, as well as numeric variables.

Example 4

Typing `infile str20 name age sex using myfile` would read

```
begin myfile.raw
"Sherri Holliday" 25 1
Branton 32 1
"Bill Ross" 27,0
```

or even

```
begin myfile.raw, variation 2
'Sherri Holliday' 25,1 "Branton" 32
1,'Bill Ross', 27,0
```

The spacing is irrelevant, and either single or double quotes may be used to delimit strings. The quotes do not count when calculating the length of strings. Quotes may be omitted altogether if the string contains no blanks or other special characters (anything other than letters, numbers, or underscores).

Typing

```
.infile str20 name age sex using myfile, clear
(3 observations read)
```

makes `name` a `str20` and `age` and `sex` floats. We might have typed

```
.infile str20 name age int sex using myfile, clear
(3 observations read)
```

to make `sex` an int or

```
.infile str20 name int(age sex) using myfile, clear
(3 observations read)
```

to make both `age` and `sex` ints.
 infile (free format) — Import unformatted text data  541

Technical note

infile can also handle nonnumeric data by using value labels. We will briefly review value
labels, but you should see [U] 12.6.3 Value labels for a complete description.

A value label is a mapping from the set of integers to words. For instance, if we had a variable
called sex in our data that represented the sex of the individual, we might code 0 for male and 1 for
female. We could then just remember that every time we see a value of 0 for sex, that observation
refers to a male, whereas 1 refers to a female.

Even better, we could inform Stata that 0 represents males and 1 represents females by typing

 . label define sexfmt 0 "Male" 1 "Female"

Then we must tell Stata that this coding scheme is to be associated with the variable sex. This is
typically done by typing

 . label values sex sexfmt

Thereafter, Stata will print Male rather than 0 and Female rather than 1 for this variable.

Stata has the ability to turn a value label around. It can go not only from numeric codes to words
such as “Male” and “Female” but also from words to numeric codes. We tell infile the value label
that goes with each variable by placing a colon (:) after the variable name and typing the name of the
value label. Before we do that, we use the label define command to inform Stata of the coding.

Let’s assume that we wish to infile a dataset containing the words Male and Female and that
we wish to store numeric codes rather than the strings themselves. This will result in considerable
data compression, especially if we store the numeric code as a byte. We have a dataset named
persons.raw that contains name, sex, and age:

```
begin persons.raw
"Arthur Doyle" Male 22
"Mary Hope" Female 37
"Guy Fawkes" Male 48
"Carrie House" Female 25
end persons.raw
```

Here is how we read and encode it at the same time:

```
. label define sexfmt 0 "Male" 1 "Female"
. infile str16 name sex:sexfmt age using persons
   (4 observations read)
. list
```

<table>
<thead>
<tr>
<th>name</th>
<th>sex</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arthur Doyle</td>
<td>Male</td>
<td>22</td>
</tr>
<tr>
<td>Mary Hope</td>
<td>Female</td>
<td>37</td>
</tr>
<tr>
<td>Guy Fawkes</td>
<td>Male</td>
<td>48</td>
</tr>
<tr>
<td>Carrie House</td>
<td>Female</td>
<td>25</td>
</tr>
</tbody>
</table>

The \texttt{str16} in the \texttt{infile} command applies only to the \texttt{name} variable; \texttt{sex} is a numeric variable, which we can prove by typing

\begin{verbatim}
.list, nolabel
\end{verbatim}

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>sex</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Arthur Doyle</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>2.</td>
<td>Mary Hope</td>
<td>1</td>
<td>37</td>
</tr>
<tr>
<td>3.</td>
<td>Guy Fawkes</td>
<td>0</td>
<td>48</td>
</tr>
<tr>
<td>4.</td>
<td>Carrie House</td>
<td>1</td>
<td>25</td>
</tr>
</tbody>
</table>

\textbf{Technical note}

When \texttt{infile} is directed to use a value label and it finds an entry in the file that does not match any of the codings recorded in the label, it prints a warning message and stores \texttt{missing} for the observation. By specifying the \texttt{automatic} option, you can instead have \texttt{infile} automatically add new entries to the value label.

Say that we have a dataset containing three variables. The first, \texttt{region} of the country, is a character string; the remaining two variables, which we will just call \texttt{var1} and \texttt{var2}, contain numbers. We have stored the data in a file called \texttt{geog.raw}:

\begin{verbatim}
begin geog.raw
"NE" 31.23 87.78
'NChntl' 29.52 98.92
South 29.62 114.69
West 28.28 218.92
NE 17.50 44.33
NChntl 22.51 55.21
end geog.raw
\end{verbatim}

The easiest way to read this dataset is to type

\begin{verbatim}
.infile str6 region var1 var2 using geog
\end{verbatim}

making \texttt{region} a string variable. We do not want to do this, however, because we are practicing for reading a dataset like this containing 20,000 observations. If \texttt{region} were numerically encoded and stored as a byte, there would be a 5-byte saving per observation, reducing the size of the data by 100,000 bytes. We also do not want to bother with first creating the value label. Using the \texttt{automatic} option, \texttt{infile} creates the value label automatically as it encounters new regions.

\begin{verbatim}
infile byte region:regfmt var1 var2 using geog, automatic clear
(6 observations read)
.list, sep(0)
\end{verbatim}

<table>
<thead>
<tr>
<th></th>
<th>region</th>
<th>var1</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>NE</td>
<td>31.23</td>
<td>87.78</td>
</tr>
<tr>
<td>2.</td>
<td>NChntl</td>
<td>29.52</td>
<td>98.92</td>
</tr>
<tr>
<td>3.</td>
<td>South</td>
<td>29.62</td>
<td>114.69</td>
</tr>
<tr>
<td>4.</td>
<td>West</td>
<td>28.28</td>
<td>218.92</td>
</tr>
<tr>
<td>5.</td>
<td>NE</td>
<td>17.5</td>
<td>44.33</td>
</tr>
<tr>
<td>6.</td>
<td>NChntl</td>
<td>22.51</td>
<td>55.21</td>
</tr>
</tbody>
</table>
infile automatically created and defined a new value label called \texttt{regfmt}. We can use the \texttt{label list} command to view its contents:

\begin{verbatim}
. label list regfmt
regfmt:
  1 NE
  2 NCntrl
  3 South
  4 West
\end{verbatim}

The value label need not be undefined before we use \texttt{infile} with the automatic option. If the value label \texttt{regfmt} had been previously defined as

\begin{verbatim}
. label define regfmt 2 "West"
\end{verbatim}

the result of \texttt{label list} after the \texttt{infile} would have been

\begin{verbatim}
regfmt:
  2 West
  3 NE
  4 NCntrl
  5 South
\end{verbatim}

The \texttt{automatic} option is convenient, but there is one reason for using it. Suppose that we had a dataset containing, among other things, information about an individual’s sex. We know that the sex variable is supposed to be coded \texttt{male} and \texttt{female}. If we read the data by using the \texttt{automatic} option and if one of the records contains \texttt{fmlae}, then \texttt{infile} will blindly create a third sex rather than print a warning.

\section*{Skipping variables}

Specifying \texttt{_skip} instead of a variable name directs \texttt{infile} to ignore the variable in that location. This feature makes it possible to extract manageable subsets from large disk datasets. A number of contiguous variables can be skipped by specifying \texttt{_skip(#)}, where \texttt{#} is the number of variables to ignore.

\section*{Example 5}

In the highway example from example 1, the data file contained three variables: \texttt{acc_rate}, \texttt{spdlimit}, and \texttt{acc_pts}. We can read the first two variables by typing

\begin{verbatim}
. infile acc_rate spdlimit _skip using highway
(4 observations read)
\end{verbatim}

We can read the first and last variables by typing

\begin{verbatim}
. infile acc_rate _skip acc_pts using highway, clear
(4 observations read)
\end{verbatim}

We can read the first variable by typing

\begin{verbatim}
. infile acc_rate _skip(2) using highway, clear
(4 observations read)
\end{verbatim}

\texttt{_skip} may be specified more than once. If we had a dataset containing four variables—say, \texttt{a}, \texttt{b}, \texttt{c}, and \texttt{d}—and we wanted to read just \texttt{a} and \texttt{c}, we could type \texttt{infile a _skip c _skip using filename}. 

Skipping observations

Subsets of observations can be extracted by specifying `if exp`, which also makes it possible to extract manageable subsets from large disk datasets. Do not, however, use the `_variable _N` in `exp`. Use the `in range` qualifier to refer to observation numbers within the disk dataset.

> Example 6

Again referring to the highway example, if we type

```
.infile acc_rate spdlimit acc_pts if acc_rate>3 using highway, clear
```

(2 observations read)

only observations for which `acc_rate` is greater than 3 will be infiled. We can type

```
.infile acc_rate spdlimit acc_pts in 2/4 using highway, clear
```

(eof not at end of obs)

(3 observations read)

to read only the second, third, and fourth observations.

Reading time-series data

If you are dealing with time-series data, you may receive datasets organized by variables rather than by observations. All the observations on the first variable appear, followed by all the observations on the second variable, and so on. The `byvariable(#)` option specifies that the external data file is organized in this way. You specify the number of observations in the parentheses, because `infile` needs to know that number to read the data properly. You can also mark the end of one variable’s data and the beginning of another’s data by placing a semicolon (";") in the raw data file. You may then specify a number larger than the number of observations in the dataset and leave it to `infile` to determine the actual number of observations. This method can also be used to read unbalanced data.

> Example 7

We have time-series data on 4 years recorded in the file `time.raw`. The dataset contains information on year, amount, and cost, and is organized by variable:

```
begin time.raw
14 17 25 30
120 135 150
180
end time.raw
```

We can read these data by typing

```
.infile year amount cost using time, byvariable(4) clear
```

(4 observations read)

```
.list
```

<table>
<thead>
<tr>
<th>year</th>
<th>amount</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1980</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>1981</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>1982</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>1983</td>
<td>30</td>
</tr>
</tbody>
</table>
If the data instead contained semicolons marking the end of each series and had no information for amount in 1983, the raw data might appear as

14 17 25 ;
120 135 150
180 ;

We could read these data by typing

```
. infile year amount cost using time, byvariable(100) clear
(4 observations read)
. list
```

<table>
<thead>
<tr>
<th>year</th>
<th>amount</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>14</td>
<td>120</td>
</tr>
<tr>
<td>1981</td>
<td>17</td>
<td>135</td>
</tr>
<tr>
<td>1982</td>
<td>25</td>
<td>150</td>
</tr>
<tr>
<td>1983</td>
<td>.</td>
<td>180</td>
</tr>
</tbody>
</table>

Also see

[D] `infile (fixed format)` — Import text data in fixed format with a dictionary
[D] `export` — Overview of exporting data from Stata
[D] `import` — Overview of importing data into Stata
[U] 22 Entering and importing data
Description

infix reads into memory from a disk dataset that is not in Stata format. infix requires that the data be in fixed-column format. Note that the column is byte based. The number of columns means the number of bytes in the file. The text file *filename* is treated as a stream of bytes, no encoding is assumed. If string data are encoded as ASCII or UTF-8, they will be imported correctly.

In the first syntax, if *using filename* is not specified on the command line and *using filename* is not specified in the dictionary, the data are assumed to begin on the line following the closing brace. infix reads the data in a two-step process. You first create a disk file describing how the data are recorded. You tell infix to read that file—called a dictionary—and from there, infix reads the data. The data can be in the same file as the dictionary or in a different file.

In its second syntax, you tell infix how to read the data right on the command line with no intermediate file.

infile and import delimited are alternatives to infix. infile can also read data in fixed format—see [D] infile (fixed format)—and it can read data in free format—see [D] infile (free format). Most people think that infix is easier to use for reading fixed-format data, but infile has more features. If your data are not fixed format, you can use import delimited; see [D] import delimited. import delimited allows you to specify the source file's encoding and then performs a conversion to UTF-8 encoding during import. If you are not certain that infix will do what you are looking for, see [D] import and [U] 22 Entering and importing data.

Quick start

Read v1 from columns 1 to 6 and v2 from column 7 using mydata.raw

```
infix v1 1-6 v2 7 using mydata
```

As above, but read v1 as a string variable

```
infix str v1 1-6 v2 7 using mydata
```

As above, but for 2-line records with v2 in column 1 of the second line

```
infix 2 lines 1: v1 1-6 2: v2 1 using mydata
```

As above, but for mydata.txt

```
infix 2 lines 1: v1 1-6 2: v2 1 using mydata.txt
```

As above, but with data beginning on line 3

```
infix 3 firstlineoffile 2 lines 1: v1 1-6 2: v2 1 using mydata.txt
```

As above, but with instructions for reading the data contained in dictionary file mydata.dct

```
infix using mydata, using(mydata.txt)
```
Menu

File > Import > Text data in fixed format

Syntax

\[ \text{infix using } dfilename \ [if] \ [in] \ [, \ using(filename_2) \ clear] \]

\[ \text{infix specifications using } filename \ [if] \ [in] \ [, \ clear] \]

If \( dfilename \) is specified without an extension, .dct is assumed. If \( dfilename \) contains embedded spaces, remember to enclose it in double quotes. \( dfilename \), if it exists, contains

\[ \text{begin dictionary file} \]
\[ \text{infix dictionary} \ [using \ filename] \ { \]
\[ \ast \text{ comments preceded by asterisk may appear freely} \]
\[ \text{Specifications} \]
\[ \} \]
\[ \text{(your data might appear here)} \]
\[ \text{end dictionary file} \]

If \( filename \) is specified without an extension, .raw is assumed. If \( filename \) contains embedded spaces, remember to enclose it in double quotes.

\[ \text{specifications is} \]
\[ \# \text{firstlineoffile} \]
\[ \# \text{lines} \]
\[ \#; \]
\[ / \]
\[ [ \text{byte | int | float | long | double | str} ] \text{ varlist [#:][-#]} \]

Options

\[ \text{using(filename_2)} \] specifies the name of a file containing the data. If \text{using()} is not specified, the data are assumed to follow the dictionary in \( dfilename \), or if the dictionary specifies the name of some other file, that file is assumed to contain the data. If \text{using(filename_2)} is specified, \( filename_2 \) is used to obtain the data, even if the dictionary says otherwise. If \( filename_2 \) is specified without an extension, .raw is assumed. If \( filename_2 \) contains embedded spaces, remember to enclose it in double quotes.

\[ \text{clear} \] specifies that it is okay for the new data to replace what is currently in memory. To ensure that you do not lose something important, \text{infix} will refuse to read new data if data are already in memory. \text{clear} allows \text{infix} to replace the data in memory. You can also drop the data yourself by typing drop _all before reading new data.
Specifications

# firstlineoffile (abbreviation first) is rarely specified. It states the line of the file at which
the data begin. You need not specify first when the data follow the dictionary; infix can figure
that out for itself. You can specify first when only the data appear in a file and the first few
lines of that file contain headers or other markers.

first appears only once in the specifications.

# lines states the number of lines per observation in the file. Simple datasets typically have “1
lines”. Large datasets often have many lines (sometimes called records) per observation. lines
is optional, even when there is more than one line per observation, because infix can sometimes
figure it out for itself. Still, if 1 lines is not right for your data, it is best to specify the appropriate
number of lines.

lines appears only once in the specifications.

#: tells infix to jump to line # of the observation. Consider a file with 4 lines, meaning four
lines per observation. 2: says to jump to the second line of the observation. 4: says to jump
to the fourth line of the observation. You may jump forward or backward: infix does not care,
and there is no inefficiency in going forward to 3:, reading a few variables, jumping back to 1:,
reading another variable, and jumping back again to 3:.

You need not ensure that, at the end of your specification, you are on the last line of the observation.
infix knows how to get to the next observation because it knows where you are and it knows
lines, the total number of lines per observation.

#: may appear many times in the specifications.

/ is an alternative to #: / goes forward one line. // goes forward two lines. We do not recommend
using / because #: is better. If you are currently on line 2 of an observation and want to get to
line 6, you could type ////, but your meaning is clearer if you type 6:.

/ may appear many times in the specifications.

[ byte | int | float | long | double | str ] varlist [#:][#-#] instructs infix to read a variable
or, sometimes, more than one.

The simplest form of this is varname #, such as sex 20. That says that variable varname be read
from column # of the current line; that variable sex be read from column 20; and that here, sex
is a one-digit number.

varname #-, such as age 21-23, says that varname be read from the column range specified;
that age be read from columns 21 through 23; and that here, age is a three-digit number.

You can prefix the variable with a storage type. str name 25-44 means read the string variable
name from columns 25 through 44. Note that the string variable name consists of $44 - 25 + 1 = 20$
bytes. If you do not specify str, the variable is assumed to be numeric. You can specify the
numeric subtype if you wish. If you specify str, infix will automatically assign the appropriate
string variable type, str# or strL. Imported strings may be up to 100,000 bytes.

You can specify more than one variable, with or without a type. byte q1-q5 51-55 means read
variables q1, q2, q3, q4, and q5 from columns 51 through 55 and store the five variables as bytes.

Finally, you can specify the line on which the variable(s) appear. age 2:21-23 says that age is
to be obtained from the second line, columns 21 through 23. Another way to do this is to put
together the #: directive with the input-variable directive: 2: age 21-23. There is a difference,
but not with respect to reading the variable age. Let's consider two alternatives:

1: str name 25-44 age 2:21-23 q1-q5 51-55
1: str name 25-44 2: age 21-23 q1-q5 51-55
The difference is that the first directive says that variables q1 through q5 are on line 1, whereas the second says that they are on line 2.

When the colon is put in front, it indicates the line on which variables are to be found when we do not explicitly say otherwise. When the colon is put inside, it applies only to the variable under consideration.

**Remarks and examples**

Remarks are presented under the following headings:

*Two ways to use infix*
*Reading string variables*
*Reading data with multiple lines per observation*
*Reading subsets of observations*

**Two ways to use infix**

There are two ways to use *infix*. One is to type the specifications that describe how to read the fixed-format data on the command line:

```
.infix acc_rate 1-4 spdlimit 6-7 acc_pts 9-11 using highway.raw
```

The other is to type the specifications into a file, begin *highway.dct*, example 1

```
infix dictionary using highway.raw {
  acc_rate 1-4
  spdlimit 6-7
  acc_pts  9-11
}
```

end *highway.dct*, example 1

and then, in Stata, type

```
.infix using highway.dct
```

The method you use makes no difference to Stata. The first method is more convenient if there are only a few variables, and the second method is less prone to error if you are reading a big, complicated file.

The second method allows two variations, the one we just showed—where the data are in another file—and one where the data are in the same file as the dictionary:

```
infix dictionary {
  acc_rate 1-4
  spdlimit 6-7
  acc_pts  9-11
}
```

```
  4.58 55 .46
  2.86 60 4.4
  1.61 2.2
  3.02 60 4.7
```

end *highway.dct*, example 2

Note that in the first example, the top line of the file read *infix dictionary using highway.raw*, whereas in the second, the line reads simply *infix dictionary*. When you do not say where the data are, Stata assumes that the data follow the dictionary.
Example 1

So, let’s complete the example we started. We have a dataset on the accident rate per million vehicle miles along a stretch of highway, the speed limit on that highway, and the number of access points per mile. We have created the dictionary file, highway.dct, which contains the dictionary and the data:

begin highway.dct, example 2

infix dictionary {
    acc_rate 1-4
    spdlimit 6-7
    acc_pts 9-11
}
4.58 55 .46
2.86 60 4.4
1.61 2.2
3.02 60 4.7

end highway.dct, example 2

We created this file outside Stata by using an editor or word processor. In Stata, we now read the data. infix lists the dictionary so that we will know the directives it follows:

. infix using highway
infix dictionary {
    acc_rate 1-4
    spdlimit 6-7
    acc_pts 9-11
}
(4 observations read)
.
list

<table>
<thead>
<tr>
<th>acc_rate</th>
<th>spdlimit</th>
<th>acc_pts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.58</td>
<td>55</td>
<td>.46</td>
</tr>
<tr>
<td>2.86</td>
<td>60</td>
<td>4.4</td>
</tr>
<tr>
<td>1.61</td>
<td>.</td>
<td>2.2</td>
</tr>
<tr>
<td>3.02</td>
<td>60</td>
<td>4.7</td>
</tr>
</tbody>
</table>

We simply typed infix using highway rather than infix using highway.dct. When we do not specify the file extension, infix assumes that we mean .dct.

Reading string variables

When you do not say otherwise in your specification—either in the command line or in the dictionary—infix assumes that variables are numeric. You specify that a variable is a string by placing str in front of its name:

. infix id 1-6 str name 7-36 age 38-39 str sex 41 using employee.raw

or

begin employee.dct

infix dictionary using employee.raw {
    id 1-6
    str name 7-36
    age 38-39
    str sex 40
}

end employee.dct
Reading data with multiple lines per observation

When a dataset has multiple lines per observation—sometimes called multiple records per observation—you specify the number of lines per observation by using `lines`, and you specify the line on which the elements appear by using `#`.

For example,

```
.infix 2 lines 1: id 1-6 str name 7-36 2: age 1-2 str sex 4 using emp2.raw
```

or

```
infix dictionary using emp2.raw {
  2 lines
  1:
    id 1-6
    str name 7-36
  2:
    age 1-2
    str sex 4
}
```

There are many different ways to do the same thing.

Example 2

Consider the following raw data:

```
begin mydata.raw
id income educ / sex age / rcode, answers to questions 1-5
1024 25000 HS
   Male 28
   1 1 9 5 0 3
1025 27000 C
   Female 24
   0 2 2 1 1 3
1035 26000 HS
   Male 32
   1 1 0 3 2 1
1036 25000 C
   Female 25
   1 3 1 2 3 2
end mydata.raw
```

This dataset has three lines per observation, and the first line is just a comment. One possible method for reading these data is

```
infix dictionary using mydata {
  2 first
  3 lines
  1: id 1-4
     income 6-10
     str educ 12-13
  2: str sex 6-11
     int age 13-14
  3: rcode 6
     q1-q5 7-16
}
```

although we prefer

```
begin mydata2.dct

infix dictionary using mydata {
  2 first
  3 lines
    id  1: 1-4
    income 1: 6-10
    str educ 1:12-13
    str sex 2: 6-11
    age 2:13-14
    rcode 3: 6
    q1-q5 3: 7-16
}
```

end mydata2.dct

Either method will read these data, so we will use the first and then explain why we prefer the second.

```
.infix using mydata1
infix dictionary using mydata {
  2 first
  3 lines
    1: id 1-4
       income 6-10
       str educ 12-13
    2: str sex 6-11
       int age 13-14
    3: rcode 6
       q1-q5 7-16
}
(4 observations read)
.list in 1/2

<table>
<thead>
<tr>
<th>id</th>
<th>income</th>
<th>educ</th>
<th>sex</th>
<th>age</th>
<th>rcode</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>q4</th>
<th>q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>25000</td>
<td>HS</td>
<td>Male</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1025</td>
<td>27000</td>
<td>C</td>
<td>Female</td>
<td>24</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
```

What is better about the second is that the location of each variable is completely documented on each line—the line number and column. Because *infix* does not care about the order in which we read the variables, we could take the dictionary and jumble the lines, and it would still work. For instance,

```
infix dictionary using mydata {
  2 first
  3 lines
    str sex 2: 6-11
    rcode 3: 6
    str educ 1:12-13
    age 2:13-14
    id 1: 1-4
    q1-q5 3: 7-16
    income 1: 6-10
}
```

```
end mydata3.dct
```
will also read these data even though, for each observation, we start on line 2, go forward to line 3, jump back to line 1, and end up on line 1. It is not inefficient to do this because \texttt{infix} does not really jump to record 2, then record 3, then record 1 again, etc. \texttt{infix} takes what we say and organizes it efficiently. The order in which we say it makes no difference, except that the order of the variables in the resulting Stata dataset will be the order we specify.

Here the reordering is senseless, but in real datasets, reordering variables is often desirable. Moreover, we often construct dictionaries, realize that we omitted a variable, and then go back and modify them. By making each line complete, we can add new variables anywhere in the dictionary and not worry that, because of our addition, something that occurs later will no longer read correctly.

\section*{Reading subsets of observations}

If you wanted to read only the information about males from some raw data file, you might type

\begin{verbatim}
.infix id 1-6 str name 7-36 age 38-39 str sex 41 using employee.raw 
> if sex=='M'
\end{verbatim}

If your specification was instead recorded in a dictionary, you could type

\begin{verbatim}
.infix using employee.dct if sex=='M'
\end{verbatim}

In another dataset, if you wanted to read just the first 100 observations, you could type

\begin{verbatim}
.infix 2 lines 1: id 1-6 str name 7-36 2: age 1-2 str sex 4 using emp2.raw 
> in 1/100
\end{verbatim}

or if the specification was instead recorded in a dictionary and you wanted observations 101–573, you could type

\begin{verbatim}
.infix using emp2.dct in 101/573
\end{verbatim}

\section*{Also see}

[D] \texttt{infile (fixed format)} — Import text data in fixed format with a dictionary

[D] \texttt{export} — Overview of exporting data from Stata

[D] \texttt{import} — Overview of importing data into Stata

[U] 22 Entering and importing data
input — Enter data from keyboard

Description

input allows you to type data directly into the dataset in memory.

For most users, edit is a better way to add observations to the dataset because it automatically adjusts the storage type of variables, if required, to accommodate new values.

Quick start

Create numeric v1, v2, and v3, and input data directly into Stata

    input v1 v2 v3

As above, but create v1 and v2 as type int, v3 as type byte

    input int (v1 v2) byte v3

Add data on string v4 of length 10

    input str10 v4

Input data for all existing variables

    input

As above, but add observations by typing strings associated with value labels of existing variables instead of numeric data

    input, label

Syntax

    input [varlist] [, automatic label]

Options

automatic causes Stata to create value labels from the nonnumeric data it encounters. It also automatically widens the display format to fit the longest label. Specifying automatic implies label, even if you do not explicitly type the label option.

label allows you to type the labels (strings) instead of the numeric values for variables associated with value labels. New value labels are not automatically created unless automatic is specified.
Remarks and examples

If no data are in memory, you must specify a *varlist* when you type *input*. Stata will then prompt you to enter the new observations until you type *end*.

Example 1

We have data on the accident rate per million vehicle miles along a stretch of highway, along with the speed limit on that highway. We wish to type these data directly into Stata:

```
. input
    nothing to input
    r(104);
```

Typing *input* by itself does not provide enough information about our intentions. Stata needs to know the names of the variables we wish to create.

```
. input acc_rate spdlimit
   acc_rate   spdlimit
     1. 4.58 55
     2. 2.86 60
     3. 1.61 .
     4. end
```

We typed *input acc_rate spdlimit*, and Stata responded by repeating the variable names and prompting us for the first observation. We entered the values for the first two observations, pressing *Return* after each value was entered. For the third observation, we entered the accident rate (1.61), but we entered a period (.) for missing because we did not know the corresponding speed limit for the highway. After entering data for the fourth observation, we typed *end* to let Stata know that there were no more observations.

We can now *list* the data to verify that we have entered the data correctly:

```
. list
```

<table>
<thead>
<tr>
<th>acc_rate</th>
<th>spdlimit</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.58</td>
<td>55</td>
</tr>
<tr>
<td>2.86</td>
<td>60</td>
</tr>
<tr>
<td>1.61</td>
<td>.</td>
</tr>
</tbody>
</table>

Example 2: Adding observations

We now have another observation that we wish to add to the dataset. Typing *input* by itself tells Stata that we wish to add new observations:

```
. input
   acc_rate   spdlimit
     4. 3.02 60
     5. end
```

If you have data in memory and type *input* without a *varlist*, you will be prompted to enter more information on all the variables. This continues until you type *end*.
Stata reminded us of the names of our variables and prompted us for the fourth observation. We entered the numbers 3.02 and 60 and pressed Return. Stata then prompted us for the fifth observation. We could add as many new observations as we wish. Because we needed to add only 1 observation, we typed `end`. Our dataset now has 4 observations.

You may add new variables to the data in memory by typing `input` followed by the names of the new variables. Stata will begin by prompting you for the first observation, then the second, and so on, until you type `end` or enter the last observation.

**Example 3: Adding variables**

In addition to the accident rate and speed limit, we now obtain data on the number of access points (on-ramps and off-ramps) per mile along each stretch of highway. We wish to enter the new data.

```
. input acc_pts
    acc_pts
    1. 4.6
    2. 4.4
    3. 2.2
    4. 4.7
```

When we typed `input acc_pts`, Stata responded by prompting us for the first observation. There are 4.6 access points per mile for the first highway, so we entered 4.6. Stata then prompted us for the second observation, and so on. We entered each of the numbers. When we entered the final observation, Stata automatically stopped prompting us—we did not have to type `end`. Stata knows that there are 4 observations in memory, and because we are adding a new variable, it stops automatically.

We can, however, type `end` anytime we wish, and Stata fills the remaining observations on the new variables with `missing`. To illustrate this, we enter one more variable to our data and then `list` the result:

```
. input junk
    junk
    1. 1
    2. 2
    3. end
```

```
. list
```

```
<table>
<thead>
<tr>
<th>acc_rate</th>
<th>spdlimit</th>
<th>acc_pts</th>
<th>junk</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.58</td>
<td>55</td>
<td>4.6</td>
<td>1</td>
</tr>
<tr>
<td>2.86</td>
<td>60</td>
<td>4.4</td>
<td>2</td>
</tr>
<tr>
<td>1.61</td>
<td>.</td>
<td>2.2</td>
<td>.</td>
</tr>
<tr>
<td>3.02</td>
<td>60</td>
<td>4.7</td>
<td>.</td>
</tr>
</tbody>
</table>
```

You can input string variables by using `input`, but you must remember to indicate explicitly that the variables are strings by specifying the type of the variable before the variable’s name.
example 4: Inputting string variables

String variables are indicated by the types str# or strL. For str#, # represents the storage length, or maximum length, in bytes of the variable. You can create variables up to str2045. You can create strL variables of arbitrary length.

For text with only plain ASCII characters, the length in bytes is equivalent to the number of characters displayed. For instance, a str4 variable has a maximum length of 4, meaning that it can contain the strings a, ab, abc, and abcd, but not abcdE. Unicode characters beyond the plain ASCII range take 2, 3, or 4 bytes each. Thus the same str4 variable could contain the strings á, âb, and âbc, but not âbcd because á takes two bytes to store. If you are using input with strings containing Unicode characters, you should allow extra room in your str# specification. See [U] 12.4.2 Handling Unicode strings.

Strings shorter than the maximum length can be stored in the variable, but strings longer than the maximum length cannot.

Although a str80 variable can store strings shorter than 80 characters, you should not make all your string variables str80 because Stata allocates space for strings on the basis of their maximum length. Thus doing so would waste the computer’s memory.

Let’s assume that we have no data in memory and wish to enter the following data:

```
. input str16 name age str6 sex
   name     age     sex
   1. "Arthur Doyle" 22 male
   2. "Mary Hope" 37 "female"
   3. Guy Fawkes 48 male
   'Fawkes' cannot be read as a number
   3. "Guy Fawkes" 48 male
   4. "Kriste Yeager" 25 female
   5. end
```

We first typed input str16 name age str6 sex, meaning that name is to be a str16 variable and sex a str6 variable. Because we did not specify anything about age, Stata made it a numeric variable.

Stata then prompted us to enter our data. On the first line, the name is Arthur Doyle, which we typed in double quotes. The double quotes are not really part of the string; they merely delimit the beginning and end of the string. We followed that with Mr. Doyle’s age, 22, and his sex, male. We did not bother to type double quotes around the word male because it contained no blanks or special characters. For the second observation, we typed the double quotes around female; it changed nothing.

In the third observation, we omitted the double quotes around the name, and Stata informed us that Fawkes could not be read as a number and reprompted us for the observation. When we omitted the double quotes, Stata interpreted Guy as the name, Fawkes as the age, and 48 as the sex. This would have been okay with Stata, except for one problem: Fawkes looks nothing like a number, so Stata complained and gave us another chance. This time, we remembered to put the double quotes around the name.
Stata was satisfied, and we continued. We entered the fourth observation and typed `end`. Here is our dataset:

```
. list

+-----+--------+-------+
<p>| 1.   |       |       |</p>
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arthur Doyle</td>
<td>22</td>
<td>male</td>
</tr>
<tr>
<td>Mary Hope</td>
<td>37</td>
<td>female</td>
</tr>
<tr>
<td>Guy Fawkes</td>
<td>48</td>
<td>male</td>
</tr>
<tr>
<td>Kriste Yeager</td>
<td>25</td>
<td>female</td>
</tr>
</tbody>
</table>
+-----+--------+-------+
```

Example 5: Specifying numeric storage types

Just as we indicated the string variables by placing a storage type in front of the variable name, we can indicate the storage type of our numeric variables as well. Stata has five numeric storage types: `byte`, `int`, `long`, `float`, and `double`. When you do not specify the storage type, Stata assumes that the variable is a `float`. See the definitions of numbers in [U] 12 Data.

There are two reasons for explicitly specifying the storage type: to induce more precision or to conserve memory. The default type `float` has plenty of precision for most circumstances because Stata performs all calculations in double precision, no matter how the data are stored. If you were storing nine-digit Social Security numbers, however, you would want to use a different storage type, or the last digit would be rounded. `long` would be the best choice; `double` would work equally well, but it would waste memory.

Sometimes you do not need to store a variable as `float`. If the variable contains only integers between $-32,767$ and $32,740$, it can be stored as an `int` and would take only half the space. If a variable contains only integers between $-127$ and $100$, it can be stored as a `byte`, which would take only half again as much space. For instance, in example 4 we entered data for `age` without explicitly specifying the storage type; hence, it was stored as a `float`. It would have been better to store it as a `byte`. To do that, we would have typed

```
. input str16 name byte age str6 sex
```

Stata understands several shorthands. For instance, typing

```
. input int(a b) c
```

allows you to input three variables—a, b, and c—and makes both a and b ints and c a float. Remember, typing
input — Enter data from keyboard

would make a an int but both b and c floats. Typing

    . input a long b double(c d) e

would make a a float, b a long, c and d doubles, and e a float.

Stata has a shorthand for variable names with numeric suffixes. Typing v1-v4 is equivalent to typing v1 v2 v3 v4. Thus typing

    . input int(v1-v4)

inputs four variables and stores them as ints.

Technical note

The rest of this section deals with using input with value labels. If you are not familiar with value labels, see [U] 12.6.3 Value labels.

Value labels map numbers into words and vice versa. There are two aspects to the process. First, we must define the association between numbers and words. We might tell Stata that 0 corresponds to Male and 1 corresponds to Female by typing label define sexlbl 0 "Male" 1 "Female". The correspondences are named, and here we have named the 0 ↔ Male 1 ↔ Female correspondence sexlbl.

Next we must associate this value label with a variable. If we had already entered the data and the variable were called sex, we would do this by typing label values sex sexlbl. We would have entered the data by typing 0s and 1s, but at least now when we list the data, we would see the words rather than the underlying numbers.

We can do better than that. After defining the value label, we can associate the value label with the variable at the time we input the data and tell Stata to use the value label to interpret what we type:

    . label define sexlbl 0 "Male" 1 "Female"
    . input str16 name byte(age sex:sexlbl), label
        name     age     sex
    1. "Arthur Doyle" 22 male
    2. "Mary Hope" 37 "female"
    3. "Guy Fawkes" 48 male
    4. "Kriste Yeager" 25 female
    5. end

After defining the value label, we typed our input command. We added the label option at the end of the command, and we typed sex:sexlbl for the name of the sex variable. The byte(...) around age and sex:sexlbl was not really necessary; it merely forced both age and sex to be stored as bytes.

Let’s first decipher sex:sexlbl. sex is the name of the variable we want to input. The :sexlbl part tells Stata that the new variable is to be associated with the value label named sexlbl. The label option tells Stata to look up any strings we type for labeled variables in their corresponding value label and substitute the number when it stores the data. Thus when we entered the first observation of our data, we typed male for Mr. Doyle’s sex, even though the corresponding variable is numeric. Rather than complaining that “"male" could not be read as a number”, Stata accepted what we typed, looked up the number corresponding to male, and stored that number in the data.
That Stata has actually stored a number rather than the words *male* or *female* is almost irrelevant. Whenever we list the data or make a table, Stata will use the words *male* and *female* just as if those words were actually stored in the dataset rather than their numeric codings:

```
. list

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arthur Doyle</td>
<td>22</td>
<td>male</td>
</tr>
<tr>
<td>Mary Hope</td>
<td>37</td>
<td>female</td>
</tr>
<tr>
<td>Guy Fawkes</td>
<td>48</td>
<td>male</td>
</tr>
<tr>
<td>Kriste Yeager</td>
<td>25</td>
<td>female</td>
</tr>
</tbody>
</table>
```

```
. tabulate sex

<table>
<thead>
<tr>
<th>sex</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>male</td>
<td>2</td>
<td>50.00</td>
<td>50.00</td>
</tr>
<tr>
<td>female</td>
<td>2</td>
<td>50.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
```

It is only almost irrelevant because we can use the underlying numbers in statistical analyses. For instance, if we were to ask Stata to calculate the mean of *sex* by typing `summarize sex`, Stata would report 0.5. We would interpret that to mean that one-half of our sample is female.

Value labels are permanently associated with variables, so once we associate a value label with a variable, we never have to do so again. If we wanted to add another observation to these data, we could type

```
. input, label

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Mark Esman&quot;</td>
<td>26</td>
<td>male</td>
</tr>
</tbody>
</table>
```

The automatic option automates the definition of the value label. In the previous example, we informed Stata that *male* corresponds to 0 and *female* corresponds to 1 by typing `label define sexlbl 0 "Male" 1 "Female"`. It was not necessary to explicitly specify the mapping. Specifying the automatic option tells Stata to interpret what we type as follows:

First, see if the value is a number. If so, store that number and be done with it. If it is not a number, check the value label associated with the variable in an attempt to interpret it. If an interpretation exists, store the corresponding numeric code. If one does not exist, add a new numeric code corresponding to what was typed. Store that new number and update the value label so that the new correspondence is never forgotten.
We can use these features to reenter our age and sex data. Before reentering the data, we drop _all and label drop _all to prove that we have nothing up our sleeve:

```
. drop _all
. label drop _all
. input str16 name byte(age sex:sexlbl), automatic
   name      age     sex
  1. "Arthur Doyle" 22 male
  2. "Mary Hope" 37 "female"
  3. "Guy Fawkes" 48 male
  4. "Kriste Yeager" 25 female
  5. end
```

We previously defined the value label sexlbl so that Male corresponded to 0 and Female corresponded to 1. The label that Stata automatically created is slightly different but is just as good:

```
. label list sexlbl
sexlbl:
    1 Male
    2 Female
```

Reference


Also see

[D] edit — Browse or edit data with Data Editor
[D] import — Overview of importing data into Stata
[D] save — Save Stata dataset
[U] 22 Entering and importing data
insobs — Add or insert observations

Description

insobs inserts new observations into the dataset. The number of new observations to insert is specified by obs. This command is primarily used by the Data Editor and is of limited use in other contexts. A more popular alternative for programmers is set obs; see [D] obs.

If option before(inspos) or after(inspos) is specified, the new observations are inserted into the middle of the dataset, and the insert position is controlled by inspos. Note that inspos must be a positive integer between 1 and the total number of observations _N_. If the dataset is empty, before() and after() may not be specified.

Menu

Data > Create or change data > Add or insert observations

Syntax

Add new observations at the end of the dataset

insobs obs

Insert new observations into the middle of the dataset

insobs obs, before(inspos)|after(inspos)

Options

before(inspos) and after(inspos) inserts new observations before and after, respectively, inspos into the dataset. These options are primarily used by the Data Editor and are of limited use in other contexts. A more popular alternative for most users is order; see [D] order.

Remarks and examples

Example 1

insobs can be useful for creating artificial datasets. For instance, if we wanted to create a new dataset with 100 observations, we could type
. clear
. insobs 100
(100 observations added)

Example 2

We are using auto.dta, but for our specific example, we need the dataset to have more observations than those provided in this dataset. To solve this problem, we could type

. sysuse auto, clear
(1978 automobile data)
. insobs 10
(10 observations added)

Typing `insobs` without an option adds the observations at the end of the dataset. Say that instead of the end, we wanted to add five new observations before observation 20. We would type

. sysuse auto, clear
(1978 automobile data)
. insobs 5, before(20)
(5 observations added)

Acknowledgment

This command was inspired by `insob`, which was written by Bas Straathof of CPB Netherlands Bureau for Economic Policy Analysis.

Also see

[D] edit — Browse or edit data with Data Editor
[D] obs — Increase the number of observations in a dataset
The `inspect` command provides a quick summary of a numeric variable that differs from the summary provided by `summarize` or `tabulate`. It reports the number of negative, zero, and positive values; the number of integers and nonintegers; the number of unique values; and the number of `missing`; and it produces a small histogram. Its purpose is not analytical but is to allow you to quickly gain familiarity with unknown data.

**Quick start**

- Summary of all numeric variables in the dataset
  ```
  inspect
  ```
- Summary of v1 for each level of catvar
  ```
  bysort catvar: inspect v1
  ```
- Summary of v1 if v2 is greater than 30
  ```
  inspect v1 if v2 > 30
  ```

**Menu**

Data > Describe data > Inspect variables

**Syntax**

```
inspect [varlist] [if] [in]
```

by and collect are allowed; see [U] 11.1.10 Prefix commands.
Remarks and examples

Typing `inspect` by itself produces an inspection for all the variables in the dataset. If you specify a `varlist`, an inspection of just those variables is presented.

> Example 1

`inspect` is not a replacement or substitute for `summarize` and `tabulate`. It is instead a data management or information tool that lets us quickly gain insight into the values stored in a variable.

For instance, we receive data that purport to be on automobiles, and among the variables in the dataset is one called `mpg`. Its variable label is `Mileage (mpg)`, which is surely suggestive. We inspect the variable,

```
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. inspect mpg
```

```
mpg: Mileage (mpg)                      Number of observations
    #  Negative   -   -   -
    #  Zero      -   -   -
    #  Positive  74  74  -
    #  #  Total   74  74  -
    #  #  #  Missing -
     12  41  74
```

and we discover that the variable is never `missing`; all 74 observations in the dataset have some value for `mpg`. Moreover, the values are all positive and are all integers, as well. Among those 74 observations are 21 unique (different) values. The variable ranges from 12 to 41, and we are provided with a small histogram that suggests that the variable appears to be what it claims.

> Example 2

Bob, a coworker, presents us with some census data. Among the variables in the dataset is one called `region`, which is labeled `Census region` and is evidently a numeric variable. We inspect this variable:

```
. use https://www.stata-press.com/data/r17/bobsdata
(1980 Census data by state)
. inspect region
```

```
region: Census region                      Number of observations
    #  Negative   -   -   -
    #  Zero      -   -   -
    #  Positive  50  50  -
    #  #  Total   50  50  -
    #  #  #  Missing -
     1   5  50
```

region is labeled but 1 value is NOT documented in the label.
In this dataset something may be wrong. The variable `region` takes on five unique values. The variable has a value label, however, and one of the observed values is not documented in the label. Perhaps there is a typographical error.

**Example 3**

There was indeed an error. Bob fixes it and returns the data to us. Here is what `inspect` produces now:

```stata
.use https://www.stata-press.com/data/r17/census
(1980 Census data by state)
.inspect region
```

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Integers</th>
<th>Nonintegers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Zero</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Positive</td>
<td>50</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>50</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>Missing</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(4 unique values)

`region` is labeled and all values are documented in the label.

**Example 4**

We receive data on the climate in 956 U.S. cities. The variable `tempjan` records the Average January temperature in degrees Fahrenheit. The results of `inspect` are

```stata
.use https://www.stata-press.com/data/r17/citytemp
(City temperature data)
.inspect tempjan
```

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Integers</th>
<th>Nonintegers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Zero</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Positive</td>
<td>954</td>
<td>78</td>
<td>876</td>
</tr>
<tr>
<td>Total</td>
<td>954</td>
<td>78</td>
<td>876</td>
</tr>
<tr>
<td>Missing</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(956 unique values)

In two of the 956 observations, `tempjan` is `missing`. Of the 954 cities that have a recorded `tempjan`, all are positive, and 78 of them are integer values. `tempjan` varies between 2.2 and 72.6. There are more than 99 unique values of `tempjan` in the dataset. (Stata stops counting unique values after 99.)
**Stored results**

`inspect` stores the following in `r()`:

Scalars
- `r(N)` number of observations
- `r(N_neg)` number of negative observations
- `r(N_0)` number of observations equal to 0
- `r(N_pos)` number of positive observations
- `r(N_negint)` number of negative integer observations
- `r(N_posint)` number of positive integer observations
- `r(N_unique)` number of unique values or . if more than 99
- `r(NUndoc)` number of undocumented values or . if not labeled

**Also see**

[D] codebook — Describe data contents
[D] compare — Compare two variables
[D] describe — Describe data in memory or in file
[D] ds — Compactly list variables with specified properties
[D] isid — Check for unique identifiers
[R] lv — Letter-value displays
[R] summarize — Summary statistics
[R] table — Table of frequencies, summaries, and command results
[R] tabulate oneway — One-way table of frequencies
[R] tabulate, summarize() — One- and two-way tables of summary statistics
[R] tabulate twoway — Two-way table of frequencies
ipolate — Linearly interpolate (extrapolate) values

Description

ipolate creates in `newvar` a linear interpolation of `yvar` on `xvar` for missing values of `yvar`. Because interpolation requires that `yvar` be a function of `xvar`, `yvar` is also interpolated for tied values of `xvar`. When `yvar` is not missing and `xvar` is neither missing nor repeated, the value of `newvar` is just `yvar`.

Quick start

Create `y2` containing a linear interpolation of `y1` on `x` for observations with missing values of `y1` or tied values of `x`

```stata
ipolate y1 x, generate(y2)
```

As above, but use interpolation and extrapolation

```stata
ipolate y1 x, generate(y2) epolate
```

As above, but perform calculation separately for each level of `catvar`

```stata
by catvar: ipolate y1 x, generate(y2) epolate
```

Menu

Data > Create or change data > Other variable-creation commands > Linearly interpolate/extrapolate values
Syntax

```
ipolate yvar xvar [if] [in], generate(newvar) [epolate]
```

by is allowed; see [D] by.

Options

generate(newvar) is required and specifies the name of the new variable to be created.
epolate specifies that values be both interpolated and extrapolated. Interpolation only is the default.

Remarks and examples

Example 1

We have data points on y and x, although sometimes the observations on y are missing. We believe that y is a function of x, justifying filling in the missing values by linear interpolation:

```
. use https://www.stata-press.com/data/r17/ipolxmpl1
. list, sep(0)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>.</td>
</tr>
<tr>
<td>3.5</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
</tr>
</tbody>
</table>
```

```
. ipolate y x, gen(y1)
(1 missing value generated)
. ipolate y x, gen(y2) epolate
. list, sep(0)

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>y1</th>
<th>y2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>.</td>
<td>4.5</td>
<td>4.5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>.</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>.</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>
```
Example 2

We have a dataset of circulations for 10 magazines from 1980 through 2003. The identity of the magazines is recorded in `magazine`, circulation is recorded in `circ`, and the year is recorded in `year`. In a few of the years, the circulation is not known, so we want to fill it in by linear interpolation.

```
. use https://www.stata-press.com/data/r17/ipolxmpl2, clear
. by magazine: ipolate circ year, gen(icirc)
```

When the `by` prefix is specified, interpolation is performed separately for each group.

Methods and formulas

The value $y$ at $x$ is found by finding the closest points $(x_0, y_0)$ and $(x_1, y_1)$, such that $x_0 < x$ and $x_1 > x$ where $y_0$ and $y_1$ are observed, and calculating

$$y = \frac{y_1 - y_0}{x_1 - x_0} (x - x_0) + y_0$$

If `epolate` is specified and if $(x_0, y_0)$ and $(x_1, y_1)$ cannot be found on both sides of $x$, the two closest points on the same side of $x$ are found, and the same formula is applied.

If there are multiple observations with the same value for $x_0$, then $y_0$ is taken as the average of the corresponding $y$ values for those observations. $(x_1, y_1)$ is handled in the same way.

Reference


Also see

[MI] *mi impute* — Impute missing values
**Description**

`isid` checks whether the specified variables uniquely identify the observations.

**Quick start**

Verify that `idvar` uniquely identifies observations
```
isid idvar
```

Verify that `idvar` uniquely identifies observations within panels identified by `pvar`
```
isid idvar pvar
```

Same as above
```
isid pvar idvar
```

As above, and indicate that the data should be sorted by `pvar` and `idvar`
```
isid pvar idvar, sort
```

Verify that `idvar` uniquely identifies observations in `mydata.dta`
```
isid idvar using mydata.dta
```

**Menu**

Data > Data utilities > Check for unique identifiers

**Syntax**

```
isid varlist [using filename] [, sort missok]
```

**Options**

`sort` indicates that the dataset be sorted by `varlist`.

`missok` indicates that missing values are permitted in `varlist`. 
Remarks and examples

➤ Example 1

Suppose that we want to check whether the mileage ratings (mpg) uniquely identify the observations in our auto dataset.

```
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. isid mpg
variable mpg does not uniquely identify the observations
r(459);
```

`isid` returns an error and reports that there are multiple observations with the same mileage rating. We can locate those observations manually:

```
. sort mpg
. by mpg: generate nobs = _N
. list make mpg if nobs >1, sepby(mpg)
```

```
make       mpg
----------
  1.       Linc. Mark V       12
  2.       Linc. Continental   12
          (output omitted)
  68.      Mazda GLC           30
  69.      Dodge Colt          30
  72.      Datsun 210          35
  73.      Subaru              35
```

➤ Example 2

`isid` is useful for checking a time-series panel dataset. For this type of dataset, we usually need two variables to identify the observations: one that labels the individual IDs and another that labels the periods. Before we set the data using `tsset`, we want to make sure that there are no duplicates with the same panel ID and time. Suppose that we have a dataset that records the yearly gross investment of 10 companies for 20 years. The panel and time variables are `company` and `year`.

```
. use https://www.stata-press.com/data/r17/grunfeld, clear
. isid company year
isid reports no error, so the two variables company and year uniquely identify the observations. Therefore, we should be able to `tsset` the data successfully:
```

```
. tsset company year
Panel variable: company (strongly balanced)
Time variable: year, 1935 to 1954
    Delta: 1 year
```
Technical note

The `sort` option is a convenient shortcut, especially when combined with `using`. The command

```
.isid patient_id date using newdata, sort
```

is equivalent to

```
preserve
use newdata, clear
sort patient_id date
.isid patient_id date
save, replace
restore
```

Also see

[D] `describe` — Describe data in memory or in file
[D] `ds` — Compactly list variables with specified properties
[D] `duplicates` — Report, tag, or drop duplicate observations
[D] `lookfor` — Search for string in variable names and labels
[D] `codebook` — Describe data contents
[D] `inspect` — Display simple summary of data’s attributes
**jdbc — Load, write, or view data from a database with a Java API**

### Description

jdbc allows you to load data from a database, execute SQL statements on a database, and insert data into a database using Java Database Connectivity (JDBC). JDBC is an application programming interface (API) for the programming language Java and defines how a client (Stata) can access a database. jdbc is oriented toward relational databases or nonrelational database-management systems that have rectangular data. NoSQL databases will not work with jdbc.

- **jdbc connect** stores all database connection settings for subsequent jdbc commands.
- **jdbc add** stores database connection settings as a data source name for a Stata session.
- **jdbc remove** removes a stored data source name for a Stata session.
- **jdbc list** displays all stored data source names for a Stata session.
- **jdbc showdbs** produces a list of all databases for a given URL.
- **jdbc showtables** retrieves a list of table names available from a specified database.
- **jdbc describe** lists column names and data types associated with a specified table.
- **jdbc load** reads a database table into Stata's memory. You can load a table specified in the table() option or load an ODBC table generated by an SQL SELECT statement specified in the exec() option.
- **jdbc insert** writes data from memory to a database table. The data can be appended to an existing table or replace an existing table.
- **jdbc exec** allows for most SQL statements to be issued directly to any database. Statements that produce output, such as SELECT, have their output neatly displayed. By using Stata's ado-language, you can also generate SQL commands on the fly to do positional updates or whatever the situation requires.

### Quick start

Store connection settings to database myDB

```stata
jdbc connect, jar("mysql-connector-java-5.1.49.jar") ///
  driverclass("com.mysql.jdbc.Driver") ///
  user("stata") password("stata")
```

List available table names in database myDB

```stata
jdbc showtables
```

Describe the column names and data types in table MyTable from myDB

```stata
jdbc describe "MyTable"
```
Load MyTable into memory from myDB
   jdbc load, table("MyTable")

Syntax

Store JDBC connection settings for all jdbc commands
   jdbc connect { DataSourceName | , connect_options }

Add JDBC connection settings as a data source name for the current Stata session
   jdbc add DataSourceName, connect_options

Remove JDBC connection settings and data source name for the current Stata session
   jdbc remove { DataSourceName | _all }

List stored data source names and URLs for the current Stata session
   jdbc list

List all databases for a given connection
   jdbc show dbs

Retrieve available table names from specified data source
   jdbc show tables [ "SearchString" ]

List column names and data types associated with specified table
   jdbc describe "TableName"

Import data from a database
   jdbc load, { table("TableName") | exec("SqlStmtList") } [ load_options ]

Export data to a database
   jdbc insert [ varlist ] [ if ] [ in ], table("TableName") [ insert_options ]

Allow SQL statements to be issued directly to a database
   jdbc exec "SqlStmtList"

DataSourceName is a name used to store connection settings.
SearchString is a database table name search string; SQL wildcard characters like % and _ are allowed.
TableName is the name of a table in the database.
SqlStmtList may be one valid SQL statement or a list of SQL statements separated by semicolons.
connect_options

* jar("JarFileName")
  JAR file name of JDBC driver
* driverclass("ClassName")
  Java class name for JDBC driver
* url("URL")
  database URL
* user("UserID")
  user ID of user establishing connection
* password("Password")
  password of user establishing connection
  connprop("ConnectionProperty")
  driver-specific connection property

JarFileName is the name of the JDBC driver JAR file.
ClassName is the Java class name stored in the JDBC driver JAR file.
URL is the database URL.
UserID is the user ID.
Password is the user’s password.

Options

Options are presented under the following headings:

Options for jdbc connect and jdbc add
Options for jdbc load
Options for jdbc insert
Options for jdbc connect and jdbc add

jar("JarFileName") specifies the JDBC driver JAR file installed along your ado-path. jar() is required with jdbc add. Also, if DataSourceName is not specified, jar() is required with jdbc connect for jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec to work.

driverclass("ClassName") specifies the Java class name stored in the JDBC driver JAR file installed along your ado-path. driverclass() is required with jdbc add. Also, if DataSourceName is not specified, driverclass() is required with jdbc connect for jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec to work.

url("URL") specifies the URL to the database the user is attempting to establish the connection to. url() is required with jdbc add. Also, if DataSourceName is not specified, url() is required with jdbc connect for jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec to work. The driver URL syntax is as follows:

jdbc:Database_type://Host:Port/Database_name?connection_properties

user("UserID") specifies the user ID of the user attempting to establish the connection to a database. user() is required with jdbc add. Also, if DataSourceName is not specified, user() is required with jdbc connect for jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec to work.

password("Password") specifies the password of the user attempting to establish the connection to a database. password() is required with jdbc add. Also, if DataSourceName is not specified, password() is required with jdbc connect for jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec to work.

connprop("ConnectionProperty") specifies the driver-specific connection properties. A connection property is a key value pair that is separated by a colon and delimited by a semicolon. For example,

jdbc connect, ... connprop("characterEncoding:ISO8859-1;")

These properties can also be set in the url() option.

Options for jdbc load

table("TableName") specifies the name of the table stored in a specified database. Either the table() option or the exec() option—but not both—is required with the jdbc load command.

exec("SqlStmtList") allows you to issue an SQL SELECT statement to generate a table to be read into Stata. Multiple SQL statements can be issued, with the last SQL statement being a SELECT. Each statement should be delimited by a semicolon. For example,

    local sql ///
    "CREATE TEMPORARY TABLE t(a INT, b INT); INSERT INTO t VALUES (1,2); ///
    SELECT * FROM t;";
    jdbc load, exec("'sql'")

An error message is returned if the SQL statements are invalid SQL. Either the table() option or the exec() option—but not both—is required with the jdbc load command.

rows(#) specifies the number of rows to be fetched from the database result set for each network call. This option may help improve command performance. The default is rows(10). Some drivers do not support this feature. Note that setting rows() to a large number might require you to change the amount of heap memory allocated for the JVM with the java set heapmax command.
clear permits the data to be loaded, even if there are data already in memory, and even if that data have changed since the data were last saved.

case(lower|upper|preserve) specifies the case of the variable names after loading. The default is case(preserve).

Options for jdbc insert

table("TableName") specifies the name of the table stored in a specified database.

rows(#) specifies the number of result set rows to be sent to the database for each network call. This option may help improve command performance. The default result set size is 1. This option does not work with datasets that contain strLs. Some drivers do not support this feature. Note that setting the rows(#) to a large number might require you to change the amount of heap memory allocated for the JVM with the java set heapmax command.

overwrite allows data to be dropped from a database table before the Stata data in memory are written to the table. All data from the table are erased, not just the data from the variable columns that will be replaced.

Remarks and examples

jdbc allows you to connect to, load data from, insert data into, and execute queries on a database using JDBC. First, you specify the connection settings with jdbc connect, including the URL for the database you are connecting to and your user ID and password. Thereafter, you can use jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec. These commands allow you to execute statements on a database and load data to and from Stata; they will use the connection information you specified with jdbc connect to open a connection and perform the specified task.

If you will be connecting to multiple databases frequently, you can store the connection settings for each database under a data source name with jdbc add. Then, whenever you wish to connect to a database, simply use jdbc connect, and specify the data source name. This avoids having to specify all the connection information every time you wish to connect to a different database.

Remarks are presented under the following headings:

JDBC drivers
Connecting to a database
Data source names
Exploring a database
Loading data from a database
Inserting data into a database
Executing SQL on a database

JDBC drivers

To use jdbc, you must first download and install your database vendor JDBC driver JAR file. To see information on Stata’s current JDBC implementation, click here.

Once you have downloaded the appropriate driver, you must install the driver along Stata’s ado-path. If the file is compressed, you can use Stata’s unzipfile with the downloaded file to extract the .jar file. Once extracted, place the .jar file along your ado-path so Stata can add it to the Java virtual machine (JVM) class-path. You can use java query to check to see whether Stata has loaded your driver along the JVM class-path.
Most users should place the .jar files in the PERSONAL directory or the current working directory. System administrators may wish to place them in the SITE directory if they have a network installation and want to make them available to all users.

Connecting to a database

jdbc connect stores all database connection settings for commands jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, and jdbc exec. Options jar(), driverclass(), url(), user(), and password() are required, unless you have already saved that information under a data source name and you are specifying that DataSourceName with jdbc connect.

If you try to use these commands before setting your connection properties, you will receive the following error message:

```
.jdbc showtables
Connection failed
JDBC driver class not found
r(681);
```

⚠️ Technical note

Storing your database name, user ID, and password in a Stata do-file, ado-file, or log file can be a security risk. Your database vendor might have software called a wallet that can store this information securely on your machine.

Example 1: Creating a connection

Below, we create a connection string for the JDBC driver in Stata:

```
.jdbc connect, jar("mysql-connector-java-8.0.22.jar")
> driverclass("com.mysql.cj.jdbc.Driver")
> url("jdbc:mysql://localhost:3306/myDB")
> user("stata") password("stata_pass")
```

Going forward, when we issue the jdbc showdbs, jdbc showtables, jdbc describe, jdbc load, jdbc insert, or jdbc exec command, each will use this information to connect to the database myDB.

Example 2: Using macros

You can also use macros to make your do-file more readable and easier to change database settings.

```
.local jar "mysql-connector-java-8.0.22.jar"
.local driverclass "com.mysql.cj.jdbc.Driver"
.local url "jdbc:mysql://localhost:3306/myDB"
.local user "stata"
.local pass "stata_pass"
.jdbc connect, jar("'jar'") driverclass("'driverclass'")
> url("'url'") user("'user'") password("'pass'")
```
Data source names

If you would like to have database connection settings stored and ready for jdbc to use every time you start a Stata session, you can place jdbc add in your profile.do to store these settings; see [GSW] B.3 Executing commands every time Stata is started, [GSM] B.1 Executing commands every time Stata is started, or [GSU] B.1 Executing commands every time Stata is started.

Use jdbc list to see the current session’s stored connection settings and jdbc remove to remove stored settings.

Exploring a database

jdbc showdbs, jdbc showtables, and jdbc describe are used, respectively, to list database names, table names, and table columns of a connection. Use these commands to search for data to load from your connection.

Example 3: Listing table names

jdbc showtables is used to list table names available from a specified database. To list all the tables stored in database myDB, type

```
  . jdbc showtables
  Database: myDB

  Tables
  ___________________________________________________________________________
    auto
  ____________________________
```

Example 4: Listing column names and data types

jdbc describe displays the column names and JDBC data types of the table listed. To describe the auto table, type

```
  . jdbc describe "auto"
  Table: auto

<table>
<thead>
<tr>
<th>column name</th>
<th>column type</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>price</td>
<td>INT</td>
</tr>
<tr>
<td>mpg</td>
<td>INT</td>
</tr>
<tr>
<td>rep78</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>headroom</td>
<td>FLOAT</td>
</tr>
<tr>
<td>trunk</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>weight</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>length</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>turn</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>displacement</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>FLOAT</td>
</tr>
<tr>
<td>domestic</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
```
Loading data from a database

jdbc load is used to load a database table into Stata’s memory; this can be an existing table or a subset of a table created by a series of SQL statements.

➢ Example 5: Loading a table

To load a database table listed in the jdbc showtables output, specify the table name in the table() option.

```
. jdbc load, table("auto")
74 observations loaded
. describe
Contains data
Observations: 74
Variables: 12
```

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display width</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str19</td>
<td>%19s</td>
<td></td>
<td>make</td>
</tr>
<tr>
<td>price</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td>price</td>
</tr>
<tr>
<td>mpg</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td>mpg</td>
</tr>
<tr>
<td>rep78</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>rep78</td>
</tr>
<tr>
<td>headroom</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td>headroom</td>
</tr>
<tr>
<td>trunk</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>trunk</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>weight</td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>length</td>
</tr>
<tr>
<td>turn</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>turn</td>
</tr>
<tr>
<td>displacement</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>displacement</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td>gear_ratio</td>
</tr>
<tr>
<td>domestic</td>
<td>str18</td>
<td>%18s</td>
<td></td>
<td>domestic</td>
</tr>
</tbody>
</table>
```

Sorted by: 
Note: Dataset has changed since last saved.

➢ Example 6: Loading part of a table

If your database table is large and the memory on your computer is limited, it is a good idea to limit the amount of data loaded from the database using a SELECT statement in the exec() option. For example, instead of loading the whole table as we did above, we can just load the mpg column:

```
. jdbc load, exec("SELECT mpg FROM auto;")
74 observations loaded
. describe
Contains data
Observations: 74
Variables: 1
```

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display width</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td>mpg</td>
</tr>
</tbody>
</table>
```

Sorted by: 
Note: Dataset has changed since last saved.
Technical note

When Stata loads a table, data are converted from JDBC data types to Stata data types. Stata does not support all JDBC data types. If the column cannot be read because of incompatible data types, Stata will issue a note and skip a column. The following table lists the supported JDBC data types and their corresponding Stata data types:

<table>
<thead>
<tr>
<th>JDBC data type</th>
<th>Stata data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>byte</td>
</tr>
<tr>
<td>BIT</td>
<td>byte</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>int</td>
</tr>
<tr>
<td>INTEGER</td>
<td>long</td>
</tr>
<tr>
<td>ROWID</td>
<td>str</td>
</tr>
<tr>
<td>BIGINT</td>
<td>str</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>double</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>double</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>DATE</td>
<td>double</td>
</tr>
<tr>
<td>TIME</td>
<td>double</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>double</td>
</tr>
<tr>
<td>TIME_WITH_TIMEZONE</td>
<td>str</td>
</tr>
<tr>
<td>TIMESTAMP_WITH_TIMEZONE</td>
<td>str</td>
</tr>
<tr>
<td>BINARY</td>
<td>strL</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>strL</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>strL</td>
</tr>
<tr>
<td>BLOB</td>
<td>strL</td>
</tr>
<tr>
<td>CHAR</td>
<td>str/strL</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>str/strL</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>str/strL</td>
</tr>
<tr>
<td>NCHAR</td>
<td>str/strL</td>
</tr>
<tr>
<td>NVARCHAR</td>
<td>str/strL</td>
</tr>
<tr>
<td>LONGNVARCHAR</td>
<td>str/strL</td>
</tr>
<tr>
<td>NCLOB</td>
<td>str/strL</td>
</tr>
<tr>
<td>CLOB</td>
<td>str/strL</td>
</tr>
<tr>
<td>STRUCT</td>
<td>skipped</td>
</tr>
<tr>
<td>ARRAY</td>
<td>skipped</td>
</tr>
<tr>
<td>SQLXML</td>
<td>skipped</td>
</tr>
<tr>
<td>NULL</td>
<td>skipped</td>
</tr>
<tr>
<td>OTHER</td>
<td>skipped</td>
</tr>
<tr>
<td>REF_CURSOR</td>
<td>skipped</td>
</tr>
<tr>
<td>JAVA_OBJECT</td>
<td>skipped</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>skipped</td>
</tr>
<tr>
<td>REF</td>
<td>skipped</td>
</tr>
<tr>
<td>DATALINK</td>
<td>skipped</td>
</tr>
</tbody>
</table>

Stata is a UTF-8 application, so all string data should be encoded as UTF-8. This can be set using a driver connection property. Check your database vendor or driver documentation to see how your string data is encoded by default to see whether this property should be set.

```
.jdb connect, ... connprop("characterEncoding:UTF8;")
```
Inserting data into a database

jdbc insert inserts data in memory into a database table. The database table and the Stata varlist must have the same column and variable names, number of columns, and compatible data types for the insert to work correctly. By default, observations are appended to the database table. When you insert data, mapping of the data types are the same as jdbc load, with one exception, Stata bytes. Stata bytes are mapped to SMALLINTs because some database vendors’ (SQLServer) BYTE data type is unsigned.

Example 7: Inserting data into a table

Below, we insert the data in memory into the table auto.

. jdbc insert, table(auto)
74 rows inserted

To replace the table with the data in memory, use the option overwrite.

. jdbc insert, table(auto) overwrite
74 rows affected
74 rows inserted

Executing SQL on a database

You use jdbc exec to execute SQL commands on the database. If an SQL command returns a result set, like SELECT, that result set will be displayed in the Stata Results window.

Example 8: Executing SQL commands

To use jdbc insert, you must have a table already created in your database. If you do not, you can use jdbc exec to create a table in your database. For example, one might create a table in a MySQL database with the SQL command below:

```
#delimit ;
local create_table_sql "CREATE TABLE auto (make varchar(19) NOT NULL, price int, mpg int, rep78 smallint, headroom float, trunk smallint, weight smallint, length smallint, turn smallint, displacement smallint, gear_ratio float, domestic varchar(18));"
jdbc exec "'create_table_sql'"
```

If your SQL statement contains double quotes, you must enclose your statement in compound double quotes, as we did with the statement above.
Stored results

jdbc showdbs stores the following in r():
Scalars
\[ r(n_{dbs}) \] number of databases displayed

jdbc showtables stores the following in r():
Scalars
\[ r(n_{tables}) \] number of tables displayed

jdbc describe stores the following in r():
Scalars
\[ r(k) \] number of columns displayed

jdbc load stores the following in r():
Scalars
\[ r(k) \] number of variables loaded
\[ r(N) \] number of observations loaded

jdbc insert stores the following in r():
Scalars
\[ r(k) \] number of columns inserted
\[ r(N) \] number of rows inserted

Reference

Also see
[D] odbc — Load, write, or view data from ODBC sources
[D] import — Overview of importing data into Stata
[D] export — Overview of exporting data from Stata
joinby — Form all pairwise combinations within groups

Description

joinby joins, within groups formed by `varlist`, observations of the dataset in memory with `filename`, a Stata-format dataset. By `join` we mean to form all pairwise combinations. If `filename` is specified without an extension, `.dta` is assumed.

If `varlist` is not specified, `joinby` takes as `varlist` the set of variables common to the dataset in memory and in `filename`.

Observations unique to one or the other dataset are ignored unless `unmatched()` specifies differently. Whether you load one dataset and join the other or vice versa makes no difference in the number of resulting observations.

If there are common variables between the two datasets, however, the combined dataset will contain the values from the master data for those observations. This behavior can be modified with the `update` and `replace` options.

Quick start

Form pairwise combinations of observations from `mydata1.dta` in memory with those from `mydata2.dta` using all common variables and drop unmatched observations

```
joinby using mydata2
```

As above, but join on `v1`, `v2`, and `v3`

```
joinby v1 v2 v3 using mydata2
```

As above, but include unmatched observations only from `mydata2.dta` and add `_merge` indicating whether the variable was in both datasets or only the using dataset

```
joinby v1 v2 v3 using mydata2, unmatched(using)
```

As above, but include unmatched observations only from `mydata1.dta` and name the variable indicating the source of the observation `newv`

```
joinby v1 v2 v3 using mydata2, unmatched(master) _merge(newv)
```

Replace missing data in `mydata1.dta` with values from `mydata2.dta`

```
joinby v1 v2 v3 using mydata2, update
```

Replace missing and conflicting data in `mydata1.dta` with values from `mydata2.dta`

```
joinby v1 v2 v3 using mydata2, update replace
```
## Syntax

```
joinby [varlist] using filename [, options]
```

### options Description

**Options**

When observations match:

- `update` replace missing data in memory with values from `filename`
- `replace` replace all data in memory with values from `filename`

When observations do not match:

- `unmatched(none)` ignore all; the default
- `unmatched(both)` include from both datasets
- `unmatched(master)` include from data in memory
- `unmatched(using)` include from data in `filename`

- `_merge(varname)` `varname` marks source of resulting observation; default is `_merge`
- `nolabel` do not copy value-label definitions from `filename`

*varlist* may not contain strLs.

### Options

UPDATE varies the action that `joinby` takes when an observation is matched. By default, values from the master data are retained when the same variables are found in both datasets. If `update` is specified, however, the values from the using dataset are retained where the master dataset contains missing.

REPLACE, allowed with `update` only, specifies that nonmissing values in the master dataset be replaced with corresponding values from the using dataset. A nonmissing value, however, will never be replaced with a missing value.

`unmatched(none | both | master | using)` specifies whether observations unique to one of the datasets are to be kept, with the variables from the other dataset set to missing. Valid values are

- `none` ignore all unmatched observations (default)
- `both` include unmatched observations from the master and using data
- `master` include unmatched observations from the master data
- `using` include unmatched observations from the using data

`_merge(varname)` specifies the name of the variable that will mark the source of the resulting observation. The default name is `_merge(_merge)`. To preserve compatibility with earlier versions of `joinby`, `_merge` is generated only if `unmatched` is specified.

`nolabel` prevents Stata from copying the value-label definitions from the dataset on disk into the dataset in memory. Even if you do not specify this option, label definitions from the disk dataset do not replace label definitions already in memory.
Remarks and examples

The following, admittedly artificial, example illustrates `joinby`.

> Example 1

We have two datasets: `child.dta` and `parent.dta`. Both contain a `family_id` variable, which identifies the people who belong to the same family.

```stata
use https://www.stata-press.com/data/r17/child (Data on Children)
describe
Contains data from https://www.stata-press.com/data/r17/child.dta
Observations: 5 Data on Children
Variables: 4 11 Dec 2020 21:08

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>family_id</td>
<td>int</td>
<td>%8.0g</td>
<td>Family ID number</td>
</tr>
<tr>
<td>child_id</td>
<td>byte</td>
<td>%8.0g</td>
<td>Child ID number</td>
</tr>
<tr>
<td>x1</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
</tr>
</tbody>
</table>
Sorted by: family_id

list

<table>
<thead>
<tr>
<th>family_id</th>
<th>child_id</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>3</td>
<td>11</td>
<td>320</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>12</td>
<td>300</td>
</tr>
<tr>
<td>1025</td>
<td>4</td>
<td>10</td>
<td>275</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>13</td>
<td>280</td>
</tr>
<tr>
<td>1027</td>
<td>5</td>
<td>15</td>
<td>210</td>
</tr>
</tbody>
</table>
```

```stata
use https://www.stata-press.com/data/r17/parent (Data on Parents)
describe
Contains data from https://www.stata-press.com/data/r17/parent.dta
Observations: 6 Data on Parents
Variables: 4 11 Dec 2020 03:06

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>family_id</td>
<td>int</td>
<td>%8.0g</td>
<td>Family ID number</td>
</tr>
<tr>
<td>parent_id</td>
<td>float</td>
<td>%9.0g</td>
<td>Parent ID number</td>
</tr>
<tr>
<td>x1</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
</tr>
</tbody>
</table>
Sorted by: |
```
We want to join the information for the parents and their children. The data on parents are in memory, and the data on children are posted at https://www.stata-press.com.

```
. joinby family_id using https://www.stata-press.com/data/r17/child
. describe
Contains data
Observations: 8 Data on Parents
Variables: 6

Variable name Storage Display Value Variable label
      type format label
family_id   int %8.0g Family ID number
parent_id   float %9.0g Parent ID number
      x1 float %9.0g
      x3 float %9.0g
child_id    byte %8.0g Child ID number
      x2   int %8.0g

Sorted by: family_id
Note: Dataset has changed since last saved.
```

```
. list, sepby(family_id) abbrev(12)
```

```
family_id parent_id x1 x3 child_id x2
1. 1025 11 20 643 1 300
2. 1025 11 20 643 3 320
3. 1025 11 20 643 4 275
4. 1025 12 27 721 4 275
5. 1025 12 27 721 1 300
6. 1025 12 27 721 3 320
7. 1026 13 30 760 2 280
8. 1026 14 26 668 2 280
```

1. family_id of 1027, which appears only in child.dta, and family_id of 1030, which appears only in parent.dta, are not in the combined dataset. Observations for which the matching variables are not in both datasets are omitted.
2. The $x_1$ variable is in both datasets. Values for this variable in the joined dataset are the values from `parent.dta`—the dataset in memory when we issued the `joinby` command. If we had `child.dta` in memory and `parent.dta` on disk when we requested `joinby`, the values for $x_1$ would have been those from `child.dta`. Values from the dataset in memory take precedence over the dataset on disk.

Acknowledgment

joinby was written by Jeroen Weesie of the Department of Sociology at Utrecht University, The Netherlands.

Reference

Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. College Station, TX: Stata Press.

Also see

[D] `append` — Append datasets
[D] `cross` — Form every pairwise combination of two datasets
[D] `fillin` — Rectangularize dataset
[D] `merge` — Merge datasets
[D] `save` — Save Stata dataset
[U] 23 Combining datasets
Description

label data attaches a label (up to 80 characters) to the dataset in memory. Dataset labels are displayed when you use the dataset and when you describe it. If no label is specified, any existing label is removed.

label variable attaches a label (up to 80 characters) to a variable. If no label is specified, any existing variable label is removed.

label define creates a value label named `lbname`, which is a set of individual numeric values and their corresponding labels. `lbname` can contain up to 65,536 individual labels; each individual label can be up to 32,000 characters long.

label values attaches a value label to `varlist`. If . is specified instead of `lbname`, any existing value label is detached from that `varlist`. The value label, however, is not deleted. The syntax `label values varname` (that is, nothing following the `varname`) acts the same as specifying the ..

label dir lists the names of value labels stored in memory.
label list lists the names and contents of value labels stored in memory.
label copy makes a copy of an existing value label.
label drop eliminates value labels.
label save saves value label definitions in a do-file. This is particularly useful for value labels that are not attached to a variable because these labels are not saved with the data. By default, .do is the filename extension used.

See [D] label language for information on the label language command.

Quick start

Label the dataset “My data”
```
label data "My data"
```

Label v1 “First variable”
```
label variable v1 "First variable"
```

Define value label named mylabel1
```
label define mylabel1 1 "Value 1" 2 "Value 2"
```

Add labels for values 0 and 3 to mylabel1
```
label define mylabel1 0 "Value 0" 3 "Value 3", add
```

Copy mylabel1 to mylabel2
```
label copy mylabel1 mylabel2
```
Redefine value 0 in mylabel2 to mean “Null”
   label define mylabel2 0 "Null", modify

Apply value label mylabel1 to v1
   label values v1 mylabel1

Save all currently defined value labels to mylabels.do for use with other datasets
   label save using mylabels.do

List names and contents of all value labels
   label list

Drop all value labels
   label drop _all

Menu

**label data**
Data > Data utilities > Label utilities > Label dataset

**label variable**
Data > Variables Manager

**label define**
Data > Variables Manager

**label values**
Data > Variables Manager

**label list**
Data > Data utilities > Label utilities > List value labels

**label copy**
Data > Data utilities > Label utilities > Copy value labels

**label drop**
Data > Variables Manager

**label save**
Data > Data utilities > Label utilities > Save value labels as do-file
Syntax

Label dataset

```
label data ["label"]
```

Label variable

```
label variable varname ["label"]
```

Define value label

```
label define lblname # "label" [# "label" ...] [ , add modify replace nofix]
```

Assign value label to variables

```
label values varlist lblname [ , nofix]
```

Remove value labels

```
label values varlist .
```

List names of value labels

```
label dir
```

List names and contents of value labels

```
label list [lblname [lblname ...]]
```

Copy value label

```
label copy lblname lblname [ , replace]
```

Drop value labels

```
label drop {lblname [lblname ...] | _all}
```

Save value labels in do-file

```
label save [lblname [lblname ...]] using filename [ , replace]
```

Labels for variables and values in multiple languages

```
label language ...
```

(see [D] label language)

where # is an integer or an extended missing value (.a, .b, ... , .z).

collect is allowed with label dir, label language, and label list; see [U] 11.1.10 Prefix commands.
Options

add allows you to add \( \# \leftrightarrow \text{label} \) correspondences to \( \text{lblname} \). If \text{add} is not specified, you may create only new \( \text{lblnames} \). If \text{add} is specified, you may create new \( \text{lblnames} \) or add new entries to existing \( \text{lblnames} \).

modify allows you to modify or delete existing \( \# \leftrightarrow \text{label} \) correspondences and add new correspondences. Specifying \text{modify} implies \text{add}, even if you do not type the \text{add} option.

replace, with \text{label define}, allows an existing value label to be redefined. replace, with \text{label copy}, allows an existing value label to be copied over. replace, with \text{label save}, allows \text{filename} to be replaced.

\text{nofix} prevents display formats from being widened according to the maximum length of the value label. Consider \text{label values myvar mylab}, and say that \text{myvar} has a %9.0g display format right now. Say that the maximum length of the strings in \text{mylab} is 12 characters. \text{label values} would change the format of \text{myvar} from %9.0g to %12.0g. \text{nofix} prevents this.

\text{nofix} is also allowed with \text{label define}, but it is relevant only when you are modifying an existing value label. Without the \text{nofix} option, \text{label define} finds all the variables that use this value label and considers widening their display formats. \text{nofix} prevents this.

Remarks and examples

See [U] 12.6 Dataset, variable, and value labels for a complete description of labels. This entry deals only with details not covered there.

Remarks are presented under the following headings:

Overview
Video examples

Overview

Value labels save us the trouble of having to remember how our variables are coded. For example, if we have a variable recording the region where people live, we might not remember if a value of 1 referred to east or west. We can use \text{label define} to create a value label attaching the labels east and west to numeric values 1 and 2. We can then attach these codings to our region variable with \text{label values} so that our labels will be displayed in the output of certain summary statistics and estimation commands instead of their corresponding numeric values. The suite of \text{label} commands makes it easy to create and manipulate these labels.

Example 1: Creating a value label

Although \text{describe} shows the names of the value labels, those value labels may not exist. Stata does not consider it an error to label the values of a variable with a nonexistent label. When this occurs, Stata still shows the association on \text{describe} but otherwise acts as if the variable's values are unlabeled. This way, you can associate a value label name with a variable before creating the corresponding label. Similarly, you can define labels that you have not yet used.
. use https://www.stata-press.com/data/r17/hbp4
. describe

Contains data from https://www.stata-press.com/data/r17/hbp4.dta
Observations: 1,130
Variables: 7

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage</th>
<th>Display</th>
<th>Value</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>str10</td>
<td>%10s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>city</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>age_grp</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hbp</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>female</td>
<td>byte</td>
<td>%8.0g</td>
<td>sexlbl</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

The dataset is using the value label sexlbl. Let’s define the value label yesno:

. label define yesno 0 "No" 1 "Yes"

. label dir

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>format</th>
<th>label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>yesno</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We have two value labels stored in memory: yesno and sexlbl.

Later, we can display the contents of a value label with the label list command:

. label list yesno

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>format</th>
<th>label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>yesno</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value label yesno labels the values 0 as no and 1 as yes.

If you do not specify the name of the value label on the label list command, Stata lists all the value labels:

. label list

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>format</th>
<th>label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>yesno</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can add new codings to an existing value label by using the add option with the label define command. You can modify existing codings by using the modify option. You can redefine a value label by specifying the replace option.
Example 2: Modifying a value label

The value label yesno codes 0 as no and 1 as yes. You might wish later to add a third coding: 2 as maybe. Typing label define with no options results in an error:

```
.label define yesno 2 maybe
r(110);
```

If you do not specify the add, modify, or replace options, label define can be used only to create new value labels. The add option lets you add codings to an existing value label:

```
.label define yesno 2 maybe, add
.label list yesno
yesno:
 0 No
 1 Yes
 2 maybe
```

Perhaps you have accidentally mislabeled a value. For instance, 2 may not mean “maybe” but may instead mean “don’t know”. add does not allow you to change an existing label:

```
.label define yesno 2 "Don't know", add
r(180);
```

Instead, you would specify the modify option:

```
.label define yesno 2 "Don't know", modify
.label list yesno
yesno:
 0 No
 1 Yes
 2 Don't know
```

In this way, Stata attempts to protect you from yourself. If you type label define with no options, you can only create a new value label—you cannot accidentally change an existing one. If you specify the add option, you can add new labels to an existing value label, but you cannot accidentally change any existing label. If you specify the modify option, which you may not abbreviate, you can change any existing label.

You can even use the modify option to eliminate existing labels. To do this, you map the numeric code to a null string, that is, "":

```
.label define yesno 2 "", modify
.label list yesno
yesno:
 0 No
 1 Yes
```

You can eliminate entire value labels by using the label drop command.
Example 3: Dropping value labels

We currently have two value labels stored in memory—sexlbl and yesno—as shown by the label dir command:

```
. label dir
yesno
sexlbl
```

The dataset that we have in memory uses only one of the labels—sexlbl. `describe` reports that yesno is not being used:

```
. describe
Contains data from https://www.stata-press.com/data/r17/hbp4.dta
Observations: 1,130
Variables: 7 22 Jan 2020 11:12

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>str10</td>
<td>%10s</td>
<td></td>
<td>Record identification number</td>
</tr>
<tr>
<td>city</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>age_grp</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hbp</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>female</td>
<td>byte</td>
<td>%8.0g</td>
<td>sexlbl</td>
<td>sexlbl</td>
</tr>
</tbody>
</table>
```

Sorted by:

Note: Dataset has changed since last saved.

We can eliminate the value label yesno by typing

```
. label drop yesno
. label dir
sexlbl
```

We could eliminate all the value labels in memory by typing

```
. label drop _all
. label dir
```

The value label sexlbl, which no longer exists, was associated with the variable female. Even after dropping the value label, sexlbl is still associated with the variable:

```
. describe
Contains data from https://www.stata-press.com/data/r17/hbp4.dta
Observations: 1,130
Variables: 7 22 Jan 2020 11:12

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>str10</td>
<td>%10s</td>
<td></td>
<td>Record identification number</td>
</tr>
<tr>
<td>city</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>age_grp</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hbp</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>female</td>
<td>byte</td>
<td>%8.0g</td>
<td>sexlbl</td>
<td>sexlbl</td>
</tr>
</tbody>
</table>
```

Sorted by:

Note: Dataset has changed since last saved.
If we wanted to disassociate this nonexistent value label from the variable it was attached to, we could issue the `label values` command without specifying a value label name.

Example 4: Copying a value label

`label copy` is useful when you want to create a new value label that is similar to an existing value label. For example, assume that we currently have the value label `yesno` in memory:

```
. label list yesno
yesno:
   1 Yes
   2 No
```

Assume that we have some variables in our dataset coded with 1 and 2 for “yes” and “no” and that we have some other variables coded with 1 for “yes”, 2 for “no”, and 3 for “maybe”.

We could make a copy of value label `yesno` and then add the new coding to that copy:

```
. label copy yesno yesnomaybe
. label define yesnomaybe 3 "Maybe", add
. label list
yesnomaybe:
   1 Yes
   2 No
   3 Maybe
yesno:
   1 Yes
   2 No
```

Example 5: Saving value labels

Data and variable labels are automatically stored with your dataset when you save it. You might have more value labels stored in memory than are actually used in the dataset, but only those value labels that are attached to variables will be stored with a dataset unless you use `save`’s `orphans` option. Conversely, the `use` command drops all in-memory labels before loading the new dataset along with any labels it might contain. You might want to store a value label not currently in use or move a value label from one dataset to another. The `label save` command allows you to do this.

For example, assume that we currently have the value label `yesnomaybe` in memory:

```
. label list yesnomaybe
yesnomaybe:
   1 Yes
   2 No
   3 Maybe
```

We have a dataset stored on disk called `survey.dta` to which we wish to add this value label. We might use `survey` and then retype the `label define yesnomaybe` command. Retyping the label would not be too tedious here but if the value label in memory mapped, say, the 50 states of the United States, retyping it would be irksome. `label save` provides an alternative:

```
. label save yesnomaybe using ynfile
file ynfile.do saved
```
Typing `label save yesnomaybe using ynfile` caused Stata to create a do-file called `ynfile.do` containing the definition of the `yesnomaybe` value label. Because we did not specify an extension for our file, `.do` was assumed. Also, if we had not specified a value label name, all value labels would have been stored in `ynfile.do`.

To see the contents of the file, we can use the `type` command:

```
    . type ynfile.do
    label define yesnomaybe 1 "Yes", modify
    label define yesnomaybe 2 "No", modify
    label define yesnomaybe 3 "Maybe", modify
```

We can now use our new dataset, `survey.dta`:

```
    . use survey, clear
    (Household survey data)
    . label dir
```

Using the new dataset causes Stata to eliminate all value labels stored in memory. The label `yesnomaybe` is now gone. Because we saved it in the file `ynfile.do`, however, we can get it back by typing either `do ynfile` or `run ynfile`. If we type `do`, we will see the commands in the file execute. If we type `run`, the file will execute silently:

```
    . run ynfile
    . label dir
    yesnomaybe
```

The value label is now restored just as if we had typed it from the keyboard.

---

**Technical note**

You can also use the `label save` command to more easily edit value labels. You can save a label in a file, leave Stata and use your word processor or editor to edit the label, and then return to Stata. Using `do` or `run`, you can load the edited values.

---

**Video examples**

- How to label variables
- How to label the values of categorical variables
Stored results

label list stores the following in r():

Scalars
- \( r(k) \): number of mapped values, including missings
- \( r(min) \): minimum nonmissing value label
- \( r(max) \): maximum nonmissing value label
- \( r(hasmiss) \): 1 if extended missing values labeled, 0 otherwise

label dir stores the following in r():

Macros
- \( r(names) \): names of value labels

References


Also see

[D] label language — Labels for variables and values in multiple languages
[D] labelbook — Label utilities
[D] encode — Encode string into numeric and vice versa
[D] varmanage — Manage variable labels, formats, and other properties
[U] 12.6 Dataset, variable, and value labels
**Description**

`label language` lets you create and use datasets that contain different sets of data, variable, and value labels. A dataset might contain one set in English, another in German, and a third in Spanish. A dataset may contain up to 100 sets of labels.

We will write about the different sets as if they reflect different spoken languages, but you need not use the multiple sets in this way. You could create a dataset with one set of long labels and another set of shorter ones.

One set of labels is in use at any instant, but a dataset may contain multiple sets. You can choose among the sets by typing

```
. label language languagename
```

When other Stata commands produce output (such as `describe` and `tabulate`), they use the currently set language. When you define or modify the labels by using the other `label` commands (see [D] `label`), you modify the current set.

`label language` (without arguments)

lists the available languages and the name of the current one. The current language refers to the labels you will see if you used, say, `describe` or `tabulate`. The available languages refer to the names of the other sets of previously created labels. For instance, you might currently be using the labels in `en` (English), but labels in `de` (German) and `es` (Spanish) may also be available.

`label language languagename`

changes the labels to those of the specified language. For instance, if `label language` revealed that `en`, `de`, and `es` were available, typing `label language de` would change the current language to German.

`label language languagename, new`

allows you to create a new set of labels and collectively name them `languagename`. You may name the set as you please, as long as the name does not exceed 24 characters. If the labels correspond to spoken languages, we recommend that you use the language’s ISO 639-1 two-letter code, such as `en` for English, `de` for German, and `es` for Spanish. A list of codes for popular languages is listed in the appendix below. For a complete list, see [https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes).

`label language languagename, rename`

changes the name of the label set currently in use. If the label set in use were named `default` and you now wanted to change that to `en`, you could type `label language en, rename`.

Our choice of the name `default` in the example was not accidental. If you have not yet used `label language` to create a new language, the dataset will have one language, named `default`.
label language `languagename`, delete
deletes the specified label set. If `languagename` is also the current language, one of the other available languages becomes the current language.

Quick start

Name unnamed default language en for English
`label language en, rename`

Create new set of labels in French named fr
`label language fr, new`

Change current label language from English to French
`label language fr`

List defined languages
`label language`

Delete English label set
`label language en, delete`

Menu

Data > Data utilities > Label utilities > Set label language

Syntax

List defined languages

`label language`

Change labels to specified language name

`label language languagename`

Create new set of labels with specified language name

`label language languagename, new [copy]`

Rename current label set

`label language languagename, rename`

Delete specified label set

`label language languagename, delete`

collect is allowed; see [U] 11.1.10 Prefix commands.
Option

copy is used with `label language, new` and copies the labels from the current language to the new language.

Remarks and examples

Remarks are presented under the following headings:

- Creating labels in the first language
- Creating labels in the second and subsequent languages
- Creating labels from a clean slate
- Creating labels from a previously existing language
- Switching languages
- Changing the name of a language
- Deleting a language
- Appendix: Selected ISO 639-1 two-letter codes

Creating labels in the first language

You can begin by ignoring the `label language` command. You create the data, variable, and value labels just as you would ordinarily; see `[D] label`.

```stata
.label data "1978 automobile data"
.label variable foreign "Car type"
.label values foreign origin
.label define origin 0 "Domestic" 1 "Foreign"
```

At some point—at the beginning, the middle, or the end—rename the language appropriately. For instance, if the labels you defined were in English, type

```stata
.label language en, rename
```

`label language, rename` simply changes the name of the currently set language. You may change the name as often as you wish.

Creating labels in the second and subsequent languages

After creating the first language, you can create a new language by typing

```stata
.label language newlanguagename, new
```

or by typing the two commands

```stata
.label language existinglanguagename
.label language newlanguagename, new copy
```

In the first case, you start with a clean slate: no data, variable, or value labels are defined. In the second case, you start with the labels from `existinglanguagename`, and you can make the changes from there.
Creating labels from a clean slate

To create new labels in the language named de, type
  
  . label language de, new

If you were now to type describe, you would find that there are no data, variable, or value labels. You can define new labels in the usual way:
  
  . label data "1978 automobil daten"
  . label variable foreign "Art auto"
  . label values foreign origin_de
  . label define origin_de 0 "Innen" 1 "Ausländisch"

Creating labels from a previously existing language

It is sometimes easier to start with the labels from a previously existing language, which you can then translate:
  
  . label language en
  . label language de, new copy

If you were now to type describe, you would see the English-language labels, even though the new language is named de. You can then work to translate the labels:
  
  . label data "1978 automobil daten"
  . label variable foreign "Art auto"

Typing describe, you might also discover that the variable foreign has the value label origin. Do not change the contents of the value label. Instead, create a new value label:
  
  . label define origin_de 0 "Innen" 1 "Ausländisch"
  . label values foreign origin_de

Creating value labels with the copy option is no different from creating them from a clean slate, except that you start with an existing set of labels from another language. Using describe can make it easier to translate them.

Switching languages

You can discover the names of the previously defined languages by typing
  
  . label language

You can switch to a previously defined language—say, to en—by typing
  
  . label language en

Changing the name of a language

To change the name of a previously defined language make it the current language and then specify the rename option:
  
  . label language de
  . label language German, rename

You may rename a language as often as you wish:
  
  . label language de, rename
Deleting a language

To delete a previously defined language, such as de, type

```
. label language de, delete
```

The `delete` option deletes the specified language and, if the language was also the currently set language, resets the current language to one of the other languages or to `default` if there are none.

Appendix: Selected ISO 639-1 two-letter codes

You may name languages as you please. You may name German labels `Deutsch`, `German`, `Aleman`, or whatever else appeals to you. For consistency across datasets, if the language you are creating is a spoken language, we suggest that you use the ISO 639-1 two-letter codes. Some of them are listed below, and the full list can be found at [https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes](https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes).
<table>
<thead>
<tr>
<th>Two-letter code</th>
<th>English name of language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar</td>
<td>Arabic</td>
</tr>
<tr>
<td>bn</td>
<td>Bengali</td>
</tr>
<tr>
<td>cs</td>
<td>Czech</td>
</tr>
<tr>
<td>de</td>
<td>German</td>
</tr>
<tr>
<td>do</td>
<td>Danish</td>
</tr>
<tr>
<td>el</td>
<td>Greek</td>
</tr>
<tr>
<td>en</td>
<td>English</td>
</tr>
<tr>
<td>es</td>
<td>Spanish; Castillian</td>
</tr>
<tr>
<td>fa</td>
<td>Persian</td>
</tr>
<tr>
<td>fi</td>
<td>Finnish</td>
</tr>
<tr>
<td>fr</td>
<td>French</td>
</tr>
<tr>
<td>ga</td>
<td>Irish</td>
</tr>
<tr>
<td>he</td>
<td>Hebrew</td>
</tr>
<tr>
<td>hi</td>
<td>Hindi</td>
</tr>
<tr>
<td>is</td>
<td>Icelandic</td>
</tr>
<tr>
<td>it</td>
<td>Italian</td>
</tr>
<tr>
<td>ja</td>
<td>Japanese</td>
</tr>
<tr>
<td>ko</td>
<td>Korean</td>
</tr>
<tr>
<td>lt</td>
<td>Lithuanian</td>
</tr>
<tr>
<td>lv</td>
<td>Latvian</td>
</tr>
<tr>
<td>nl</td>
<td>Dutch; Flemish</td>
</tr>
<tr>
<td>no</td>
<td>Norwegian</td>
</tr>
<tr>
<td>pa</td>
<td>Punjabi</td>
</tr>
<tr>
<td>pl</td>
<td>Polish</td>
</tr>
<tr>
<td>pt</td>
<td>Portuguese</td>
</tr>
<tr>
<td>ro</td>
<td>Romanian; Moldavian</td>
</tr>
<tr>
<td>ru</td>
<td>Russian</td>
</tr>
<tr>
<td>sk</td>
<td>Slovak</td>
</tr>
<tr>
<td>sr</td>
<td>Serbian</td>
</tr>
<tr>
<td>sv</td>
<td>Swedish</td>
</tr>
<tr>
<td>te</td>
<td>Telugu</td>
</tr>
<tr>
<td>tr</td>
<td>Turkish</td>
</tr>
<tr>
<td>uk</td>
<td>Ukrainian</td>
</tr>
<tr>
<td>ur</td>
<td>Urdu</td>
</tr>
<tr>
<td>zh</td>
<td>Chinese</td>
</tr>
</tbody>
</table>

**Stored results**

`label language` without arguments stores the following in `r()`:

Scalars

- `r(k)` number of languages defined

Macros

- `r(languages)` list of languages, listed one after the other
- `r(language)` name of current language
Methods and formulas

This section is included for programmers who wish to access or extend the services `label language` provides.

Language sets are implemented using [P] char. The names of the languages and the name of the current language are stored in

```
_dta[lang_list]        list of defined languages
_dta[lang_c]          currently set language
```

If these characteristics are undefined, results are as if each contained the word “default”. Do not change the contents of the above two macros except by using `label language`.

For each language `language_name` except the current language, data, variable, and value labels are stored in

```
_dta[lang_v_language_name]        data label
varname[lang_v_language_name]    variable label
varname[lang_l_language_name]    value-label name
```

References


Also see

[D] label — Manipulate labels
[D] labelbook — Label utilities
[D] codebook — Describe data contents
Description

`labelbook` displays information for the value labels specified or, if no labels are specified, all the labels in the data.

For multilingual datasets (see [D] label language), `labelbook` lists the variables to which value labels are attached in all defined languages.

`numlabel` prefixes numeric values to value labels. For example, a value mapping of 2 -> "catholic" will be changed to 2 -> "2. catholic". See option `mask()` for the different formats. Stata commands that display the value labels also show the associated numeric values. Prefixes are removed with the `remove` option.

`uselabel` is a programmer's command that reads the value-label information from the currently loaded dataset or from an optionally specified filename.

`uselabel` creates a dataset in memory that contains only that value-label information. The new dataset has four variables named `label`, `lname`, `value`, and `trunc`; is sorted by `lname` value; and has 1 observation per mapping. Value labels can be longer than the maximum string length in Stata; see [R] Limits. The new variable `trunc` contains 1 if the value label is truncated to fit in a string variable in the dataset created by `uselabel`.

`uselabel` complements `label`, `save`, which produces a text file of the value labels in a format that allows easy editing of the value-label texts.

Specifying no list or `-all` is equivalent to specifying all value labels. Value-label names may not be abbreviated or specified with wildcards.

Quick start

Codebook of all currently defined value labels

```
labelbook
```

As above, but only include labels `mylabel1`, `mylabel2`, and `mylabel3`

```
labelbook mylabel1 mylabel2 mylabel3
```

As above, and check that value labels are unique to the first 8 characters

```
labelbook mylabel1 mylabel2 mylabel3, length(8)
```

Prefix numeric values to `mylabel1` with the number separated from the text by a hyphen

```
umlabel mylabel1, add mask("# - ")
```

Remove a prefixed numeric value from a value label when the “# -” mask was used

```
numlabel mylabel1, remove mask("# - ")
```
Menu

**labelbook**
Data > Data utilities > Label utilities > Produce codebook of value labels

**numlabel**
Data > Data utilities > Label utilities > Prepend values to value labels

**uselabel**
Data > Data utilities > Label utilities > Create dataset from value labels

Syntax

**Produce a codebook describing value labels**

```plaintext
labelbook [lblname-list] [ , labelbook_options ]
```

**Prefix numeric values to value labels**

```plaintext
numlabel [lblname-list], {add|remove} [numlabel_options]
```

**Make dataset containing value-label information**

```plaintext
uselabel [lblname-list] [using filename] [ , clear var]
```

**labelbook_options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>alpha</strong></td>
<td>alphabetize label mappings</td>
</tr>
<tr>
<td><strong>length(#)</strong></td>
<td>check if value labels are unique to length #; default is length(12)</td>
</tr>
<tr>
<td><strong>list(#)</strong></td>
<td>list maximum of # mappings; default is list(32000)</td>
</tr>
<tr>
<td><strong>problems</strong></td>
<td>describe potential problems in a summary report</td>
</tr>
<tr>
<td><strong>detail</strong></td>
<td>do not suppress detailed report on variables or value labels</td>
</tr>
</tbody>
</table>

*collect* is allowed with *labelbook*: see [U] 11.1.10 Prefix commands.

**numlabel_options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>add</em></td>
<td>prefix numeric values to value labels</td>
</tr>
<tr>
<td><em>remove</em></td>
<td>remove numeric values from value labels</td>
</tr>
<tr>
<td><strong>mask(str)</strong></td>
<td>mask for formatting numeric labels; default mask is &quot;#. &quot;</td>
</tr>
<tr>
<td><strong>force</strong></td>
<td>force adding or removing of numeric labels</td>
</tr>
<tr>
<td><strong>detail</strong></td>
<td>provide details about value labels, where some labels are prefixed with numbers and others are not</td>
</tr>
</tbody>
</table>

* Either *add* or *remove* must be specified.

**Options**

Options are presented under the following headings:

- Options for *labelbook*
- Options for *numlabel*
- Options for *uselabel*
Options for `labelbook`

- **alpha** specifies that the list of value-label mappings be sorted alphabetically on label. The default is to sort the list on value.
- **length(#)** specifies the minimum length that `labelbook` checks to determine whether shortened value labels are still unique. It defaults to 12, the width used by most Stata commands. `labelbook` also reports whether value labels are unique at their full length.
- **list(#)** specifies the maximum number of value-label mappings to be listed. If a value label defines more mappings, a random subset of # mappings is displayed. By default, `labelbook` displays all mappings. **list(0)** suppresses the listing of the value-label definitions.
- **problems** specifies that a summary report be produced describing potential problems that were diagnosed:
  1. Value label has gaps in mapped values (for example, values 0 and 2 are labeled, while 1 is not)
  2. Value label strings contain leading or trailing blanks
  3. Value label contains duplicate labels, that is, there are different values that map into the same string
  4. Value label contains duplicate labels at length 12
  5. Value label contains numeric → numeric mappings
  6. Value label contains numeric → null string mappings
  7. Value label is not used by variables
- **detail** may be specified only with **problems**. It specifies that the detailed report on the variables or value labels not be suppressed.

Options for `numlabel`

- **add** specifies that numeric values be prefixed to value labels. Value labels that are already numlabeled (using the same mask) are not modified.
- **remove** specifies that numeric values be removed from the value labels. If you added numeric values by using a nondefault mask, you must specify the same mask to remove them. Value labels that are not numlabeled or are numlabeled using a different mask are not modified.
- **mask(str)** specifies a mask for formatting the numeric labels. In the mask, # is replaced by the numeric label. The default mask is "#." so that numeric value 3 is shown as "3.". Spaces are relevant. For the mask 
  \[
  \text{"[\#]"},
  \text{numeric value 3 would be shown as "[3].".}
  \]
- **force** specifies that adding or removing numeric labels be performed, even if some value labels are numlabeled using the mask and others are not. Here only labels that are not numlabeled will be modified.
- **detail** specifies that details be provided about the value labels that are sometimes, but not always, numlabeled using the mask.

Options for `uselabel`

- **clear** permits the dataset to be created, even if the dataset already in memory has changed since it was last saved.
- **var** specifies that the varlists using value label vl be returned in r(vl).
Remarks and examples

Remarks are presented under the following headings:

- `labelbook`
- `Diagnosing problems`
- `numlabel`
- `uselabel`

**labelbook**

`labelbook` produces a detailed report of the value labels in your data. You can restrict the report to a list of labels, meaning that no abbreviations or wildcards will be allowed. `labelbook` is a companion command to [D] `codebook`, which describes the data, focusing on the variables.

For multilingual datasets (see [D] `label language`), `labelbook` lists the variables to which value labels are attached in any of the languages.

**Example 1**

We request a `labelbook` report for value labels in a large dataset on the internal organization of households. We restrict output to three value labels: `agree5` (used for five-point Likert-style items), `divlabor` (division of labor between husband and wife), and `noyes` for simple no-or-yes questions.

```stata
.use https://www.stata-press.com/data/r17/labelbook1
.labelbook agree5 divlabor noyes
```

---

<table>
<thead>
<tr>
<th>Values</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: [1,5]</td>
<td>String length: [8,11]</td>
</tr>
<tr>
<td>N: 5</td>
<td>Unique at full length: yes</td>
</tr>
<tr>
<td>Gaps: no</td>
<td>Unique at length 12: yes</td>
</tr>
<tr>
<td>Missing .*: 0</td>
<td>Null string: no</td>
</tr>
<tr>
<td></td>
<td>Leading/trailing blanks: no</td>
</tr>
<tr>
<td></td>
<td>Numeric -&gt; numeric: no</td>
</tr>
</tbody>
</table>

**Definition**

1. -- disagree
2. - disagree
3. indifferent
4. + agree
5. ++ agree

Variables: rs056 rs057 rs058 rs059 rs060 rs061 rs062 rs063 rs064 rs065 rs066 rs067 rs068 rs069 rs070 rs071 rs072 rs073 rs074 rs075 rs076 rs077 rs078 rs079 rs080 rs081
Value label divlabor

<table>
<thead>
<tr>
<th>Values</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: [1,7]</td>
<td>String length: [7,16]</td>
</tr>
<tr>
<td>N: 7</td>
<td>Unique at full length: yes</td>
</tr>
<tr>
<td>Gaps: no</td>
<td>Unique at length 12: yes</td>
</tr>
<tr>
<td>Missing .*: 0</td>
<td>Null string: no</td>
</tr>
<tr>
<td>Leading/trailing blanks: yes</td>
<td>Numeric -&gt; numeric: no</td>
</tr>
</tbody>
</table>

**Definition**

1. wife only
2. wife >> husband
3. wife > husband
4. equally
5. husband > wife
6. husband >> wife
7. husband only

Variables: hm01_a hm01_b hm01_c hm01_d hm01_e hn19 hn21 hn25_a hn25_b hn25_c hn25_d hn25_e hn27_a hn27_b hn27_c hn27_d hn27_e hn31 hn36 hn38 hn42 hn46_a hn46_b hn46_c hn46_d hn46_e ho01_a ho01_b ho01_c ho01_d ho01_e

Value label noyes

<table>
<thead>
<tr>
<th>Values</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: [1,2]</td>
<td>String length: [2,16]</td>
</tr>
<tr>
<td>N: 4</td>
<td>Unique at full length: yes</td>
</tr>
<tr>
<td>Gaps: no</td>
<td>Unique at length 12: yes</td>
</tr>
<tr>
<td>Missing .*: 2</td>
<td>Null string: no</td>
</tr>
<tr>
<td>Leading/trailing blanks: no</td>
<td>Numeric -&gt; numeric: no</td>
</tr>
</tbody>
</table>

**Definition**

1. no
2. yes
   .a not applicable
   .b ambiguous answer

Variables: hb12 hd01_a hd01_b hd03 hd04_a hd04_b he03_a he03_b hlat hn09_b hn24_a hn34 hn49 hu05_a hu06_1c hu06_2c hx07_a hx08 hlat2 hfinish rh02 rj10_01 rk16_a rk16_b rl01 rl03 rl08_a rl08_b rl09_a rs047 rs048 rs049 rs050 rs051 rs052 rs053 rs054 rs093 rs095 rs096 rs098

The report is largely self-explanatory. Extended missing values are denoted by “.*”. In the definition of the mappings, the leading 12 characters of longer value labels are underlined to make it easier to check that the value labels still make sense after truncation. The following example emphasizes this feature. The option alpha specifies that the value-label mappings be sorted in alphabetical order by the label strings rather than by the mapped values.
. use https://www.stata-press.com/data/r17/labelbook2
. labelbook sports, alpha

Value label sports

<table>
<thead>
<tr>
<th>Values</th>
<th>Labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range: [1,5]</td>
<td>String length: [16,23]</td>
</tr>
<tr>
<td>N: 4</td>
<td>Unique at full length: yes</td>
</tr>
<tr>
<td>Gaps: yes</td>
<td>Unique at length 12: no</td>
</tr>
<tr>
<td>Missing .*: 0</td>
<td>Null string: no</td>
</tr>
</tbody>
</table>

Definition

5 college baseball
4 college basketball
2 professional baseball
1 professional basketball

Variables: active passive

The report includes information about potential problems in the data. These are discussed in greater detail in the next section.

Diagnosing problems

labelbook can diagnose a series of potential problems in the value-label mappings. labelbook produces warning messages for a series of problems:

1. Gaps in the labeled values (for example, values 0 and 2 are labeled, whereas 1 is not) may occur when value labels of the intermediate values have not been defined.

2. Leading or trailing blanks in the value labels may distort Stata output.

3. Stata allows you to define blank labels, that is, the mapping of a number to the empty string. Below we give you an example of the unexpected output that may result. Blank labels are most often the result of a mistaken value-label definition, for instance, the expansion of a nonexistent macro in the definition of a value label.

4. Stata does not require that the labels within each value label consist of unique strings, that is, that different values be mapped into different strings. For instance, you might accidentally define the value label gender as

   label define gender 1 female 2 female

You will probably catch most of the problems, but in more complicated value labels, it is easy to miss the error. labelbook finds such problems and displays a warning.

5. Stata allows long value labels (32,000 characters), so labels can be long. However, some commands may need to display truncated value labels, typically at length 12. Consequently, even if the value labels are unique, the truncated value labels may not be, which can cause problems. labelbook warns you for value labels that are not unique at length 12.

6. Stata allows value labels that can be interpreted as numbers. This is sometimes useful, but it can cause highly misleading output. Think about tabulating a variable for which the associated value label incorrectly maps 1 into “2”, 2 into “3”, and 3 into “1”. labelbook looks for such problematic labels and warns you if they are found.
7. In Stata, value labels are defined as separate objects that can be associated with more than one variable:

    label define labname # str # str ....
    label value varname1 labname
    label value varname2 labname

If you forget to associate a variable label with a variable, Stata considers the label unused and drops its definition. `labelbook` reports unused value labels so that you may fix the problem.

The related command `codebook` reports on two other potential problems concerning value labels:

a. A variable is value labeled, but some values of the variable are not labeled. You may have forgotten to define a mapping for some values, or you generated a variable incorrectly; for example, your `sex` variable has an unlabeled value 3, and you are not working in experimental genetics!

b. A variable has been associated with an undefined value label.

`labelbook` can also be invoked with the `problems` option, specifying that only a report on potential problems be displayed without the standard detailed description of the value labels.

Technical note

The following two examples demonstrate some features of value labels that may be difficult to understand. In the first example, we `encode` a string variable with blank strings of various sizes; that is, we turn a string variable into a value-labeled numeric variable. Then we tabulate the generated variable.

```
. clear all
. set obs 5
   Number of observations (_N) was 0, now 5.
. generate str10 horror = substr(" ", 1, _n)
. encode horror, gen(Ihorror)
. tabulate horror

   horror | Freq.  Percent Cum.  
----------|--------|-------|
          1 | 20.00  | 20.00 |
          1 | 20.00  | 40.00 |
          1 | 20.00  | 60.00 |
          1 | 20.00  | 80.00 |
          1 | 20.00  | 100.00|

   Total | 5      | 100.00|
```

It may look as if you have discovered a bug in Stata because there are no value labels in the first column of the table. This happened because we encoded a variable with only blank strings, so the associated value label maps integers into blank strings.

```
. label list Ihorror
Ihorror:
   1    2    3    4    5
```

In the first column of the table, `tabulate` displayed the value-label texts, just as it should. Because these texts are all blank, the first column is empty. As illustrated below, `labelbook` would have warned you about this odd value label.
Our second example illustrates what could go wrong with numeric values stored as string values. We want to turn this into a numeric variable, but we incorrectly encode the variable rather than using the appropriate command, desting.

```
. generate str10 horror2 = string(_n+1)
. encode horror2, gen(Ihorror2)
. tabulate Ihorror2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>20.00</td>
<td>20.00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>20.00</td>
<td>40.00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>20.00</td>
<td>60.00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>20.00</td>
<td>80.00</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>20.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Total 5 100.00

. tabulate Ihorror2, nolabel

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>20.00</td>
<td>20.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>20.00</td>
<td>40.00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>20.00</td>
<td>60.00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>20.00</td>
<td>80.00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>20.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Total 5 100.00

. label list Ihorror2
Ihorror2:
1 2
2 3
3 4
4 5
5 6

labelbook skips the detailed descriptions of the value labels and reports only the potential problems in the value labels if the problems option is specified. This report would have alerted you to the problems with the value labels we just described.

```

. use https://www.stata-press.com/data/r17/data_in_trouble, clear
. labelbook, problem
```

<table>
<thead>
<tr>
<th>Potential problem</th>
<th>Value labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric -&gt; numeric</td>
<td>Ihorror2</td>
</tr>
<tr>
<td>Leading or trailing blanks</td>
<td>Ihorror</td>
</tr>
<tr>
<td>Numeric -&gt; null str</td>
<td>Ihorror</td>
</tr>
</tbody>
</table>

Running labelbook, problems and codebook, problems on new data might catch a series of annoying problems.
The `numlabel` command allows you to prefix numeric codes to value labels. The reason you might want to do this is best seen in an example using the automobile data. First, we create a value label for the variable `rep78` (repair record in 1978),

```
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. label define repair 1 "very poor" 2 "poor" 3 "medium" 4 good 5 "very good"
. label values rep78 repair
```

and tabulate it.

```
. tabulate rep78
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>very poor</td>
<td>2</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td>poor</td>
<td>8</td>
<td>11.59</td>
<td>14.49</td>
</tr>
<tr>
<td>medium</td>
<td>30</td>
<td>43.48</td>
<td>57.97</td>
</tr>
<tr>
<td>good</td>
<td>18</td>
<td>26.09</td>
<td>84.06</td>
</tr>
<tr>
<td>very good</td>
<td>11</td>
<td>15.94</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>69</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

Suppose that we want to recode the variable by joining the categories `poor` and `very poor`. To do this, we need the numerical codes of the categories, not the value labels. However, Stata does not display both the numeric codes and the value labels. We could redisplay the table with the `nolabel` option. The `numlabel` command provides a simple alternative: it modifies the value labels so that they also contain the numeric codes.

```
. numlabel, add
. tabulate rep78
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. very poor</td>
<td>2</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td>2. poor</td>
<td>8</td>
<td>11.59</td>
<td>14.49</td>
</tr>
<tr>
<td>3. medium</td>
<td>30</td>
<td>43.48</td>
<td>57.97</td>
</tr>
<tr>
<td>4. good</td>
<td>18</td>
<td>26.09</td>
<td>84.06</td>
</tr>
<tr>
<td>5. very good</td>
<td>11</td>
<td>15.94</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>69</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

If you do not like the way the numeric codes are formatted, you can use `numlabel` to change the formatting. First, we remove the numeric codes again:

```
. numlabel repair, remove
```

In this example, we specified the name of the label. If we had not typed it, `numlabel` would have removed the codes from all the value labels. We can include the numeric codes while specifying a mask:
. numlabel, add mask("[#] ")
. tabulate rep78

<table>
<thead>
<tr>
<th>Repair record</th>
<th>1978</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] very poor</td>
<td>2</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td>[3] medium</td>
<td>30</td>
<td>43.48</td>
<td>57.97</td>
</tr>
<tr>
<td>[4] good</td>
<td>18</td>
<td>26.09</td>
<td>84.06</td>
</tr>
<tr>
<td>[5] very good</td>
<td>11</td>
<td>15.94</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>69</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

numlabel prefixes rather than postfixes the value labels with numeric codes. Because value labels can be fairly long (up to 80 characters), Stata usually displays only the first 12 characters.

uselabel

uselabel is of interest primarily to programmers. Here we briefly illustrate it with the auto dataset.

Example 2

. use https://www.stata-press.com/data/r17/auto, clear
   (1978 automobile data)
. uselabel
. describe
Contains data
Observations: 2
Variables: 4

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage</th>
<th>Display</th>
<th>Value</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>type</td>
<td>format</td>
<td>label</td>
<td></td>
</tr>
<tr>
<td>lname</td>
<td>str6</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>byte</td>
<td>%10.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>label</td>
<td>str8</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trunc</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: lname  value
   Note: Dataset has changed since last saved.
. list

<table>
<thead>
<tr>
<th>lname</th>
<th>value</th>
<th>label</th>
<th>trunc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>origin</td>
<td>0</td>
<td>Domestic</td>
</tr>
<tr>
<td>2.</td>
<td>origin</td>
<td>1</td>
<td>Foreign</td>
</tr>
</tbody>
</table>

uselabel created a dataset containing the labels and values for the value label origin.

The maximum length of the text associated with a value label is 32,000 characters, whereas the maximum length of a string variable in a Stata dataset is 2,045. uselabel uses only the first 2,045 characters of the label. The trunc variable will record a 1 if the text was truncated for this reason.
Stored results

`labelbook` stores the following in `r()`:

Macros
- `r(names)`  `lblname-list`
- `r(gaps)`  gaps in mapped values
- `r(blanks)`  leading or trailing blanks
- `r(null)`  name of value label containing null strings
- `r(nuniq)`  duplicate labels
- `r(nuniq_sh)`  duplicate labels at length 12
- `r(ntruniq)`  duplicate labels at maximum string length
- `r(notused)`  not used by any of the variables
- `r(numeric)`  name of value label containing mappings to numbers

`uselabel` stores the following in `r()`:

Macros
- `r(lbname)`  list of variables that use value label `lbname` (only when `var` option is specified)

Acknowledgments

`labelbook` and `numlabel` were written by Jeroen Weesie of the Department of Sociology at Utrecht University, The Netherlands. A command similar to `numlabel` was written by J. M. Lauritsen (2001) of Odense Universitieshospital, Denmark.

References


Also see

[D] `codebook` — Describe data contents
[D] `describe` — Describe data in memory or in file
[D] `ds` — Compactly list variables with specified properties
[D] `encode` — Encode string into numeric and vice versa
[D] `label` — Manipulate labels
[U] 12.6 Dataset, variable, and value labels
[U] 15 Saving and printing output—log files
list — List values of variables

Description

list displays the values of variables. If no varlist is specified, the values of all the variables are displayed. Also see browse in [D] edit.

Quick start

List the data in memory
list

List only data in variables v1, v2, and v3
list v1 v2 v3

As above, but include only the first 10 observations and suppress numbering
list v1 v2 v3 in f/10, noobs

As above, but list the last 10 observations
list v1 v2 v3 in -10/l, noobs

Draw separator line every 10 observations, and repeat header row every 20 observations
list v1 v2 v3, separator(10) header(20)

As above, but draw separator line between values of v1 and do not show the header
list v1 v2 v3, sepby(v1) noheader

Add the mean and sum of the observations at the end of the table, and suppress separator and divider lines
list v1 v2 v3, mean sum clean

Menu

Data > Describe data > List data
## Syntax

```
list [varlist] [if] [in] [, options]
```

`flist` is equivalent to `list` with the `fast` option.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
<td></td>
</tr>
<tr>
<td><code>compress</code></td>
<td>compress width of columns in both table and display formats</td>
</tr>
<tr>
<td><code>nocompress</code></td>
<td>use display format of each variable</td>
</tr>
<tr>
<td><code>fast</code></td>
<td>synonym for <code>nocompress</code>; no delay in output of large datasets</td>
</tr>
<tr>
<td><code>abbreviate(#)</code></td>
<td>abbreviate variable names to <code>#</code> display columns; default is <code>ab(8)</code></td>
</tr>
<tr>
<td><code>string(#)</code></td>
<td>truncate string variables to <code>#</code> display columns</td>
</tr>
<tr>
<td><code>noobs</code></td>
<td>do not list observation numbers</td>
</tr>
<tr>
<td><code>fvall</code></td>
<td>display all levels of factor variables</td>
</tr>
<tr>
<td><strong>Options</strong></td>
<td></td>
</tr>
<tr>
<td><code>table</code></td>
<td>force table format</td>
</tr>
<tr>
<td><code>display</code></td>
<td>force display format</td>
</tr>
<tr>
<td><code>header</code></td>
<td>display variable header once; default is table mode</td>
</tr>
<tr>
<td><code>noheader</code></td>
<td>suppress variable header</td>
</tr>
<tr>
<td><code>header(#)</code></td>
<td>display variable header every <code>#</code> lines</td>
</tr>
<tr>
<td><code>clean</code></td>
<td>force table format with no divider or separator lines</td>
</tr>
<tr>
<td><code>divider</code></td>
<td>draw divider lines between columns</td>
</tr>
<tr>
<td><code>separator(#)</code></td>
<td>draw a separator line every <code>#</code> lines; default is <code>separator(5)</code></td>
</tr>
<tr>
<td><code>sepby(varlist2)</code></td>
<td>draw a separator line whenever <code>varlist2</code> values change</td>
</tr>
<tr>
<td><code>ds</code></td>
<td>use double-spaced lines</td>
</tr>
<tr>
<td><code>nolabel</code></td>
<td>display numeric codes rather than label values</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td></td>
</tr>
<tr>
<td><code>mean[(varlist2)]</code></td>
<td>add line reporting the mean for the (specified) variables</td>
</tr>
<tr>
<td><code>sum[(varlist2)]</code></td>
<td>add line reporting the sum for the (specified) variables</td>
</tr>
<tr>
<td><code>N[(varlist2)]</code></td>
<td>add line reporting the number of nonmissing values for the (specified) variables</td>
</tr>
<tr>
<td><code>labvar(varname)</code></td>
<td>substitute <code>Mean</code>, <code>Sum</code>, or <code>N</code> for value of <code>varname</code> in last row of table</td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
<td></td>
</tr>
<tr>
<td><code>constant[(varlist2)]</code></td>
<td>separate and list variables that are constant only once</td>
</tr>
<tr>
<td><code>notrim</code></td>
<td>suppress string trimming</td>
</tr>
<tr>
<td><code>absolute</code></td>
<td>display overall observation numbers when using <code>by varlist:</code></td>
</tr>
<tr>
<td><code>nodotz</code></td>
<td>display numerical values equal to <code>.z</code> as field of blanks</td>
</tr>
<tr>
<td><code>subvarname</code></td>
<td>substitute characteristic for variable name in header</td>
</tr>
<tr>
<td><code>linesize(#)</code></td>
<td>columns per line; default is <code>linesize(79)</code></td>
</tr>
</tbody>
</table>

*varlist* may contain factor variables; see [U] 11.4.3 Factor variables.

*varlist* may contain time-series operators; see [U] 11.4.4 Time-series varlists.

`by` is allowed with `list`; see [D] `by`. 
Options

`compress` and `nocompress` change the width of the columns in both table and display formats. By default, `list` examines the data and allocates the needed width to each variable. For instance, a variable might be a string with a `%18s` format, and yet the longest string will be only 12 characters long. Or a numeric variable might have a `%9.0g` format, and yet, given the values actually present, the widest number needs only four columns.

`nocompress` prevents `list` from examining the data. Widths will be set according to the display format of each variable. Output generally looks better when `nocompress` is not specified, but for very large datasets (say, 1,000,000 observations or more), `nocompress` can speed up the execution of `list`.

`compress` allows `list` to engage in a little more compression than it otherwise would by telling `list` to abbreviate variable names to fewer than eight characters.

`fast` is a synonym for `nocompress`. `fast` may be of interest to those with very large datasets who wish to see output appear without delay.

`abbreviate(#)` is an alternative to `compress` that allows you to specify the minimum abbreviation of variable names to be considered. For example, you could specify `abbreviate(16)` if you never wanted variables abbreviated to less than 16 display columns. For most users, the number of display columns is equal to the number of characters. However, some languages, such as Chinese, Japanese, and Korean (CJK), require two display columns per character.

`string(#)` specifies that when string variables are listed, they be truncated to `#` display columns in the output. Any value that is truncated will be appended with “…” to indicate the truncation. `string()` is useful for displaying just a part of long strings.

`noobs` suppresses the listing of the observation numbers.

`fvall` specifies that the entire dataset be used to determine how many levels are in any factor variables specified in `varlist`. The default is to determine the number of levels by using only the observations in the `if` and in `qualifiers`.

`table` and `display` determine the style of output. By default, `list` determines whether to use `table` or `display` on the basis of the width of your screen and the `linesize()` option, if you specify it.

`table` forces table format. Forcing table format when `list` would have chosen otherwise generally produces impossible-to-read output because of the linewraps. However, if you are logging output in SMCL format and plan to print the output on wide paper later, specifying `table` can be a reasonable thing to do.

`display` forces display format.

`header`, `noheader`, and `header(#)` specify how the variable header is to be displayed.

`header` is the default in table mode and displays the variable header once, at the top of the table. `noheader` suppresses the header altogether.

`header(#)` redisplay the variable header every `#` observations. For example, `header(10)` would display a new header every 10 observations.
The default in display mode is to display the variable names interweaved with the data:

<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>mpg</th>
<th>rep78</th>
<th>headroom</th>
<th>trunk</th>
<th>weight</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC Concord</td>
<td>4,099</td>
<td>22</td>
<td>3</td>
<td>2.5</td>
<td>11</td>
<td>2,930</td>
<td>186</td>
</tr>
<tr>
<td>turn</td>
<td>displa-t</td>
<td>gear_r-o</td>
<td>foreign</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>121</td>
<td>3.58</td>
<td>Domestic</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However, if you specify `header`, the header is displayed once, at the top of the table:

<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>mpg</th>
<th>rep78</th>
<th>headroom</th>
<th>trunk</th>
<th>weight</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>turn</td>
<td>displa-t</td>
<td>gear_r-o</td>
<td>foreign</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cleaning is a better alternative to `table` when you want to force table format and your goal is to produce more readable output on the screen. `clean` implies `table`, and it removes dividing and separating lines, which is what makes wrapped table output nearly impossible to read. Blank separator lines may be included by specifying the `ds` option.

divider, `separator(#)`, `sepby(varlist)`, and `ds` specify how dividers and separator lines should be displayed. These four options affect only table format.

divider specifies that divider lines be drawn between columns. The default is `nodivider`.

`separator(#)` and `sepby(varlist)` indicate when separator lines should be drawn between rows.

To make these separator lines blank, specify the `ds` option.

`separator(#)` specifies how often separator lines should be drawn between rows. The default is `separator(5)`, meaning every 5 observations. You may specify `separator(0)` to suppress separators altogether.

`sepby(varlist)` specifies that a separator line be drawn whenever any of the variables in `sepby(varlist)` change their values; up to 10 variables may be specified. You need not make sure the data were sorted on `sepby(varlist)` before issuing the `list` command. The variables in `sepby(varlist)` also need not be among the variables being listed.

ds specifies that the lines be double spaced, meaning that a blank separator line be inserted after every observation. To control when blank separator lines are inserted, specify `ds` with `separator(#)` or `sepby(varlist)`.

By default, separator lines are suppressed when specifying the `clean` option unless `ds` is specified, in which case blank separator lines will be used.

`nolabel` specifies that numeric codes be displayed rather than the label values.

`mean, sum, N, mean(varlist), sum(varlist), and N(varlist)` all specify that lines be added to the output reporting the mean, sum, or number of nonmissing values for the (specified) variables. If you do not specify the variables, all numeric variables in the `varlist` following `list` are used.
labvar(varname) is for use with mean[], sum[], and N[]. list displays Mean, Sum, or N where the observation number would usually appear to indicate the end of the table—where a row represents the calculated mean, sum, or number of observations.

labvar(varname) changes that. Instead, Mean, Sum, or N is displayed where the value for varname would be displayed. For instance, you might type

```
. list group costs profits, sum(costs profits) labvar(group)
```

<table>
<thead>
<tr>
<th>group</th>
<th>costs</th>
<th>profits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>47</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>123</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Sum</td>
<td>192</td>
<td>17</td>
</tr>
</tbody>
</table>

and then also specify the noobs option to suppress the observation numbers.

```Advanced```

constant and constant(varlist) specify that variables that do not vary observation by observation be separated out and listed only once.

constant specifies that list determine for itself which variables are constant.

constant(varlist) allows you to specify which of the constant variables you want listed separately.
list verifies that the variables you specify really are constant and issues an error message if they are not.
constant and constant() respect if exp and in range. If you type

```
. list if group==3 variable x might be constant in the selected observations, even though the variable varies in the entire dataset.
```

notrim affects how string variables are listed. The default is to trim strings at the width implied by the widest possible column given your screen width (or linesize(), if you specified that). notrim specifies that strings not be trimmed. notrim implies clean (see above) and, in fact, is equivalent to the clean option, so specifying either makes no difference.

absolute affects output only when list is prefixed with by varlist:. Observation numbers are displayed, but the overall observation numbers are used rather than the observation numbers within each by-group. For example, if the first group had 4 observations and the second had 2, by default the observations would be numbered 1, 2, 3, 4 and 1, 2. If absolute is specified, the observations will be numbered 1, 2, 3, 4 and 5, 6.

nodotz is a programmer’s option that specifies that numerical values equal to .z be listed as a field of blanks rather than as .z.

subvarname is a programmer’s option. If a variable has the characteristic var[varname] set, then the contents of that characteristic will be used in place of the variable’s name in the headers.

linesize(#) specifies the width of the page to be used for determining whether table or display format should be used and for formatting the resulting table. Specifying a value of linesize() that is wider than your screen width can produce truly ugly output on the screen, but that output can nevertheless be useful if you are logging output and plan to print the log later on a wide printer.
Remarks and examples

list, typed by itself, lists all the observations and variables in the dataset. If you specify varlist, only those variables are listed. Specifying one or both of in range and if exp limits the observations listed.

list respects line size. That is, if you resize the Results window (in windowed versions of Stata) before running list, it will take advantage of the available horizontal space. Stata for Unix(console) users can instead use the set linesize command to take advantage of this feature; see [R] log.

list may not display all the large strings. You have two choices: 1) you can specify the clean option, which makes a different, less attractive listing, or 2) you can increase line size, as discussed above.

Example 1

list has two output formats, known as table and display. The table format is suitable for listing a few variables, whereas the display format is suitable for listing an unlimited number of variables. Stata chooses automatically between those two formats:

```
use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
list in 1/2
1.   make          price     mpg   rep78  headroom  trunk  weight  length
       AMC Concord 4,099  22      3   2.5         11 2,930    186
              turn   displa-t  gear_r-o foreign
       40       121       3.58 Domestic

2.   make          price     mpg   rep78  headroom  trunk  weight  length
       AMC Pacer  4,749  17      3   3.0         11 3,350    173
              turn   displa-t  gear_r-o foreign
       40       258       2.53 Domestic

list make mpg weight displ rep78 in 1/5
```

The first case is an example of display format; the second is an example of table format. The table format is more readable and takes less space, but it is effective only if the variables can fit on one line across the screen. Stata chose to list all 12 variables in display format, but when the varlist was restricted to five variables, Stata chose table format.

If you are dissatisfied with Stata’s choice, you can decide for yourself. You can specify the display option to force display format and the nodisplay option to force table format.
If you have long string variables in your data—say, `str75` or longer—by default, `list` displays only the first 70 or so characters of each; the exact number is determined by the width of your Results window. The first 70 or so characters will be shown followed by “...”. If you need to see the entire contents of the string, you can

1. specify the `clean` option, which makes a different (and uglier) style of list, or
2. make your Results window wider [Stata for Unix(console) users: increase `set linesize`].

Among the things that determine the widths of the columns, the variable names play a role. Left to itself, `list` will never abbreviate variable names to fewer than eight characters. You can use the `compress` option to abbreviate variable names to fewer characters than that.

When Stata lists a string variable in table output format, the variable is displayed right-justified by default.

When Stata lists a string variable in display output format, it decides whether to display the variable right-justified or left-justified according to the display format for the string variable; see [U] 12.5 Formats: Controlling how data are displayed. In our previous example, `make` has a display format of `%−18s`.

```
. describe make
Variable                      Storage  Display          Value
  name     type   format   label          Variable label
make      str18  %−18s   Make and model
```

The negative sign in the `%−18s` instructs Stata to left-justify this variable. If the display format had been `%18s`, Stata would have right-justified the variable.

The `foreign` variable appears to be string, but if we `describe` it, we see that it is not:

```
. describe foreign
Variable                      Storage  Display          Value
  name     type  format  label          Variable label
foreign   byte    %8.0g  origin         Car origin
```

`foreign` is stored as a `byte`, but it has an associated value label named `origin`; see [U] 12.6.3 Value labels. Stata decides whether to right-justify or left-justify a numeric variable with an associated value label by using the same rule used for string variables: it looks at the display format of the variable. Here the display format of `%8.0g` tells Stata to right-justify the variable. If the display format had been `%−8.0g`, Stata would have left-justified this variable.
Technical note

You can list the variables in any order. When you specify the `varlist`, list displays the variables in the order you specify. You may also include variables more than once in the `varlist`.

Example 2

Sometimes you may wish to suppress the observation numbers. You do this by specifying the `noobs` option:

```
. list make mpg weight displ foreign in 46/55, noobs
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
<th>displa-t</th>
<th>foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plym. Volare</td>
<td>18</td>
<td>3,330</td>
<td>225</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Catalina</td>
<td>18</td>
<td>3,700</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Firebird</td>
<td>18</td>
<td>3,470</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>19</td>
<td>3,210</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Le Mans</td>
<td>19</td>
<td>3,200</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Phoenix</td>
<td>19</td>
<td>3,420</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Sunbird</td>
<td>24</td>
<td>2,690</td>
<td>151</td>
<td>Domestic</td>
</tr>
<tr>
<td>Audi 5000</td>
<td>17</td>
<td>2,830</td>
<td>131</td>
<td>Foreign</td>
</tr>
<tr>
<td>Audi Fox</td>
<td>23</td>
<td>2,070</td>
<td>97</td>
<td>Foreign</td>
</tr>
<tr>
<td>BMW 320i</td>
<td>25</td>
<td>2,650</td>
<td>121</td>
<td>Foreign</td>
</tr>
</tbody>
</table>

After seeing the table, we decide that we want to separate the “Domestic” observations from the “Foreign” observations, so we specify `sepby(foreign)`.

```
. list make mpg weight displ foreign in 46/55, noobs sepby(foreign)
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
<th>displa-t</th>
<th>foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plym. Volare</td>
<td>18</td>
<td>3,330</td>
<td>225</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Catalina</td>
<td>18</td>
<td>3,700</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Firebird</td>
<td>18</td>
<td>3,470</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>19</td>
<td>3,210</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Le Mans</td>
<td>19</td>
<td>3,200</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Phoenix</td>
<td>19</td>
<td>3,420</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Sunbird</td>
<td>24</td>
<td>2,690</td>
<td>151</td>
<td>Domestic</td>
</tr>
<tr>
<td>Audi 5000</td>
<td>17</td>
<td>2,830</td>
<td>131</td>
<td>Foreign</td>
</tr>
<tr>
<td>Audi Fox</td>
<td>23</td>
<td>2,070</td>
<td>97</td>
<td>Foreign</td>
</tr>
<tr>
<td>BMW 320i</td>
<td>25</td>
<td>2,650</td>
<td>121</td>
<td>Foreign</td>
</tr>
</tbody>
</table>
Example 3

We want to add vertical lines in the table to separate the variables, so we specify the `divider` option. We also want to draw a horizontal line after every 2 observations, so we specify `separator(2)`.

```
. list make mpg weight displ foreign in 46/55, divider separator(2)
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
<th>displa-t</th>
<th>foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plym. Volare</td>
<td>18</td>
<td>3,330</td>
<td>225</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Catalina</td>
<td>18</td>
<td>3,700</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Firebird</td>
<td>18</td>
<td>3,470</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>19</td>
<td>3,210</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Le Mans</td>
<td>19</td>
<td>3,200</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Phoenix</td>
<td>19</td>
<td>3,420</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Sunbird</td>
<td>24</td>
<td>2,690</td>
<td>151</td>
<td>Domestic</td>
</tr>
<tr>
<td>Audi 5000</td>
<td>17</td>
<td>2,830</td>
<td>131</td>
<td>Domestic</td>
</tr>
<tr>
<td>Audi Fox</td>
<td>23</td>
<td>2,070</td>
<td>97</td>
<td>Foreign</td>
</tr>
<tr>
<td>BMW 320i</td>
<td>25</td>
<td>2,650</td>
<td>121</td>
<td>Foreign</td>
</tr>
</tbody>
</table>

After seeing the table, we decide that we do not want to abbreviate `displacement`, so we specify `abbreviate(12)`.

```
. list make mpg weight displ foreign in 46/55, divider sep(2) abbreviate(12)
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
<th>displacement</th>
<th>foreign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plym. Volare</td>
<td>18</td>
<td>3,330</td>
<td>225</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Catalina</td>
<td>18</td>
<td>3,700</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Firebird</td>
<td>18</td>
<td>3,470</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>19</td>
<td>3,210</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Le Mans</td>
<td>19</td>
<td>3,200</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Phoenix</td>
<td>19</td>
<td>3,420</td>
<td>231</td>
<td>Domestic</td>
</tr>
<tr>
<td>Pont. Sunbird</td>
<td>24</td>
<td>2,690</td>
<td>151</td>
<td>Domestic</td>
</tr>
<tr>
<td>Audi 5000</td>
<td>17</td>
<td>2,830</td>
<td>131</td>
<td>Domestic</td>
</tr>
<tr>
<td>Audi Fox</td>
<td>23</td>
<td>2,070</td>
<td>97</td>
<td>Foreign</td>
</tr>
<tr>
<td>BMW 320i</td>
<td>25</td>
<td>2,650</td>
<td>121</td>
<td>Foreign</td>
</tr>
</tbody>
</table>

Technical note

You can suppress the use of value labels by specifying the `nolabel` option. For instance, the `foreign` variable in the examples above really contains numeric codes, with 0 meaning `Domestic` and 1 meaning `Foreign`. When we `list` the variable, however, we see the corresponding value labels rather than the underlying numeric code:
. list foreign in 51/55

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>51.</td>
<td>Domestic</td>
</tr>
<tr>
<td>52.</td>
<td>Domestic</td>
</tr>
<tr>
<td>53.</td>
<td>Foreign</td>
</tr>
<tr>
<td>54.</td>
<td>Foreign</td>
</tr>
<tr>
<td>55.</td>
<td>Foreign</td>
</tr>
</tbody>
</table>

Specifying the `nolabel` option displays the underlying numeric codes:

. list foreign in 51/55, nolabel

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>51.</td>
<td>0</td>
</tr>
<tr>
<td>52.</td>
<td>0</td>
</tr>
<tr>
<td>53.</td>
<td>1</td>
</tr>
<tr>
<td>54.</td>
<td>1</td>
</tr>
<tr>
<td>55.</td>
<td>1</td>
</tr>
</tbody>
</table>

References


Also see

[D] edit — Browse or edit data with Data Editor

[P] display — Display strings and values of scalar expressions

[P] tabdisp — Display tables

[R] table — Table of frequencies, summaries, and command results
lookfor — Search for string in variable names and labels

Description

lookfor helps you find variables by searching for `string` among all variable names and labels. If multiple `strings` are specified, `lookfor` will search for each of them separately. You may search for a phrase by enclosing `string` in double quotes.

Quick start

Search variable names and variable labels for the phrase “my text” regardless of case

```
lookfor "my text"
```

Search for “word1” or “word2”

```
lookfor word1 word2
```

Syntax

```
lookfor string [ string [...]]
```

`collect` is allowed; see [U] 11.1.10 Prefix commands.

Remarks and examples

Example 1

```
lookfor finds variables by searching for `string`, ignoring case, among the variable names and labels.

. use https://www.stata-press.com/data/r17/nlswork
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. lookfor code
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>idcode</td>
<td>int</td>
<td>%8.0g</td>
<td>NLS ID</td>
</tr>
<tr>
<td>ind_code</td>
<td>byte</td>
<td>%8.0g</td>
<td>Industry of employment</td>
</tr>
<tr>
<td>occ_code</td>
<td>byte</td>
<td>%8.0g</td>
<td>Occupation</td>
</tr>
</tbody>
</table>

Three variable names contain the word `code`.

```
. lookfor married
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>msp</td>
<td>byte</td>
<td>%8.0g</td>
<td>1 if married, spouse present</td>
</tr>
<tr>
<td>nev_mar</td>
<td>byte</td>
<td>%8.0g</td>
<td>1 if never married</td>
</tr>
</tbody>
</table>
Two variable labels contain the word *married*.

```stata
. lookfor gnp
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln_wage</td>
<td>float</td>
<td>%9.0g</td>
<td>ln(wage/GNP deflator)</td>
</tr>
</tbody>
</table>

`lookfor` ignores case, so `lookfor gnp` found GNP in a variable label.

**Example 2**

If multiple strings are specified, all variable names or labels containing any of the strings are listed.

```stata
. lookfor code married
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>idcode</td>
<td>int</td>
<td>%8.0g</td>
<td>NLS ID</td>
</tr>
<tr>
<td>msp</td>
<td>byte</td>
<td>%8.0g</td>
<td>1 if married, spouse present</td>
</tr>
<tr>
<td>never_mar</td>
<td>byte</td>
<td>%8.0g</td>
<td>1 if never married</td>
</tr>
<tr>
<td>ind_code</td>
<td>byte</td>
<td>%8.0g</td>
<td>Industry of employment</td>
</tr>
<tr>
<td>occ_code</td>
<td>byte</td>
<td>%8.0g</td>
<td>Occupation</td>
</tr>
</tbody>
</table>

To search for a phrase, enclose *string* in double quotes.

```stata
. lookfor "never married"
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>never_mar</td>
<td>byte</td>
<td>%8.0g</td>
<td>1 if never married</td>
</tr>
</tbody>
</table>

**Stored results**

`lookfor` stores the following in `r()`:

- Macros
  - `r(varlist)` the varlist of found variables

**References**


**Also see**

[D] `describe` — Describe data in memory or in file

[D] `ds` — Compactly list variables with specified properties
memory — Memory management

Description

Memory usage and settings are described here.

memory displays a report on Stata’s current memory usage.

query memory displays the current values of Stata’s memory settings.

set maxvar, set niceness, set min_memory, set max_memory, and set segmentsize change the values of the memory settings.

If you are a Unix user, see Serious bug in Linux OS under Remarks and examples below.

Quick start

Display memory usage report

memory

Display memory settings

query memory

Increase the maximum number of variables to 8,000 in Stata/MP or Stata/SE

set maxvar 8000

Set maximum memory allocation to avoid potential memory allocation bug in Linux

set max_memory 16g, permanently
Syntax

Display memory usage report

    memory

Display memory settings

    query memory

Modify memory settings

    set maxvar   # [, permanently]
    set niceness # [, permanently]
    set min_memory amt [, permanently]
    set max_memory amt [, permanently]
    set segmentsize amt [, permanently]

where amt is # |b|k|m|g], and the default unit is b.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxvar</td>
<td>5000</td>
<td>2048</td>
<td>120000</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>2048</td>
<td>32767</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>niceness</td>
<td>5</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>min_memory</td>
<td>0</td>
<td>0</td>
<td>max_memory</td>
</tr>
<tr>
<td>max_memory</td>
<td>.</td>
<td>2×segmentsize</td>
<td>.</td>
</tr>
<tr>
<td>segmentsize</td>
<td>32m</td>
<td>1m</td>
<td>32g</td>
</tr>
</tbody>
</table>

Notes:

1. The maximum number of variables in your dataset is limited to maxvar. The default value of maxvar is 5,000 for Stata/MP and Stata/SE, and 2,048 for Stata/BE. With Stata/MP and Stata/SE, this default value may be increased by using set maxvar. The default value is fixed for Stata/BE.

2. Most users do not need to read beyond this point. Stata’s memory management is completely automatic. If, however, you are using the Linux operating system, see Serious bug in Linux OS under Remarks and examples below.

3. The maximum number of observations is fixed at 1,099,511,627,775 for Stata/MP and is fixed at 2,147,483,619 for Stata/SE and Stata/BE regardless of computer size or memory settings. Depending on the amount of memory on your computer, you may face a lower practical limit. See help obs_advice.
4. `max_memory` specifies the maximum amount of memory Stata can use to store your data. The default of missing (.) means all the memory the operating system is willing to supply. There are three reasons to change the value from missing to a finite number.

1. You are a Linux user; see *Serious bug in Linux OS* under Remarks and examples below.

2. You wish to reduce the chances of accidents, such as typing `expand 100000` with a large dataset in memory and actually having Stata do it. You would rather see an insufficient-memory error message. Set `max_memory` to the amount of physical memory on your computer or more than that if you are willing to use virtual memory.

3. You are a system administrator; see *Notes for system administrators* under Remarks and examples below.

5. The remaining memory parameters—`niceness`, `min_memory`, and `segment_size`—affect efficiency only; they do not affect the size of datasets you can analyze.

6. Memory amounts for `min_memory`, `max_memory`, and `segment_size` may be specified in bytes, kilobytes, megabytes, or gigabytes; suffix `b`, `k`, `m`, or `g` to the end of the number. The following are equivalent ways of specifying 1 gigabyte:

   1073741824
   1048576k
   1024m
   1g

   Suffix `k` is defined as (multiply by) 1024, `m` is defined as `1024^2`, and `g` is defined as `1024^3`.

7. 64-bit computers can theoretically provide up to 18,446,744,073,709,551,616 bytes of memory, equivalent to 17,179,869,184 gigabytes, 16,777,216 terabytes, 16,384 petabytes, or 16 exabytes. Real computers have less.

8. Stata allocates memory for data in units of `segment_size`. Smaller values of `segment_size` can result in more efficient use of available memory but require Stata to jump around more. The default provides a good balance. We recommend resetting `segment_size` only if your computer has large amounts of memory.

9. If you have large amounts of memory and you use it to process large datasets, you may wish to increase `segment_size`. Suggested values are

<table>
<thead>
<tr>
<th>memory</th>
<th>segmentsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>32g</td>
<td>64m</td>
</tr>
<tr>
<td>64g</td>
<td>128m</td>
</tr>
<tr>
<td>128g</td>
<td>256m</td>
</tr>
<tr>
<td>256g</td>
<td>512m</td>
</tr>
<tr>
<td>512g</td>
<td>1g</td>
</tr>
<tr>
<td>1024g</td>
<td>2g</td>
</tr>
</tbody>
</table>

10. `niceness` affects how soon Stata gives back unused segments to the operating system. If Stata releases them too soon, it often needs to turn around and get them right back. If Stata waits too long, Stata is consuming memory that it is not using. One reason to give memory back is to be nice to other users on multiuser systems or to be nice to yourself if you are running other processes.
The default value of 5 is defined to provide good performance. Waiting times are currently defined as

<table>
<thead>
<tr>
<th>niceness</th>
<th>waiting time (m:s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0:00.000</td>
</tr>
<tr>
<td>9</td>
<td>0:00.125</td>
</tr>
<tr>
<td>8</td>
<td>0:00.500</td>
</tr>
<tr>
<td>7</td>
<td>0:01</td>
</tr>
<tr>
<td>6</td>
<td>0:30</td>
</tr>
<tr>
<td>5</td>
<td>1:00</td>
</tr>
<tr>
<td>4</td>
<td>5:00</td>
</tr>
<tr>
<td>3</td>
<td>10:00</td>
</tr>
<tr>
<td>2</td>
<td>15:00</td>
</tr>
<tr>
<td>1</td>
<td>20:00</td>
</tr>
<tr>
<td>0</td>
<td>30:00</td>
</tr>
</tbody>
</table>

Niceness 10 corresponds to being totally nice. Niceness 0 corresponds to being an incon- siderate, self-centered, totally selfish jerk.

11. `min_memory` specifies an amount of memory Stata will not fall below. For instance, you have a long do-file. You know that late in the do-file, you will need 8 gigabytes. You want to ensure that the memory will be available later. At the start of your do-file, you set `set min_memory 8g`.

12. Concerning `min_memory` and `max_memory`, be aware that Stata allocates memory in `segmentsize` blocks. Both `min_memory` and `max_memory` are rounded down. Thus the actual minimum memory Stata will reserve will be

\[
\text{segmentsize} \times \text{trunc}(\text{min_memory}/\text{segmentsize})
\]

The effective maximum memory is calculated similarly. (Stata does not round up `min_memory` because some users set `min_memory` equal to `max_memory`.)

**Options**

- `permanently` specifies that, in addition to making the change right now, the new limit be remembered and become the default setting when you invoke Stata.
- `once` is not shown in the syntax diagram but is allowed with `set niceness`, `set min_memory`, `set max_memory`, and `set segmentsize`. It is for use by system administrators; see *Notes for system administrators* under Remarks and examples below.

**Remarks and examples**

Remarks are presented under the following headings:

- **Examples**
- **Serious bug in Linux OS**
- **Notes for system administrators**
Examples

Here is our memory-usage report after we load auto.dta that comes with Stata using Stata/MP:

```
. sysuse auto
(1978 automobile data)
. memory

Memory usage

<table>
<thead>
<tr>
<th></th>
<th>Used</th>
<th>Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>3,182</td>
<td>100,663,296</td>
</tr>
<tr>
<td>strLs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Data &amp; strLs</td>
<td>3,182</td>
<td>100,663,296</td>
</tr>
<tr>
<td>Variable names, %fmts, ...</td>
<td>4,178</td>
<td>396,279</td>
</tr>
<tr>
<td>Overhead</td>
<td>1,081,344</td>
<td>1,082,136</td>
</tr>
<tr>
<td>Stata matrices</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ado-files</td>
<td>53,718</td>
<td>53,718</td>
</tr>
<tr>
<td>Stored results</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mata matrices</td>
<td>10,880</td>
<td>10,880</td>
</tr>
<tr>
<td>Mata functions</td>
<td>2,720</td>
<td>2,720</td>
</tr>
<tr>
<td>set maxvar usage</td>
<td>4,636,521</td>
<td>4,636,521</td>
</tr>
<tr>
<td>Other</td>
<td>3,497</td>
<td>3,497</td>
</tr>
<tr>
<td>Total</td>
<td>5,773,999</td>
<td>106,849,047</td>
</tr>
</tbody>
</table>
```

We could then obtain the current memory-settings report by typing

```
. query memory
```

```
Memory settings
set maxvar     5000  2048-120000; max. vars allowed
set niceness   5    0-10
set min_memory 0    0-1600g
set max_memory .  32m-1600g or .
set segmentsize 32m 1m-32g
set adosize 1000 kilobytes
set max_preservemem 1g 0-1600g
```

Serious bug in Linux OS

If you use Linux OS, we strongly suggest that you set `max_memory`. Here’s why:

“By default, Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory really is available. This is a really bad bug. In case it turns out that the system is out of memory, one or more processes will be killed by the infamous OOM killer. In case Linux is employed under circumstances where it would be less desirable to suddenly lose some randomly picked processes, and moreover the kernel version is sufficiently recent, one can switch off this overcommitting behavior using [...]”

– Output from Unix command `man malloc`.

What this means is that Stata requests memory from Linux, Linux says yes, and then later when Stata uses that memory, the memory might not be available and Linux crashes Stata, or worse. The Linux documentation writer exercised admirable restraint. This bug can cause Linux itself to crash. It is easy.
The proponents of this behavior call it “optimistic memory allocation”. We will, like the documentation writer, refer to it as a bug.

The bug is fixable. Type `man malloc` at the Unix prompt for instructions. Note that `man malloc` is an instruction of Unix, not Stata. If the bug is not mentioned, perhaps it has been fixed. Before assuming that, we suggest using a search engine to search for “linux optimistic memory allocation”.

Alternatively, Stata can live with the bug if you set `max_memory`. Find out how much physical memory is on your computer and set `max_memory` to that. If you want to use virtual memory, you might set it larger, just make sure your Linux system can provide the requested memory. Specify the option permanently so you only need to do this once. For example,

```
set max_memory 16g, permanently
```

Doing this does not guarantee that the bug does not bite, but it makes it unlikely.

### Notes for system administrators

System administrators can set `max_memory`, `min_memory`, and `niceness` so that Stata users cannot change them. They can also do this with `max_preservemem` (see [P] preserve). You may want to do this on shared computers to prevent individual users from hogging resources.

There is no reason you would want to do this on users’ personal computers.

You can also set `segmentsize`, but there is no reason to do this even on shared systems.

The instructions are to create (or edit) the text file `sysprofile.do` in the directory where the Stata executable resides. Add the lines

```
set min_memory 0, once
set max_memory 16g, once
set niceness 5, once
```

The file must be plain text, and there must be end-of-line characters at the end of each line, including the last line. Blank lines at the end are recommended.

The 16g on `set max_memory` is merely for example. Choose an appropriate number.

The values of 0 for `min_memory` and 5 for `niceness` are recommended.

### Stored results

`memory` stores all reported numbers in `r()`. StataCorp may change what `memory` reports, and you should not expect the same `r()` results to exist in future versions of Stata. To see the stored results from `memory`, type `return list, all`.

### Also see

[P] creturn — Return c-class values
[R] query — Display system parameters
[U] 6 Managing memory
**merge** — Merge datasets

**Description**

*merge* joins corresponding observations from the dataset currently in memory (called the master dataset) with those from *filename*.dta (called the using dataset), matching on one or more key variables. *merge* can perform match merges (one-to-one, one-to-many, many-to-one, and many-to-many), which are often called *joins* by database people. *merge* can also perform sequential merges, which have no equivalent in the relational database world.

*merge* is for adding new variables from a second dataset to existing observations. You use *merge*, for instance, when combining hospital patient and discharge datasets. If you wish to add new observations to existing variables, then see \[D\] *append*. You use *append*, for instance, when adding current discharges to past discharges.

To link datasets in separate frames, you can use the *frlink* and *frget* commands. Linking and merging solve similar problems, and each is better than the other in some ways. You may prefer linking, for instance, when dealing with an individual-level dataset and a county-level dataset. Linking also works well when you have nested linkages such as linking a county dataset, a school-within-county dataset, and a student-within-school dataset or when you need to link a dataset to itself. See \[D\] *frlink* for more information and examples.

By default, *merge* creates a new variable, `_merge`, containing numeric codes concerning the source and the contents of each observation in the merged dataset. These codes are explained below in the match results table.

Key variables cannot be strLs.

If *filename* is specified without an extension, then .dta is assumed.

**Quick start**

One-to-one merge of *mydata1*.dta in memory with *mydata2*.dta on v1

```
merge 1:1 v1 using mydata2
```

As above, and also treat v2 as a key variable and name the new variable indicating the merge result for each observation newv

```
merge 1:1 v1 v2 using mydata2, generate(newv)
```

As above, but keep only v3 from *mydata2*.dta and use default merge result variable `_merge`

```
merge 1:1 v1 v2 using mydata2, keepusing(v3)
```

As above, but keep only observations in both datasets

```
merge 1:1 v1 v2 using mydata2, keepusing(v3) keep(match)
```

Same as above

```
merge 1:1 v1 v2 using mydata2, keepusing(v3) keep(3)
```
As above, but assert that all observations should match or return an error otherwise
   merge 1:1 v1 v2 using mydata2, keepusing(v3) assert(3)

Replace missing data in mydata1.dta with values from mydata2.dta
   merge 1:1 v1 v2 using mydata2, update

Replace missing and conflicting data in mydata1.dta with values from mydata2.dta
   merge 1:1 v1 v2 using mydata2, update replace

Many-to-one merge on v1 and v2
   merge m:1 v1 v2 using mydata2

One-to-many merge on v1 and v2
   merge 1:m v1 v2 using mydata2

Menu

Data > Combine datasets > Merge two datasets
Syntax

One-to-one merge on specified key variables

```
merge 1:1 varlist using filename [, options]
```

Many-to-one merge on specified key variables

```
merge m:1 varlist using filename [, options]
```

One-to-many merge on specified key variables

```
merge 1:m varlist using filename [, options]
```

Many-to-many merge on specified key variables

```
merge m:m varlist using filename [, options]
```

One-to-one merge by observation

```
merge 1:1 _n using filename [, options]
```

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keepusing(varlist)</td>
<td>variables to keep from using data; default is all</td>
</tr>
<tr>
<td>generate(newvar)</td>
<td>name of new variable to mark merge results; default is _merge</td>
</tr>
<tr>
<td>nogenerate</td>
<td>do not create _merge variable</td>
</tr>
<tr>
<td>nolabel</td>
<td>do not copy value-label definitions from using</td>
</tr>
<tr>
<td>nonotes</td>
<td>do not copy notes from using</td>
</tr>
<tr>
<td>update</td>
<td>update missing values of same-named variables in master with values from using</td>
</tr>
<tr>
<td>replace</td>
<td>replace all values of same-named variables in master with nonmissing values from using (requires update)</td>
</tr>
<tr>
<td>noreport</td>
<td>do not display match result summary table</td>
</tr>
<tr>
<td>force</td>
<td>allow string/numeric variable type mismatch without error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>do not sort; dataset already sorted</td>
</tr>
</tbody>
</table>

sorted does not appear in the dialog box.

Options

keepusing(varlist) specifies the variables from the using dataset that are kept in the merged dataset. By default, all variables are kept. For example, if your using dataset contains 2,000 demographic characteristics but you want only sex and age, then type merge ..., keepusing(sex age) ....
generate(newvar) specifies that the variable containing match results information should be named newvar rather than _merge.

nogenerate specifies that _merge not be created. This would be useful if you also specified keep(match), because keep(match) ensures that all values of _merge would be 3.

nolabel specifies that value-label definitions from the using file be ignored. This option should be rare, because definitions from the master are already used.

nonotes specifies that notes in the using dataset not be added to the merged dataset; see [D] notes.

update and replace both perform an update merge rather than a standard merge. In a standard merge, the data in the master are the authority and inviolable. For example, if the master and using datasets both contain a variable age, then matched observations will contain values from the master dataset, while unmatched observations will contain values from their respective datasets.

If update is specified, then matched observations will update missing values from the master dataset with values from the using dataset. Nonmissing values in the master dataset will be unchanged.

If replace is specified, then matched observations will contain values from the using dataset, unless the value in the using dataset is missing.

Specifying either update or replace affects the meanings of the match codes. See Treatment of overlapping variables for details.

noreport specifies that merge not present its summary table of match results.

force allows string/numeric variable type mismatches, resulting in missing values from the using dataset. If omitted, merge issues an error; if specified, merge issues a warning.

---

assert(results) specifies the required match results. The possible results are

<table>
<thead>
<tr>
<th>Numeric code</th>
<th>Equivalent word (results)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td>observation appeared in master only</td>
</tr>
<tr>
<td>2</td>
<td>using</td>
<td>observation appeared in using only</td>
</tr>
<tr>
<td>3</td>
<td>match</td>
<td>observation appeared in both</td>
</tr>
<tr>
<td>4</td>
<td>match_update</td>
<td>observation appeared in both, missing values updated</td>
</tr>
<tr>
<td>5</td>
<td>match_conflict</td>
<td>observation appeared in both, conflicting nonmissing values</td>
</tr>
</tbody>
</table>

Codes 4 and 5 can arise only if the update option is specified. If codes of both 4 and 5 could pertain to an observation, then 5 is used.

Numeric codes and words are equivalent when used in the assert() or keep() options.

The following synonyms are allowed: masters for master, usings for using, matches and matched for match, match_updates for match_update, and match_conflicts for match_conflict.

Using assert(match master) specifies that the merged file is required to include only matched master or using observations and unmatched master observations, and may not include unmatched using observations. Specifying assert() results in merge issuing an error message if there are match results you did not explicitly allow.
The order of the words or codes is not important, so all the following `assert()` specifications would be the same:

```
assert(match master)
assert(master matches)
assert(1 3)
```

When the match results contain codes other than those allowed, return code 9 is returned, and the merged dataset with the unanticipated results is left in memory to allow you to investigate.

`keep(results)` specifies which observations are to be kept from the merged dataset. Using `keep(match master)` specifies keeping only matched observations and unmatched master observations after merging.

`keep()` differs from `assert()` because it selects observations from the merged dataset rather than enforcing requirements. `keep()` is used to pare the merged dataset to a given set of observations when you do not care if there are other observations in the merged dataset. `assert()` is used to verify that only a given set of observations is in the merged dataset.

You can specify both `assert()` and `keep()`. If you require matched observations and unmatched master observations but you want only the matched observations, then you could specify `assert(match master) keep(match)`.

`assert()` and `keep()` are convenience options whose functionality can be duplicated using `_merge` directly.

```
. merge ..., assert(match master) keep(match)
```

is identical to

```
. merge ...
    . assert _merge==1 | _merge==3
    . keep if _merge==3
```

The following option is available with `merge` but is not shown in the dialog box:

`sorted` specifies that the master and using datasets are already sorted by `varlist`. If the datasets are already sorted, then `merge` runs a little more quickly; the difference is hardly detectable, so this option is of interest only where speed is of the utmost importance.

## Remarks and examples

Remarks are presented under the following headings:

- Overview
- Basic description
- 1:1 merges
- m:1 merges
- 1:m merges
- m:m merges
- Sequential merges
- Treatment of overlapping variables
- Sort order
- Troubleshooting m:m merges
- Examples
- Video example
Overview

```
merge 1:1 varlist ... specifies a one-to-one match merge. varlist specifies variables common to both datasets that together uniquely identify single observations in both datasets. For instance, suppose you have a dataset of customer information, called `customer.dta`, and have a second dataset of other information about roughly the same customers, called `other.dta`. Suppose further that both datasets identify individuals by using the `pid` variable, and there is only one observation per individual in each dataset. You would merge the two datasets by typing

```
use customer
merge 1:1 pid using other
```

Reversing the roles of the two files would be fine. Choosing which dataset is the master and which is the using matters only if there are overlapping variable names. 1:1 merges are less common than 1:m and m:1 merges.

```
merge 1:m varlist ... specifies a one-to-many match merge.
use hospitals
merge 1:m hospitalid using discharges
```

would join the discharge data to the hospital data. This is a 1:m merge because `hospitalid` uniquely identifies individual observations in the dataset in memory (`hospitals`), but could correspond to many observations in the using dataset.

```
merge m:1 varlist ... specifies a many-to-one match merge.
use discharges
merge m:1 hospitalid using hospitals
```

would join the hospital data to the discharge data. This is an m:1 merge because `hospitalid` can correspond to many observations in the master dataset, but uniquely identifies individual observations in the using dataset.

```
merge m:m varlist ... specifies a many-to-many match merge. This is allowed for completeness, but it is difficult to imagine an example of when it would be useful. For an m:m merge, varlist does not uniquely identify the observations in either dataset. Matching is performed by combining observations with equal values of varlist; within matching values, the first observation in the master dataset is matched with the first matching observation in the using dataset; the second, with the second; and so on. If there is an unequal number of observations within a group, then the last observation of the shorter group is used repeatedly to match with subsequent observations of the longer group. Use of `merge m:m` is not encouraged.

merge 1:1 _n performs a sequential merge. _n is not a variable name; it is Stata syntax for observation number. A sequential merge performs a one-to-one merge on observation number. The first observation of the master dataset is matched with the first observation of the using dataset; the second, with the second; and so on. If there is an unequal number of observations, the remaining observations are unmatched. Sequential merges are dangerous, because they require you to rely on sort order to know that observations belong together. Use this merge at your own risk.
```
Basic description

Think of merge as being master + using = merged result.

Call the dataset in memory the master dataset, and the dataset on disk the using dataset. This way we have general names that are not dependent on individual datasets.

Suppose we have two datasets,

<table>
<thead>
<tr>
<th>master in memory</th>
<th>on disk in file filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>age</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>wgt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>130</td>
</tr>
<tr>
<td>2</td>
<td>180</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
</tr>
</tbody>
</table>

We would like to join together the age and weight information. We notice that the id variable identifies unique observations in both datasets: if you tell me the id number, then I can tell you the one observation that contains information about that id. This is true for both the master and the using datasets.

Because id uniquely identifies observations in both datasets, this is a 1:1 merge. We can bring in the dataset from disk by typing

```
. merge 1:1 id using filename
```

The original data in memory are called the master data. The data in filename.dta are called the using data. After merge, the merged result is left in memory. The id variable is called the key variable. Stata jargon is that the datasets were merged on id.

Observations for id==1 existed in both the master and using datasets and so were combined in the merged result. The same occurred for id==2. For id==5 and id==4, however, no matches were found and thus each became a separate observation in the merged result. Thus each observation in the merged result came from one of three possible sources:

<table>
<thead>
<tr>
<th>Numeric code</th>
<th>Equivalent word</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td>originally appeared in master only</td>
</tr>
<tr>
<td>2</td>
<td>using</td>
<td>originally appeared in using only</td>
</tr>
<tr>
<td>3</td>
<td>match</td>
<td>originally appeared in both</td>
</tr>
</tbody>
</table>
merge encodes this information into new variable _merge, which merge adds to the merged result:

<table>
<thead>
<tr>
<th>in memory</th>
<th>in filename.dta</th>
<th>=</th>
<th>merged result</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>using</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>age</td>
<td>id</td>
<td>wgt</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>1</td>
<td>130</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
<td>2</td>
<td>180</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>4</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
<td>4</td>
<td>.</td>
</tr>
</tbody>
</table>

Note: Above we show the master and using data sorted by id before merging; this was for illustrative purposes. The dataset resulting from a 1:1 merge will have the same data, regardless of the sort order of the master and using datasets.

The formal definition for merge behavior is the following: Start with the first observation of the master. Find the corresponding observation in the using data, if there is one. Record the matched or unmatched result. Proceed to the next observation in the master dataset. When you finish working through the master dataset, work through unused observations from the using data. By default, unmatched observations are kept in the merged data, whether they come from the master dataset or the using dataset.

Remember this formal definition. It will serve you well.

1:1 merges

The example shown above is called a 1:1 merge, because the key variable uniquely identified each observation in each of the datasets.

A variable or variable list uniquely identifies the observations if each distinct value of the variable(s) corresponds to one observation in the dataset.

In some datasets, multiple variables are required to identify the observations. Imagine data obtained by observing patients at specific points in time so that variables pid and time, taken together, identify the observations. Below we have two such datasets and run a 1:1 merge on pid and time,

```
.merge 1:1 pid time using filename
    master + using = merged result
```

<table>
<thead>
<tr>
<th>pid</th>
<th>time</th>
<th>x1</th>
<th>pid</th>
<th>time</th>
<th>x2</th>
<th>pid</th>
<th>time</th>
<th>x1</th>
<th>x2</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>7</td>
<td>14</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>2</td>
<td>9</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>4</td>
<td>.</td>
<td>14</td>
<td>4</td>
<td>0</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
<td>16</td>
<td>2</td>
<td>3</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td>1</td>
<td>5</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>.</td>
<td>17</td>
<td>2</td>
<td>.</td>
<td>17</td>
<td>2</td>
<td>.</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

This is a 1:1 merge because the combination of the values of pid and time uniquely identifies observations in both datasets.

By default, there is nothing about a 1:1 merge that implies that all, or even any of, the observations match. Above five observations matched, one observation was only in the master (subject 14 at time 4), and another was only in the using (subject 17 at time 2).
**m:1 merges**

In an m:1 merge, the key variable or variables uniquely identify the observations in the using data, but not necessarily in the master data. Suppose you had person-level data within regions and you wished to bring in regional data. Here is an example:

```
. merge m:1 region using filename
    master + using = merged result
```

<table>
<thead>
<tr>
<th>id</th>
<th>region</th>
<th>a</th>
<th>region x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>29</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>22</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

To bring in the regional information, we need to merge on `region`. The values of `region` identify individual observations in the using data, but it is not an identifier in the master data.

We show the merged dataset sorted by `id` because this makes it easier to see how the merged dataset was constructed. For each observation in the master data, `merge` finds the corresponding observation in the using data. `merge` combines the values of the variables in the using dataset to the observations in the master dataset.

**1:m merges**

1:m merges are similar to m:1, except that now the key variables identify unique observations in the master dataset. Any datasets that can be merged using an m:1 merge may be merged using a 1:m merge by reversing the roles of the master and using datasets. Here is the same example as used previously, with the master and using datasets reversed:

```
. merge 1:m region using filename
    master + using = merged result
```

<table>
<thead>
<tr>
<th>region x</th>
<th>id</th>
<th>region</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>region x</th>
<th>id</th>
<th>region</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>2</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

This merged result is identical to the merged result in the previous section, except for the sort order and the contents of `_merge`. This time, we show the merged result sorted by `region` rather than `id`. Reversing the roles of the files causes a reversal in the 1s and 2s for `_merge`: where `_merge` was previously 1, it is now 2, and vice versa. These exchanged `_merge` values reflect the reversed roles of the master and using data.

For each observation in the master data, `merge` found the corresponding observation(s) in the using data and then wrote down the matched or unmatched result. Once the master observations were exhausted, `merge` wrote down any observations from the using data that were never used.
m:m merges

m:m specifies a many-to-many merge and is a bad idea. In an m:m merge, observations are matched within equal values of the key variable(s), with the first observation being matched to the first; the second, to the second; and so on. If the master and using have an unequal number of observations within the group, then the last observation of the shorter group is used repeatedly to match with subsequent observations of the longer group. Thus m:m merges are dependent on the current sort order—something which should never happen.

Because m:m merges are such a bad idea, we are not going to show you an example. If you think that you need an m:m merge, then you probably need to work with your data so that you can use a 1:m or m:1 merge. Tips for this are given in Troubleshooting m:m merges below.

Sequential merges

In a sequential merge, there are no key variables. Observations are matched solely on their observation number:

```
. merge 1:1 _n using filename
```

```
master + using = merged result
```

<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x1</th>
<th>x2</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>30</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>5</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

In the example above, the using data are longer than the master, but that could be reversed. In most cases where sequential merges are appropriate, the datasets are expected to be of equal length, and you should type

```
. merge 1:1 _n using filename, assert(match) nogenerate
```

Sequential merges, like m:m merges, are dangerous. Both depend on the current sort order of the data.

Treatment of overlapping variables

When performing merges of any type, the master and using datasets may have variables in common other than the key variables. We will call such variables overlapping variables. For instance, if the variables in the master and using datasets are

```
master: id, region, sex, age, race
using: id, sex, bp, race
```

and id is the key variable, then the overlapping variables are sex and race.

By default, merge treats values from the master as inviolable. When observations match, it is the master's values of the overlapping variables that are recorded in the merged result.
If you specify the `update` option, however, then all missing values of overlapping variables in matched observations are replaced with values from the using data. Because of this new behavior, the merge codes change somewhat. Codes 1 and 2 keep their old meaning. Code 3 splits into codes 3, 4, and 5. Codes 3, 4, and 5 are filtered according to the following rules; the first applicable rule is used.

5 corresponds to matched observations where at least one overlapping variable had conflicting nonmissing values.

4 corresponds to matched observations where at least one missing value was updated, but there were no conflicting nonmissing values.

3 means observations matched, and there were neither updated missing values nor conflicting nonmissing values.

If you specify both the `update` and `replace` options, then the `_merge==5` cases are updated with values from the using data.

**Sort order**

As we have mentioned, in the 1:1, 1:m, and m:1 match merges, the sort orders of the master and using datasets do not affect the data in the merged dataset. This is not the case of m:m, which we recommend you never use.

Sorting is used by `merge` internally for efficiency, so the merged result can be produced most quickly when the master and using datasets are already sorted by the key variable(s) before merging. You are not required to have the dataset sorted before using `merge`, however, because `merge` will sort behind the scenes, if necessary. If the using dataset is not sorted, then a temporary copy is made and sorted to ensure that the current sort order on disk is not affected.

All of this is to reassure you that 1) your datasets on disk will not be modified by `merge` and 2) despite the fact that our discussion has ignored sort issues, `merge` is, in fact, efficient behind the scenes.

It hardly makes any difference in run times, but if you know that the master and using data are already sorted by the key variable(s), then you can specify the `sorted` option. All that will be saved is the time `merge` would spend discovering that fact for itself.

The merged result produced by `merge` orders the variables and observations in a special and sometimes useful way. If you think of datasets as tables, then the columns for the new variables appear to the right of what was the master. If the master data originally had \( k \) variables, then the new variables will be the \((k+1)\)st, \((k+2)\)nd, and so on. The new observations are similarly ordered so that they all appear at the end of what was the master. If the master originally had \( N \) observations, then the new observations, if any, are the \((N+1)\)st, \((N+2)\)nd, and so on. Thus the original master data can be found from the merged result by extracting the first \( k \) variables and first \( N \) observations.

If `merge` with the `update` option was specified, however, then be aware that the extracted master may have some updated values.

If you care about the ordering of observations in the data after a merge, then you should sort the data after the merge. You should sort it in such a way that it has a unique ordering; see `Sorting with ties in [D] sort`. If, against this recommendation, you wish to have a reproducible ordering after a merge, then read the next paragraph. But be forewarned; just because something is reproducible does not mean it is useful. Again, see `Sorting with ties`.

The resulting dataset after any merge is unsorted. That is to say, if you type `describe`, the “Sorted by” result will be empty. That is not to say that the data will not be ordered; a dataset always has an order. After 1:1 merges, the ordering will always be in the original order of the master dataset,
Troubleshooting m:m merges

First, if you think you need to perform an m:m merge, then we suspect you are wrong. If you
would like to match every observation in the master to every observation in the using with the same
values of the key variable(s), then you should be using joinby; see [D] joinby.

If you still want to use merge, then it is likely that you have forgotten one or more key variables that
could be used to identify observations within groups. Perhaps you have panel data with 4 observations
on each subject, and you are thinking that what you need to do is

```
   . merge m:m subjectid using filename
```

Ask yourself if you have a variable that identifies observation within panel, such as a sequence
number or a time. If you have, say, a time variable, then you probably should try something like

```
   . merge 1:m subjectid time using filename
```

(You might need a 1:1 or m:1 merge; 1:m was arbitrarily chosen for the example.)

If you do not have a time or time-like variable, then ask yourself if there is a meaning to matching
the first observations within subject, the second observations within subject, and so on. If so, then
there is a concept of sequence within subject.

Suppose you do indeed have a sequence concept, but in your dataset it is recorded via the ordering
of the observations. Here you are in a dangerous situation because any kind of sorting would lose
the identity of the first, second, and nth observation within subject. Your first goal should be to fix
this problem by creating an explicit sequence variable from the current ordering—your merge can
come later.

Start with your master data. Type

```
   . sort subjectid, stable
   . by subjectid: generate seqnum = _n
```

Do not omit sort’s stable option. That is what will keep the observations in the same order
within subject. Save the data. Perform these same three steps on your using data.

After fixing the datasets, you can now type

```
   . merge 1:m subjectid seqnum using filename
```

If you do not think there is a meaning to being the first, second, and nth observation within subject,
then you need to ask yourself what it means to match the first observations within subject, the
second observations within subject, and so on. Would it make equal sense to match the first with
the third, the second with the fourth, or any other haphazard matching? If so, then there is no real
ordering, so there is no real meaning to merging. You are about to obtain a haphazard result; you
need to rethink your merge.
Examples

> Example 1: A 1:1 merge

We have two datasets, one of which has information about the size of old automobiles, and the other of which has information about their expense:

```
. use https://www.stata-press.com/data/r17/autosize
(1978 automobile data)
. list
```

<table>
<thead>
<tr>
<th>make</th>
<th>weight</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toyota Celica</td>
<td>2,410</td>
<td>174</td>
</tr>
<tr>
<td>BMW 320i</td>
<td>2,650</td>
<td>177</td>
</tr>
<tr>
<td>Cad. Seville</td>
<td>4,290</td>
<td>204</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>3,210</td>
<td>201</td>
</tr>
<tr>
<td>Datsun 210</td>
<td>2,020</td>
<td>165</td>
</tr>
<tr>
<td>Plym. Arrow</td>
<td>3,260</td>
<td>170</td>
</tr>
</tbody>
</table>

```
. use https://www.stata-press.com/data/r17/autoexpense
(1978 automobile data)
. list
```

<table>
<thead>
<tr>
<th>make</th>
<th>price</th>
<th>mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toyota Celica</td>
<td>5,899</td>
<td>18</td>
</tr>
<tr>
<td>BMW 320i</td>
<td>9,735</td>
<td>25</td>
</tr>
<tr>
<td>Cad. Seville</td>
<td>15,906</td>
<td>21</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>5,222</td>
<td>19</td>
</tr>
<tr>
<td>Datsun 210</td>
<td>4,589</td>
<td>35</td>
</tr>
</tbody>
</table>
```

We can see that these datasets contain different information about nearly the same cars—the autosize file has one more car. We would like to get all the information about all the cars into one dataset.

Because we are adding new variables to old variables, this is a job for the `merge` command. We need only to decide what type of match merge we need.

Looking carefully at the datasets, we see that the `make` variable, which identifies the cars in each of the two datasets, also identifies individual observations within the datasets. What this means is that if you tell me the make of car, I can tell you the one observation that corresponds to that car. Because this is true for both datasets, we should use a 1:1 merge.
We will start with a clean slate to show the full process:

```
. use https://www.stata-press.com/data/r17/autosize
(1978 automobile data)
. merge 1:1 make using https://www.stata-press.com/data/r17/autoexpense
```

Result Number of obs
Not matched 1
  from master 1 ( _merge==1)
  from using 0 ( _merge==2)
Matched 5 ( _merge==3)

```
. list

<table>
<thead>
<tr>
<th>make</th>
<th>weight</th>
<th>length</th>
<th>price</th>
<th>mpg</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMW 320i</td>
<td>2,650</td>
<td>177</td>
<td>9,735</td>
<td>25</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>Cad. Seville</td>
<td>4,290</td>
<td>204</td>
<td>15,906</td>
<td>21</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>Datsun 210</td>
<td>2,020</td>
<td>165</td>
<td>4,589</td>
<td>35</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>Plym. Arrow</td>
<td>3,260</td>
<td>170</td>
<td>.</td>
<td>.</td>
<td>Master only (1)</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>3,210</td>
<td>201</td>
<td>5,222</td>
<td>19</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>Toyota Celica</td>
<td>2,410</td>
<td>174</td>
<td>5,899</td>
<td>18</td>
<td>Matched (3)</td>
</tr>
</tbody>
</table>
```

The merge is successful—all the data are present in the combined dataset, even that from the one car that has only size information. If we wanted only those makes for which all information is present, it would be up to us to drop the observations for which _merge < 3.

➤ Example 2: Requiring matches

Suppose we had the same setup as in the previous example, but we erroneously think that we have all the information on all the cars. We could tell `merge` that we expect only matches by using the `assert` option.

```
. use https://www.stata-press.com/data/r17/autosize, clear
(1978 automobile data)
. merge 1:1 make using https://www.stata-press.com/data/r17/autoexpense,
  > assert(match)
```

```
merge: after merge, not all observations matched
     (merged result left in memory)
     r(9);  
```

`merge` tells us that there is a problem with our assumption. To see how many mismatches there were, we can tabulate _merge:
If we would like to list the problem observation, we can type

```
.list if _merge < 3
```

```
<table>
<thead>
<tr>
<th>make</th>
<th>weight</th>
<th>length</th>
<th>price</th>
<th>mpg</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plym. Arrow</td>
<td>3,260</td>
<td>170</td>
<td>.</td>
<td>.</td>
<td>master only (1)</td>
</tr>
</tbody>
</table>
```

If we were convinced that all data should be complete in the two datasets, we would have to rectify the mismatch in the original datasets.

### Example 3: Keeping just the matches

Once again, suppose that we had the same datasets as before, but this time we want the final dataset to have only those observations for which there is a match. We do not care if there are mismatches—all that is important are the complete observations. By using the `keep(match)` option, we will guarantee that this happens. Because we are keeping only those observations for which the key variable matches, there is no need to generate the `_merge` variable. We could do the following:

```
.use https://www.stata-press.com/data/r17/autosize, clear
(1978 automobile data)
.merge 1:1 make using https://www.stata-press.com/data/r17/autoexpense, > keep(match) nogenerate
```

```
Result       Number of obs
Not matched  0
Matched      5
```

```
<table>
<thead>
<tr>
<th>make</th>
<th>weight</th>
<th>length</th>
<th>price</th>
<th>mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMW 320i</td>
<td>2,650</td>
<td>177</td>
<td>9,735</td>
<td>25</td>
</tr>
<tr>
<td>Cad. Seville</td>
<td>4,290</td>
<td>204</td>
<td>15,906</td>
<td>21</td>
</tr>
<tr>
<td>Datsun 210</td>
<td>2,020</td>
<td>165</td>
<td>4,589</td>
<td>35</td>
</tr>
<tr>
<td>Pont. Grand Prix</td>
<td>3,210</td>
<td>201</td>
<td>5,222</td>
<td>19</td>
</tr>
<tr>
<td>Toyota Celica</td>
<td>2,410</td>
<td>174</td>
<td>5,899</td>
<td>18</td>
</tr>
</tbody>
</table>
```

### Example 4: Many-to-one matches

We have two datasets: one has salespeople in regions and the other has regional data about sales. We would like to put all the information into one dataset. Here are the datasets:
. use https://www.stata-press.com/data/r17/sforce, clear  
(Sales Force)

. list

<table>
<thead>
<tr>
<th>region</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Cntrl</td>
<td>Krantz</td>
</tr>
<tr>
<td>N Cntrl</td>
<td>Phipps</td>
</tr>
<tr>
<td>N Cntrl</td>
<td>Willis</td>
</tr>
<tr>
<td>NE</td>
<td>Ecklund</td>
</tr>
<tr>
<td>NE</td>
<td>Franks</td>
</tr>
<tr>
<td>South</td>
<td>Anderson</td>
</tr>
<tr>
<td>South</td>
<td>Dubnoff</td>
</tr>
<tr>
<td>South</td>
<td>Lee</td>
</tr>
<tr>
<td>South</td>
<td>McNeil</td>
</tr>
<tr>
<td>West</td>
<td>Charles</td>
</tr>
<tr>
<td>West</td>
<td>Cobb</td>
</tr>
<tr>
<td>West</td>
<td>Grant</td>
</tr>
</tbody>
</table>

. use https://www.stata-press.com/data/r17/dollars  
(Regional Sales & Costs)

. list

<table>
<thead>
<tr>
<th>region</th>
<th>sales</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Cntrl</td>
<td>419,472</td>
<td>227,677</td>
</tr>
<tr>
<td>NE</td>
<td>360,523</td>
<td>138,097</td>
</tr>
<tr>
<td>South</td>
<td>532,399</td>
<td>330,499</td>
</tr>
<tr>
<td>West</td>
<td>310,565</td>
<td>165,348</td>
</tr>
</tbody>
</table>

We can see that the region would be used to match observations in the two datasets, and this time we see that region identifies individual observations in the dollars dataset but not in the sforce dataset. This means we will have to use either an m:1 or a 1:m merge. Here we will open the sforce dataset and then merge the dollars dataset. This will be an m:1 merge, because region does not identify individual observations in the dataset in memory but does identify them in the using dataset. Here is the command and its result:

. use https://www.stata-press.com/data/r17/sforce  
(Sales Force)

. merge m:1 region using https://www.stata-press.com/data/r17/dollars  
(label region already defined)

<table>
<thead>
<tr>
<th>Result</th>
<th>Number of obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not matched</td>
<td>0</td>
</tr>
<tr>
<td>Matched</td>
<td>12 batually (_merge==3)</td>
</tr>
</tbody>
</table>
. list

<table>
<thead>
<tr>
<th>region</th>
<th>name</th>
<th>sales</th>
<th>cost</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Ctrl</td>
<td>Krantz</td>
<td>419,472</td>
<td>227,677</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>N Ctrl</td>
<td>Phipps</td>
<td>419,472</td>
<td>227,677</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>N Ctrl</td>
<td>Willis</td>
<td>419,472</td>
<td>227,677</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>NE</td>
<td>Ecklund</td>
<td>360,523</td>
<td>138,097</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>NE</td>
<td>Franks</td>
<td>360,523</td>
<td>138,097</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>South</td>
<td>Anderson</td>
<td>532,399</td>
<td>330,499</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>South</td>
<td>Dubnoff</td>
<td>532,399</td>
<td>330,499</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>South</td>
<td>Lee</td>
<td>532,399</td>
<td>330,499</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>South</td>
<td>McNeil</td>
<td>532,399</td>
<td>330,499</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>West</td>
<td>Charles</td>
<td>310,565</td>
<td>165,348</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>West</td>
<td>Cobb</td>
<td>310,565</td>
<td>165,348</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>West</td>
<td>Grant</td>
<td>310,565</td>
<td>165,348</td>
<td>Matched (3)</td>
</tr>
</tbody>
</table>

We can see from the result that all the values of `region` were matched in both datasets. This is a rare occurrence in practice!

Had we had the `dollars` dataset in memory and merged in the `sforce` dataset, we would have done a 1:m merge.

We would now like to use a series of examples that shows how `merge` treats nonkey variables, which have the same names in the two datasets. We will call these “overlapping” variables.

### Example 5: Overlapping variables

Here are two datasets whose only purpose is for this illustration:

. use https://www.stata-press.com/data/r17/overlap1, clear
. list, sepby(id)

<table>
<thead>
<tr>
<th>id</th>
<th>seq</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>6</td>
<td>a</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

. use https://www.stata-press.com/data/r17/overlap2
We can see that `id` can be used as the key variable for putting the two datasets together. We can also see that there are two overlapping variables: `x1` and `x2`.

We will start with a simple `m:1` merge:

```
. use https://www.stata-press.com/data/r17/overlap1
. merge m:1 id using https://www.stata-press.com/data/r17/overlap2
```

<table>
<thead>
<tr>
<th>Result</th>
<th>Number of obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not matched</td>
<td>2</td>
</tr>
<tr>
<td>from master</td>
<td>1 (_merge==1)</td>
</tr>
<tr>
<td>from using</td>
<td>1 (_merge==2)</td>
</tr>
<tr>
<td>Matched</td>
<td>14 (_merge==3)</td>
</tr>
</tbody>
</table>

```
. list, sepby(id)
```

Careful inspection shows that for the matched `id`, the values of `x1` and `x2` are still the values that were originally in the `overlap1` dataset. This is the default behavior of `merge`—the data in the master dataset are the authority and are kept intact.
Example 6: Updating missing data

Now we would like to investigate the update option. Used by itself, it will replace missing values in the master dataset with values from the using dataset:

```stata
. use https://www.stata-press.com/data/r17/overlap1, clear
. merge m:1 id using https://www.stata-press.com/data/r17/overlap2, update
```

<table>
<thead>
<tr>
<th>Result</th>
<th>Number of obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not matched</td>
<td>2</td>
</tr>
<tr>
<td>from master</td>
<td>1 (_merge==1)</td>
</tr>
<tr>
<td>from using</td>
<td>1 (_merge==2)</td>
</tr>
<tr>
<td>Matched</td>
<td>14</td>
</tr>
<tr>
<td>not updated</td>
<td>5 (_merge==3)</td>
</tr>
<tr>
<td>missing updated</td>
<td>4 (_merge==4)</td>
</tr>
<tr>
<td>nonmissing conflict</td>
<td>5 (_merge==5)</td>
</tr>
</tbody>
</table>

```stata
. list, sepsby(id)
```

<table>
<thead>
<tr>
<th>id</th>
<th>seq</th>
<th>x1</th>
<th>x2</th>
<th>bar</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>11</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>12</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>.</td>
<td>2</td>
<td>12</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>5</td>
<td>.</td>
<td>1</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>6</td>
<td>.</td>
<td>2</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>.</td>
<td>.a</td>
<td>14</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>.</td>
<td>.a</td>
<td>14</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>4</td>
<td>.</td>
<td>.a</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>.</td>
<td>Master only (1)</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>.</td>
<td>1</td>
<td>18</td>
<td>Using only (2)</td>
</tr>
</tbody>
</table>

Looking through the resulting dataset observation by observation, we can see both what the update option updated as well as how the _merge variable gets its values.

The following is a listing that shows what is happening, where x1_m and x2_m come from the master dataset (overlap1), x1_u and x2_u come from the using dataset (overlap2), and x1 and x2 are the values that appear when using merge with the update option.
From this, we can see two important facts: if there are both a conflict and an updated value, the value of _merge will reflect that there was a conflict, and missing values in the master dataset are updated by missing values in the using dataset.

Example 7: Updating all common observations

We would like to see what happens if the update and replace options are specified. The replace option extends the action of update to use nonmissing values of the using dataset to replace values in the master dataset. The values of _merge are unaffected by using both update and replace.

```
. use https://www.stata-press.com/data/r17/overlap1, clear
. merge m:1 id using https://www.stata-press.com/data/r17/overlap2, update replace
```

<table>
<thead>
<tr>
<th>id</th>
<th>x1_m</th>
<th>x1_u</th>
<th>x1</th>
<th>x2_m</th>
<th>x2_u</th>
<th>x2</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>matched (3)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>missing updated (4)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>nonmissing conflict (5)</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>.</td>
<td>1</td>
<td>.</td>
<td>2</td>
<td>1</td>
<td>nonmissing conflict (5)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>matched (3)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>1</td>
<td>nonmissing conflict (5)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>matched (3)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>nonmissing conflict (5)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>.a</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>missing updated (4)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>.a</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td>1</td>
<td>nonmissing conflict (5)</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.a</td>
<td>.a</td>
<td>matched (3)</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.a</td>
<td>1</td>
<td>matched (3)</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.a</td>
<td>.a</td>
<td>missing updated (4)</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>.a</td>
<td>.</td>
<td>.</td>
<td>.a</td>
<td>.a</td>
<td>missing updated (4)</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>5</td>
<td>.</td>
<td>5</td>
<td>8</td>
<td>.</td>
<td>master only (1)</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>1</td>
<td>using only (2)</td>
</tr>
</tbody>
</table>

```

Not matched 2
from master 1 (_merge==1)
from using 1 (_merge==2)
Matched 14
not updated 5 (_merge==3)
missing updated 4 (_merge==4)
nonmissing conflict 5 (_merge==5)
. list, sepby(id)

<table>
<thead>
<tr>
<th>id</th>
<th>seq</th>
<th>x1</th>
<th>x2</th>
<th>bar</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>5</td>
<td>.</td>
<td>1</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>6</td>
<td>.</td>
<td>1</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1</td>
<td>.a</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>3</td>
<td>.a</td>
<td>1</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>4</td>
<td>.a</td>
<td>1</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>Master only (1)</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>Using only (2)</td>
</tr>
</tbody>
</table>

Example 8: More on the keep() option

Suppose we would like to use the update option, as we did above, but we would like to keep only those observations for which the value of the key variable, id, was found in both datasets. This will be more complicated than in our earlier example, because the update option splits the matches into matches, match_updates, and match_conflicts. We must either use all of these code words in the keep option or use their numerical equivalents, 3, 4, and 5. Here the latter is simpler.

. use https://www.stata-press.com/data/r17/overlap1, clear
. merge m:1 id using https://www.stata-press.com/data/r17/overlap2, update > keep(3 4 5)

<table>
<thead>
<tr>
<th>Result</th>
<th>Number of obs</th>
<th>_merge==3</th>
<th>_merge==4</th>
<th>_merge==5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not matched</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matched</td>
<td>14</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Example 9: A one-to-many merge

As a final example, we would like to show one example of a 1:m merge. There is nothing conceptually different here; what is interesting is the order of the observations in the final dataset:

```
. use https://www.stata-press.com/data/r17/overlap2, clear
. merge 1:m id using https://www.stata-press.com/data/r17/overlap1
```

<table>
<thead>
<tr>
<th>Result</th>
<th>Number of obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not matched</td>
<td>2</td>
</tr>
<tr>
<td>from master</td>
<td>1 (_merge=1)</td>
</tr>
<tr>
<td>from using</td>
<td>1 (_merge=2)</td>
</tr>
<tr>
<td>Matched</td>
<td>14 (_merge=3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>seq</th>
<th>x1</th>
<th>x2</th>
<th>bar</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>11</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>12</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>.</td>
<td>12</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>12</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>12</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>5</td>
<td>.</td>
<td>12</td>
<td>Missing updated (4)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>6</td>
<td>.</td>
<td>12</td>
<td>Nonmissing conflict (5)</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1</td>
<td>.</td>
<td>.a</td>
<td>14 Matched (3)</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>2</td>
<td>.</td>
<td>14</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>3</td>
<td>.</td>
<td>.a</td>
<td>14 Missing updated (4)</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>4</td>
<td>.</td>
<td>.a</td>
<td>14 Missing updated (4)</td>
</tr>
</tbody>
</table>
. list, sepby(id)

<table>
<thead>
<tr>
<th>id</th>
<th>bar</th>
<th>x1</th>
<th>x2</th>
<th>seq</th>
<th>_merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>.a</td>
<td>.a</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td>Master only (1)</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>2</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>3</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>4</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>5</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>6</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>.a</td>
<td>.a</td>
<td>2</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>.a</td>
<td>.a</td>
<td>3</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>.a</td>
<td>.a</td>
<td>4</td>
<td>Matched (3)</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>Using only (2)</td>
</tr>
</tbody>
</table>

We can see here that the first four observations come from the master dataset, and all additional observations, whether matched or unmatched, come below these observations. This illustrates that the master dataset is always in the upper-left corner of the merged dataset.

Video example

How to merge files into a single dataset

References


Also see

[D] **append** — Append datasets

[D] **cross** — Form every pairwise combination of two datasets

[D] **frget** — Copy variables from linked frame

[D] **frlink** — Link frames

[D] **joinby** — Form all pairwise combinations within groups

[D] **save** — Save Stata dataset

[U] 23 Combining datasets
Missing values — Quick reference for missing values

Description

This entry provides a quick reference for Stata’s missing values.

Remarks and examples

Stata has 27 numeric missing values:

., the default, which is called the system missing value or sysmiss

and

.a, .b, .c, ..., .z, which are called the extended missing values.

Numeric missing values are represented by large positive values. The ordering is

all nonmissing numbers < . < .a < .b < ... < .z

Thus the expression

    age > 60

is true if variable age is greater than 60 or missing.

To exclude missing values, ask whether the value is less than ‘.’.

    . list if age > 60 & age < .

To specify missing values, ask whether the value is greater than or equal to ‘.’. For instance,

    . list if age >=.

Stata has one string missing value, which is denoted by "" (blank).

References


Also see

[U] 12.2.1 Missing values
mkdir — Create directory

Description

mkdir creates a new directory (folder).

Quick start

Create mysubdir in the current working directory

```
mkdir mysubdir
```

As above, but make mysubdir readable by everyone regardless of default permissions

```
mkdir mysubdir, public
```

Create mysubdir in C:\mydir using Stata for Windows

```
mkdir c:\mydir\mysubdir
```

Create mysubdir in ~/mydir using Stata for Mac or Unix

```
mkdir ~/mydir/mysubdir
```

Create my folder in C:\my dir using Stata for Windows

```
mkdir "c:\my dir\my folder"
```

Syntax

```
mkdir directoryname [, public]
```

Double quotes may be used to enclose directoryname, and the quotes must be used if directoryname contains embedded spaces.

Option

public specifies that directoryname be readable by everyone; otherwise, the directory will be created according to the default permissions of your operating system.
Remarks and examples

Examples:

Windows

. mkdir myproj
. mkdir c:\projects\myproj
. mkdir "c:\My Projects\Project 1"

Mac and Unix

. mkdir myproj
. mkdir ~/projects/myproj

Also see

[D] cd — Change directory
[D] copy — Copy file from disk or URL
[D] dir — Display filenames
[D] erase — Erase a disk file
[D] rmdir — Remove directory
[D] shell — Temporarily invoke operating system
[D] type — Display contents of a file
[U] 11.6 Filenaming conventions
mvencode — Change missing values to numeric values and vice versa

Description

cmpencode
changes missing values in the specified varlist to numeric values.

mvdecode
changes occurrences of a numlist in the specified varlist to a missing-value code.

Missing-value codes may be sysmiss (.) and the extended missing-value codes .a, .b, ..., .z.

String variables in varlist are ignored.

Quick start

Replace all missing values in v1 with 99

mvencode v1, mv(99)

Replace extended missing value .a with 888 and .b with 999 in v2

mvencode v2, mv(.a=888 \ .b=999)

Replace .a with 888, .b with 999, and other missing values with 99 in numeric variables

mvencode _all, mv(.a=888 \ .b=999 \ else=99)

As above, but only for observations where catvar equals 1

mvencode _all if catvar==1, mv(.a=888 \ .b=999 \ else=99)

Replace 888 and 999 with system missing . in all numeric variables

mvdecode _all, mv(888 999)

As above, but replace 888 with .a and 999 with .b

mvdecode _all, mv(888=.a \ 999=.b)

Menu

mvencode

Data > Create or change data > Other variable-transformation commands > Change missing values to numeric

mvdecode

Data > Create or change data > Other variable-transformation commands > Change numeric values to missing
Syntax

Change missing values to numeric values

```
mvencode varlist [if] [in], mv(# | mvc=# [ \ mvc=#... ] [ \ else=#]) [override]
```

Change numeric values to missing values

```
mvdecode varlist [if] [in], mv(numlist | numlist=mvc [ \ numlist=mvc ... ])
```

where mvc is one of . | .a | .b | ... | .z

Options

- **Main**

  `mv(# | mvc=# [ \ mvc=#... ] [ \ else=#])` is required and specifies the numeric values to which the missing values are to be changed.

  `mv(#)` specifies that all types of missing values be changed to #.

  `mv(mvc=##)` specifies that occurrences of missing-value code mvc be changed to #. Multiple transformation rules may be specified, separated by a backward slash (\). The list may be terminated by the special rule else=#, specifying that all types of missing values not yet transformed be set to #.

  Examples: `mv(9), mv(.=99\.a=98\.b=97), mv(.=99 \ else=98)`

  `mv(numlist | numlist=mvc [ \ numlist=mvc ... ])` is required and specifies the numeric values that are to be changed to missing values.

  `mv(numlist=mvc)` specifies that the values in numlist be changed to missing-value code mvc. Multiple transformation rules may be specified, separated by a backward slash (\). See [P] numlist for the syntax of a numlist.

  Examples: `mv(9), mv(99=\.98=.a\97=.b), mv(99=\. \ 100/999=.a)`

  override specifies that the protection provided by mvencode be overridden. Without this option, mvencode refuses to make the requested change if any of the numeric values are already used in the data.

Remarks and examples

Remarks are presented under the following headings:

- **Overview**
- **Video example**

Overview

You may occasionally read data in which missing (for example, a respondent failed to answer a survey question or the data were not collected) is coded with a special numeric value. Popular codings are 9, 99, −9, −99, and the like. If missing were encoded as −99, then

```
.mvdecode _all, mv(-99)
```
would translate the special code to the Stata missing value ".". Use this command cautiously because, even if \textasciitilde99 were not a special code, all \textasciitilde99s in the data would be changed to missing.

Sometimes different codes are used to represent different reasons for missing values. For instance, 98 may be used for "refused to answer" and 99 for "not applicable". Extended missing values (.a, .b, and so on) may be used to code these differences.

\begin{verbatim}
. mvdecode _all, mv(98=.a \ 99=.b)
\end{verbatim}

Conversely, you might need to export data to software that does not understand that "." indicates a missing value, so you might code missing with a special numeric value. To change all missings to \textasciitilde99, you could type

\begin{verbatim}
. mvencode _all, mv(-99)
\end{verbatim}

To change extended missing values back to numeric values, type

\begin{verbatim}
. mvencode _all, mv(.a=98 \ .b=99)
\end{verbatim}

This would leave sysmiss and all other extended missing values unchanged. To encode in addition sysmiss . to 999 and all other extended missing values to 97, you might type

\begin{verbatim}
. mvencode _all, mv(.=999 \ .a=98 \ .b=99 \ else=97)
\end{verbatim}

\texttt{mvencode} will automatically recast variables upward, if necessary, so even if a variable is stored as a \texttt{byte}, its missing values can be recoded to, say, 999. Also \texttt{mvencode} refuses to make the change if \texttt{#} (\textasciitilde99 here) is already used in the data, so you can be certain that your coding is unique. You can override this feature by including the \texttt{override} option.

Be aware of another potential problem with encoding and decoding missing values: value labels are not automatically adapted to the changed codings. You have to do this yourself. For example, the value label \texttt{divlabor} maps the value 99 to the string "not applicable". You used \texttt{mvdecode} to recode 99 to \texttt{.a} for all variables that are associated with this label. To fix the value label, clear the mapping for 99 and define it again for \texttt{.a}.

\begin{verbatim}
. label define divlabor 99 "", modify
. label define divlabor .a "not applicable", add
\end{verbatim}

\textbf{Example 1}

Our automobile dataset contains 74 observations and 12 variables. Let’s first attempt to translate the missing values in the data to 1:

\begin{verbatim}
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. mvencode _all, mv(1)
make: string variable ignored
rep78: already 1 in 2 observations
foreign: already 1 in 22 observations
no action taken
r(9);
\end{verbatim}

Our attempt failed. \texttt{mvencode} first informed us that \texttt{make} is a string variable—this is not a problem but is reported merely for our information. String variables are ignored by \texttt{mvencode}. It next informed us that \texttt{rep78} was already coded 1 in 2 observations and that \texttt{foreign} was already coded 1 in 22 observations. Thus 1 would be a poor choice for encoding missing values because, after encoding, we could not tell a real 1 from a coded missing value 1.
We could force `mvencode` to encode the data with 1, anyway, by typing `mvencode _all, mv(1) override`. That would be appropriate if the 1s in our data already represented missing data. They do not, however, so we code missing as 999:

```
. mvencode _all, mv(999)
    make: string variable ignored
    rep78: 5 missing values
```

This worked, and we are informed that the only changes necessary were to 5 observations of `rep78`.

Example 2

Let’s now pretend that we just read in the automobile data from some raw dataset in which all the missing values were coded 999. We can convert the 999s to real missings by typing

```
. mvdecode _all, mv(999)
    make: string variable ignored
    rep78: 5 missing values
```

We are informed that `make` is a string variable, so it was ignored, and that `rep78` contained 5 observations with 999. Those observations have now been changed to contain missing.

Video example

How to convert missing value codes to missing values

Acknowledgment

These versions of `mvencode` and `mvdecode` were written by Jeroen Weesie of the Department of Sociology at Utrecht University, The Netherlands.

Also see

[D] `generate` — Create or change contents of variable
[D] `recode` — Recode categorical variables
notes attaches notes to the dataset in memory. These notes become a part of the dataset and are saved when the dataset is saved and retrieved when the dataset is used; see [D] save and [D] use. notes can be attached generically to the dataset or specifically to a variable within the dataset.

Quick start

Attach “My note about data” to current dataset
notes: My note about data

Add note “There is one note for v1” to v1
notes v1: There is one note for v1

Add note “A note was added to v2 on” and a time stamp for the note
notes v2: A note was added to v2 on TS

Add note “Data have changed” to the dataset
notes: Data have changed

Remove the first note from the dataset
notes drop _dta in 1

Renumber notes after removing a note from the dataset
notes renumber _dta

As above, but for a variable
notes renumber v1

List all notes
notes

List notes for the dataset but omit notes applied to variables
notes _dta

List only notes for variables
notes *

Search all notes for the word “check”
notes search check
Menu

notes (add)
Data > Variables Manager

notes list and notes search
Data > Data utilities > Notes utilities > List or search notes

notes replace
Data > Variables Manager

notes drop
Data > Variables Manager

notes renumber
Data > Data utilities > Notes utilities > Renumber notes
Syntax

Attach notes to dataset

```
notes [evarname] : text
```

List all notes

```
notes
```

List specific notes

```
notes [list] evarlist [in #[/]]
```

Search for a text string across all notes in all variables and _dta

```
notes search [sometext]
```

Replace a note

```
notes replace evarname in # : text
```

Drop notes

```
notes drop evarlist [in #[/]]
```

Renumber notes

```
notes renumber evarname
```

where `evarname` is _dta or a varname, `evarlist` is a varlist that may contain the _dta, and # is a number or the letter 1.

If `text` includes the letters TS surrounded by blanks, the TS is removed, and a time stamp is substituted in its place.

Remarks and examples

Remarks are presented under the following headings:

- How notes are numbered
- Attaching and listing notes
- Selectively listing notes
- Searching and replacing notes
- Deleting notes
- Warnings
- Video example
How notes are numbered

Notes are numbered sequentially, with the first note being 1. Say the `myvar` variable has four notes numbered 1, 2, 3, and 4. If you type `notes drop myvar in 3`, the remaining notes will be numbered 1, 2, and 4. If you now add another note, it will be numbered 5. That is, notes are not renumbered and new notes are added immediately after the highest numbered note. Thus, if you now dropped notes 4 and 5, the next note added would be 3.

You can renumber notes by using `notes renumber`. Going back to when `myvar` had notes numbered 1, 2, and 4 after dropping note 3, if you typed `notes renumber myvar`, the notes would be renumbered 1, 2, and 3. If you added a new note after that, it would be numbered 4.

Attaching and listing notes

A note is nothing formal; it is merely a string of text reminding you to do something, cautioning you against something, or saying anything else you might feel like jotting down. People who work with real data invariably end up with paper notes plastered around their terminal saying things like, “Send the new sales data to Bob”, “Check the income variable in `salary95`; I don’t believe it”, or “The gender dummy was significant!” It would be better if these notes were attached to the dataset.

Adding a note to your dataset requires typing `note` or `notes` (they are synonyms), a colon (``), and whatever you want to remember. The note is added to the dataset currently in memory.

```
    . note: Send copy to Bob once verified.
```

You can replay your notes by typing `notes` (or `note`) by itself.

```
    . notes
    _dta:
     1. Send copy to Bob once verified.
```

Once you resave your data, you can replay the note in the future, too. You add more notes just as you did the first:

```
    . note: Mary wants a copy, too.
    . notes
    _dta:
     1. Send copy to Bob once verified.
     2. Mary wants a copy, too.
```

You can place time stamps on your notes by placing the word `TS` (in capitals) in the text of your note:

```
    . note: TS merged updates from JJ&F
    . notes
    _dta:
     1. Send copy to Bob once verified.
     2. Mary wants a copy, too.
     3. 19 Apr 2020 15:38 merged updates from JJ&F
```

Notes may contain SMCL directives:

```
    . use https://www.stata-press.com/data/r17/auto
       (1978 automobile data)
    . note: check reason for missing values in `rep78'
    . notes
    _dta:
     1. from Consumer Reports with permission
     2. check reason for missing values in `rep78`
```
The notes we have added so far are attached to the dataset generically, which is why Stata prefixes them with _dta when it lists them. You can attach notes to variables:

```
. note mpg: is the 44 a mistake? Ask Bob.
. note mpg: what about the two missing values?
. notes
_dta:
  1. Send copy to Bob once verified.
  2. Mary wants a copy, too.
  3. 19 Apr 2020 15:38 merged updates from JJ&F

_mpg:
  1. is the 44 a mistake? Ask Bob.
  2. what about the two missing values?
```

Up to 9,999 generic notes can be attached to _dta, and another 9,999 notes can be attached to each variable.

**Selectively listing notes**

Typing `notes` by itself lists all the notes. In full syntax, `notes` is equivalent to typing `notes _all in 1/l`. Here are some variations:

```
notes _dta          list all generic notes
notes mpg           list all notes for variable mpg
notes _dta mpg      list all generic notes and mpg notes
notes _dta in 3     list generic note 3
notes _dta in 3/5   list generic notes 3–5
notes mpg in 3/5    list mpg notes 3–5
notes _dta in 3/l   list generic notes 3 through last
```

**Searching and replacing notes**

You had a bad day yesterday, and you want to recheck the notes that you added to your dataset. Fortunately, you always put a time stamp on your notes.

```
. notes search "29 Jan"
_dta:
  2. 29 Jan 2020 13:40 check reason for missing values in foreign
```

Good thing you checked. It is `rep78` that has missing values.

```
. notes replace _dta in 2: TS check reason for missing values in rep78
   (note 2 for _dta replaced)
. notes
_dta:
  1. from Consumer Reports with permission
  2. 30 Jan 2020 12:32 check reason for missing values in rep78
```
Deleting notes

notes drop works much like listing notes, except that typing `notes drop` by itself does not delete all notes; you must type `notes drop _all`. Here are some variations:

- `notes drop _dta` delete all generic notes
- `notes drop _dta in 3` delete generic note 3
- `notes drop _dta in 3/5` delete generic notes 3–5
- `notes drop _dta in 3/l` delete generic notes 3 through last
- `notes drop mpg in 4` delete mpg note 4

Warnings

- Notes are stored with the data, and as with other updates you make to the data, the additions and deletions are not permanent until you save the data; see `[D] save`.
- The maximum length of one note is 67,784 characters for Stata/MP, Stata/SE, and Stata/BE.

Video example

How to add notes to a variable

Reference


Also see

- `[D] codebook` — Describe data contents
- `[D] describe` — Describe data in memory or in file
- `[D] ds` — Compactly list variables with specified properties
- `[D] save` — Save Stata dataset
- `[D] varmanage` — Manage variable labels, formats, and other properties
- `[U] 12.8 Characteristics`
Description

`set obs` changes the number of observations in the current dataset. `#` must be at least as large as the current number of observations. If there are variables in memory, the values of all new observations are set to missing.

Quick start

Add 100 observations with no observations currently in memory
```
set obs 100
```
Add 100 observations with 100 observations currently in memory
```
set obs 200
```

Syntax

```
set obs #
```

Remarks and examples

Example 1

`set obs` can be useful for creating artificial datasets. For instance, if we wanted to graph the function $y = x^2$ over the range 1–100, we could type

```
. drop _all
. set obs 100
Number of observations (_N) was 0, now 100.
. generate x = _n
. generate y = x^2
. scatter y x
(graph not shown)
```
Example 2

If we want to add an extra data point in a program, we could type

```
. local np1 = _N + 1
. set obs `np1'
Number of observations (_N) was 0, now 1.
```

or

```
. set obs `=_N + 1'
```

Also see

[D] describe — Describe data in memory or in file
[D] insobs — Add or insert observations
odbc — Load, write, or view data from ODBC sources

Description

odbc allows you to load, write, and view data from Open DataBase Connectivity (ODBC) sources into Stata. ODBC is a standardized set of function calls for accessing data stored in both relational and nonrelational database-management systems. By default on Unix platforms, iODBC is the ODBC driver manager Stata uses, but you can use unixODBC by using the command `set odbcmgr unixodbc`.

ODBC’s architecture consists of four major components (or layers): the client interface, the ODBC driver manager, the ODBC drivers, and the data sources. Stata provides odbc as the client interface.

The system is illustrated as follows:

```
Client interface
   (Stata)
   odbc list
   odbc query
   odbc describe
   odbc load
   odbc insert
   odbc exec
   odbc sqlfile
```

```
ODBC driver manager
```

```
ODBC driver
```

```
ODBC data source
```

**odbc list** produces a list of ODBC data source names to which Stata can connect.

**odbc query** retrieves a list of table names available from a specified data source’s system catalog.

**odbc describe** lists column names and types associated with a specified table.

**odbc load** reads an ODBC table into memory. You can load an ODBC table specified in the `table()` option or load an ODBC table generated by an SQL SELECT statement specified in the `exec()` option. In both cases, you can choose which columns and rows of the ODBC table to read by specifying `extvarlist` and `if` and `in` conditions. `extvarlist` specifies the columns to be read and allows you to rename variables. For example,

```
odbc load id=ID name="Last Name", table(Employees) dsn(Northwind)
```

reads two columns, ID and Last Name, from the Employees table of the Northwind data source. It will also rename variable ID to id and variable Last Name to name.

**odbc insert** writes data from memory to an ODBC table. The data can be appended to an existing table or replace an existing table.

**odbc exec** allows for most SQL statements to be issued directly to any ODBC data source. Statements that produce output, such as SELECT, have their output neatly displayed. By using Stata’s ado language, you can also generate SQL commands on the fly to do positional updates or whatever the situation requires.

**odbc sqlfile** provides a “batch job” alternative to the `odbc exec` command. A file is specified that contains any number of any length SQL commands. Every SQL command in this file should be delimited by a semicolon and must be constructed as pure SQL. Stata macros and ado-language syntax
are not permitted. The advantage in using this command, as opposed to `odbc exec`, is that only one connection is established for multiple SQL statements. A similar sequence of SQL commands used via `odbc exec` would require constructing an ado-file that issued a command and, thus, a connection for every SQL command. Another slight difference is that any output that might be generated from an SQL command is suppressed by default. A `loud` option is provided to toggle output back on.

`set odbcdriver unicode` specifies that the ODBC driver is a Unicode driver (the default). `set odbcdriver ansi` specifies that the ODBC driver is an ANSI driver. You must restart Stata for the setting to take effect.

`set odbcmgr iodbc` specifies that the ODBC driver manager is iODBC (the default). `set odbcmgr unixodbc` specifies that the ODBC driver manager is unixODBC.

**Quick start**

List all defined data source names (DSNs) to which Stata can connect
```
odbc list
```

List available table names in `MyDSN`
```
odbc query "MyDSN"
```

Describe the column names and data types in table `MyTable` from `MyDSN`
```
odbc describe "MyTable", dsn("MyDSN")
```

Load `MyTable` into memory from `MyDSN`
```
odbc load, table("MyTable") dsn("MyDSN")
```

**Menu**

`odbc load`
```
File > Import > ODBC data source
```

`odbc insert`
```
File > Export > ODBC data source
```
Syntax

List ODBC sources to which Stata can connect

    odbc list

Retrieve available names from specified data source

    odbc query ["DataSourceName", verbose schema connect_options]

List column names and types associated with specified table

    odbc describe ["TableName", connect_options]

Import data from an ODBC data source

    odbc load [extvarlist] [if] [in], {table("TableName")|exec("Stmt")}
    [load_options connect_options]

Export data to an ODBC data source

    odbc insert [varlist] [if] [in], table("TableName")
    {dsn("DataSourceName")|connectionstring("ConnectStr")}
    [insert_options connect_options]

Allow SQL statements to be issued directly to ODBC data source

    odbc exec("Stmt"),
    {dsn("DataSourceName")|connectionstring("ConnectStr")}
    [connect_options]

Batch job alternative to odbc exec

    odbc sqlfile("filename"),
    {dsn("DataSourceName")|connectionstring("ConnectStr")}
    [loud connect_options]

Specify ODBC driver type

    set odbcdriver {unicode|ansi} [, permanently]

Specify ODBC driver manager (Mac and Unix only)

    set odbcmgr {iodbc|unixodbc} [, permanently]
DataSourceName is the name of the ODBC source (database, spreadsheet, etc.)
ConnectStr is a valid ODBC connection string
TableName is the name of a table within the ODBC data source
SqlStmt is an SQL SELECT statement
filename is pure SQL commands separated by semicolons
extvarlist contains
  sqlvarname = varname

connect_options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user(UserID)</td>
<td>user ID of user establishing connection</td>
</tr>
<tr>
<td>password(Password)</td>
<td>password of user establishing connection</td>
</tr>
<tr>
<td>dialog(noprompt)</td>
<td>do not display ODBC connection-information dialog, and do not prompt user for connection information</td>
</tr>
<tr>
<td>dialog(prompt)</td>
<td>display ODBC connection-information dialog</td>
</tr>
<tr>
<td>dialog(complete)</td>
<td>display ODBC connection-information dialog only if there is not enough information</td>
</tr>
<tr>
<td>dialog(required)</td>
<td>display ODBC connection-information dialog only if there is not enough mandatory information provided</td>
</tr>
</tbody>
</table>

*dsn("DataSourceName") name of data source
*connectionstring("ConnectStr") ODBC connection string

*dsn("DataSourceName") is not allowed with odbc query. You may not specify both DataSourceName and connectionstring() with odbc query. Either dsn() or connectionstring() is required with odbc insert, odbc exec, and odbc sqlfile.

load_options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*table(&quot;TableName&quot;)</td>
<td>name of table stored in data source</td>
</tr>
<tr>
<td>*exec(&quot;SqlStmt&quot;)</td>
<td>SQL SELECT statement to generate a table to be read into Stata</td>
</tr>
<tr>
<td>clear</td>
<td>load dataset even if there is one in memory</td>
</tr>
<tr>
<td>noquote</td>
<td>alter Stata’s internal use of SQL commands; seldom used</td>
</tr>
<tr>
<td>lowercase</td>
<td>read variable names as lowercase</td>
</tr>
<tr>
<td>sqlshow</td>
<td>show all SQL commands issued</td>
</tr>
<tr>
<td>allstring</td>
<td>read all variables as strings</td>
</tr>
<tr>
<td>datestring</td>
<td>read date-formatted variables as strings</td>
</tr>
<tr>
<td>multistatement</td>
<td>allow multiple SQL statements delimited by ; when using exec()</td>
</tr>
<tr>
<td>bigintasdouble</td>
<td>store BIGINT columns as Stata doubles on 64-bit operating systems</td>
</tr>
</tbody>
</table>

*Either table("TableName") or exec("SqlStmt") must be specified with odbc load.
**insert_options**

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>name of table stored in data source</td>
<td><em>table(&quot;TableName&quot;)</em></td>
</tr>
<tr>
<td>clear data in ODBC table before data in memory is written to the table</td>
<td>overwrite</td>
</tr>
<tr>
<td>default mode of operation for the odbc insert command</td>
<td>insert</td>
</tr>
<tr>
<td>quote all values with single quotes as they are inserted in ODBC table</td>
<td>quoted</td>
</tr>
<tr>
<td>show all SQL commands issued</td>
<td>sqlshow</td>
</tr>
<tr>
<td>ODBC variables on the data source that correspond to the variables in Stata’s memory</td>
<td>as(&quot;varlist&quot;)</td>
</tr>
<tr>
<td>use block inserts</td>
<td>block</td>
</tr>
</tbody>
</table>

*table("TableName")* is required for odbc insert.

**Options**

`user(UserID)` specifies the user ID of the user attempting to establish the connection to the data source. By default, Stata assumes that the user ID is the same as the one specified in the previous `odbc` command or is empty if `user()` has never been specified in the current session of Stata.

`password(Password)` specifies the password of the user attempting to establish the connection to the data source. By default, Stata assumes that the password is the same as the one previously specified or is empty if the password has not been used during the current session of Stata. Typically, the password() option will not be specified apart from the `user()` option.

`dialog(noprompt | prompt | complete | required)` specifies the mode the ODBC Driver Manager uses to display the ODBC connection-information dialog to prompt for more connection information.

- `noprompt` is the default value. The ODBC connection-information dialog is not displayed, and you are not prompted for connection information. If there is not enough information to establish a connection to the specified data source, an error is returned.
- `prompt` causes the ODBC connection-information dialog to be displayed.
- `complete` causes the ODBC connection-information dialog to be displayed only if there is not enough information, even if the information is not mandatory.
- `required` causes the ODBC connection-information dialog to be displayed only if there is not enough mandatory information provided to establish a connection to the specified data source. You are prompted only for mandatory information; controls for information that is not required to connect to the specified data source are disabled.

`dsn("DataSourceName")` specifies the name of a data source, as listed by the `odbc list` command. If a name contains spaces, it must be enclosed in double quotes. By default, Stata assumes that the data source name is the same as the one specified in the previous `odbc` command. This option is not allowed with `odbc query`. Either the `dsn()` option or the `connectionstring()` option may be specified with `odbc describe` and `odbc load`, and one of these options must be specified with `odbc insert`, `odbc exec`, and `odbc sqlfile`.

`connectionstring("ConnectStr")` specifies a connection string rather than the name of a data source. Stata does not assume that the connection string is the same as the one specified in the previous `odbc` command. Either `DataSourceName` or the `connectionstring()` option may be specified with `odbc query`; either the `dsn()` option or the `connectionstring()` option can be specified with `odbc describe` and `odbc load`, and one of these options must be specified with `odbc insert`, `odbc exec`, and `odbc sqlfile`. 
table("TableName") specifies the name of an ODBC table stored in a specified data source’s system catalog, as listed by the odbc query command. If a table name contains spaces, it must be enclosed in double quotes. Either the table() option or the exec() option—but not both—is required with the odbc load command.

eexec("SqlStmt") allows you to issue an SQL SELECT statement to generate a table to be read into Stata. An error message is returned if the SELECT statement is an invalid SQL statement. The statement must be enclosed in double quotes. Either the table() option or the exec() option—but not both—is required with the odbc load command.
clear permits the data to be loaded, even if there is a dataset already in memory, and even if that dataset has changed since the data were last saved.
nouquote alters Stata’s internal use of SQL commands, specifically those relating to quoted table names, to better accommodate various drivers. This option has been particularly helpful for DB2 drivers.
lowercase causes all the variable names to be read as lowercase.
sqlshow is a useful option for showing all SQL commands issued to the ODBC data source from the odbc insert or odbc load command. This can help you debug any issues related to inserting or loading.
alstring causes all variables to be read as string data types.
datestring causes all date- and time-formatted variables to be read as string data types.
multistatement specifies that multiple SQL statements delimited by ; be allowed when using the exec() option. Some drivers do not support multiple SQL statements.
bigintasdouble specifies that data stored in 64-bit integer (BIGINT) database columns be converted to Stata doubles. If any integer value is larger than 9,007,199,254,740,965 or less than -9,007,199,254,740,992, this conversion is not possible, and odbc load will issue an error message.
overwrite allows data to be cleared from an ODBC table before the data in memory are written to the table. All data from the ODBC table are erased, not just the data from the variable columns that will be replaced.
insert appends data to an existing ODBC table and is the default mode of operation for the odbc insert command.
quoted is useful for ODBC data sources that require all inserted values to be quoted. This option specifies that all values be quoted with single quotes as they are inserted into an ODBC table.
as("varlist") allows you to specify the ODBC variables on the data source that correspond to the variables in Stata’s memory. If this option is specified, the number of variables must equal the number of variables being inserted, even if some names are identical.
loud specifies that output be displayed for SQL commands.
verbose specifies that odbc query list any data source alias, nickname, typed table, typed view, and view along with tables so that you can load data from these table types.
schema specifies that odbc query return schema names with the table names from a data source. Note: The schema names returned from odbc query will also be used with the odbc describe and odbc load commands. When using odbc load with a schema name, you might also need to specify the noquote option because some drivers do not accept quotes around table or schema names.
block specifies that odbc insert use block inserts to speed up data-writing performance. Some drivers do not support block inserts.

permanently (set odbcdriver and set odbcmgr only) specifies that, in addition to making the change right now, the setting be remembered and become the default setting when you invoke Stata.

Remarks and examples

When possible, the examples in this manual entry are developed using the Northwind sample database that is automatically installed with Microsoft Access. If you do not have Access, you can still use odbc, but you will need to consult the documentation for your other ODBC sources to determine how to set them up.

Remarks are presented under the following headings:

- Unicode and ODBC
- Setting up the data sources
- Listing ODBC data source names
- Listing available table names from a specified data source’s system catalog
- Describing a specified table
- Loading data from ODBC sources

Unicode and ODBC

Stata supports accessing databases with Unicode data through Unicode ODBC drivers on the following platforms:

- Microsoft Windows through ODBC driver manager (version 3.5 or higher).
- Unix through unixODBC driver manager with ODBC drivers compiled for unixODBC. Stata does not support Unicode drivers when using iODBC as your driver manager. Stata requires that the driver support UTF-8.
- macOS through unixODBC driver manager with ODBC drivers compiled for unixODBC. Stata does not support Unicode drivers when using iODBC as your driver manager. Stata requires that the driver support UTF-8.

Stata supports non-Unicode databases through ASCII drivers with all driver managers.

Setting up the data sources

Before using Stata’s ODBC commands, you must register your ODBC database with the ODBC Data Source Administrator. This process varies depending on platform, but the following example shows the steps necessary for Windows.

Using Windows 10, follow these steps to create an ODBC User Data Source for the Northwind sample database:

1. On the Start page, type ODBC Data Sources. From the list that appears, select the ODBC Data Sources Desktop App.
2. In the Data Sources (ODBC) dialog box,
   a. click on the User DSN tab;
   b. click on Add...;
c. choose Microsoft Access Driver (*.mdb,*.accdb) on the Create New Data Source dialog box; and
d. click on Finish.

3. In the ODBC Microsoft Access Setup dialog box, type Northwind in the Data Source Name field and click on Select.... Locate the Northwind.mdb database and click on OK to finish creating the data source.

Using Windows 7, follow these steps to create an ODBC User Data Source for the Northwind sample database:

1. From the Start Menu, select the Control Panel.
2. In the Control Panel window, click on System and Security > Administrative Tools.
3. In the Data Sources (ODBC) dialog box,
   a. click on the User DSN tab;
   b. click on Add...;
   c. choose Microsoft Access Driver (*.mdb,*.accdb) on the Create New Data Source dialog box; and
   d. click on Finish.
4. In the ODBC Microsoft Access Setup dialog box, type Northwind in the Data Source Name field and click on Select.... Locate the Northwind.mdb database and click on OK to finish creating the data source.

Technical note

In earlier versions of Windows, the exact location of the Data Source (ODBC) dialog varies, but it is always somewhere within the Control Panel.

Listing ODBC data source names

odbc list is used to produce a list of data source names to which Stata can connect. For a specific data source name to be shown in the list, the data source has to be registered with the ODBC Data Source Administrator. See Setting up the data sources for information on how to do this.

Example 1

```
. odbc list

Data Source Name       Driver
------------------------ -----------------------------------
dBASE Files             Microsoft Access dBASE Driver (*.dbf, *.ndx
Excel Files             Microsoft Excel Driver (*.xls, *.xlsx, *.xl
MS Access Database      Microsoft Access Driver (*.mdb, *.accdb)
Northwind               Microsoft Access Driver (*.mdb, *.accdb)
```

In the above list, Northwind is one of the sample Microsoft Access databases that Access installs by default.
Listing available table names from a specified data source's system catalog

odbc query is used to list table names available from a specified data source.

Example 2

. odbc query "Northwind"
DataSource: Northwind
Path : C:\Program Files\Microsoft Office\Office\Samples\Northwind.accdb

Customers
Employee Privileges
Employees
Inventory Transaction Types
Inventory Transactions
Invoices
Order Details
Order Details Status
Orders
Orders Status
Orders Tax Status
Privileges
Products
Purchase Order Details
Purchase Order Status
Purchase Orders
Sales Reports
Shippers
Strings
Suppliers
Describing a specified table

`odbc describe` is used to list column (variable) names and their SQL data types that are associated with a specified table.

Example 3

Here we specify that we want to list all variables in the `Employees` table of the `Northwind` data source.

```
    odbc describe "Employees", dsn("Northwind")
```

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Variable Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>COUNTER</td>
</tr>
<tr>
<td>Company</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Last Name</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>First Name</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>E-mail Address</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Job Title</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Business Phone</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Home Phone</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Mobile Phone</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Fax Number</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Address</td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>City</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>State/Province</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>ZIP/Postal Code</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Country/Region</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>Web Page</td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>Notes</td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>Attachments</td>
<td>LONGCHAR</td>
</tr>
</tbody>
</table>

Loading data from ODBC sources

`odbc load` is used to load an ODBC table into memory.

To load an ODBC table listed in the `odbc query` output, specify the table name in the `table()` option and the data source name in the `dsn()` option.

Example 4

We want to load the `Employees` table from the `Northwind` data source.

```
    clear
    odbc load, table("Employees") dsn("Northwind")
```

- converted E-mail_Address to var5
- converted State/Province to var13
- converted ZIP/Postal_Code to var14
- converted Country/Region to var15
. describe

Contains data
Observations: 9
Variables: 18

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Company</td>
<td>str17</td>
<td>%17s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last_Name</td>
<td>str14</td>
<td>%14s</td>
<td>Last Name</td>
<td></td>
</tr>
<tr>
<td>First_Name</td>
<td>str7</td>
<td>%9s</td>
<td>First Name</td>
<td></td>
</tr>
<tr>
<td>var5</td>
<td>str28</td>
<td>%28s</td>
<td>E-mail Address</td>
<td></td>
</tr>
<tr>
<td>Job_Title</td>
<td>str21</td>
<td>%21s</td>
<td>Job Title</td>
<td></td>
</tr>
<tr>
<td>Business_Phone</td>
<td>str13</td>
<td>%13s</td>
<td>Business Phone</td>
<td></td>
</tr>
<tr>
<td>Home_Phone</td>
<td>str13</td>
<td>%13s</td>
<td>Home Phone</td>
<td></td>
</tr>
<tr>
<td>Mobile_Phone</td>
<td>str1</td>
<td>%9s</td>
<td>Mobile Phone</td>
<td></td>
</tr>
<tr>
<td>Fax_Number</td>
<td>str13</td>
<td>%13s</td>
<td>Fax Number</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>strL</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>City</td>
<td>str8</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>var13</td>
<td>str2</td>
<td>%9s</td>
<td>State/Province</td>
<td></td>
</tr>
<tr>
<td>var14</td>
<td>str5</td>
<td>%9s</td>
<td>ZIP/Postal Code</td>
<td></td>
</tr>
<tr>
<td>var15</td>
<td>str3</td>
<td>%9s</td>
<td>Country/Region</td>
<td></td>
</tr>
<tr>
<td>Web_Page</td>
<td>strL</td>
<td>%9s</td>
<td>Web Page</td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>strL</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attachments</td>
<td>strL</td>
<td>%9s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.

Technical note

When Stata loads the ODBC table, data are converted from SQL data types to Stata data types. Stata does not support all SQL data types. If the column cannot be read because of incompatible data types, Stata will issue a note and skip a column. The following table lists the supported SQL data types and their corresponding Stata data types:
You can also load an ODBC table generated by an SQL SELECT statement specified in the `exec()` option.

**Example 5**

Suppose that, from the Northwind data source, we want a list of all the customers who have placed orders. We might use the SQL SELECT statement

```sql
SELECT DISTINCT c.ID, c.Company
FROM Customers c
INNER JOIN Orders o
    ON c.[Customer ID] = o.CustomerID
```

To load the table into Stata, we use `odbc load` with the `exec()` option.
. odbc load, exec('"SELECT DISTINCT c.ID, c.Company FROM Customers c INNER JOIN Orders o ON c.ID = o.[Customer ID]"') dsn("Northwind") clear
.describe
Contains data
Observations: 15
Variables: 2

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>long</td>
<td>%12.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Company</td>
<td>str10</td>
<td>%10s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.

The *extvarlist* is optional. It allows you to choose which columns (variables) are to be read and to rename variables when they are read.

> **Example 6**

Suppose that we want to load the ID column and the Last Name column from the Employees table of the Northwind data source. Moreover, we want to rename ID as *id* and Last Name as *name*.

. odbc load id=ID name="Last Name", table("Employees") dsn("Northwind") clear
.describe
Contains data
Observations: 9
Variables: 2

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>long</td>
<td>%12.0g</td>
<td>ID</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>str14</td>
<td>%14s</td>
<td>Last Name</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
Note: Dataset has changed since last saved.

The *if* and *in* qualifiers allow you to choose which rows are to be read. You can also use a *WHERE* clause in the SQL *SELECT* statement to select the rows to be read.
Example 7

Suppose that we want the information from the Order Details table, where Quantity is greater than 50. We can specify the if and in qualifiers,

```
. odbc load if Quantity>50, table("Order Details") dsn("Northwind") clear
. sum Quantity
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantity</td>
<td>10</td>
<td>177.7</td>
<td>94.21966</td>
<td>87</td>
<td>300</td>
</tr>
</tbody>
</table>

or we can issue the SQL SELECT statement directly:

```
. odbc load, exec("SELECT * FROM [Order Details] WHERE Quantity>50")
```

```
> dsn("Northwind") clear
. sum Quantity
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantity</td>
<td>10</td>
<td>177.7</td>
<td>94.21966</td>
<td>87</td>
<td>300</td>
</tr>
</tbody>
</table>

Example 8

To use odbc insert, you must have an SQL table already created in your data source. If you do not, you can use odbc exec to create a table in your data source. For example, one might create a table in an Oracle database with the SQL command below:

```
#delimit ;
llocal cols '" ID NUMBER(5,0),
               PAY NUMBER(8,2),
               TITLE NVARCHAR2(18),
               LOCATION VARCHAR(40)"; ;
#delimit cr
odbc exec("CREATE TABLE JOB_TYPES ('cols');"), dsn(oracle_dsn) ///
   user(username) password(password)
```

You must create a table using the correct data type for each table column for your data to transfer correctly. Note that the SQL syntax to create a table differs across data sources, as do column data types.

Reference


Also see

[D] jdbc — Load, write, or view data from a database with a Java API
[D] export — Overview of exporting data from Stata
[D] import — Overview of importing data into Stata
**Description**

`order` relocates `varlist` to a position depending on which option you specify. If no option is specified, `order` relocates `varlist` to the beginning of the dataset in the order in which the variables are specified.

**Quick start**

Move `v1` to the beginning of the dataset

```
order v1
```

As above, but instead move `v1` to the end of the dataset

```
order v1, last
```

Move `v3` before `v2`

```
order v3, before(v2)
```

Move `x` and `z` after `y`

```
order x z, after(y)
```

Alphabetize `y`, `x`, and `z`, and move them to the beginning of the dataset

```
order y x z, alphabetic
```

Alphabetize `x`, `y`, `z`, `v3`, `v2`, and `v1`, and sort numbers in sequential order

```
order x y z v*, sequential
```

**Menu**

Data > Data utilities > Change order of variables
Syntax

order varlist [ , options ]

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>move varlist to beginning of dataset; the default</td>
</tr>
<tr>
<td>last</td>
<td>move varlist to end of dataset</td>
</tr>
<tr>
<td>before(varname)</td>
<td>move varlist before varname</td>
</tr>
<tr>
<td>after(varname)</td>
<td>move varlist after varname</td>
</tr>
<tr>
<td>alphabetic</td>
<td>alphabetize varlist and move it to beginning of dataset</td>
</tr>
<tr>
<td>sequential</td>
<td>alphabetize varlist keeping numbers sequential and move it to beginning of dataset</td>
</tr>
</tbody>
</table>

Options

first shifts varlist to the beginning of the dataset. This is the default.

last shifts varlist to the end of the dataset.

before(varname) shifts varlist before varname.

after(varname) shifts varlist after varname.

alphabetic alphabetizes varlist and moves it to the beginning of the dataset. For example, here is a varlist in alphabetic order: a x7 x70 x8 x80 z. If combined with another option, alphabetic just alphabetizes varlist, and the movement of varlist is controlled by the other option.

sequential alphabetizes varlist, keeping variables with the same ordered letters but with differing appended numbers in sequential order. varlist is moved to the beginning of the dataset. For example, here is a varlist in sequential order: a x7 x8 x70 x80 z.

Remarks and examples

Example 1

When using order, you must specify a varlist, but you do not need to specify all the variables in the dataset. For example, we want to move the make and mpg variables to the front of the auto dataset.
. use https://www.stata-press.com/data/r17/auto4
(1978 automobile data)
. describe
Contains data from https://www.stata-press.com/data/r17/auto4.dta
Observations: 74  1978 automobile data
Variables: 6  6 Apr 2020  00:20

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>int</td>
<td>%8.0gc</td>
<td>Price</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td>Weight (lbs.)</td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>make</td>
<td>str17</td>
<td>%-17s</td>
<td>Make and model</td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%8.0g</td>
<td>Length (in.)</td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%8.0g</td>
<td>Repair record 1978</td>
</tr>
</tbody>
</table>

Sorted by:
. order make mpg
. describe
Contains data from https://www.stata-press.com/data/r17/auto4.dta
Observations: 74  1978 automobile data
Variables: 6  6 Apr 2020  00:20

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%-17s</td>
<td>Make and model</td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%8.0gc</td>
<td>Price</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td>Weight (lbs.)</td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%8.0g</td>
<td>Length (in.)</td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%8.0g</td>
<td>Repair record 1978</td>
</tr>
</tbody>
</table>

Sorted by:

We now want length to be the last variable in our dataset, so we could type order make mpg price weight rep78 length, but it would be easier to use the last option:

. order length, last
. describe
Contains data from https://www.stata-press.com/data/r17/auto4.dta
Observations: 74  1978 automobile data
Variables: 6  6 Apr 2020  00:20

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str17</td>
<td>%-17s</td>
<td>Make and model</td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%8.0gc</td>
<td>Price</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td>Weight (lbs.)</td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%8.0g</td>
<td>Repair record 1978</td>
</tr>
<tr>
<td>length</td>
<td>int</td>
<td>%8.0g</td>
<td>Length (in.)</td>
</tr>
</tbody>
</table>

Sorted by:
We now change our mind and decide that we prefer that the variables be alphabetized.

```stata
. order _all, alphabetic
. describe
Contains data from https://www.stata-press.com/data/r17/auto4.dta
Observations: 74 1978 automobile data
Variables: 6 6 Apr 2020 00:20

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage</th>
<th>Display</th>
<th>Value</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>int</td>
<td>%8.0g</td>
<td></td>
<td>Length (in.)</td>
</tr>
<tr>
<td>make</td>
<td>str17</td>
<td>%-17s</td>
<td></td>
<td>Make and model</td>
</tr>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>price</td>
<td>int</td>
<td>%8.0gc</td>
<td></td>
<td>Price</td>
</tr>
<tr>
<td>rep78</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td>Repair record 1978</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td></td>
<td>Weight (lbs.)</td>
</tr>
</tbody>
</table>
```

Sorted by:

—if your data contain variables named `year1`, `year2`, ..., `year19`, `year20`, specify the `sequential` option to obtain this ordering. If you specify the `alphabetic` option, `year10` will appear between `year1` and `year11`.

Also see

[D] describe — Describe data in memory or in file
[D] ds — Compactly list variables with specified properties
[D] edit — Browse or edit data with Data Editor
[D] rename — Rename variable
outfile — Export dataset in text format

**Description**

outfile writes data to a disk file in plain-text format, which can be read by other programs. The new file is *not* in Stata format; see [D] save for instructions on saving data for later use in Stata.

The data saved by outfile can be read back by infile; see [D] import. If filename is specified without an extension, .raw is assumed unless the dictionary option is specified, in which case .dct is assumed. If your filename contains embedded spaces, remember to enclose it in double quotes.

**Quick start**

Export current dataset to space-separated mydata.raw
   outfile using mydata

As above, but export only v1, v2, and v3
   outfile v1 v2 v3 using mydata

As above, but export to comma-separated mydata.csv
   outfile v1 v2 v3 using mydata.csv, comma

Export current dataset in Stata’s dictionary format to myfile.dct
   outfile v1 v2 v3 using mydata, dictionary

Do not allow observations to break across lines
   outfile using mydata, wide

**Menu**

File > Export > Text data (fixed- or free-format)
Syntax

```
outfile [varlist] using filename [if] [in] [, options]
```

**options**

<table>
<thead>
<tr>
<th>Main</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary</td>
<td>write the file in Stata’s dictionary format</td>
</tr>
<tr>
<td>nolabel</td>
<td>output numeric values (not labels) of labeled variables; the default is to write labels in double quotes</td>
</tr>
<tr>
<td>noquote</td>
<td>do not enclose strings in double quotes</td>
</tr>
<tr>
<td>comma</td>
<td>write file in comma-separated (instead of space-separated) format</td>
</tr>
<tr>
<td>wide</td>
<td>force one observation per line (no matter how wide)</td>
</tr>
</tbody>
</table>

**Advanced**

| rjs                   | right-justify string variables; the default is to left-justify |
| fjs                   | left-justify if format width < 0; right-justify if format width > 0 |
| runtogether           | all on one line, no quotes, no space between, and ignore formats |
| missing               | retain missing values; use only with comma         |
| replace               | overwrite the existing file                        |

**Options**

- **Main**
  - `dictionary`: writes the file in Stata’s data dictionary format. See [D] `infile` (fixed format) for a description of dictionaries. `comma`, `missing`, and `wide` are not allowed with `dictionary`.
  - `nolabel`: causes Stata to write the numeric values of labeled variables. The default is to write the labels enclosed in double quotes.
  - `noquote`: prevents Stata from placing double quotes around the contents of strings, meaning string variables and value labels.
  - `comma`: causes Stata to write the file in comma-separated–value format. In this format, values are separated by commas rather than by blanks. Missing values are written as two consecutive commas unless `missing` is specified.
  - `wide`: causes Stata to write the data with 1 observation per line. The default is to split observations into lines of 80 characters or fewer, but strings longer than 80 characters are never split across lines.

- **Advanced**
  - `rjs` and `fjs` affect how strings are justified; you probably do not want to specify either of these options. By default, `outfile` outputs strings left-justified in their field.
  - If `rjs` is specified, strings are output right-justified. `rjs` stands for “right-justified strings”.
  - If `fjs` is specified, strings are output left- or right-justified according to the variable’s format: left-justified if the format width is negative and right-justified if the format width is positive. `fjs` stands for “format-justified strings”.

replace does not appear in the dialog box.
runtogether is a programmer’s option that is valid only when all variables of the specified varlist are of type string. runtogether specifies that the variables be output in the order specified, without quotes, with no spaces between, and ignoring the display format attached to each variable. Each observation ends with a new line character.

missing, valid only with comma, specifies that missing values be retained. When comma is specified without missing, missing values are changed to null strings ("").

The following option is available with outfile but is not shown in the dialog box:

replace permits outfile to overwrite an existing dataset.

Remarks and examples

outfile enables data to be sent to a disk file for processing by a non-Stata program. Each observation is written as one or more records that will not exceed 80 characters unless you specify the wide option. Each column other than the first is prefixed by two blanks.

outfile is careful to put the data in columns in case you want to read the data by using formatted input. String variables and value labels are output in left-justified fields by default. You can change this behavior by using the rjs or fjs options.

Numeric variables are output right-justified in the field width specified by their display format. A numeric variable with a display format of %9.0g will be right-justified in a nine-character field. Commas are not written in numeric variables, even if a comma format is used.

If you specify the dictionary option, the data are written in the same way, but preceding the data, outfile writes a data dictionary describing the contents of the file.

Example 1: Basic usage

We have entered into Stata some data on seven employees in our firm. The data contain employee name, employee identification number, salary, and sex:

```
.list

<table>
<thead>
<tr>
<th>name</th>
<th>empno</th>
<th>salary</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carl Marks</td>
<td>57213</td>
<td>24,000</td>
<td>male</td>
</tr>
<tr>
<td>Irene Adler</td>
<td>47229</td>
<td>27,000</td>
<td>female</td>
</tr>
<tr>
<td>Adam Smith</td>
<td>57323</td>
<td>24,000</td>
<td>male</td>
</tr>
<tr>
<td>David Wallis</td>
<td>57401</td>
<td>24,500</td>
<td>male</td>
</tr>
<tr>
<td>Mary Rogers</td>
<td>57802</td>
<td>27,000</td>
<td>female</td>
</tr>
<tr>
<td>Carolyn Frank</td>
<td>57805</td>
<td>24,000</td>
<td>female</td>
</tr>
<tr>
<td>Robert Lawson</td>
<td>57824</td>
<td>22,500</td>
<td>male</td>
</tr>
</tbody>
</table>
```

The last variable in our data, sex, is really a numeric variable, but it has an associated value label.

If we now wish to use a program other than Stata with these data, we must somehow get the data over to that other program. The standard Stata-format dataset created by save will not do the job—it is written in a special format that only Stata understands. Most programs, however, understand plain-text datasets, such as those produced by a text editor. We can tell Stata to produce such a dataset by using outfile. Typing outfile using employee creates a dataset called employee.raw that contains all the data. We can use the Stata type command to review the resulting file:
We see that the file contains the four variables and that Stata has surrounded the string variables with double quotes.

**Technical note**

The `nolabel` option prevents Stata from substituting value-label strings for the underlying numeric values; see [U] 12.6.3 Value labels. The last variable in our data is really a numeric variable:

```
. outfile using employ2, nolabel
```

```
   Carl Marks   57213   24000    0
   Irene Adler  47229   27000    1
   Adam Smith   57323   24000    0
   David Wallis 57401   24500    0
   Mary Rogers  57802   27000    1
   Carolyn Frank 57805   24000    1
   Robert Lawson 57824   22500    0
```

**Technical note**

If you do not want Stata to place double quotes around the contents of string variables, you can specify the `noquote` option:

```
. outfile using employ3, noquote
```

```
   Carl Marks   57213   24000   male
   Irene Adler  47229   27000   female
   Adam Smith   57323   24000   male
   David Wallis 57401   24500   male
   Mary Rogers  57802   27000   female
   Carolyn Frank 57805   24000   female
   Robert Lawson 57824   22500   male
```

**Example 2: Overwriting an existing file**

Stata never writes over an existing file unless explicitly told to do so. For instance, if the file `employee.raw` already exists and we attempt to overwrite it by typing `outfile using employee`, here is what would happen:

```
. outfile using employee
file employee.raw already exists
r(602);
```
We can tell Stata that it is okay to overwrite a file by specifying the `replace` option:

```
    . outfile using employee, replace
```

**Technical note**

Some programs prefer data to be separated by commas rather than by blanks. Stata produces such a dataset if you specify the `comma` option:

```
    . outfile using employee, comma replace
    . type employee.raw
    "Carl Marks",57213,24000,"male"
    "Irene Adler",47229,27000,"female"
    "Adam Smith",57323,24000,"male"
    "David Wallis",57401,24500,"male"
    "Mary Rogers",57802,27000,"female"
    "Carolyn Frank",57805,24000,"female"
    "Robert Lawson",57824,22500,"male"
```

**Example 3: Creating data dictionaries**

Finally, `outfile` can create data dictionaries that `infile` can read. Dictionaries are perhaps the best way to organize your raw data. A dictionary describes your data so that you do not have to remember the order of the variables, the number of variables, the variable names, or anything else. The file in which you store your data becomes self-documenting so that you can understand the data in the future. See [D] `infile (fixed format)` for a full description of data dictionaries.

When you specify the `dictionary` option, Stata writes a `.dct` file:

```
    . outfile using employee, dict replace
    . type employee.dct
    dictionary { 
      str15 name   "Employee name"
      float empno "Employee number"
      float salary "Annual salary"
      float sex   :sexlbl "Sex"
    }
    "Carl Marks" 57213 24000 "male"
    "Irene Adler" 47229 27000 "female"
    "Adam Smith" 57323 24000 "male"
    "David Wallis" 57401 24500 "male"
    "Mary Rogers" 57802 27000 "female"
    "Carolyn Frank" 57805 24000 "female"
    "Robert Lawson" 57824 22500 "male"
```
Example 4: Working with dates

We have historical data on the S&P 500 for the month of January 2001.

```
. use https://www.stata-press.com/data/r17/outfilexmpl, clear
(S&P 500)
. describe
Contains data from https://www.stata-press.com/data/r17/outfilexmpl.dta
Observations: 21  S&P 500
Variables:  6  6 Apr 2020 16:02
(_dta has notes)
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>int</td>
<td>%td</td>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>open</td>
<td>float</td>
<td>%9.0g</td>
<td>Opening price</td>
<td></td>
</tr>
<tr>
<td>high</td>
<td>float</td>
<td>%9.0g</td>
<td>High price</td>
<td></td>
</tr>
<tr>
<td>low</td>
<td>float</td>
<td>%9.0g</td>
<td>Low price</td>
<td></td>
</tr>
<tr>
<td>close</td>
<td>float</td>
<td>%9.0g</td>
<td>Closing price</td>
<td></td>
</tr>
<tr>
<td>volume</td>
<td>int</td>
<td>%12.0gc</td>
<td>Volume (thousands)</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: date

The `date` variable has a display format of `%td` so that it is displayed as `ddmmmyyyy`.

```
. list
```

<table>
<thead>
<tr>
<th>date</th>
<th>open</th>
<th>high</th>
<th>low</th>
<th>close</th>
<th>volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>02jan2001</td>
<td>1320.28</td>
<td>1320.28</td>
<td>1276.05</td>
<td>1283.27</td>
<td>11,294</td>
</tr>
<tr>
<td>03jan2001</td>
<td>1283.27</td>
<td>1347.76</td>
<td>1274.62</td>
<td>1347.56</td>
<td>18,807</td>
</tr>
<tr>
<td>04jan2001</td>
<td>1347.56</td>
<td>1350.24</td>
<td>1329.14</td>
<td>1333.34</td>
<td>21,310</td>
</tr>
<tr>
<td>05jan2001</td>
<td>1333.34</td>
<td>1334.77</td>
<td>1294.95</td>
<td>1298.35</td>
<td>14,308</td>
</tr>
<tr>
<td>08jan2001</td>
<td>1298.35</td>
<td>1298.35</td>
<td>1276.29</td>
<td>1295.86</td>
<td>11,155</td>
</tr>
<tr>
<td>09jan2001</td>
<td>1295.86</td>
<td>1311.72</td>
<td>1295.14</td>
<td>1300.8</td>
<td>11,913</td>
</tr>
<tr>
<td>10jan2001</td>
<td>1300.8</td>
<td>1313.76</td>
<td>1287.28</td>
<td>1313.27</td>
<td>12,965</td>
</tr>
<tr>
<td>11jan2001</td>
<td>1313.27</td>
<td>1332.19</td>
<td>1309.72</td>
<td>1326.82</td>
<td>14,112</td>
</tr>
<tr>
<td>12jan2001</td>
<td>1326.82</td>
<td>1333.21</td>
<td>1311.59</td>
<td>1318.55</td>
<td>12,760</td>
</tr>
<tr>
<td>16jan2001</td>
<td>1318.32</td>
<td>1327.81</td>
<td>1313.33</td>
<td>1326.65</td>
<td>12,057</td>
</tr>
<tr>
<td>17jan2001</td>
<td>1326.65</td>
<td>1346.92</td>
<td>1325.41</td>
<td>1329.47</td>
<td>13,491</td>
</tr>
<tr>
<td>18jan2001</td>
<td>1329.89</td>
<td>1352.71</td>
<td>1327.41</td>
<td>1347.97</td>
<td>14,450</td>
</tr>
<tr>
<td>19jan2001</td>
<td>1347.97</td>
<td>1354.55</td>
<td>1336.74</td>
<td>1342.54</td>
<td>14,078</td>
</tr>
<tr>
<td>22jan2001</td>
<td>1342.54</td>
<td>1353.62</td>
<td>1333.84</td>
<td>1342.9</td>
<td>11,640</td>
</tr>
<tr>
<td>23jan2001</td>
<td>1342.9</td>
<td>1362.9</td>
<td>1339.63</td>
<td>1360.4</td>
<td>12,326</td>
</tr>
<tr>
<td>24jan2001</td>
<td>1360.4</td>
<td>1369.75</td>
<td>1357.28</td>
<td>1364.3</td>
<td>13,090</td>
</tr>
<tr>
<td>25jan2001</td>
<td>1364.3</td>
<td>1367.35</td>
<td>1354.63</td>
<td>1357.5</td>
<td>12,580</td>
</tr>
<tr>
<td>26jan2001</td>
<td>1357.51</td>
<td>1357.51</td>
<td>1342.75</td>
<td>1354.95</td>
<td>10,980</td>
</tr>
<tr>
<td>29jan2001</td>
<td>1354.92</td>
<td>1365.54</td>
<td>1350.36</td>
<td>1364.17</td>
<td>10,531</td>
</tr>
<tr>
<td>30jan2001</td>
<td>1364.17</td>
<td>1375.68</td>
<td>1356.2</td>
<td>1373.73</td>
<td>11,498</td>
</tr>
<tr>
<td>31jan2001</td>
<td>1373.73</td>
<td>1383.37</td>
<td>1364.66</td>
<td>1366.01</td>
<td>12,953</td>
</tr>
</tbody>
</table>
We outfile our data and use the `type` command to view the result.

```plaintext
.outfile using sp
.type sp.raw
"02jan2001" 1320.28 1320.28 1276.05 1283.27 11294
"03jan2001" 1283.27 1347.76 1274.62 1347.56 18807
"04jan2001" 1347.56 1350.24 1329.14 1333.34 21310
"05jan2001" 1333.34 1334.77 1294.95 1298.35 14308
"08jan2001" 1298.35 1296.35 1276.29 1295.86 11155
"09jan2001" 1295.86 1311.72 1295.14 1300.8 11913
"10jan2001" 1300.8 1313.76 1287.28 1313.27 12965
"11jan2001" 1313.27 1332.19 1309.72 1326.82 14112
"12jan2001" 1326.82 1333.21 1311.59 1318.55 12760
"16jan2001" 1318.32 1327.81 1313.33 1326.65 12057
"17jan2001" 1326.65 1346.92 1325.41 1329.47 13491
"18jan2001" 1329.89 1352.71 1327.41 1347.97 14450
"19jan2001" 1347.97 1354.55 1336.74 1342.54 14078
"22jan2001" 1342.54 1353.62 1333.84 1342.9 11640
"23jan2001" 1342.9 1362.9 1339.63 1360.4 12326
"24jan2001" 1360.4 1369.75 1357.28 1364.3 13090
"25jan2001" 1364.3 1367.35 1354.63 1357.51 12580
"26jan2001" 1357.51 1357.51 1342.75 1354.95 10980
"29jan2001" 1354.92 1365.54 1350.36 1364.17 10531
"30jan2001" 1364.17 1375.68 1356.2 1373.73 11498
"31jan2001" 1373.73 1383.37 1364.66 1366.01 12953
```

The `date` variable, originally stored as an `int`, was outfiled as a string variable. Whenever Stata outfiles a variable with a date format, Stata outfiles the variable as a string.

---

Also see

[D] export — Overview of exporting data from Stata

[D] import — Overview of importing data into Stata

[U] 22 Entering and importing data
**Description**

*pctile* creates a new variable containing the percentiles of `exp`, where the expression `exp` is typically just another variable.

*xtile* creates a new variable that categorizes `exp` by its quantiles. If the `cutpoints(varname)` option is specified, it categorizes `exp` using the values of `varname` as category cutpoints. For example, `varname` might contain percentiles of another variable, generated by `pctile`.

`pctile` is a programmer’s command that computes up to 4,096 percentiles and places the results in `r()`; see [U] 18.8 Accessing results calculated by other programs. `summarize, detail` computes some percentiles (1, 5, 10, 25, 50, 75, 90, 95, and 99th); see [R] `summarize`.

**Quick start**

Create `qrt1` containing the quartiles of `v`

```
pctile qrt1 = v, nq(4)
```

As above, and create `percent` containing the percentages

```
pctile qrt1 = v, nq(4) genp(percent)
```

As above, but apply sampling weights `wvar1`

```
pctile qrt1 = v [pweight=wvar1], nq(4) genp(percent)
```

Create `dec1` containing the deciles of `v`

```
pctile dec1 = v, nq(10)
```

As above, but create `dec2` indicating to which decile each observation belongs

```
xtile dec2 = v, nq(10)
```

As above, but apply frequency weights `wvar2`

```
xtile dec2 = v [fweight=wvar2], nq(10)
```

Compute the 10th and 90th percentiles, and store them in `r(r1)` and `r(r2)`

```
_pctile v, percentiles(10 90)
```

**Menu**

```
pctile
```

Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Create variable of percentiles

```
xtile
```

Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Create variable of quantiles
Syntax

Create variable containing percentiles

```
pctile [type] newvar = exp [if] [in] [weight] [ , pctile_options ]
```

Create variable containing quantile categories

```
xtile newvar = exp [if] [in] [weight] [ , xtile_options ]
```

Compute percentiles and store them in r()

```
_pctile varname [if] [in] [weight] [ , _pctile_options ]
```

**pctile_options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
</tr>
<tr>
<td>nquantiles(#)</td>
</tr>
<tr>
<td>genp(newvar_p)</td>
</tr>
<tr>
<td>altdef</td>
</tr>
</tbody>
</table>

**xtile_options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
</tr>
<tr>
<td>nquantiles(#)</td>
</tr>
<tr>
<td>cutpoints(varname)</td>
</tr>
<tr>
<td>altdef</td>
</tr>
</tbody>
</table>

**_pctile_options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
</tr>
<tr>
<td>nquantiles(#)</td>
</tr>
<tr>
<td>percentiles(numlist)</td>
</tr>
<tr>
<td>altdef</td>
</tr>
</tbody>
</table>

Collect is allowed with pctile; see [U] 11.1.10 Prefix commands.

aweights, fweights, and pweights are allowed (see [U] 11.1.6 weight), except when the altdef option is specified, in which case no weights are allowed.

**Options**

<table>
<thead>
<tr>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>nquantiles(#)</td>
</tr>
<tr>
<td>genp(newvar_p) (pctile only)</td>
</tr>
</tbody>
</table>
altdef uses an alternative formula for calculating percentiles. The default method is to invert the empirical distribution function by using averages, \((x_i + x_{i+1})/2\), where the function is flat (the default is the same method used by summarize; see [R] summarize). The alternative formula uses an interpolation method. See Methods and formulas at the end of this entry. Weights cannot be used when altdef is specified.

cutpoints(varname) (xtile only) requests that xtile use the values of varname, rather than quantiles, as cutpoints for the categories. All values of varname are used, regardless of any if or in restriction; see the technical note in the xtile section below.

percentiles(numlist) (pctile only) requests percentiles corresponding to the specified percentages. Percentiles are placed in \(r(r_1), r(r_2), \ldots\), etc. For example, percentiles(10(20)90) requests that the 10th, 30th, 50th, 70th, and 90th percentiles be computed and placed into \(r(r_1), r(r_2), r(r_3), r(r_4),\) and \(r(r_5)\). Up to 4,096 (inclusive) percentiles can be requested. See [P] numlist for the syntax of a numlist.

Remarks and examples

Remarks are presented under the following headings:

- pctile
- xtile
- _pctile

pctile

pctile creates a new variable containing percentiles. You specify the number of quantiles that you want, and pctile computes the corresponding percentiles. Here we use Stata’s auto dataset and compute the deciles of mpg:

```
. use https://www.stata-press.com/data/r17/auto
   (1978 automobile data)
. pctile pct = mpg, nq(10)
. list pct in 1/10
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>.</td>
</tr>
</tbody>
</table>
If we use the `genp()` option to generate another variable with the corresponding percentages, it is easier to distinguish between the percentiles.

```
. drop pct
. pctile pct = mpg, nq(10) genp(percent)
. list percent pct in 1/10

 percent pct
1. 10 14
2. 20 17
3. 30 18
4. 40 19
5. 50 20
6. 60 22
7. 70 24
8. 80 25
9. 90 29
10. . .
```

`summarize, detail` calculates standard percentiles.

```
. summarize mpg, detail
  
  Mileage (mpg)
    
    +--------------------------------------------------+
    | Percentiles | Smallest                     | Largest          |
    |            |  | Obs | Sum of wgt. | Mean    | Std. dev.  |
    | Percentile |  |     |   |            |         |            |
    | 1%         | 12 | 12  | 74 | 21.2973    | 5.785503|
    | 5%         | 14 | 14  | 74 |             |          |
    | 10%        | 14 | 14  | 74 |             |          |
    | 25%        | 18 | 14  | 74 |             |          |
    | 50%        | 20 | 20  | 74 |             |          |
    | 75%        | 25 | 25  | 34 |             |          |
    | 90%        | 29 | 29  | 35 | Variance    | 33.47205|
    | 95%        | 34 | 34  | 35 | Skewness    | 0.9487176|
    | 99%        | 41 | 41  | 41 | Kurtosis    | 3.975005|
    +--------------------------------------------------+
```

`summarize, detail` can calculate only these particular percentiles. The `pctile` and `_pctile` commands allow you to compute any percentile.
Weights can be used with `pctile`, `xtile`, and `_pctile`:

```
drop pct percent
.pctile pct = mpg [w=weight], nq(10) genp(percent)
(list analytic weights assumed)
.list percent pct in 1/10

<table>
<thead>
<tr>
<th>percent</th>
<th>pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>30</td>
<td>17</td>
</tr>
<tr>
<td>40</td>
<td>18</td>
</tr>
<tr>
<td>50</td>
<td>19</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
</tr>
<tr>
<td>70</td>
<td>22</td>
</tr>
<tr>
<td>80</td>
<td>24</td>
</tr>
<tr>
<td>90</td>
<td>28</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>
```

The result is the same, no matter which weight type you specify—`aweight`, `fweight`, or `pweight`.

**xtile**

`xtile` creates a categorical variable that contains categories corresponding to quantiles. We illustrate this with a simple example. Suppose that we have a variable, `bp`, containing blood pressure measurements:

```
.use https://www.stata-press.com/data/r17/bp1, clear
.list
```

```
<table>
<thead>
<tr>
<th>bp</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>104</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>125</td>
</tr>
<tr>
<td>130</td>
</tr>
<tr>
<td>132</td>
</tr>
</tbody>
</table>
```
xtile can be used to create a variable, quart, that indicates the quartiles of bp.

```
.xtile quart = bp, nq(4)
.list bp quart, sepby(quart)
```

<table>
<thead>
<tr>
<th>bp</th>
<th>quart</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
</tr>
<tr>
<td>125</td>
<td>3</td>
</tr>
<tr>
<td>130</td>
<td>4</td>
</tr>
<tr>
<td>132</td>
<td>4</td>
</tr>
</tbody>
</table>

The categories created are

\[ (-\infty, x_{25}] , \quad (x_{25}, x_{50}] , \quad (x_{50}, x_{75}] , \quad (x_{75}, +\infty) \]

where \( x_{25} \), \( x_{50} \), and \( x_{75} \) are, respectively, the 25th, 50th (median), and 75th percentiles of bp. We could use the pctile command to generate these percentiles:

```
.pctile pct = bp, nq(4) genp(percent)
.list bp quart percent pct, sepby(quart)
```

<table>
<thead>
<tr>
<th>bp</th>
<th>quart</th>
<th>percent</th>
<th>pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>1</td>
<td>25</td>
<td>104</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>50</td>
<td>120</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>75</td>
<td>125</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>125</td>
<td>3</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>130</td>
<td>4</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>132</td>
<td>4</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

xtile can categorize a variable on the basis of any set of cutpoints, not just percentiles. Suppose that we wish to create the following categories for blood pressure:

\[ (-\infty, 100] , \quad (100, 110] , \quad (110, 120] , \quad (120, 130] , \quad (130, +\infty) \]
To do this, we simply create a variable containing the cutpoints,

```
    . input class
class
    1. 100
    2. 110
    3. 120
    4. 130
    5. end
```

and then use `xtile` with the `cutpoints()` option:

```
    . xtile category = bp, cutpoints(class)
    . list bp class category, sepby(category)
```

<table>
<thead>
<tr>
<th>bp</th>
<th>class</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>104</td>
<td>120</td>
<td>2</td>
</tr>
<tr>
<td>110</td>
<td>130</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>.</td>
<td>3</td>
</tr>
<tr>
<td>120</td>
<td>.</td>
<td>3</td>
</tr>
<tr>
<td>120</td>
<td>.</td>
<td>3</td>
</tr>
<tr>
<td>120</td>
<td>.</td>
<td>3</td>
</tr>
<tr>
<td>125</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>130</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>132</td>
<td>.</td>
<td>5</td>
</tr>
</tbody>
</table>

The cutpoints can, of course, come from anywhere. They can be the quantiles of another variable or the quantiles of a subgroup of the variable. Suppose that we had a variable, `case`, that indicated whether an observation represented a case (`case = 1`) or control (`case = 0`).

```
    . use https://www.stata-press.com/data/r17/bp2, clear
    . list in 1/11, sep(4)
```

<table>
<thead>
<tr>
<th>bp</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
</tr>
<tr>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>130</td>
<td>1</td>
</tr>
<tr>
<td>132</td>
<td>1</td>
</tr>
</tbody>
</table>
We can categorize the cases on the basis of the quantiles of the controls. To do this, we first generate a variable, pct, containing the percentiles of the controls’ blood pressure data:

```
. ptcile pct = bp if case==0, nq(4)
. list pct in 1/4
```

<table>
<thead>
<tr>
<th>pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 104</td>
</tr>
<tr>
<td>2. 117</td>
</tr>
<tr>
<td>3. 124</td>
</tr>
<tr>
<td>4. .</td>
</tr>
</tbody>
</table>

Then we use these percentiles as cutpoints to classify bp: for all subjects.

```
. xtile category = bp, cutpoints(pct)
. gsort -case bp
. list bp case category in 1/11, sepby(category)
```

<table>
<thead>
<tr>
<th>bp</th>
<th>case</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>120</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>125</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>130</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>132</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

**Technical note**

In the last example, if we wanted to categorize only cases, we could have issued the command

```
. xtile category = bp if case==1, cutpoints(pct)
```

Most Stata commands follow the logic that using an `if exp` is equivalent to dropping observations that do not satisfy the expression and running the command. This is not true of `xtile` when the `cutpoints()` option is used. (When the `cutpoints()` option is not used, the standard logic is true.) `xtile` uses all nonmissing values of the `cutpoints()` variable whether or not these values belong to observations that satisfy the `if` expression.

If you do not want to use all the values in the `cutpoints()` variable as cutpoints, simply set the ones that you do not need to missing. `xtile` does not care about the order of the values or whether they are separated by missing values.
Technical note

Quantiles are not always unique. If we categorize our blood pressure data by quintiles rather than quartiles, we get

```
.use https://www.stata-press.com/data/r17/bp1, clear
.xtile quint = bp, nq(5)
.pctile pct = bp, nq(5) genp(percent)
.list bp quint pct percent, sepby(quint)
```

<table>
<thead>
<tr>
<th></th>
<th>bp</th>
<th>quint</th>
<th>pct</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98</td>
<td>1</td>
<td>104</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>1</td>
<td>120</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>104</td>
<td>1</td>
<td>120</td>
<td>60</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
<td>2</td>
<td>125</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>120</td>
<td>2</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>9</td>
<td>125</td>
<td>4</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>130</td>
<td>5</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>11</td>
<td>132</td>
<td>5</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

The 40th and 60th percentile are the same; they are both 120. When two (or more) percentiles are the same, they are given the lower category number.

_pctile

_pctile is a programmer’s command. It computes percentiles and stores them in r(); see [U] 18.8 Accessing results calculated by other programs.
You can use _pctile to compute quantiles, just as you can with pctile:

```
use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
_pctile weight, nq(10)
return list
 scalars:
  r(r1) = 2020
  r(r2) = 2160
  r(r3) = 2520
  r(r4) = 2730
  r(r5) = 3190
  r(r6) = 3310
  r(r7) = 3420
  r(r8) = 3700
  r(r9) = 4060
```

The percentiles() option (abbreviation p()) can be used to compute any percentile you wish:

```
_pctile weight, p(10, 33.333, 45, 50, 55, 66.667, 90)
return list
 scalars:
  r(r1) = 2020
  r(r2) = 2640
  r(r3) = 2830
  r(r4) = 3190
  r(r5) = 3250
  r(r6) = 3400
  r(r7) = 4060
```

_pctile, pctile, and xtile each have an option that uses an alternative definition of percentiles, based on an interpolation scheme; see Methods and formulas below.

```
_pctile weight, p(10, 33.333, 45, 50, 55, 66.667, 90) altdef
return list
 scalars:
  r(r1) = 2005
  r(r2) = 2639.985
  r(r3) = 2830
  r(r4) = 3190
  r(r5) = 3252.5
  r(r6) = 3400.005
  r(r7) = 4060
```

The default formula inverts the empirical distribution function. The default formula is more commonly used, although some consider the “alternative” formula to be the standard definition. One drawback of the alternative formula is that it does not have an obvious generalization to noninteger weights.

Technical note

summarize, detail computes the 1st, 5th, 10th, 25th, 50th (median), 75th, 90th, 95th, and 99th percentiles. There is no real advantage in using _pctile to compute these percentiles. Both summarize, detail and _pctile use the same internal code. _pctile is slightly faster because summarize, detail computes a few extra things. The value of _pctile is its ability to compute percentiles other than these standard ones.
Stored results

_pctile_ and __pctile__ store the following in r():

Scalars

\( r(#) \) value of \#-requested percentile

Methods and formulas

The default formula for percentiles is as follows: Let \( x(j) \) refer to the \( x \) in ascending order for \( j = 1, 2, \ldots, n \). Let \( w(j) \) refer to the corresponding weights of \( x(j) \); if there are no weights, \( w(j) = 1 \). Let \( N = \sum_{j=1}^{n} w(j) \).

To obtain the \( p \)th percentile, which we will denote as \( x[p] \), let \( P = Np/100 \), and let

\[
W(i) = \sum_{j=1}^{i} w(j)
\]

Find the first index, \( i \), such that \( W(i) > P \). The \( p \)th percentile is then

\[
x[p] = \begin{cases} 
\frac{x(i-1) + x(i)}{2} & \text{if } W(i-1) = P \\
x(i) & \text{otherwise}
\end{cases}
\]

When the altdef option is specified, the following alternative definition is used. Here weights are not allowed.

Let \( i \) be the integer floor of \((n+1)p/100\); that is, \( i \) is the largest integer \( i \leq (n+1)p/100 \). Let \( h \) be the remainder \( h = (n+1)p/100 - i \). The \( p \)th percentile is then

\[
x[p] = (1 - h)x(i) + hx(i+1)
\]

where \( x(0) \) is taken to be \( x(1) \) and \( x(n+1) \) is taken to be \( x(n) \).

_xtile_ produces the categories

\((-\infty, x[p_1]), (x[p_1], x[p_2]), \ldots, (x[p_{m-2}], x[p_{m-1}]), (x[p_{m-1}], +\infty)\)

numbered, respectively, 1, 2, \ldots, \( m \), based on the \( m \) quantiles given by the \( p_k \)th percentiles, where \( p_k = 100k/m \) for \( k = 1, 2, \ldots, m-1 \).

If \( x[p_{k-1}] = x[p_k] \), the \( k \)th category is empty. All elements \( x = x[p_{k-1}] = x[p_k] \) are put in the \((k-1)\)th category: \((x[p_{k-2}], x[p_{k-1}])\).

If _xtile_ is used with the _cutpoints(varname)_ option, the categories are

\((-\infty, y(1)], (y(1), y(2)], \ldots, (y(m-1), y(m)], (y(m), +\infty)\)

and they are numbered, respectively, 1, 2, \ldots, \( m + 1 \), based on the \( m \) nonmissing values of _varname_: \( y(1), y(2), \ldots, y(m) \).
Acknowledgment

xtile is based on a command originally posted on Statalist (see [U] 3.2.4 The Stata Forum) by Philip Ryan of the Discipline of Public Health at the University of Adelaide, Australia.

Also see

[R] centile — Report centile and confidence interval
[R] summarize — Summary statistics
[U] 18.8 Accessing results calculated by other programs
Description

putmata exports the contents of Stata variables to Mata vectors and matrices.
getmata imports the contents of Mata vectors and matrices to Stata variables.
putmata and getmata are useful for creating solutions to problems more easily solved in Mata. The commands are also useful in teaching.

Quick start

Create a Mata vector for each Stata variable in memory
putmata *

As above, but create a vector only for nonmissing values of idvar, v1, and v2
putmata idvar v1 v2, omitmissing

Place variables v1 and v2 into column vectors x1 and x2
putmata idvar x1=v1 x2=v2

Create Mata matrix X from v1 and v2
putmata X=(v1 v2)

Create Stata variables newv1 and newv2 from Mata matrix X
getmata (newv1 newv2)=X

Replace v1 and v2 with columns from Mata matrix X
getmata (v1 v2)=X, replace

As above, and match observations using idvar Mata vector
getmata (v1 v2)=X, replace id(idvar)
**Syntax**

```plaintext
putmata putlist [ if ] [ in ] [ , putmata_options ]

getmata getlist [ , getmata_options ]
```

**putmata_options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>omitmissing</td>
</tr>
<tr>
<td>view</td>
</tr>
<tr>
<td>replace</td>
</tr>
</tbody>
</table>

A `putlist` can be as simple as a list of Stata variable names. See below for details.

**getmata_options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
</tr>
<tr>
<td>update</td>
</tr>
<tr>
<td>replace</td>
</tr>
<tr>
<td>id(name)</td>
</tr>
<tr>
<td>force</td>
</tr>
</tbody>
</table>

A `getlist` can be as simple as a list of Mata vector names. See below for details.

`collect` is allowed with `putmata` and `getmata`; see [U] 11.1.10 Prefix commands.

Definition of `putlist` for use with `putmata`:

A `putlist` is one or more of any of the following:

* `varname`
* `varlist`
* `vecname=varname`
* `matname=(varlist)`
* `matname=([ varlist ] # [ varlist ] [...])`

Example: `putmata *`

Creates a vector in Mata for each of the Stata variables in memory. Vectors contain the same data as Stata variables. Vectors have the same names as the corresponding variables.

Example: `putmata mpg weight displ`

Creates a vector in Mata for each variable specified. Vectors have the same names as the corresponding variables. In this example, `displ` is an abbreviation for the variable `displacement`; thus the vector will also be named `displacement`.

Example: `putmata mileage=mpg pounds=weight`

Creates a vector for each variable specified. Vector names differ from the corresponding variable names. In this example, vectors will be named `mileage` and `pounds`.

Example: `putmata y=mpg X=(weight displ)`

Creates \( N \times 1 \) Mata vector \( y \) equal to Stata variable `mpg`, and creates \( N \times 2 \) Mata matrix \( X \) containing the values of Stata variables `weight` and `displacement`. 
Example: `putmata y=mpg X=(weight displ 1)`

Creates $N \times 1$ Mata vector $y$ containing mpg, and creates $N \times 3$ Mata matrix $X$ containing weight, displacement, and a column of 1s. After typing this example, you could enter Mata and type `invsym(X'X)*X'y` to obtain the regression coefficients.

Syntactical elements may be combined. It is valid to type

```
. putmata mpg foreign X=(weight displ) Z=(foreign 1)
```

No matter how you specify the `putlist`, you will need to specify the `replace` option if some or all vectors already exist in Mata:

```
. putmata mpg foreign X=(weight displ) Z=(foreign 1), replace
```

Definition of `getlist` for use with `getmata`:

- A `getlist` is one or more of any of the following:
  - `vecname`
  - `varname=vecname`
  - `(varname varname ... varname)=matname`
  - `(varname*)=matname`

Example: `getmata x1 x2`

Creates a Stata variable for each Mata vector specified. Variables will have the same names as the corresponding vectors. Names may not be abbreviated.

Example: `getmata myvar1=x1 myvar2=x2`

Creates a Stata variable for each Mata vector specified. Variable names will differ from the corresponding vector names.

Example: `getmata (firstvar secondvar)=X`

Creates one Stata variable corresponding to each column of the Mata matrix specified. In this case, the matrix has two columns, and corresponding variables will be named `firstvar` and `secondvar`. If the matrix had three columns, then three variable names would need to be specified.

Example: `getmata (myvar*)=X`

Creates one Stata variable corresponding to each column of the Mata matrix specified. Variables will be named `myvar1`, `myvar2`, etc. The matrix may have any number of columns, even zero!

Syntactical elements may be combined. It is valid to type

```
. getmata r1 r2 final=r3 (rplus*)=X
```

No matter how you specify the `getlist`, you will need to specify the `replace` or `update` option if some or all variables already exist in Stata:

```
. getmata r1 r2 final=r3 (rplus*)=X, replace
```

Options for `putmata`

- `omitmissing` specifies that observations containing a missing value in any of the numeric variables specified be omitted from the vectors and matrices created in Mata. In

```
. putmata y=mpg X=(weight displ 1), omitmissing
```

rows would be omitted from $y$ and $X$ in which the corresponding observation contained missing in any of `mpg`, `weight`, or `displ`. In this case, specifying `omitmissing` would be equivalent to typing
. putmata y=mpg X=(weight displ 1) if !missing(mpg) & !missing(weight) & !missing(displ)

All vectors and matrices created by a single putmata command will have the same number of rows (observations). That is true whether you specify if, in, or the omitmissing option.

view specifies that putmata create views rather than copies of the Stata data in the Mata vectors and matrices. Views require less memory than copies and offer the advantage (and disadvantage) that changes in the Stata data are immediately reflected in the Mata vectors and matrices, and vice versa.

If you specify numeric constants using the matname=(...) syntax, matname is created as a copy even if the view option is specified. Other vectors and matrices created by the command, however, would be views.

Use of the view option with putmata often obviates the need to use getmata to import results back into Stata.

Warning 1: Mata records views as “this vector is a view onto variable 3, observations 2 through 5 and 7”. If you change the order of the variables, the order of the observations, or drop variables once the views are created, then the contents of the views will change.

Warning 2: When assigning values in Mata to view vectors, code

   v[] = ...

   not v = ....

To have changes reflected in the underlying Stata data, you must update the elements of the view v, not redefine it. To update all the elements of v, you literally code v[.]. In the matrix case, you code X[.,].

replace specifies that existing Mata vectors or matrices be replaced should that be necessary.

**Options for getmata**

double specifies that Stata numeric variables be created as doubles. The default is that they be created as floats. Actually, variables start out as floats or doubles, but then they are compressed (see [D] compress).

update and replace are alternatives. They have the same meaning unless the id() or force option is specified.

When id() or force is not specified, both replace and update specify that it is okay to replace the values in existing Stata variables. By default, vectors can be posted to new Stata variables only.

When id() or force is specified, replace and update allow posting of values of existing variables, just as usual. The options differ in how the posting is performed when the id() or force option causes only a subset of the observations of the variables to be updated. update specifies that the remaining values be left as they are. replace specifies that the remaining values be set to missing, just as if the existing variable(s) were being created for the first time.

id(name) and id(varname=vecname) specify how the rows in the Mata vectors and matrices match the observations in the Stata data. Observation i matches row j if variable name[i] equals vector name[j], or in the second syntax, if varname[i] = vecname[j]. The ID variable (vector) must contain values that uniquely identify the observations (rows). Only in observations that contain matching values will the variable be modified. Values in observations that have no match will not
be modified or will be set to missing, as appropriate; values in the ID vector that have no match will be ignored.

Example: You wish to run a regression of $y$ on $x_1$ and $x_2$ on the males in the data and use that result to obtain the fitted values for the males. Stata already has commands that will do this, namely, `regress y x1 x2 if male` followed by `predict yhat if male`. For instructional purposes, let’s say you wish to do this in Mata. You type

```
. putmata myid y X=(x1 x2 1) if male
. mata
: b = invsym(X'*X)*X'y
: yhat = X*b
: end
. getmata yhat, id(myid)
```

The new Stata variable $yhat$ will contain the predicted values for males and missing values for the females. If the $yhat$ variable already existed, you would type

```
. getmata yhat, id(myid) replace
```

or

```
. getmata yhat, id(myid) update
```

The `replace` option would set the female observations to missing. The `update` option would leave the female observations unchanged.

If you do not have an identification variable, create one first by typing `generate myid = _n`

`force` specifies that it is okay to post vectors and matrices with fewer or with more rows than the number of observations in the data. The `force` option is an alternative to `id()`, and usually, `id()` is the appropriate choice.

If you specify `force` and if there are fewer rows in the vectors and matrices than observations in the data, new variables will be padded with missing values. If there are more rows than observations, observations will be added to the data and previously existing variables will be padded with missing values.

### Remarks and examples

Remarks are presented under the following headings:

- Use of `putmata`
- Use of `putmata` and `getmata`
- Using `putmata` and `getmata` on subsets of observations
- Using views
- Constructing do-files

### Use of `putmata`

In this example, we will use Mata to make a calculation and report the result, but we will not post results back to Stata. We will use `putmata` but not `getmata`.

Consider solving for $b$ the set of linear equations

$$y = Xb$$

where $y: N \times 1$, $X: N \times k$, and $b: k \times 1$. If $N = k$, then $y = Xb$ amounts to solving $k$ equations for $k$ unknowns, and the solution is

$$b = X^{-1}y$$

That solution is obtained by premultiplying both sides of (1) by $X^{-1}$. 


When $N > k$, (2) can be used to obtain least-square results if matrix inversion is appropriately defined. Assume that you wish to demonstrate this when matrix inversion is defined as the Moore–Penrose generalized inverse for nonsquare matrices. The demonstration can be obtained by typing

```stata
.sysuse auto, clear
.regress mpg weight displacement
.putmata y=mpg X=(weight displacement 1)
.mata
: pinv(X)*y
: end
```

The Mata expression `pinv(X)*y` will display a $3 \times 1$ column vector. The elements of the vector will equal the coefficients reported by `regress mpg weight displacement`.

For your information, the Moore–Penrose inverse of rectangular matrix $X$: $N \times k$ is a $k \times N$ rectangular matrix. Among other properties, $\text{pinv}(X)^T X = I$, where $I$ is the $k \times k$ identity matrix. You can demonstrate that using Mata, too:

```mata
: mata: pinv(X)*X
```

Use of `putmata` and `getmata`

In this example, we will use Mata to calculate a result that we wish to post back to Stata. We will use both `putmata` and `getmata`.

Some problems are more easily solved in Mata than in Stata. For instance, say that you need to create new Stata variable $D$ from existing variable $C$, defined as

$$D[i] = \sum(C[j] - C[i]) \text{ for all } C[j] > C[i]$$

where $i$ and $j$ index observations.

This problem can be solved in Stata, but the solution is elusive to most people. The solution is more natural in Mata because the Mata solution corresponds almost letter for letter with the mathematical statement of the problem. If $C$ and $D$ were Mata vectors rather than Stata variables, the solution would be

```mata
D = J(rows(C), 1, 0)
for (i=1; i<rows(C); i++) {
    for (j=1; j<rows(C); j++) {
        if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
    }
}
```

The most difficult part of this solution to understand is the first line, $D = J(rows(C), 1, 0)$, and that is because you may not be familiar with Mata’s $J()$ function. $D = J(rows(C), 1, 0)$ creates a $rows(C) \times 1$ column vector of 0s. The arguments of $J()$ are in just that order.

$C$ and $D$ are not vectors in Mata, or at least they are not yet. Using `getmata`, we can create vector $C$ from variable $C$ and run our Mata solution. Then using `putmata`, we can post Mata vector $D$ back to new Stata variable $D$. The solution includes these three steps, also shown in the do-file below:

1. In Stata, use `putmata` to create vector $C$ in Mata equal to variable $C$ in Stata: `putmata C`.
2. Use Mata to solve the problem, creating new Mata vector $D$.
3. In Stata again, use `getmata` to create new variable $D$ equal to Mata vector $D$. 
Because of the typing involved in the solution, we would package the code in a do-file:

```
begin myfile.do
use mydata, clear
putmata C
mata:
D = J(rows(C), 1, 0)
for (i=1; i<=rows(C); i++) {
    for (j=1; j<=rows(C); j++) {
        if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
    }
}
getmata D
save mydata, replace
end
```

With `myfile.do` now in place, in Stata we would type
```
.do myfile
```

Notes:

1. Our program might be better if we changed `putmata C` to read `putmata C, replace` and if we changed `getmata D` to read `getmata D, replace`. As things are right now, typing `do myfile` works, but if we were then to run it a second time, it would not work. Stata would encounter the `putmata` command and issue an error that matrix `C` already exists. Even if Stata got through that, it would encounter the `getmata` command and issue an error that variable `D` already exists. Perhaps that is an advantage. You cannot run `myfile.do` again without dropping matrix `C` and variable `D`. If you consider that a disadvantage, however, include the `replace` option.

2. In our solution, we entered Mata by typing `mata:`, which is to say, `mata` with a colon. Interactively, we usually enter Mata by just typing `mata`. The colon affects how Mata treats errors. When working interactively, we want Mata to note errors but then to continue running so we can correct ourselves. In do-files, we want Mata to note the error and stop. That is the difference between `mata` without the colon and `mata` with the colon. Remember to use `mata:` when writing do-files.

3. Rather than specify the `replace` option, you could modify the do-file to drop any preexisting Mata vector `C` and any preexisting variable `D`. To drop vector `C`, in Mata you can type `mata drop C`, or in Stata, you can type `mata drop C`. To drop variable `D`, in Stata you can type `drop D`. You must worry that the variables do not exist, so in your do-file, you would code
```
capture mata: mata drop C
capture drop D
```

Rather than dropping vector `C`, you might prefer just to clear Mata:
```
clear mata
```
Using putmata and getmata on subsets of observations

Putmata can be used to create Mata vectors that contain a subset of the observations in the Stata data, and getmata can be used to fetch such vectors back into Stata. Thus you can work with only the males or only outcomes in which failures are observed, and so on. Below we work with only the observations in which C does not contain missing values.

In the create-variable-D-from-C example above, we assumed that there were no missing values in C, or at least we did not consider the issue. It turns out that our code produces several missing values in the presence of just one missing value in C. Perhaps, if there are missing values, we want to exclude them from our calculation. We could complicate our Mata code to handle that. We could modify our Mata code to read

```mata
use mydata, clear
putmata C
    D = J(rows(C), 1, 0)
    for (i=1; i<=rows(C); i++) {
        if (C[i]>=.) D[i] = . // new
        else for (j=1; j<=rows(C); j++) {
            if (C[j]<.) { // new
                if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
            }
        }
    }
end
getmata D
save mydata, replace
```

Easier, however, is simply to restrict Mata vector C to the nonmissing elements of Stata variable C, which we could do by replacing putmata C with

```mata
putmata C if !missing(C)
```
or, equivalently,

```mata
putmata C, omitmissing
```

Whichever way we coded it, if the data contained 100 observations and variable C contained 82 nonmissing values, new Mata vector C would contain 82 rows rather than 100. The observations corresponding to missing(C) would be omitted from the vector, and that means we could run our original Mata solution without modification.

There is, however, an issue. At the end of our code when we post the Mata solution vector D to Stata variable D—getmata D—we will need to specify which of the 100 observations are to receive the 82 results stored in the vector. Getmata has an option to handle this situation—id(varname), where varname is the name of an identification variable.

An identification variable is a variable that takes on different values for each observation in the data. The values could be 1, 2, ..., 100; or they could be 1.25, −2, ..., 16.5; or they could be Nick, Bill, ..., Mary. The values can be numeric or string, and they need not be in order. All that is important is that the variable contain a unique (different) value in each observation. Possibly, the data already contain such a variable. If not, you can create one by typing

```mata
generate fid = _n
```

When we use putmata to create vector C, we will need simultaneously to create vector fid containing the selected values of variable fid, which we can do by adding fid to the putlist:

```mata
putmata fid C if !missing(C)
```
The above command creates two vectors in Mata: \texttt{fid} and \texttt{C}. When we post the resulting vector \texttt{D} back to Stata, we will specify the \texttt{id(fid)} option to indicate into which observations \texttt{getmata} is to post the results:

\begin{verbatim}
getmata D, id(fid)
\end{verbatim}

The \texttt{id(fid)} option is taken to mean that there exists a variable named \texttt{fid} and a vector named \texttt{fid}. It is by comparing the values in each that \texttt{getmata} determines how the rows of the vectors correspond to the observations of the data.

The entire solution is

\begin{verbatim}
begin myfile.do
use mydata, clear
putmata fid C if !missing(C) // new: we put fid & add if !missing(C)
mata:
D = J(rows(C), 1, 0)
for (i=1; i<=rows(C); i++) {
    for (j=1; j<=rows(C); j++) {
        if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
    }
}
end
getmata D, id(fid) // new: we add option id(fid)
save mydata, replace
end myfile.do
\end{verbatim}

The above code will run on data with or without missing values. New variable \texttt{D} will be missing in observations where \texttt{C} is missing, but \texttt{D} will otherwise contain nonmissing values.

Using views

When you type or code \texttt{putmata C}, vector \texttt{C} is created as a copy of the Stata data. The variable and the vector are separate things. An alternative is to make the Mata vector a view onto the Stata variable. By that, we mean that both the variable and the vector share the same recording of the values. Views save memory but are slightly less efficient in terms of execution time. Views have other advantages and disadvantages, too.

For instance, if you type \texttt{putmata mpg} and then, in Mata, type \texttt{mpg[1]=20}, you will change not only the Mata vector but also the Stata data! Or if, after typing \texttt{putmata mpg}, you typed \texttt{replace mpg = 20 in 1}, that would modify both the data and the Mata vector! This is an advantage if you are fixing real errors and a disadvantage if you intend to do something else.

If in the middle of your Mata session where you are working with views you take a break and return to Stata, it is important that you do not modify the Stata data in certain ways. Rather than recording copies of the data, views record notes about the mapping. A view might record that this Mata vector corresponds to variable 3, observations 2 through 20 and 39. If you change the sort order of the data, the view will still be working with observations 2 through 20 and 39 even though those physical observations now contain different data. If you drop the first or second variable, the view will still be working with the third variable even though that will now be a different variable!

The memory savings offered by views are considerable, at least when working with large datasets. Say that you have a dataset containing 200 variables and 1,000,000 observations. Your data might be 1 GB in size. Even so, typing \texttt{putmata *, view}, and thus creating 200 vectors each with 1,000,000 rows, would consume only a few dozen kilobytes of memory.
All the examples shown above work equally well with copies or views. We have been working with copies, but in the previous example, where we coded

   putmata fid C if !missing(C)

we could switch to working with views by coding

   putmata fid C if !missing(C), view

With that one change, our code would still work and it would use less memory.

With that one change, we would still not be working with views everywhere we could, however. Vector D—the vector we create in Mata and then post back to Stata—would still be a regular vector. We can save additional memory by making D a view, too. Before we do that, let us warn you that we do not recommend doing this unless the memory savings is vitally important. The result, when complete, will be elegant and memory efficient, but the extra memory savings is seldom worth the debugging effort.

No extra changes are required to your code when the vectors you make into views contain values that are not modified in the code. Vector C is such a vector. We use the values stored in C, but we do not change them. Vector D, on the other hand, is a vector in which we change values. It is usually easier if you do not convert such vectors into views.

With that proviso, we are going to make D into a view, too, and in the process, we will drop the use of fid altogether:

```
begin myfile.do
use mydata, clear
generate D = . // new
putmata C D if !missing(C), view // changed
mata:
D[.] = J(rows(C), 1, 0) // changed
for (i=1; i<=rows(C); i++) {
    for (j=1; j<=rows(C); j++) {
        if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
    }
}
end // we drop the getmata
save mydata, replace
end myfile.do
```

In this solution, we create new Stata variable D at the outset, and then we modify the putmata command to create view vectors for both C and D. Our code, which stores results in vector D, now simultaneously posts to variable D when we store results in vector D, so we can omit the getmata D at the end because results are already posted! Moreover, we no longer have to concern ourselves with matching observations to rows via fid. Rows of D now automatically align themselves with the selected observations in variable D by the mere fact of D being a view.

The beginning of our Mata code has an important change, however. We change

   D = J(rows(C), 1, 0)

to

   D[.] = J(rows(C), 1, 0)
That change is very important. What we coded previously created vector $D$. What we now code changes the values stored in existing vector $D$. If we left what we coded previously, Mata would discard the view currently stored in $D$ and create a new $D$—a regular Mata vector unconnected to Stata—containing 0s.

**Constructing do-files**

`putmata` and `getmata` can be used interactively, but if you have much Mata code between the put and the get, you will be better off using a do-file because do-files can be easily edited when they have a mistake in them. We recommend the following outline for such do-files:

```
begin outline.do

version 17.0
mata clear
// Stata code for setup goes here
putmata ...
mata:
// Mata code goes here
end
getmata
mata clear

end outline.do
```

Notes on do-file steps:

1. A do-file should always start with a `version` statement; it ensures that the do-file continues to work in the years to come as new versions of Stata are released. See `[P] version.`

2. The do-file should not depend on Mata having certain vectors, matrices, or programs already loaded and set up because if you attempt to run the do-file again later, what you assumed may not be true. A do-file should be self-contained. To ensure that is true the first time we write and run the do-file and to ensure on subsequent runs that nothing lying around in Mata gets in our way, we clear Mata.

3. You may need to sort your data, create extra variables that your do-file will use, or drop variables that you are assuming do not already exist. In the last iteration of `myfile.do`, we needed to `generate D = .`, and it would not have been a bad idea to `capture drop D` before we did that. Our example did not depend on the sort order of the data, but if it had, we would have included the sort even if we were certain that the data would already be in the right order.

4. Put the `putmata` command here. If `putmata` includes the `omitmissing` option, then put everything you need to put in a single `putmata` command. Otherwise, you can use multiple `putmata` commands if you find that more convenient. If you use multiple `putmata` commands, be sure to include the same `if expression` and `in range` qualifiers on each one.

5. The Mata code goes here. Note that we type `mata:` (mata with a colon) to enter Mata. `mata:` ensures that errors stop Mata and thus our do-file.

6. The `getmata` command goes here if you need it. Be sure to include `getmata`'s `id(name)` or `id(vecname=varname)` option if, on the `putmata` command in step 4, you included the `if expression` qualifier or the `in range` qualifier or the `omitmissing` option. If you include `id()`, be sure you included the ID variable in the `putmata` command in step 4.
We conclude by clearing Mata again to avoid leaving memory allocated needlessly and to avoid causing problems for poorly written do-files that we might subsequently run.

`putmata` and `getmata` are designed to work interactively and in do-files. The commands are not designed to work with ado-files. An ado-file is something like a do-file, but it defines a program that implements a new command of Stata, and well-written ado-files do not use globals such as the global vectors and matrices that `putmata` creates. Ado-files use local variables. Ado-file programmers should use the Mata functions `st_data()` and `st_view()` (see [M-5] `st_data()` and [M-5] `st_view()`) to create vectors and matrices, and if necessary, use `st_store()` (see [M-5] `st_store()`) to post the contents of those vectors and matrices back to Stata.

### Stored results

`putmata` stores the following in `r()`:

 Scalars  
- `r(N)` number of rows in created vectors and matrices  
- `r(K_views)` number of vectors and matrices created as views  
- `r(K_copies)` number of vectors and matrices created as copies

The total number of vectors and matrices created is `r(K_views) + r(K_copies)`.  

If `r(N) = 0`, then `r(K_views) + r(K_copies) = 0`. A `r(N)=0` means that zero-observation vectors and matrices were created, which is to say, vectors and matrices dimensioned `0 x 1` and `0 x k`.

`getmata` stores the following in `r()`:

 Scalars  
- `r(K_new)` number of new variables created  
- `r(K_existing)` number of existing variables modified

The total number of variables modified is `r(K_new) + r(K_existing)`.

### Reference


### Also see

[M-4] `Stata` — Stata interface functions  
[M-5] `st_data()` — Load copy of current Stata dataset  
[M-5] `st_store()` — Modify values stored in current Stata dataset  
[M-5] `st_view()` — Make matrix that is a view onto current Stata dataset
range — Generate numerical range

Description

range generates a numerical range, which is useful for evaluating and graphing functions.

Quick start

Generate newv1 that ranges from 0 to π
range newv1 0 _pi

As above, but only for the first 50 observations in the dataset
range newv1 0 _pi 50

Generate newv2 that ranges from the minimum to the maximum of v2 after summarize
range newv2 r(min) r(max)

Menu

Data > Create or change data > Other variable-creation commands > Generate numerical range

Syntax

range varname #first #last [ #obs ]

Remarks and examples

range constructs the variable varname, taking on values #first to #last, inclusive, over #obs. If #obs is not specified, the number of observations in the current dataset is used.

range can be used to produce increasing sequences, such as
range x 0 12.56 100

or it can be used to produce decreasing sequences:
range z 100 1

Example 1

To graph \( y = e^{-x/6}\sin(x) \) over the interval \([0, 12.56]\), we can type
range x 0 12.56 100
Number of observations (_N) was 0, now 100.
generate y = exp(-x/6)*sin(x)
Example 2

Stata is not limited solely to graphing functions—it can draw parameterized curves as well. For instance, consider the curve given by the polar coordinate relation \( r = 2 \sin(2\theta) \). The conversion of polar coordinates to parameterized form is \((y, x) = (r \sin \theta, r \cos \theta)\), so we can type

```stata
. clear
. range theta 0 2*_pi 400
   Number of observations (_N) was 0, now 400.
. generate r = 2*sin(2*theta)
. generate y = r*sin(theta)
. generate x = r*cos(theta)
. line y x, c(l) m(i) yline(0) xline(0) aspectratio(1)
```
Also see

[D] `egen` — Extensions to generate

[D] `obs` — Increase the number of observations in a dataset
### Description

`recast` changes the storage type of the variables identified in `varlist` to `type`.

### Quick start

- Recast numeric variable `v1` to type `double` from any other numeric type
  ```
  recast double v1
  ```

- Recast string variable `v2` to `str30` from any length less than 30
  ```
  recast str30 v2
  ```

- As above, but for length longer than 30
  ```
  recast str30 v2, force
  ```

### Syntax

```
recast type varlist [, force]
```

where `type` is `byte`, `int`, `long`, `float`, `double`, `str1`, `str2`, ..., `str2045`, or `strL`.

### Option

`force` makes `recast` unsafe by causing the variables to be given the new storage type even if that will cause a loss of precision, introduction of missing values, or, for string variables, the truncation of strings.

`force` should be used with caution. `force` is for those instances where you have a variable saved as a `double` but would now be satisfied to have the variable stored as a `float`, even though that would lead to a slight rounding of its values.

### Remarks and examples

See [U] 12 Data for a description of storage types. Also see [D] `compress` and [D] `destring` for alternatives to `recast`.

Note that `recast` is not a command to change, or to map, string variables to numeric variables or numeric variables to string variables. For that, one of `encode`, `decode`, `destring`, or `tostring` is likely to be appropriate.
Example 1

`recast` refuses to change a variable’s type if that change is inappropriate for the values actually stored, so it is always safe to try:

```stata
use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
describe headroom
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage Type</th>
<th>Display Format</th>
<th>Value Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>headroom</td>
<td>float</td>
<td>%6.1f</td>
<td>Headroom (in.)</td>
</tr>
</tbody>
</table>

```stata
recast int headroom
```

`headroom: 37 values would be changed; not changed`

Our attempt to change `headroom` from a float to an int was ignored—if the change had been made, 37 values would have changed. Here is an example where the type can be changed:

```
describe mpg
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage Type</th>
<th>Display Format</th>
<th>Value Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>int</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
</tr>
</tbody>
</table>

```stata
recast byte mpg
describe mpg
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage Type</th>
<th>Display Format</th>
<th>Value Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>byte</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
</tr>
</tbody>
</table>

`recast` works with string variables as well as numeric variables, and it provides all the same protections:

```
describe make
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage Type</th>
<th>Display Format</th>
<th>Value Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str18</td>
<td>%-18s</td>
<td>Make and model</td>
</tr>
</tbody>
</table>

```stata
recast str16 make
make: 2 values would be changed; not changed
```

`recast` can be used both to promote and to demote variables:

```stata
recast str20 make
describe make
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage Type</th>
<th>Display Format</th>
<th>Value Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str20</td>
<td>%-20s</td>
<td>Make and model</td>
</tr>
</tbody>
</table>

Also see

[D] `compress` — Compress data in memory
[D] `destring` — Convert string variables to numeric variables and vice versa
[U] `12.2.2 Numeric storage types`
[U] `12.4 Strings`
**recode — Recode categorical variables**

**Description**

`recode` changes the values of numeric variables according to the rules specified. Values that do not meet any of the conditions of the rules are left unchanged, unless an `otherwise` rule is specified.

A range `#1/#2` refers to all (real and integer) values between `#1` and `#2`, including the boundaries `#1` and `#2`. This interpretation of `#1/#2` differs from that in `numlists`.

`min` and `max` provide a convenient way to refer to the minimum and maximum for each variable in `varlist` and may be used in both the from-value and the to-value parts of the specification. Combined with `if` and `in`, the minimum and maximum are determined over the restricted dataset.

The keyword rules specify transformations for values not changed by the previous rules:

- `nonmissing` — all nonmissing values not changed by the rules
- `missing` — all missing values (., .a, .b, ..., .z) not changed by the rules
- `else` — all nonmissing and missing values not changed by the rules
- `*` — synonym for `else`

`recode` provides a convenient way to define value labels for the generated variables during the definition of the transformation, reducing the risk of inconsistencies between the definition and value labeling of variables. Value labels may be defined for integer values and for the extended missing values (., .a, ..., .z), but not for noninteger values or for `sysmiss` (.)

Although this is not shown in the syntax diagram, the parentheses around the `rules` and keyword clauses are optional if you transform only one variable and if you do not define value labels.

**Quick start**

Recode 3 to 0, 4 to −1, and 5 to −2 in `v1`, and store result in `newv1`

```
recode v1 (3=0) (4=-1) (5=-2), generate(newv1)
```

As above, and recode missing values to 9

```
recode v1 (3=0) (4=-1) (5=-2) (missing=9), gen(newv1)
```

Also recode `v2` using the same rule and store result in `newv2`

```
recode v1 v2 (3=0) (4=-1) (5=-2) (missing=9), gen(newv1 newv2)
```

Same as above when adding a prefix to the old variable name

```
recode v1 v2 (3=0) (4=-1) (5=-2) (missing=9), prefix(new)
```

Recode 3 through 5 to 0 and 1 through 2 to 1, and create value label `mylabel`

```
recode v1 (3/5=0 "Value 0") (1/2=1 "Value 1"), gen(newv1) ///
    label(mylabel)
```
As above, but set all other values to 9 and label them “Invalid”

```
recode v1 (3/5=0 "Value 0") (1/2=1 "Value 1") ///
   (else=9 "Invalid"), gen(newv1) label(mylabel)
```

### Syntax

#### Basic syntax

```
recode varlist (rule) [(rule) . .] [, generate(newvar)]
```

#### Full syntax

```
recode varlist (erule) [(erule) . .] [if] [in] [, options]
```

where the most common forms for `rule` are

<table>
<thead>
<tr>
<th>rule</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td># =</td>
<td>3 = 1</td>
<td>3 recoded to 1</td>
</tr>
<tr>
<td># # =</td>
<td>2 . = 9</td>
<td>2 and . recoded to 9</td>
</tr>
<tr>
<td>#/# =</td>
<td>1/5 = 4</td>
<td>1 through 5 recoded to 4</td>
</tr>
<tr>
<td>nonmissing =</td>
<td>nonmiss = 8</td>
<td>all other nonmissing to 8</td>
</tr>
<tr>
<td>missing =</td>
<td>miss = 9</td>
<td>all other missings to 9</td>
</tr>
</tbody>
</table>

where `erule` has the form

```
element [element . .] = el ["label"]
nonmissing = el ["label"]
missing = el ["label"]
else | * = el ["label"]
```

`element` has the form

```
el | el/el
```

and `el` is

```
# | min | max
```

The keyword rules `missing`, `nonmissing`, and `else` must be the last rules specified. `else` may not be combined with `missing` or `nonmissing`. 
**options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generate(newvar)</code></td>
<td>generate <code>newvar</code> containing transformed variables; default is to replace existing variables</td>
</tr>
<tr>
<td><code>prefix(str)</code></td>
<td>generate new variables with <code>str</code> prefix</td>
</tr>
<tr>
<td><code>label(name)</code></td>
<td>specify a name for the value label defined by the transformation rules</td>
</tr>
<tr>
<td><code>copyrest</code></td>
<td>copy out-of-sample values from original variables</td>
</tr>
<tr>
<td><code>test</code></td>
<td>test that rules are invoked and do not overlap</td>
</tr>
</tbody>
</table>

**Options**

`generate(newvar)` specifies the names of the variables that will contain the transformed variables. `into()` is a synonym for `generate()`. Values outside the range implied by `if` or `in` are set to missing (.), unless the `copyrest` option is specified.

If `generate()` is not specified, the input variables are overwritten; values outside the `if` or `in` range are not modified. Overwriting variables is dangerous (you cannot undo changes, value labels may be wrong, etc.), so we strongly recommend specifying `generate()`.

`prefix(str)` specifies that the recoded variables be returned in new variables formed by prefixing the names of the original variables with `str`.

`label(name)` specifies a name for the value label defined from the transformation rules. `label()` may be defined only with `generate()` (or its synonym, `into()`) and `prefix()`. If a variable is recoded, the label name defaults to `newvar` unless a label with that name already exists.

`copyrest` specifies that out-of-sample values be copied from the original variables. In line with other data management commands, `recode` defaults to setting `newvar` to missing (.) outside the observations selected by `if exp` and `in range`.

`test` specifies that Stata test whether rules are ever invoked or that rules overlap; for example, `(1/5=1) (3=2)`.

**Remarks and examples**

Remarks are presented under the following headings:

Simple examples
Setting up value labels with recode
Referring to the minimum and maximum in rules
Recoding missing values
Recoding subsets of the data
Otherwise rules
Test for overlapping rules
Video example
Simple examples

Many users experienced with other statistical software use the `recode` command often, but easier and faster solutions in Stata are available. On the other hand, `recode` often provides simple ways to manipulate variables that are not easily accomplished otherwise. Therefore, we show other ways to perform a series of tasks with and without `recode`.

We want to change 1 to 2, leave all other values unchanged, and store the results in the new variable `nx`.

```
.recode x (1 = 2), gen(nx)
```

or

```
.generate nx = x
.replace nx = 2 if nx==1
```

or

```
.generate nx = cond(x==1,2,x)
```

We want to swap 1 and 2, saving them in `nx`.

```
.recode x (1 = 2) (2 = 1), gen(nx)
```

or

```
.generate nx = cond(x==1,2,cond(x==2,1,x))
```

We want to recode `item` by collapsing 1 and 2 into 1, 3 into 2, and 4 to 7 (boundaries included) into 3.

```
.recode item (1 2 = 1) (3 = 2) (4/7 = 3), gen(Ritem)
```

or

```
.generate Ritem = item
.replace Ritem = 1 if inlist(item,1,2)
.replace Ritem = 2 if item==3
.replace Ritem = 3 if inrange(item,4,7)
```

We want to change the “direction” of the 1,...,5 valued variables `x1`, `x2`, `x3`, storing the transformed variables in `nx1`, `nx2`, and `nx3` (that is, we form new variable names by prefixing old variable names with an “n”).

```
.recode x1 x2 x3 (1=5) (2=4) (3=3) (4=2) (5=1), pre(n) test
```

or

```
.generate nx1 = 6-x1
.generate nx2 = 6-x2
.generate nx3 = 6-x3
.forvalues i = 1/3 {
    generate nx'i' = 6-x'i'
}
```

In the categorical variable `religion`, we want to change 1, 3, and the real and integer numbers 3 through 5 into 6; we want to set 2, 8, and 10 to 3 and leave all other values unchanged.

```
.recode religion 1 3/5 = 6 2 8 10 = 3
```

or

```
.replace religion = 6 if religion==1 | inrange(religion,3,5)
.replace religion = 3 if inlist(religion,2,8,10)
```
This example illustrates two features of `recode` that were included for backward compatibility with previous versions of `recode` but that we do not recommend. First, we omitted the parentheses around the rules. This is allowed if you recode one variable and you do not plan to define value labels with `recode` (see below for an explanation of this feature). Personally, we find the syntax without parentheses hard to read, although we admit that we could have used blanks more sensibly. Because difficulties in reading may cause us to overlook errors, we recommend always including parentheses.

Second, because we did not specify a `generate()` option, we overwrite the `religion` variable. This is often dangerous, especially for “original” variables in a dataset. We recommend that you always specify `generate()` unless you want to overwrite your data.

**Setting up value labels with recode**

The `recode` command is most often used to transform categorical variables, which are many times value labeled. When a value-labeled variable is overwritten by `recode`, it may well be that the value label is no longer appropriate. Consequently, output that is labeled using these value labels may be misleading or wrong.

When `recode` creates one or more new variables with a new classification, you may want to put value labels on these new variables. It is possible to do this in three steps:

1. Create the new variables (`recode ... , gen()`).
2. Define the value label (`label define ...`).
3. Link the value label to the variables (`label value ...`).

Inconsistencies may emerge from mistakes between steps 1 and 2. Especially when you make a change to the recode 1, it is easy to forget to make a similar adjustment to the value label 2. Therefore, `recode` can perform steps 2 and 3 itself.

Consider recoding a series of items with values

1 = strongly agree
2 = agree
3 = neutral
4 = disagree
5 = strongly disagree

into three items:

1 = positive (= “strongly agree” or “agree”)
2 = neutral
3 = negative (= “strongly disagree” or “disagree”)

This is accomplished by typing

```
. recode item* (1 2 = 1 positive) (3 = 2 neutral) (4 5 = 3 negative), pre(R)
> label(Item3)
```

which is much simpler and safer than

```
. recode item1-item7 (1 2 = 1) (3 = 2) (4 5 = 3), pre(R)
. label define Item3 1 positive 2 neutral 3 negative
. forvalues i = 1/7 {
    label value Ritem'i' Item3
}
```
Example 1

As another example, let’s recode vote (voting intentions) for 12 political parties in the Dutch parliament into left, center, and right parties. We then tabulate the original and new variables so that we can check that everything came out correctly.

```
use https://www.stata-press.com/data/r17/recodexmpl
label list pparty
pparty:
  1 pvda
  2 cda
  3 d66
  4 vvd
  5 groenlinks
  6 sgp
  7 rpf
  8 gpv
  9 aov
 10 unie55
11 sp
12 cd
recode polpref (1 5 11 = 1 left) (2 3 = 2 center) (4 6/10 12 = 3 right),
gen(polpref3)
(2020 differences between polpref and polpref3)
tabulate polpref polpref3
```

```
<table>
<thead>
<tr>
<th>pol party</th>
<th>RECODE of polpref (pol party choice if elections)</th>
<th>right</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left</td>
<td>center</td>
<td></td>
</tr>
<tr>
<td>pvda</td>
<td>622</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cda</td>
<td>0</td>
<td>525</td>
<td>0</td>
</tr>
<tr>
<td>d66</td>
<td>0</td>
<td>634</td>
<td>0</td>
</tr>
<tr>
<td>vvd</td>
<td>0</td>
<td>0</td>
<td>930</td>
</tr>
<tr>
<td>groenlinks</td>
<td>199</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sgp</td>
<td>0</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>rpf</td>
<td>0</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>gpv</td>
<td>0</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>aov</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>unie55</td>
<td>0</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>sp</td>
<td>45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cd</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>866</td>
<td>1,159</td>
<td>1,142</td>
</tr>
</tbody>
</table>
```

Referring to the minimum and maximum in rules

recode allows you to refer to the minimum and maximum of a variable in the transformation rules. The keywords min and max may be included as a from-value, as well as a to-value.

For example, we might divide age into age categories, storing in iage.

```
recode age (0/9=1) (10/19=2) (20/29=3) (30/39=4) (40/49=5) (50/max=6),
gen(iage)
```
or

```
generate iage = 1 + irecode(age, 9, 19, 29, 39, 49)
```
. generate iage = min(6, 1+int(age/10))

As another example, we could set all incomes less than 10,000 to 10,000 and those more than 200,000 to 200,000, storing the data in ninc.

. recode inc (min/10000 = 10000) (200000/max = 200000), gen(ninc)

or

. generate ninc = inc
. replace ninc = 10000 if ninc<10000
. replace ninc = 200000 if ninc>200000 & !missing(ninc)

or

. generate ninc = max(min(inc,200000),10000)

or

. generate ninc = clip(inc,10000,200000)

Recoding missing values

You can also set up rules in terms of missing values, either as from-values or as to-values. Here `recode` mimics the functionality of `mvdecode` and `mvencode` (see [D] `mvencode`), although these specialized commands execute much faster.

Say that we want to change missing (.) to 9, storing the data in X:

. recode x (.=9), gen(X)

or

. generate X = cond(x==., 9, x)

or

. mvencode x, mv(.=9) gen(X)

We want to change 9 to .a and 8 to ., storing the data in z.

. recode x (9=.a) (8=.), gen(z)

or

. generate z = cond(x==9, .a, cond(x==8, ., x))

or

. mvdecode x, mv(9=.a, 8=. ) gen(z)

Recoding subsets of the data

We want to swap in x the values 1 and 2 only for those observations for which `age>40`, leaving all other values unchanged. We issue the command

. recode x (1=2) (2=1) if age>40, gen(y)

or

. generate y = cond(x==1,2,cond(x==2,1,x)) if age>40
We are in for a surprise. \( y \) is missing for observations that do not satisfy the \( \text{if} \) condition. This outcome is in accordance with how Stata’s data manipulation commands usually work. However, it may not be what you intend. The \texttt{copyrest} option specifies that \( x \) be copied into \( y \) for all nonselected observations:

\[
\text{. recode } x \ (1=2) \ (2=1) \ \text{if age>40, gen(y) copy}
\]

or

\[
\text{. generate } y = x
\]

\[
\text{. recode } y \ (1=2) \ (2=1) \ \text{if age>40}
\]

or

\[
\text{. generate } y = \text{cond(age>40, cond(x==1,2, cond(x==2,1,x)),x))}
\]

**Otherwise rules**

In all our examples so far, \texttt{recode} had an implicit rule that specified that values that did not meet the conditions of any of the rules were to be left unchanged. \texttt{recode} also allows you to use an “otherwise rule” to specify how untransformed values are to be transformed. \texttt{recode} supports three kinds of otherwise conditions:

- **nonmissing**: all nonmissing not yet transformed
- **missing**: all missing values not yet transformed
- **else**: all values, missing or nonmissing, not yet transformed

The otherwise rules are to be specified after the standard transformation rules. \texttt{nonmissing} and \texttt{missing} may be combined with each other, but not with \texttt{else}.

Consider a recode that swaps the values 1 and 2, transforms all other nonmissing values to 3, and transforms all missing values (that is, sysmiss and the extended missing values) to \( . \) (sysmiss). We could type

\[
\text{. recode } x \ (1=2) \ (2=1) \ (\text{nonmissing}=3) \ (\text{missing}=.), \ \text{gen(z)}
\]

or

\[
\text{. generate } z = \text{cond(x==1,2, cond(x==2,1, cond(!missing(x),3),.))}
\]

As a variation, if we had decided to recode all extended missing values to \( .a \) but to keep sysmiss \( . \) distinct at \( . \), we could have typed

\[
\text{. recode } x \ (1=2) \ (2=1) \ (\text{.}=. \) \ (\text{nonmissing}=3) \ (\text{missing}=.a), \ \text{gen(z)}
\]

**Test for overlapping rules**

\texttt{recode} evaluates the rules from left to right. Once a value has been transformed, it will not be transformed again. Thus if rules “overlap”, the first matching rule is applied, and further matches are ignored. A common form of overlapping is illustrated in the following example:

\[
\ldots \ 1/5 = 1 \ 5/10 = 2
\]

Here 5 occurs in the condition parts of both rules. Because rules are matched left to right, 5 matches the first rule, and the second rule will not be tested for 5, unless \texttt{recode} is instructed to test for rule overlap with the \texttt{test} option.
Other instances of overlapping rules usually arise because you mistyped the rules. For instance, you are recoding voting intentions for parties in elections into three groups of parties (left, center, right), and you type

... (1/5 = 1) ... (3 = 2)

Party 3 matches the conditions 1/5 and 3. Because `recode` applies the first matching rule, party 3 will be mapped into party category 1. The second matching rule is ignored. It is not clear what was wrong in this example. You may have included party 3 in the range 1/5 or mistyped 3 in the second rule. Either way, `recode` did not notice the problem and your data analysis is in jeopardy. The `test` option specifies that `recode` display a warning message if values are matched by more than one rule. With the `test` option specified, `recode` also tests whether all rules were applied at least once and displays a warning message otherwise. Rules that never matched any data may indicate that you mistyped a rule, although some conditions may not have applied to (a selection of) your data.

Video example

How to create a categorical variable from a continuous variable

Acknowledgment

This version of `recode` was written by Jeroen Weesie of the Department of Sociology at Utrecht University, The Netherlands.

Also see

[D] `generate` — Create or change contents of variable

[D] `mvencode` — Change missing values to numeric values and vice versa
rename — Rename variable

Description

rename changes the name of an existing variable *old_varname* to *new_varname*; the contents of the variable are unchanged. Also see [D] rename group for renaming groups of variables.

Quick start

Change the name of *v1* to *var1*

```
rename v1 var1
```

Also change the name of *v2* to *var2*

```
rename v2 var2
```

Menu

Data > Data utilities > Rename groups of variables

Syntax

```
rename *old_varname* *new_varname*
```

collect is allowed; see [U] 11.1.10 Prefix commands.

Remarks and examples

Example 1

rename allows you to change variable names. Say that we have labor market data for siblings.

```
. use https://www.stata-press.com/data/r17/renamexmpl
. describe
Contains data from https://www.stata-press.com/data/r17/renamexmpl.dta
Observations: 277
Variables: 6
  9 Jan 2020 11:57
```

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>famid</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edu</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>promo</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td>float</td>
<td>%9.0g</td>
<td>sex</td>
<td></td>
</tr>
<tr>
<td>inc</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sorted by: famid

738
We decide to rename the `exp` and `inc` variables.

```
. rename exp experience
. rename inc income
. describe
Contains data from https://www.stata-press.com/data/r17/renamexmpl.dta
Observations: 277
Variables: 6
9 Jan 2020 11:57

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>famid</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edu</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>experience</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>promo</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td>float</td>
<td>%9.0g</td>
<td>sex</td>
<td></td>
</tr>
<tr>
<td>income</td>
<td>float</td>
<td>%9.0g</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Sorted by: famid
Note: Dataset has changed since last saved.

The `exp` variable is now called `experience`, and the `inc` variable is now called `income`.

References


Also see

[D] rename group — Rename groups of variables
[D] generate — Create or change contents of variable
[D] varmanage — Manage variable labels, formats, and other properties
rename changes the names of existing variables to the new names specified. See \[D\] rename for the base rename syntax. Documented here is the advanced syntax for renaming groups of variables.

Quick start

Change the name of v1 to var1 and v2 to var2
   rename (v1 v2) (var1 var2)

Change the name of V1 to v1 and V2 to v2
   rename V1 V2, lower

Add suffix old to variables v1, v2, ... for one or more digits
   rename v# =old

Remove suffix old from all variables ending in old
   rename *old *

Remove prefix old from all variables beginning with old
   rename old* *

Note: A complete list of rules for renaming groups of variables appears below the syntax diagram.
Syntax

Rename a single variable

rename old new [ , options1 ]

Rename groups of variables

rename (old1 old2 ...) (new1 new2 ...) [ , options1 ]

Change the case of groups of variable names

rename old1 old2 ..., { upper | lower | proper } [ options2 ]

where old and new specify the existing and the new variable names. The rules for specifying them are

1. rename stat status: Renames stat to status.
   - Rule 1: This is the same rename command documented in [D] rename, with which you are familiar.
2. rename (stat inc) (status income): Renames stat to status and inc to income.
   - Rule 2: Use parentheses to specify multiple variables for old and new.
3. rename (v1 v2) (v2 v1): Swaps v1 and v2.
   - Rule 3: Variable names may be interchanged.
4. rename (a b c) (b c a): Swaps names. Renames a to b, b to c, and c to a.
   - Rule 4: There is no limit to how many names may be interchanged.
5. rename (a b c) (c b a): Renames a to c and c to a, but leaves b as is.
   - Rule 5: Renaming variables to themselves is allowed.
6. rename jan* *1: Renames all variables starting with jan to instead end with 1, for example, janstat to stat1, janinc to inc1, etc.
   - Rule 6.1: * in old selects the variables to be renamed. * means that zero or more characters go here.
   - Rule 6.2: * in new corresponds with * in old and stands for the text that * in old matched.
   - * in new or old is called a wildcard character, or just a wildcard.
   rename jan* *: Removes prefix jan.
   rename *jan *: Removes suffix jan.
7. rename jan? ?1: Renames all variables starting with jan and ending in one character by removing jan and adding 1 to the end; for example, jans is renamed to s1, but janstat remains unchanged. ? means that exactly one character goes here, just as * means that zero or more characters go here.
   - Rule 7: ? means exactly one character, ?? means exactly two characters, etc.
8. rename *jan* **: Removes prefix, midfix, and suffix jan, for example, janstat to stat, injanstat to instat, and subjan to sub.
   - Rule 8: You may specify more than one wildcard in old and in new. They correspond in the order given.
rename *jan* *s*1: Renames all variables that start with jan and contain s to instead end in 1, dropping the jan, for example, janstat to stat1 and janest to est1, but not janinc to inc1.

9. rename jan*s* *s*1: Renames all variables that start with jan and contain s to instead end in 1, dropping the jan, for example, janstat to stat1 and janest to est1, but not janinc to inc1.

Rule 9: You may specify more wildcards in old than in new.

10. rename *jan**: Removes jan and whatever follows from variable names, thereby renaming statjan to stat, incjan71 to inc, ....

Rule 10: Wildcard . (dot) in new skips over the corresponding wildcard in old.

11. rename *pop jan=: Adds prefix jan to all variables ending in pop, for example, age1pop to janage1pop, ....

rename (status bp time) admit=: Renames status to admitstatus, bp to admitbp, and time to admittime.

rename whatever pre=: Adds prefix pre to all variables selected by whatever, however whatever is specified.

Rule 11: Wildcard = in new specifies the original variable name.

rename whatever =jan: Adds suffix jan to all variables selected by whatever.

rename whatever pre=fix: Adds prefix pre and suffix fix to all variables selected by whatever.

12. rename v# stat#: Renames v1 to stat1, v2 to stat2, ...., v10 to stat10, ....

Rule 12.1: # is like * but for digits. # in old selects one or more digits.

Rule 12.2: # in new copies the digits just as they appear in the corresponding old.

13. rename v(#) stat(#): Renames v1 to stat1, v2 to stat2, ...., but does not rename v10, ....

Rule 13.1: (#) in old selects exactly one digit. Similarly, (##) selects exactly two digits, and so on, up to ten # symbols.

Rule 13.2: (#) in new means reformat to one or more digits. Similarly, (##) reformats to two or more digits, and so on, up to ten # symbols.

rename v(##) stat(#): Renames v01 to stat01, v02 to stat02, ...., v10 to stat10, ...., but does not rename v0, v1, v2, ...., v9, v100, ....

14. rename v# v(##): Renames v1 to v01, v2 to v02, ...., v10 to v10, v11 to v11, ...., v100 to v100, v101 to v101, ....

Rule 14: You may combine #, (#), (##), ... in old with any of #, (#), (##), ... in new.

rename v(##) v(#): Renames v01 to v1, v02 to v2, ...., v10 to v10, ...., but does not rename v001, etc.

rename stat(##) stat_20(##): Renames stat10 to stat_2010, stat11 to stat_2011, ...., but does not rename stat1, stat2, ....

rename stat(#) to stat_200(#): Renames stat1 to stat_2001, stat2 to stat_2002, ...., but does not rename stat10 or stat_2010.
15. rename v# (a b c): Renames v1 to a, v10 to b, and v2 to c if variables v1, v10, v2 appear in that order in the data. Because three variables were specified in new, v# in old must select three variables or rename will issue an error.

Rule 15.1: You may mix syntaxes. Note that the explicit and implied numbers of variables must agree.

rename v# (a b c), sort: Renames (for instance) v1 to a, v2 to b, and v10 to c.

Rule 15.2: The sort option places the variables selected by old in order and does so smartly. In the case where #, (#), (##), ... appear in old, sort places the variables in numeric order.

rename v* (a b c), sort: Renames (for instance) valpha to a, vbeta to b, and vgamma to c regardless of the order of the variables in the data.

Rule 15.3: In the case where * or ? appears in old, sort places the variables in alphabetical order.

16. rename v# v#, renumber: Renames (for instance) v9 to v1, v10 to v2, v8 to v3, ..., assuming that variables v9, v10, v8, ... appear in that order in the data.

Rule 16.1: The renumber option resequences the numbers.

rename v# v#, renumber sort: Renames (for instance) v8 to v1, v9 to v2, v10 to v3, ... Concerning option sort, see rule 15.2 above.

rename v# v#, renumber(10) sort: Renames (for instance) v8 to v10, v9 to v11, v10 to v12, ...

Rule 16.2: The renumber(#) option allows you to specify the starting value.

17. rename v* v#, renumber: Renames (for instance) valpha to v1, vgamma to v2, vbeta to v3, ..., assuming variables valpha, vgamma, vbeta, ... appear in that order in the data.

Rule 17: # in new may correspond to *, ?, #, (#), (##), ... in old.

rename v* v#, renumber sort: Renames (for instance) valpha to v1, vbeta to v2, vgamma to v3, ... Also see rule 15.3 above concerning the sort option.

rename *stat stat#, renumber: Renames, for instance, janstat to stat1, febstat to stat2, ... Note that # in new corresponds to * in old, just as in the previous example.

rename *stat stat(##), renumber: Renames, for instance, janstat to stat01, febstat to stat02, ....

rename *stat stat#, renumber(0): Renames, for instance, janstat to stat0, febstat to stat1, ....

rename *stat stat#, renumber sort: Renames, for instance, aprstat to stat1, augstat to stat2, ....

18. rename (a b c) v#, addnumber: Renames a to v1, b to v2, and c to v3.

Rule 18: The addnumber option allows you to add numbering. More formally, if you specify addnumber, you may specify one more wildcard in new than is specified in old, and that extra wildcard must be #, (#), (##), ....

19. rename a(#)(#) a(#) [2] (#) [1]: Renames a12 to a21, a13 to a31, a14 to a41, ..., a21 to a12, ....

Rule 19.1: You may specify explicit subscripts with wildcards in new to make explicit its matching wildcard in old. Subscripts are specified in square brackets after a wildcard in new. The number refers to the number of the wildcard in old.
rename *stat* *[2]stat*[1]: Swaps prefixes and suffixes; it renames bpstata to astatbp, rstater to erstatr, etc.

rename *stat* *[2]stat* : Does the same as above; it swaps prefixes and suffixes.

Rule 19.2: After specifying a subscripted wildcard, subsequent unsubscripted wildcards correspond to the same wildcards in old as they would if you had removed the subscripted wildcards altogether.

rename v#a# v##[1]_a#[2] : Renames v1a1 to v1_1_a1, v1a2 to v1_1_a2, ..., v2a1 to v2_2_a1, ...

Rule 19.3: Using subscripts, you may refer to the same wildcard in old more than once. Subscripts are commonly used to interchange suffixes at the ends of variable names. For instance, you have districts and schools within them, and many of the variable names in your data match *__#_.#_. The first number records district and the second records school within district. To reverse the ordering, you type rename *__.##.*.##[3]_##[2]. When specifying subscripts, you refer to them by the position number in the original name. For example, our original name was *__.##_. so [1] refers to *, [2] refers to the first #, and [3] refers to the last #.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning in old</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more characters</td>
</tr>
<tr>
<td>?</td>
<td>1 character exactly</td>
</tr>
<tr>
<td>#</td>
<td>1 or more digits</td>
</tr>
<tr>
<td>(#)</td>
<td>1 digit exactly</td>
</tr>
<tr>
<td>(##)</td>
<td>2 digits exactly</td>
</tr>
<tr>
<td>(###)</td>
<td>3 digits exactly</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>(##########)</td>
<td>10 digits exactly</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specifier</th>
<th>May correspond in old with</th>
<th>Meaning in new</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*, ?, #, (#), ...</td>
<td>copies matched text</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>copies a character</td>
</tr>
<tr>
<td>#</td>
<td>#, (#), ...</td>
<td>copies a number as is</td>
</tr>
<tr>
<td>(#)</td>
<td>#, (#), ...</td>
<td>reformats to 1 or more digits</td>
</tr>
<tr>
<td>(##)</td>
<td>#, (#), ...</td>
<td>reformats to 2 or more digits</td>
</tr>
<tr>
<td>...</td>
<td>#, (#), ...</td>
<td>reformats to 10 digits</td>
</tr>
<tr>
<td>(##########)</td>
<td>#, (#), ...</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>*, ?, #, (#), ...</td>
<td>skip</td>
</tr>
<tr>
<td>=</td>
<td>nothing</td>
<td>copies entire variable name</td>
</tr>
</tbody>
</table>

Specifier # in any of its guises may also correspond with * or ? if the renumber option is specified.
Options for renaming variables

`addnumber` and `addnumber(#)` specify to add a sequence number to the variable names. See item 18 of Syntax. If `#` is not specified, the sequence number begins with 1.

`renumber` and `renumber(#)` specify to replace existing numbers or text in a set of variable names with a sequence number. See items 16 and 17 of Syntax. If `#` is not specified, the sequence number begins with 1.

`sort` specifies that the existing names be placed in order before the renaming is performed. See item 15 of Syntax for details. This ordering matters only when `addnumber` or `renumber` is also specified or when specifying a list of variable names for `old` or `new`.

`dryrun` specifies that the requested renaming not be performed but instead that a table be displayed showing the old and new variable names. It is often a good idea to specify this option before actually renaming the variables.

`r` is a programmer’s option that requests that old and new variable names be stored in `r()` for programming use.

Options for changing the case of groups of variable names

`upper`, `lower`, and `proper` specify how the variables are to be renamed. `upper` specifies that ASCII letters in variable names be changed to uppercase; `lower`, to lowercase; and `proper`, to having the first ASCII letter capitalized and the remaining ASCII letters in lowercase. One of these three options must be specified. Note that these options do not handle Unicode characters beyond the plain ASCII range. To change Unicode characters in the variable names to uppercase, lowercase, or titlecase, use functions `ustrupper()`, `ustrlower()`, and `ustrtitle()`. See the technical note in Remarks and examples.

dryrun and `r` are the same options as documented directly above.
Remarks and examples

Remarks are presented under the following headings:

Advice
Explanation

* matches 0 or more characters; use *? to match 1 or more
* is greedy
# is greedier

Advice

1. Read [D] rename before reading this entry.
2. Read items 1–19 (the Rules) under Syntax above before reading the rest of these remarks.
3. Specify the dryrun option when using complicated patterns. dryrun presents a table of the old and new variable names rather than actually renaming the variables, so you can check that the patterns you have specified produce the desired result.

Explanation

The rename command has three syntaxes; see Syntax. See [D] rename for details on the first syntax, renaming a single variable. The remaining two syntaxes are for renaming groups of variables and for changing the case of groups of variables. These two syntaxes are the ones we will focus on for the remainder of this manual entry. Here they are again:

\[
\text{rename } (old_1 \ldots, \{upper|lower|proper\})
\]

The second syntax shown above merely changes the case of variables, such as MPG or mpg or Mpg. For instance, to rename all variables to be lowercase, type

\[
\text{rename } *, \text{lower}
\]

The first syntax shown above is more daunting and more powerful. The first syntax has two styles, with and without parentheses:

\[
\text{rename } (bp_0 bp_1) (bp_1 bp_0)
\]

You can combine the two styles whenever it is convenient.

\[
\text{rename } v* (mpg weight displacement)
\]

We summarize all of this by simply writing the syntax as

\[
\text{rename } old \ new, \ldots
\]

and referring to old and new.

Wildcards play different but related roles in old and new. When you type

\[
\text{rename pop*80 pop*80}
\]
the wildcard (* in this case) in old specifies which variables are to be renamed, and in new the wildcard stands for the text that appears in the variables to be renamed. In this case, there is just one wildcard, but sometimes there are more.

In old, * means zero or more characters go here. Specifying pop*80 means find all variables that begin with pop and end in 80. Say that doing so results in three variables being found: poplt2080, pop204080, and pop41plus80. To understand how * is interpreted in new, it is useful to write the three found variables like this:

\[
\begin{align*}
\text{pop*80} &= \text{pop} + \ast + 80 \\
\text{poplt2080} &= \text{pop} + \text{lt20} + 80 \\
\text{pop204080} &= \text{pop} + 2040 + 80 \\
\text{pop41plus80} &= \text{pop} + 41\text{plus} + 80
\end{align*}
\]

* in new refers to what was found by * in old. So the new pattern pop_--1980 will assemble the following new variable names for each of the old names:

<table>
<thead>
<tr>
<th>old variable</th>
<th>* is</th>
<th>→ pop_--1980 is</th>
</tr>
</thead>
<tbody>
<tr>
<td>poplt2080</td>
<td>lt20</td>
<td>pop_lt20_1980</td>
</tr>
<tr>
<td>pop204080</td>
<td>2040</td>
<td>pop_2040_1980</td>
</tr>
<tr>
<td>pop41plus80</td>
<td>41plus</td>
<td>pop_41plus_1980</td>
</tr>
</tbody>
</table>

Thus typing rename pop*80 pop_--1980 is equivalent to typing

rename poplt2080 pop_lt20_1980
rename pop204080 pop_2040_1980
rename pop41plus80 pop_41plus_1980

There are three basic wildcard characters for specification in old, and they filter the variables to be renamed:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more characters go here</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>exactly 1 character goes here</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>number goes here (this one comes in 11 flavors!)</td>
<td></td>
</tr>
</tbody>
</table>

The generic # listed above collects all the digits. The other 10 flavors are (#), which means exactly 1 digit goes here; (##), which means exactly 2 digits go here; and so on, up to exactly 10 digits go here.

All the above, the \(3 + 10 = 13\) wildcard characters, can appear in new, where each has a different but related meaning:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>copy corresponding text from old as is</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>copy corresponding character from old</td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>copy corresponding number from old as is</td>
<td></td>
</tr>
<tr>
<td>(#)</td>
<td>reformat corresponding number from old to 1 or more digits</td>
<td></td>
</tr>
<tr>
<td>(##)</td>
<td>reformat corresponding number from old to 2 or more digits</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In addition, new allows two special wildcard characters of its own:

- `=` copy the entire original variable name
- `.` skip the corresponding text in old

With the above information and the definitions of the options, you can derive on your own the first eighteen rules given in Syntax. The nineteenth rule concerns subscripting. In new, you can specify explicitly to which wildcard in old you are referring. You can type

rename pop*80 pop_*_1980

or you can type

rename pop*80 pop_*[1]_1980

thus making it explicit that the * in new is referring to the text matched by the first wildcard in old. That * corresponds to * is hardly surprising, especially when there is only one * in old, so let's complicate the example:

rename v*_* outcome_*_*

You can type that command, or you can type

rename v*_* outcome_*[1]_*[2]

More importantly, you can specify the subscripts in whatever order you wish, so you could type

rename v*_* outcome_*[2]_*[1]

That command would interchange the text in old matched by the two wildcards.

**matches 0 or more characters; use ?* to match 1 or more**

l*a in old matches louisiana and it matches la because * means zero or more characters. What if you want to match louisiana and lymphoma but not la?

For instance, say you have from–to variables named from*to* and from variables named from*. The problem is that variable fromtoledo would match from*to*. To avoid that, rather than describing the from–to pattern from*to*, you use from?*to?*. Thus you could type

rename from?*to?* from_?*_to_?*

?* is not a secret wildcard we have yet to tell you about—it is merely the two wildcards ? and * in sequence. ? means exactly one character goes here, and * means zero or more characters go here, so ?* means one or more characters go here. In the same way, ??* means two or more characters go here, and so on.

**is greedy**

Consider the existing variable assessment and pattern *s* in old. Clearly, *s* matches assessment, but how? That is, among these possibilities,

<table>
<thead>
<tr>
<th>assessment</th>
<th>s</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>+s+</td>
<td>sessment</td>
</tr>
<tr>
<td>as</td>
<td>+s+</td>
<td>essment</td>
</tr>
<tr>
<td>asse</td>
<td>+s+</td>
<td>sment</td>
</tr>
<tr>
<td>asses</td>
<td>+s+</td>
<td>ment</td>
</tr>
</tbody>
</table>

...
which one is true? We need to know the answer to know what each of the corresponding wildcards in new will mean. The answer is that * is greedy, and the pattern is matched from left to right. As we move through the variable name from left to right, at each step * takes the most characters possible, subject to the pattern working out.

\[
\begin{array}{c}
* \\
\text{assessment} = \text{asses} + s + \text{ment}
\end{array}
\]

Thus the first * in new would stand for asses and the second would stand for ment.

The “subject to the pattern working out” part is important. Variable sunglasses would be broken out by *s* as

\[
\begin{array}{c}
* \\
\text{sunglasses} = \text{sunglasse} + s + \text{nothing}
\end{array}
\]

But by *s?*, the breakout would be

\[
\begin{array}{c}
* \\
\text{sunglasses} = \text{sunglas} + s + e + s
\end{array}
\]

### # is greedier

Wildcard # in old is greedier than *, which means that when * and # are up against each other, # wins.

Consider the pattern **#** and the variable name v1234. Given that * is greedy and that the # specifies one or more digits, the possible solutions are

\[
\begin{array}{c}
v1234 = * \\
v123 + 4 \\
v12 + 34 \\
v1 + 234 \\
v + 1234
\end{array}
\]

The solution chosen by rename is the last one, v + 1234. Thus you can type

```
rename **# period_#[2]
```

without concern that some digits might be lost.

⚠️ **Technical note**

You cannot directly use functions `ustrupper()`, `ustrlower()`, and `ustrtitle()` in your rename command. You must first create a local macro with the new variable name and then use that macro in your rename command. For example,

```bash
.local new = ustrlower(Ubicación)
.rename Ubicación 'new'
```
You can use multiple local macros in a varlist. For example,

```stata
local new1 = ustrlower(Ubicación1)
local new2 = ustrlower(Ubicación2)
rename (Ubicación1 Ubicación2) ('new1' 'new2')
```

For more information about local macros, see [U] 18.3.1 Local macros.

### Stored results

rename stores nothing in r() by default. If the r option is specified, then rename stores the following in r():

Scalar
- r(n) number of variables to be renamed

Macros
- r(oldnames) original variable names
- r(newnames) new variable names

Variables that are renamed to themselves are omitted from the recorded lists.

### Also see

[D] rename — Rename variable
[D] generate — Create or change contents of variable
[D] varmanage — Manage variable labels, formats, and other properties
reshape — Convert data from wide to long form and vice versa

Description

 reshape converts data from *wide* to *long* form and vice versa.

Quick start

Create \( v \) from 2 time periods stored in \( v1 \) and \( v2 \) for observations identified by \( idvar \) and add \( tvar \) identifying time period

\[
\text{reshape long } v, \text{ i(idvar) } j(tvar)
\]

Create \( v \) from 2 subobservations stored in \( v1 \) and \( v2 \) for observations identified by \( idvar \) and add \( subobs \) identifying each subobservation

\[
\text{reshape long } v, \text{ i(idvar) } j(subobs)
\]

As above, but allow \( subobs \) to contain strings

\[
\text{reshape long } v, \text{ i(idvar) } j(subobs) \text{ string}
\]

Undo results from above

\[
\text{reshape wide}
\]

Create \( v1 \) and \( v2 \) from \( v \) with observations identified by \( idvar \) and time period identified by \( tvar \)

\[
\text{reshape wide } v, \text{ i(idvar) } j(tvar)
\]

Undo results from above

\[
\text{reshape long}
\]

Create \( var \) and time identifier \( tvar \) from \( v1ar \) and \( v2ar \) with observation identifier \( idvar \)

\[
\text{reshape long } v@ar, \text{ i(idvar) } j(tvar)
\]

Menu

Data > Create or change data > Other variable-transformation commands > Convert data between wide and long
Syntax

Overview

<table>
<thead>
<tr>
<th>long</th>
<th>wide</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i ) ( j ) ( stub )</td>
<td>( i ) ( stub_1 ) ( stub_2 )</td>
</tr>
<tr>
<td>1 1 4.1</td>
<td>1 4.1 4.5</td>
</tr>
<tr>
<td>1 2 4.5</td>
<td>2 3.3 3.0</td>
</tr>
<tr>
<td>2 1 3.3</td>
<td></td>
</tr>
<tr>
<td>2 2 3.0</td>
<td></td>
</tr>
</tbody>
</table>

To go from long to wide:

\[
\text{reshape wide } stub, i(i) \ j(j)
\]

To go from wide to long:

\[
\text{reshape long } stub, i(i) \ j(j)
\]

To go back to long after using \text{reshape wide}:

\[
\text{reshape long}
\]

To go back to wide after using \text{reshape long}:

\[
\text{reshape wide}
\]

Basic syntax

Convert data from wide form to long form

\[
\text{reshape long stubnames, i(varlist) [options]}
\]

Convert data from long form to wide form

\[
\text{reshape wide stubnames, i(varlist) [options]}
\]

Convert data back to long form after using \text{reshape wide}

\[
\text{reshape long}
\]

Convert data back to wide form after using \text{reshape long}

\[
\text{reshape wide}
\]

List problem observations when \text{reshape} fails

\[
\text{reshape error}
\]
### reshape — Convert data from wide to long form and vice versa

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>i(varlist)</em></td>
<td>use varlist as the ID variables</td>
</tr>
</tbody>
</table>
| j(varname [values]) | long→wide: varname, existing variable  
wide→long: varname, new variable  
optionally specify values to subset *varname* |
| **string** | *varname* is a string variable (default is numeric) |

* i(varlist) is required.

where values is

- #[-#][... ] if *varname* is numeric (default)
- "string" ["string" ...] if *varname* is string

and where *stubnames* are variable names (long→wide), or stubs of variable names (wide→long), and

either way, may contain @, denoting where *j* appears or is to appear in the name.

In the example above, when we wrote “reshape wide stub”, we could have written “reshape wide stub@” because *j* by default ends up as a suffix. Had we written *stu@b*, then the wide variables would have been named *stu1b* and *stu2b*.

**Advanced syntax**

- reshape i varlist
- reshape j varname [values] [, string]
- reshape xij fvarnames [, atwl(chars)]
- reshape xi [varlist]
- reshape [query]
- reshape clear

**Options**

- i(varlist) specifies the variables whose unique values denote a logical observation. i() is required.
- j(varname [values]) specifies the variable whose unique values denote a subobservation. values lists the unique values to be used from *varname*, which typically are not explicitly stated because *reshape* will determine them automatically from the data.
- **string** specifies that j() may contain string values.
- atwl(chars), available only with the advanced syntax and not shown in the dialog box, specifies that plain ASCII chars be substituted for the @ character when converting the data from wide to long form.
Remarks and examples

Remarks are presented under the following headings:

- Description of basic syntax
- Wide and long data forms
- Avoiding and correcting mistakes
- reshape long and reshape wide without arguments
- Missing variables
- Advanced issues with basic syntax: i()
- Advanced issues with basic syntax: j()
- Advanced issues with basic syntax: xij
- Advanced issues with basic syntax: String identifiers for j()
- Advanced issues with basic syntax: Second-level nesting
- Description of advanced syntax
- Video examples

See Mitchell (2020, chap. 9) for information and examples using reshape.

Description of basic syntax

Before using reshape, you need to determine whether the data are in long or wide form. You also must determine the logical observation (i) and the subobservation (j) by which to organize the data. Suppose that you had the following data, which could be organized in wide or long form as follows:

<table>
<thead>
<tr>
<th>i</th>
<th>X_{ij}</th>
<th>i</th>
<th>X_{ij}</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>sex</td>
<td>id</td>
<td>sex</td>
</tr>
<tr>
<td></td>
<td>inc80</td>
<td></td>
<td>inc80</td>
</tr>
<tr>
<td></td>
<td>inc81</td>
<td></td>
<td>inc81</td>
</tr>
<tr>
<td></td>
<td>inc82</td>
<td></td>
<td>inc82</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>0</td>
<td>5000</td>
<td>0</td>
<td>5000</td>
</tr>
<tr>
<td>2000</td>
<td>2200</td>
<td>3300</td>
<td>3300</td>
</tr>
<tr>
<td>3000</td>
<td>2000</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Given these data, you could use reshape to convert from one form to the other:

```plaintext
. reshape long inc, i(id) j(year) /* goes from left form to right */
. reshape wide inc, i(id) j(year) /* goes from right form to left */
```

Because we did not specify sex in the command, Stata assumes that it is constant within the logical observation, here id.

Wide and long data forms

Think of the data as a collection of observations $X_{ij}$, where $i$ is the logical observation, or group identifier, and $j$ is the subobservation, or within-group identifier.

Wide-form data are organized by logical observation, storing all the data on a particular observation in one row. Long-form data are organized by subobservation, storing the data in multiple rows.
Example 1

For example, we might have data on a person’s ID, gender, and annual income over the years 1980–1982. We have two $X_{ij}$ variables with the data in wide form:

```
. use https://www.stata-press.com/data/r17/reshape1
. list

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
<th>inc80</th>
<th>inc81</th>
<th>inc82</th>
<th>ue80</th>
<th>ue81</th>
<th>ue82</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5000</td>
<td>5500</td>
<td>6000</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2000</td>
<td>2200</td>
<td>3300</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3000</td>
<td>2000</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

To convert these data to the long form, we type

```
. reshape long inc ue, i(id) j(year)
   (j = 80 81 82)
```

<table>
<thead>
<tr>
<th>Data</th>
<th>Wide</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>3 -&gt; 9</td>
<td></td>
</tr>
<tr>
<td>Number of variables</td>
<td>8 -&gt; 5</td>
<td></td>
</tr>
<tr>
<td>j variable (3 values)</td>
<td>-&gt; year</td>
<td></td>
</tr>
<tr>
<td>xij variables:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inc80 inc81 inc82 -&gt; inc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ue80 ue81 ue82 -&gt; ue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There is no variable named `year` in our original, wide-form dataset. `year` will be a new variable in our long dataset. After this conversion, we have

```
. list, sep(3)

<table>
<thead>
<tr>
<th>id</th>
<th>year</th>
<th>sex</th>
<th>inc</th>
<th>ue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>0</td>
<td>5000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>81</td>
<td>0</td>
<td>5500</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>82</td>
<td>0</td>
<td>6000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>80</td>
<td>1</td>
<td>2000</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
<td>1</td>
<td>2200</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>82</td>
<td>1</td>
<td>3300</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td>0</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>81</td>
<td>0</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>82</td>
<td>0</td>
<td>1000</td>
<td>1</td>
</tr>
</tbody>
</table>
We can return to our original, wide-form dataset by using \texttt{reshape wide}.

```
. reshape wide inc ue, i(id) j(year)
  (j = 80 81 82)
```

<table>
<thead>
<tr>
<th>Data</th>
<th>Long</th>
<th>Wide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Number of variables</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>j variable (3 values)</td>
<td>year -&gt; (dropped)</td>
<td></td>
</tr>
<tr>
<td>xij variables:</td>
<td>inc -&gt; inc80 inc81 inc82</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ue -&gt; ue80 ue81 ue82</td>
<td></td>
</tr>
</tbody>
</table>

```
. list
   id   inc80   ue80   inc81   ue81   inc82   ue82   sex
  1.  1   5000   0   5500    1   6000   0   0
  2.  2   2000   1   2200    0   3300   0   1
  3.  3   3000   0   2000    0   1000   1   0
```

Converting from wide to long creates the \texttt{j (year)} variable. Converting back from long to wide drops the \texttt{j (year)} variable.

\section*{Technical note}

If your data are in wide form and you do not have a group identifier variable (the \texttt{i(varlist)} required option), you can create one easily by using \texttt{generate}; see \texttt{[D] generate}. For instance, in the last example, if we did not have the \texttt{id} variable in our dataset, we could have created it by typing

```
. generate id = _n
```

\section*{Avoiding and correcting mistakes}

\texttt{reshape} often detects when the data are not suitable for reshaping; an error is issued, and the data remain unchanged.

\section*{Example 2}

The following wide data contain a mistake:

```
. use https://www.stata-press.com/data/r17/reshape2, clear
. list
   id   sex   inc80   inc81   inc82
  1.   1    0   5000    5500    6000
  2.   2    1   2000    2200    3300
  3.   3    0   3000    2000    1000
  4.   2    0   2400    2500    2400
```
Your data are currently wide. You are performing a reshape long. You specified i(id) and j(year). In the current wide form, variable id should uniquely identify the observations. Remember this picture:

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Type reshape error for a list of the problem observations.

r(9);

The i variable must be unique when the data are in the wide form; we typed i(id), yet we have 2 observations for which id is 2. (Is person 2 a male or female?)

Example 3

It is not a mistake when the i variable is repeated when the data are in long form, but the following data have a similar mistake:

. use https://www.stata-press.com/data/r17/reshapexp1
. list

<table>
<thead>
<tr>
<th>id</th>
<th>year</th>
<th>sex</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>0</td>
<td>5000</td>
</tr>
<tr>
<td>1</td>
<td>81</td>
<td>0</td>
<td>5500</td>
</tr>
<tr>
<td>3</td>
<td>81</td>
<td>0</td>
<td>5400</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
<td>0</td>
<td>6000</td>
</tr>
</tbody>
</table>

. reshape wide inc, i(id) j(year)
(note: j = 80 81 82)

Values of variable year not unique within id
Your data are currently long. You are performing a reshape wide. You specified i(id) and j(year). There are observations within i(id) with the same value of j(year). In the long data, variables i() and j() together must uniquely identify the observations.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>a</th>
<th>b</th>
<th>i</th>
<th>a1</th>
<th>a2</th>
<th>b1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Type reshape error for a list of the problem variables.

r(9);

In the long form, i(id) does not have to be unique, but j(year) must be unique within i; otherwise, what is the value of inc in 1981 for which id==1?
reshape told us to type `reshape error` to view the problem observations.

```
. reshape error
  (j = 80 81 82)
```

- `i (id)` indicates the top-level grouping such as subject id.
- `j (year)` indicates the sub-grouping such as time.

The data are in the long form; `j` should be unique within `i`

There are multiple observations on the same year within `id`.

The following 2 of 4 observations have repeated year values:

```
<table>
<thead>
<tr>
<th>id</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>81</td>
</tr>
<tr>
<td>3.</td>
<td>81</td>
</tr>
</tbody>
</table>
```

(data now sorted by `id year`)

### Example 4

Consider some long-form data that have no mistakes. We list the first 4 observations.

```
. use https://www.stata-press.com/data/r17/reshape6
. list in 1/4
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>year</th>
<th>sex</th>
<th>inc</th>
<th>ue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>80</td>
<td>0</td>
<td>5000</td>
<td>0</td>
</tr>
<tr>
<td>2.</td>
<td>81</td>
<td>0</td>
<td>5500</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>82</td>
<td>0</td>
<td>6000</td>
<td>0</td>
</tr>
<tr>
<td>4.</td>
<td>80</td>
<td>1</td>
<td>2000</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Say that when converting the data to wide form, however, we forget to mention the `ue` variable (which varies within person).

```
. reshape wide inc, i(id) j(year)
  (j = 80 81 82)
```

Variable `ue` not constant within `id`

Your data are currently long. You are performing a `reshape wide`. You typed something like

```
. reshape wide a b, i(id) j(year)
```

There are variables other than `a`, `b`, `id`, `year` in your data. They must be constant within `id` because that is the only way they can fit into wide data without loss of information.

The variable or variables listed above are not constant within `id`. Perhaps the values are in error. Type `reshape error` for a list of the problem observations.

Either that, or the values vary because they should vary, in which case you must either add the variables to the list of `xij` variables to be reshaped, or drop them.

```
r(9);
```

Here reshape observed that `ue` was not constant within `id` and so could not restructure the data so that there were single observations on `id`. We should have typed

```
. reshape wide inc ue, i(id) j(year)
```
In summary, there are three cases in which `reshape` will refuse to convert the data:

1. The data are in wide form and \( i \) is not unique.
2. The data are in long form and \( j \) is not unique within \( i \).
3. The data are in long form and an unmentioned variable is not constant within \( i \).

### Example 5

With some mistakes, `reshape` will probably convert the data and produce a surprising result. Suppose that we forget to mention that the `ue` variable varies within `id` in the following wide data:

```
. use https://www.stata-press.com/data/r17/reshape1
. list
    +-----+--------+--------+-------+-------+-------+-------+-------+-------+
   | id  | sex    | inc80  | inc81  | inc82  | ue80  | ue81  | ue82  |
   +-----+--------+--------+-------+-------+-------+-------+-------+
   | 1.   | 0      | 5000   | 5500   | 6000   | 0     | 1     | 0     |
   | 2.   | 1      | 2000   | 2200   | 3300   | 1     | 0     | 0     |
   | 3.   | 0      | 3000   | 2000   | 1000   | 0     | 0     | 1     |
   +-----+--------+--------+-------+-------+-------+-------+-------+

. reshape long inc, i(id) j(year)
(j = 80 81 82)
```

```
Data Wide -> Long
Number of observations 3 -> 9
Number of variables 8 -> 7
j variable (3 values) -> year
xij variables:
      inc80 inc81 inc82 -> inc
```

```
. list, sep(3)
    +-----+---+--------+-------+-------+-------+-------+-------+-------+
   | id  | y  | sex    | inc   | ue80  | ue81  | ue82  |
   +-----+---+--------+-------+-------+-------+-------+
   | 1.   | 80 | 0      | 5000  | 0     | 1     | 0     |
   | 2.   | 81 | 0      | 5500  | 0     | 1     | 0     |
   | 3.   | 82 | 0      | 6000  | 0     | 1     | 0     |
   | 4.   | 80 | 1      | 2000  | 1     | 0     | 0     |
   | 5.   | 81 | 1      | 2200  | 1     | 0     | 0     |
   | 6.   | 82 | 1      | 3300  | 1     | 0     | 0     |
   | 7.   | 80 | 0      | 3000  | 0     | 0     | 1     |
   | 8.   | 81 | 0      | 2000  | 0     | 0     | 1     |
   | 9.   | 82 | 0      | 1000  | 0     | 0     | 1     |
   +-----+---+--------+-------+-------+-------+-------+
```

We did not state that `ue` varied within `i`, so the variables `ue80`, `ue81`, and `ue82` were left as is. `reshape` did not complain. There is no real problem here because no information has been lost. In fact, this may actually be the result we wanted. Probably, however, we simply forgot to include `ue` among the \( X_{ij} \) variables.

If you obtain an unexpected result, here is how to undo it:

1. If you typed `reshape long ...` to produce the result, type `reshape wide` (without arguments) to undo it.
2. If you typed `reshape wide ...` to produce the result, type `reshape long` (without arguments) to undo it.
So, we can type
   . reshape wide

to get back to our original, wide-form data and then type the `reshape long' command that we intended:
   . reshape long inc ue, i(id) j(year)

**reshape long and reshape wide without arguments**

Whenever you type a `reshape long' or `reshape wide' command with arguments, `reshape' remembers it. Thus you might type
   . reshape long inc ue, i(id) j(year)

and work with the data like that. You could then type
   . reshape wide

to convert the data back to the wide form. Then later you could type
   . reshape long

to convert them back to the long form. If you save the data, you can even continue using `reshape wide' and `reshape long' without arguments during a future Stata session.

Be careful. If you create new $X_{ij}$ variables, you must tell `reshape' about them by typing the full `reshape' command, although no real damage will be done if you forget. If you are converting from long to wide form, `reshape' will catch your error and refuse to make the conversion. If you are converting from wide to long, `reshape' will convert the data, but the result will be surprising: remember what happened when we forgot to mention the `ue' variable and ended up with `ue80', `ue81', and `ue82' in our long data; see example 5. You can `reshape long' to undo the unwanted change and then try again.

**Missing variables**

When converting data from wide form to long form, `reshape' does not demand that all the variables exist. Missing variables are treated as variables with missing observations.

▶ **Example 6**

Let’s drop `ue81' from the wide form of the data:

   . use `https://www.stata-press.com/data/r17/reshape1', clear
   . drop ue81
   . list

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
<th>inc80</th>
<th>inc81</th>
<th>inc82</th>
<th>ue80</th>
<th>ue82</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5000</td>
<td>5500</td>
<td>6000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2000</td>
<td>2200</td>
<td>3300</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3000</td>
<td>2000</td>
<td>1000</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
. reshape long inc ue, i(id) j(year)  
( j = 80 81 82)  
( variable ue81 not found)  

Data | Wide | Long
---|---|---
Number of observations | 3 | 9
Number of variables | 7 | 5
j variable (3 values) | - | year
xij variables: | inc80 inc81 inc82 | inc
ue80 ue81 ue82 | ue |  

. list, sep(3)

<table>
<thead>
<tr>
<th>id</th>
<th>year</th>
<th>sex</th>
<th>inc</th>
<th>ue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80</td>
<td>0</td>
<td>5000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
<td>0</td>
<td>5500</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>82</td>
<td>0</td>
<td>6000</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>1</td>
<td>2000</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>81</td>
<td>1</td>
<td>2200</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>82</td>
<td>1</td>
<td>3300</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>80</td>
<td>0</td>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>81</td>
<td>0</td>
<td>2000</td>
<td>.</td>
</tr>
<tr>
<td>9</td>
<td>82</td>
<td>0</td>
<td>1000</td>
<td>1</td>
</tr>
</tbody>
</table>

reshape placed missing values where ue81 values were unavailable. If we reshaped these data back to wide form by typing

. reshape wide inc ue, i(id) j(year)

the ue81 variable would be created and would contain all missing values.

Advanced issues with basic syntax: i()

The i() option can indicate one i variable (as our past examples have illustrated) or multiple variables. An example of multiple i variables would be hospital ID and patient ID within each hospital.

. reshape ... , i(hid pid)

Unique pairs of values for hid and pid in the data define the grouping variable for reshape.

Advanced issues with basic syntax: j()

The j() option takes a variable name (as our past examples have illustrated) or a variable name and a list of values. When the values are not provided, reshape deduces them from the data. Specifying the values with the j() option is rarely needed.

reshape never makes a mistake when the data are in long form and you type reshape wide. The values are easily obtained by tabulating the j variable.
`reshape` can make a mistake when the data are in wide form and you type `reshape long` if your variables are poorly named. Say that you have the `inc80`, `inc81`, and `inc82` variables, recording income in each of the indicated years, and you have a variable named `inc2`, which is not income but indicates when the area was reincorporated. You type

`. reshape long inc, i(id) j(year)`

`reshape` sees the `inc2`, `inc80`, `inc81`, and `inc82` variables and decides that there are four groups in which \( j = 2, 80, 81, \) and \( 82 \).

The easiest way to solve the problem is to rename the `inc2` variable to something other than “inc” followed by a number; see [D] `rename`.

You can also keep the name and specify the \( j \) values. To perform the reshape, you can type

`. reshape long inc, i(id) j(year 80-82)`

or

`. reshape long inc, i(id) j(year 80 81 82)`

You can mix the dash notation for value ranges with individual numbers. `reshape` would understand `80 82-87 89 91-95` as a valid values specification.

At the other extreme, you can omit the \( j() \) option altogether with `reshape long`. If you do, the \( j \) variable will be named \(_j\).

**Advanced issues with basic syntax: xij**

When specifying variable names, you may include @ characters to indicate where the numbers go.

> **Example 7**

Let’s reshape the following data from wide to long form:

```
. use https://www.stata-press.com/data/r17/reshape3, clear
. list
    +---+-----+-------+-------+-------+-----+-----+-----+
<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
<th>inc80r</th>
<th>inc81r</th>
<th>inc82r</th>
<th>ue80</th>
<th>ue81</th>
<th>ue82</th>
</tr>
</thead>
</table>
  1.  | 1   | 0    | 5000  | 5500  | 6000 | 0   | 1   | 0   |
  2.  | 2   | 1    | 2000  | 2200  | 3300 | 1   | 0   | 0   |
  3.  | 3   | 0    | 3000  | 2000  | 1000 | 0   | 0   | 1   |
```

`. reshape long inc@r ue, i(id) j(year)
  (j = 80 81 82)`

**Data**

<table>
<thead>
<tr>
<th>Wide</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>3 → 9</td>
</tr>
<tr>
<td>Number of variables</td>
<td>8 → 5</td>
</tr>
<tr>
<td>( j ) variable (3 values)</td>
<td>→ year</td>
</tr>
<tr>
<td>xij variables:</td>
<td>inc80r, inc81r, inc82r → incr</td>
</tr>
<tr>
<td></td>
<td>ue80, ue81, ue82 → ue</td>
</tr>
</tbody>
</table>
reshape — Convert data from wide to long form and vice versa

At most one @ character may appear in each name. If no @ character appears, results are as if the @ character appeared at the end of the name. So, the equivalent reshape command to the one above is

```
. reshape long incr ue, i(id) j(year)
```

inc@r specifies variables named inc#r in the wide form and incr in the long form. The @ notation may similarly be used for converting data from long to wide format:

```
. reshape wide inc@r ue, i(id) j(year)
```

## Advanced issues with basic syntax: String identifiers for j()

The string option allows j to take on string values.

### Example 8

Consider the following wide data on husbands and wives. In these data, incm is the income of the man and incf is the income of the woman.

```
. use https://www.stata-press.com/data/r17/reshape4, clear
. list
```

<table>
<thead>
<tr>
<th>id</th>
<th>kids</th>
<th>incm</th>
<th>incf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5000</td>
<td>5500</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2000</td>
<td>2200</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3000</td>
<td>2000</td>
</tr>
</tbody>
</table>

These data can be reshaped into separate observations for males and females by typing

```
. reshape long inc, i(id) j(sex) string (j = f m)
```

<table>
<thead>
<tr>
<th>Data</th>
<th>Wide -&gt;</th>
<th>Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>3 -&gt; 6</td>
<td></td>
</tr>
<tr>
<td>Number of variables</td>
<td>4 -&gt; 4</td>
<td></td>
</tr>
<tr>
<td>j variable (2 values)</td>
<td>-&gt; sex</td>
<td></td>
</tr>
<tr>
<td>xij variables:</td>
<td>incf incm -&gt; inc</td>
<td></td>
</tr>
</tbody>
</table>
The string option specifies that \( j \) take on nonnumeric values. The result is

\[
\text{. list, sep(2)}
\]

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
<th>kids</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f</td>
<td>0</td>
<td>5500</td>
</tr>
<tr>
<td>1</td>
<td>m</td>
<td>0</td>
<td>5000</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>1</td>
<td>2200</td>
</tr>
<tr>
<td>2</td>
<td>m</td>
<td>1</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>m</td>
<td>2</td>
<td>3000</td>
</tr>
</tbody>
</table>

sex will be a string variable. Similarly, these data can be converted from long to wide form by typing

\[
\text{. reshape wide inc, i(id) j(sex) string}
\]

Strings are not limited to being single characters or even having the same length. You can specify the location of the string identifier in the variable name by using the @ notation.

\[\text{Example 9}\]

Suppose that our variables are named \texttt{id}, \texttt{kids}, \texttt{incmale}, and \texttt{incfem}.

\[
\text{. use https://www.stata-press.com/data/r17/reshapexp2, clear}
\]

\[
\text{. list}
\]

<table>
<thead>
<tr>
<th>id</th>
<th>kids</th>
<th>incmale</th>
<th>incfem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5000</td>
<td>5500</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2000</td>
<td>2200</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3000</td>
<td>2000</td>
</tr>
</tbody>
</table>

\[
\text{. reshape long inc, i(id) j(sex) string}
\]

(j = fem male)

\[]

Data          Wide  ->  Long
Number of observations  \(3 \rightarrow 6\)
Number of variables     \(4 \rightarrow 4\)
j variable (2 values)  \(\rightarrow\) sex
xij variables:
\[
\text{incfem incmale}  \rightarrow  \text{inc}
\]

\[
\text{. list, sep(2)}
\]

<table>
<thead>
<tr>
<th>id</th>
<th>sex</th>
<th>kids</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fem</td>
<td>0</td>
<td>5500</td>
</tr>
<tr>
<td>1</td>
<td>male</td>
<td>0</td>
<td>5000</td>
</tr>
<tr>
<td>2</td>
<td>fem</td>
<td>1</td>
<td>2200</td>
</tr>
<tr>
<td>2</td>
<td>male</td>
<td>1</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>fem</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>3</td>
<td>male</td>
<td>2</td>
<td>3000</td>
</tr>
</tbody>
</table>
If the wide data had variables named \texttt{minc} and \texttt{finc}, the appropriate \texttt{reshape} command would have been

```
. reshape long @inc, i(id) j(sex) string
```

The resulting variable in the long form would be named \texttt{inc}.

We can also place strings in the middle of the variable names. If the variables were named \texttt{incMome} and \texttt{incFome}, the \texttt{reshape} command would be

```
. reshape long inc@ome, i(id) j(sex) string
```

Be careful with string identifiers because it is easy to be surprised by the result. Say that we have wide data having variables named \texttt{incm}, \texttt{incf}, \texttt{uem}, \texttt{uef}, \texttt{agem}, and \texttt{agef}. To make the data long, we might type

```
. reshape long inc ue age, i(id) j(sex) string
```

Along with these variables, we also have the variable \texttt{agenda}. \texttt{reshape} will decide that the sexes are \texttt{m}, \texttt{f}, and \texttt{nda}. This would not happen without the \texttt{string} option if the variables were named \texttt{inc0}, \texttt{inc1}, \texttt{ue0}, \texttt{ue1}, \texttt{age0}, and \texttt{age1}, even with the \texttt{agenda} variable present in the data.

---

### Advanced issues with basic syntax: Second-level nesting

Sometimes the data may have more than one possible \texttt{j} variable for reshaping. Suppose that your data have both a year variable and a sex variable. One logical observation in the data might be represented in any of the following four forms:

```
. list in 1/4 // The long-long form

<table>
<thead>
<tr>
<th>hid</th>
<th>sex</th>
<th>year</th>
<th>inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f</td>
<td>90</td>
<td>3200</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>91</td>
<td>4700</td>
</tr>
<tr>
<td>3</td>
<td>m</td>
<td>90</td>
<td>4500</td>
</tr>
<tr>
<td>4</td>
<td>m</td>
<td>91</td>
<td>4600</td>
</tr>
</tbody>
</table>

. list in 1/2 // The long-year wide-sex form

<table>
<thead>
<tr>
<th>hid</th>
<th>year</th>
<th>minc</th>
<th>finc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90</td>
<td>4500</td>
<td>3200</td>
</tr>
<tr>
<td>2</td>
<td>91</td>
<td>4600</td>
<td>4700</td>
</tr>
</tbody>
</table>

. list in 1/2 // The wide-year long-sex form

<table>
<thead>
<tr>
<th>hid</th>
<th>sex</th>
<th>inc90</th>
<th>inc91</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f</td>
<td>3200</td>
<td>4700</td>
</tr>
<tr>
<td>2</td>
<td>m</td>
<td>4500</td>
<td>4600</td>
</tr>
</tbody>
</table>

. list in 1 // The wide-wide form

<table>
<thead>
<tr>
<th>hid</th>
<th>minc90</th>
<th>minc91</th>
<th>finc90</th>
<th>finc91</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4500</td>
<td>4600</td>
<td>3200</td>
<td>4700</td>
</tr>
</tbody>
</table>
**reshape** can convert any of these forms to any other. Converting data from the long–long form to the wide–wide form (or any of the other forms) takes two **reshape** commands. Here is how we would do it:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>sex</td>
<td>year sex</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>reshape wide inc, i(hid year) j(sex) string</td>
</tr>
<tr>
<td>long</td>
<td>wide</td>
<td>reshape long inc, i(hid year) j(sex) string</td>
</tr>
<tr>
<td>wide</td>
<td>long</td>
<td>reshape long inc, i(hid sex) j(year)</td>
</tr>
<tr>
<td>wide</td>
<td>wide</td>
<td>reshape long inc, i(hid sex) j(year)</td>
</tr>
</tbody>
</table>

**Description of advanced syntax**

The advanced syntax is simply a different way of specifying the **reshape** command, and it has one seldom-used feature that provides extra control. Rather than typing one **reshape** command to describe the data and perform the conversion, such as

```
. reshape long inc, i(id) j(year)
```

you type a sequence of **reshape** commands. The initial commands describe the data, and the last command performs the conversion:

```
. reshape i id
. reshape j year
. reshape xij inc
. reshape long
```

**reshape** i corresponds to i() in the basic syntax.

**reshape** j corresponds to j() in the basic syntax.

**reshape** xij corresponds to the variables specified in the basic syntax. **reshape** xij also accepts the **atwl**() option for use when @ characters are specified in the **fvarnames**. **atwl** stands for at-when-long. When you specify names such as inc@r or ue@, in the long form the names become incr and ue, and the @ character is ignored. **atwl**() allows you to change @ into whatever you specify. For example, if you specify **atwl**(X), the long-form names become incXr and ueX.

There is also one more specification, which has no counterpart in the basic syntax:

```
. reshape xi varlist
```

In the basic syntax, Stata assumes that all unspecified variables are constant within i. The advanced syntax works the same way, unless you specify the **reshape** xi command, which names the constant-within-i variables. If you specify **reshape** xi, any variables that you do not explicitly specify are dropped from the data during the conversion.

As a practical matter, you should explicitly drop the unwanted variables before conversion. For instance, suppose that the data have variables inc80, inc81, inc82, sex, age, and age2 and that you no longer want the age2 variable. You could specify

```
. reshape xi sex age
```

or

```
. drop age2
```

and leave **reshape** xi unspecified.
reshape xi does have one minor advantage. It saves reshape the work of determining which variables are unspecified. This saves a relatively small amount of computer time.

Another advanced-syntex feature is reshape query, which is equivalent to typing reshape by itself. reshape query reports which reshape parameters have been defined. reshape i, reshape j, reshape xij, and reshape xi specifications may be given in any order and may be repeated to change or correct what has been specified.

Finally, reshape clear clears the definitions. reshape definitions are stored with the dataset when you save it. reshape clear allows you to erase these definitions.

The basic syntax of reshape is implemented in terms of the advanced syntax, so you can mix basic and advanced syntaxes.

Video examples

How to reshape data from long format to wide format
How to reshape data from wide format to long format

Stored results

 reshape stores the following characteristics with the data (see [P] char):

- _dta[Res_S_i] i variable names
- _dta[Res_S_j] j variable name
- _dta[Res_S_jv] j values, if specified
- _dta[Res_S_Xij] Xij variable names
- _dta[Res_S_Xij_n] number of Xij variables
- _dta[Res_S_Xij_long#] name of #th Xij variable in long form
- _dta[Res_S_Xij_wide#] name of #th Xij variable in wide form
- _dta[Res_S_Xi] Xi variable names, if specified
- _dta[Res_atwl] atwl() value, if specified
- _dta[Res_str] 1 if option string specified, 0 otherwise

Acknowledgment

This version of reshape was based in part on the work of Jeroen Weesie (1997) of the Department of Sociology at Utrecht University, The Netherlands.

References

Also see

[D] save — Save Stata dataset
[D] stack — Stack data
[D] xpose — Interchange observations and variables
[P] char — Characteristics
**rmdir — Remove directory**

Description

`rmdir` removes an empty directory (folder).

Quick start

Remove empty `myfolder` from the current working directory

```bash
rmdir myfolder
```

Remove `myfolder` from `C:\mydir` using Stata for Windows

```bash
rmdir c:\mydir\myfolder
```

Remove `myfolder` from `/mydir` using Stata for Mac or Unix

```bash
rmdir ~/mydir/myfolder
```

Remove my folder from `C:\my dir` using Stata for Windows

```bash
rmdir "c:\my dir\my folder"
```

Syntax

`rmdir directory_name`

Double quotes may be used to enclose the directory name, and the quotes must be used if the directory name contains embedded blanks.

Remarks and examples

Examples:

Windows

```bash
. rmdir myproj
. rmdir c:\projects\myproj
. rmdir "c:\My Projects\Project 1"
```

Mac and Unix

```bash
. rmdir myproj
. rmdir ~/projects/myproj
```
Also see

[D] cd — Change directory
[D] copy — Copy file from disk or URL
[D] dir — Display filenames
[D] erase — Erase a disk file
[D] mkdir — Create directory
[D] shell — Temporarily invoke operating system
[D] type — Display contents of a file
[U] 11.6 Filenaming conventions
**Description**

`sample` draws random samples of the data in memory. “Sampling” here is defined as drawing observations without replacement; see [R] `bsample` for sampling with replacement.

The size of the sample to be drawn can be specified as a percentage or as a count:

- `sample` without the `count` option draws a `#`% pseudorandom sample of the data in memory, thus discarding `(100 - #)`% of the observations.

- `sample` with the `count` option draws a `-#`-observation pseudorandom sample of the data in memory, thus discarding `-N - #` observations. `#` can be larger than `-N`, in which case all observations are kept.

In either case, observations not meeting the optional `if` and `in` criteria are kept (sampled at 100%).

If you are interested in reproducing results, you must first set the random-number seed; see [R] `set seed`.

**Quick start**

Draw 10% pseudorandom sample without replacement from data in memory

```
sample 10
```

As above, but perform sampling within strata identified by `svar`

```
sample 10, by(svar)
```

Sample 100 observations from data in memory

```
sample 100, count
```

As above, but only sample observations where `catvar` equals 5

```
sample 100 if catvar==5, count
```

**Menu**

Statistics > Resampling > Draw random sample
sample # [if] [in] [ , count by(groupvars) ]

by is allowed; see [D] by.

Options
count specifies that # in sample # be interpreted as an observation count rather than as a percentage.
Typing sample 5 without the count option means that a 5% sample be drawn: typing sample 5, count, however, would draw a sample of 5 observations.
Specifying # as greater than the number of observations in the dataset is not considered an error.
by(groupvars) specifies that a #% sample be drawn within each set of values of groupvars, thus maintaining the proportion of each group.
count may be combined with by(). For example, typing sample 50, count by(sex) would draw a sample of size 50 for men and 50 for women.
Specifying by varlist: sample # is equivalent to specifying sample #, by(varlist); use whichever syntax you prefer.

Remarks and examples

Example 1

We have NLSY data on young women aged 14–24 years in 1968 and wish to draw a 10% sample of the data in memory.

```
. use https://www.stata-press.com/data/r17/nlswork
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. describe, short
Contains data from https://www.stata-press.com/data/r17/nlswork.dta
Observations: 28,534 National Longitudinal Survey of Young Women, 14-24 years old in 1968
Variables: 21 27 Nov 2020 08:14
Sorted by: idcode year
. sample 10
(25,681 observations deleted)
. describe, short
Contains data from https://www.stata-press.com/data/r17/nlswork.dta
Observations: 2,853 National Longitudinal Survey of Young Women, 14-24 years old in 1968
Variables: 21 27 Nov 2020 08:14
Sorted by:
Note: Dataset has changed since last saved.
```

Our original dataset had 28,534 observations. The sample-10 dataset has 2,853 observations, which is the nearest number to 0.10 \times 28534.
Example 2

Among the variables in our data is `race`. By typing `label list`, we see that `race = 1` denotes whites, `race = 2` denotes blacks, and `race = 3` denotes other races. We want to keep 100% of the nonwhite women but only 10% of the white women.

```
. use https://www.stata-press.com/data/r17/nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. tab race
```

```
<table>
<thead>
<tr>
<th>Race</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>20,180</td>
<td>70.72</td>
<td>70.72</td>
</tr>
<tr>
<td>Black</td>
<td>8,051</td>
<td>28.22</td>
<td>98.94</td>
</tr>
<tr>
<td>Other</td>
<td>303</td>
<td>1.06</td>
<td>100.00</td>
</tr>
</tbody>
</table>
```

```
. sample 10 if race == 1
(18,162 observations deleted)
. describe, short
Contains data from https://www.stata-press.com/data/r17/nlswork.dta
Observations: 10,372 National Longitudinal Survey of Young Women, 14-24 years old in 1968
Variables: 21 27 Nov 2020 08:14
Sorted by:
Note: Dataset has changed since last saved.
. display .10*20180 + 8051 + 303
10372
```

Example 3

Now let's suppose that we want to keep 10% of each of the three categories of `race`.

```
. use https://www.stata-press.com/data/r17/nlswork, clear
(National Longitudinal Survey of Young Women, 14-24 years old in 1968)
. sample 10, by(race)
(25,681 observations deleted)
. tab race
```

```
<table>
<thead>
<tr>
<th>Race</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>2,018</td>
<td>70.73</td>
<td>70.73</td>
</tr>
<tr>
<td>Black</td>
<td>805</td>
<td>28.22</td>
<td>98.95</td>
</tr>
<tr>
<td>Other</td>
<td>30</td>
<td>1.05</td>
<td>100.00</td>
</tr>
</tbody>
</table>
```

This differs from simply typing `sample 10` in that with `by()`, `sample` holds constant the percentages of white, black, and other women.

Technical note

We have a large dataset on disk containing 125,235 observations. We wish to draw a 10% sample of this dataset without loading the entire dataset (perhaps because the dataset will not fit in memory). `sample` will not solve this problem—the dataset must be loaded first—but it is rather easy to solve it ourselves. Say that `bigdata.dct` contains the dictionary for this dataset; see `import`. One solution is to type
The `if` qualifier on the end of `infile` drew uniformly distributed random numbers over the interval 0 and 1 and kept each observation if the random number was less than or equal to 0.1. This, however, did not draw an exact 10% sample—the sample was expected to contain only 10% of the observations, and here we obtained just more than 10%. This is probably a reasonable solution.

If the sample must contain precisely 12,524 observations, however, after getting too many observations, we could type

```stata
    . generate u=runiform()
    . sort u
    . keep in 1/12524
```

That is, we put the resulting sample in random order and keep the first 12,524 observations. Now our only problem is making sure that, at the first step, we have more than 12,524 observations. Here we were lucky, but half the time we will not be so lucky—after typing `infile ... if runiform()<=.1`, we will have less than a 10% sample. The solution, of course, is to draw more than a 10% sample initially and then cut it back to 10%.

How much more than 10% do we need? That depends on the number of records in the original dataset, which in our example is 125,235.

A little experimentation with `bitesti` (see [R] `bitest`) provides the answer:

```stata
    . bitesti 125235 12524 .102

<table>
<thead>
<tr>
<th>N</th>
<th>Observed k</th>
<th>Expected k</th>
<th>Assumed p</th>
<th>Observed p</th>
</tr>
</thead>
<tbody>
<tr>
<td>125,235</td>
<td>12,524</td>
<td>12,773.97</td>
<td>0.10200</td>
<td>0.10000</td>
</tr>
</tbody>
</table>
```

Initially drawing a 10.2% sample will yield a sample larger than 10% 99 times of 100. If we draw a 10.4% sample, we are virtually assured of having enough observations (type `bitesti 125235 12524 .104` for yourself).
sample — Draw random sample

References


Also see

[D] splitsample — Split data into random samples

[R] bsample — Sampling with replacement
Description

save stores the dataset currently in memory on disk under the name filename. If filename is not specified, the name under which the data were last known to Stata (c(filename)) is used. If filename is specified without an extension, .dta is used. If your filename contains embedded spaces, remember to enclose it in double quotes.

Stata 14 through 17 have the same dataset format so long as the dataset has 32,767 variables or less. Since Stata/MP 15, Stata/MP has supported more than 32,767 variables and thus has a slightly different dataset format when there are that many variables. If you are using Stata 17 and want to save a dataset so that it may be read by someone using Stata 16 or Stata 15, simply use the save command; those older versions will be able to read it. If the dataset has more than 32,767 variables, it can be read by Stata/MP 16 and Stata/MP 15. If you want to save a dataset so that it may be read by someone using Stata 14, again simply use the save command; Stata 14 will be able to read it so long as it does not have more than 32,767 variables. Stata 14 supports at most 32,767 variables.

saveold saves the dataset currently in memory on disk under the name filename in previous .dta formats, namely, those for Stata 13, 12, or 11. If you are using Stata 17 and want to save a file so that it may be read by someone using an older version of Stata, use the saveold command.

Quick start

Save data in memory to mydata.dta in the current directory
save mydata

As above, but overwrite mydata.dta if it exists
save mydata, replace

Also save value labels that have not been applied to variables
save mydata, replace orphans

Save data in Stata 13 format
saveold mydata

Menu

File > Save as...
Syntax

Save data in memory to file

```
save [ filename ] [ , save_options ]
```

Save data in memory to file in Stata 13, 12, or 11 format

```
saveold filename [ , saveold_options ]
```

<table>
<thead>
<tr>
<th>save_options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nolabel</td>
<td>omit value labels from the saved dataset</td>
</tr>
<tr>
<td>replace</td>
<td>overwrite existing dataset</td>
</tr>
<tr>
<td>all</td>
<td>save ( e(sample) ) with the dataset; programmer’s option</td>
</tr>
<tr>
<td>orphans</td>
<td>save all value labels</td>
</tr>
<tr>
<td>emptyok</td>
<td>save dataset even if zero observations and zero variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>saveold_options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>version(#)</td>
<td>specify version ( 11 \leq # \leq 16 ); default is version(13), meaning Stata 13 format</td>
</tr>
<tr>
<td>nolabel</td>
<td>omit value labels from the saved dataset</td>
</tr>
<tr>
<td>replace</td>
<td>overwrite existing dataset</td>
</tr>
<tr>
<td>all</td>
<td>save ( e(sample) ) with the dataset; programmer’s option</td>
</tr>
</tbody>
</table>

Options for save

- **nolabel** omits value labels from the saved dataset. The associations between variables and value-label names, however, are saved along with the dataset label and the variable labels.
- **replace** permits `save` to overwrite an existing dataset.
- **all** is for use by programmers. If specified, \( e(sample) \) will be saved with the dataset. You could run a regression; `save mydata, all; drop _all; use mydata; and predict yhat if e(sample)`.
- **orphans** saves all value labels, including those not attached to any variable.
- **emptyok** is a programmer’s option. It specifies that the dataset be saved, even if it contains zero observations and zero variables. If `emptyok` is not specified and the dataset is empty, `save` responds with the message “no variables defined”.

Options for saveold

- **version(#)** specifies which previous .dta file format is to be used. \( # \) may be 16, 15, 14, 13, 12, or 11. The default is `version(13)`, meaning Stata 13 format. To save datasets in the modern, Stata 17 format, use the `save` command, not `saveold`. Stata 14 through Stata 17 share the same format, so you do not have to use `saveold` to save a Stata 14, 15, or 16 dataset; simply use `save`.
- **nolabel** omits value labels from the saved dataset. The associations between variables and value-label names, however, are saved along with the dataset label and the variable labels.
- **replace** permits `saveold` to overwrite an existing dataset.
all is for use by programmers. If specified, \texttt{e(sample)} will be saved with the dataset. You could run a regression; \texttt{save mydata, all}; \texttt{drop _all}; \texttt{use mydata}; and predict \texttt{yhat} if \texttt{e(sample)}.

**Remarks and examples**

Stata keeps the data on which you are currently working in your computer’s memory. You put the data there in the first place; see [U] 22 Entering and importing data. Thereafter, you can save the dataset on disk so that you can use it easily in the future. Stata stores your data on disk in a compressed format that only Stata understands. This does not mean, however, that you are locked into using only Stata. Any time you wish, you can export the data to a format other software packages understand; see [D] export.

Stata goes to a lot of trouble to keep you from accidentally losing your data. When you attempt to leave Stata by typing \texttt{exit}, Stata checks that your data have been safely stored on disk. If not, Stata refuses to let you leave. (You can tell Stata that you want to leave anyway by typing \texttt{exit, clear}.) Similarly, when you \texttt{save} your data in a disk file, Stata ensures that the disk file does not already exist. If it does exist, Stata refuses to save it. You can use the \texttt{replace} option to tell Stata that it is okay to overwrite an existing file.

▶ Example 1

We have entered data into Stata for the first time. We have the following data:

```
. describe
Contains data
   Observations: 39
   Variables: 5

Variable name Storage type Display format Value label Variable label

  acc_rate  float %9.0g  Accident rate
  spdlimit  float %9.0g  Speed limit
  acc_pts   float %9.0g  Access points per mile
  rate      float %9.0g  rcat  Accident rate per million vehicle miles
  spdcat    float %9.0g  scat  Speed limit category

Sorted by:  
   Note: Dataset has changed since last saved.
```

We have a dataset containing 39 observations on five variables, and, evidently, we have gone to a lot of trouble to prepare this dataset. We have used the \texttt{label data} command to label the data Minnesota Highway Data, the \texttt{label variable} command to label all the variables, and the \texttt{label define} and \texttt{label values} commands to attach value labels to the last two variables. (See [U] 12.6.3 Value labels for information about doing this.)

At the end of the \texttt{describe}, Stata notes that the “dataset has changed since last saved”. This is Stata’s way of gently reminding us that these data need to be saved. Let’s \texttt{save} our data:

```
. save hiway
file hiway.dta saved
```

We type \texttt{save hiway}, and Stata stores the data in a file named \texttt{hiway.dta}. (Stata automatically added the \texttt{.dta} suffix.) Now when we \texttt{describe} our data, we no longer get the warning that our dataset has not been saved; instead, we are told the name of the file in which the data are saved:
Just to prove to you that the data have really been saved, let’s eliminate the copy of the data in memory by typing `drop _all`:

```
. drop _all
. describe
```

We now have no data in memory. Because we saved our dataset, we can retrieve it by typing `use hiway`:

```
. use hiway
(Minnesota Highway Data, 1973)
. describe
```

```
Variable name          Storage type Display format Value label            Variable label
acc_rate              float      %9.0g                Accident rate
spdlimit              float      %9.0g                Speed limit
acc_pts               float      %9.0g                Access points per mile
rate                  float      %9.0g                Accident rate per million vehicle miles
spdcat                float      %9.0g                Speed limit category
```

Sorted by:
Example 2

Continuing with our previous example, we have saved our data in the file hiway.dta. We continue to work with our data and discover an error; we made a mistake when we typed one of the values for the spdlimit variable:

```stata
    . list in 1/3

   +-----------------+-----------------+-----------------+-----------------+-----------------+
   | acc_rate | spdlimit | acc_pts | rate | spdcat        |
   |----------+----------+----------+------|-------------+|
   | 1. 1.61  |  50      |  2.2    | Below 4| Above 60     |
   | 2. 1.81  |  60      |  6.8    | Below 4| 55 to 60     |
   | 3. 1.84  |  55      | 14      | Below 4| 55 to 60     |
   +-----------------+-----------------+-----------------+-----------------+-----------------+
```

In the first observation, the spdlimit variable is 50, whereas the spdcat variable indicates that the speed limit is more than 60 miles per hour. We check our original copy of the data and discover that the spdlimit variable ought to be 70. We can fix it with the replace command:

```stata
    . replace spdlimit=70 in 1
    (1 real change made)
```

If we were to describe our data now, Stata would warn us that our data have changed since they were last saved:

```stata
    . describe
    Contains data from hiway.dta
    Observations: 39 Minnesota Highway Data, 1973
    Variables: 5 21 Jul 2000 11:42
    +--------------------------------------------------+
    | Variable name | Storage type | Display format | Value label                          |
    +--------------------------------------------------+
    | acc_rate      | float        | %9.0g          | Accident rate                        |
    | spdlimit      | float        | %9.0g          | Speed limit                          |
    | acc_pts       | float        | %9.0g          | Access points per mile               |
    | rate          | float        | %9.0g          | rcat Accident rate per million vehicle |
    | spdcat        | float        | %9.0g          | scat Speed limit category            |
    +--------------------------------------------------+
    Sorted by:                                           
    Note: Dataset has changed since last saved.
```

We take our cue and attempt to save the data again:

```stata
    . save hiway
    file hiway.dta already exists
    r(602);
```

Stata refuses to honor our request, telling us instead that “file hiway.dta already exists”. Stata will not let us accidentally overwrite an existing dataset. To replace the data, we must do so explicitly by typing save hiway, replace. If we want to save the file under the same name as it was last known to Stata, we can omit the filename:

```stata
    . save, replace
    file hiway.dta saved
```

Now our data are saved.
Also see

[D] `compress` — Compress data in memory

[D] `export` — Overview of exporting data from Stata

[D] `import` — Overview of importing data into Stata

[D] `use` — Load Stata dataset

[P] `File formats .dta` — Description of .dta file format

[U] 11.6 Filenaming conventions
**separate — Create separate variables**

**Description**

`separate` creates new variables containing values from `varname`.

**Quick start**

Create one variable for each level of `catvar` containing value of `v1` or missing

`separate v1, by(catvar)`

As above, but treat missing values of `catvar` as a valid category

`separate v1, by(catvar) missing`

Create `v10` as the value of `v1` when `v2 ≥ 20` or missing and missing otherwise and `v11` as the value of `v1` when `v2 < 20` and missing otherwise

`separate v1, by(v2 < 20)`

As above, but name new variables `newv1` and `newv2`

`separate v1, by(v2 < 20) generate(newv) sequential`

**Menu**

Data > Create or change data > Other variable-transformation commands > Create separate variables
Syntax

```
separate varname [if] [in], by(groupvar | exp) [options]
```

**options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
</tr>
<tr>
<td>* by(groupvar)</td>
</tr>
<tr>
<td>* by(exp)</td>
</tr>
<tr>
<td><strong>Options</strong></td>
</tr>
<tr>
<td>generate(stubname)</td>
</tr>
<tr>
<td>sequential</td>
</tr>
<tr>
<td>missing</td>
</tr>
<tr>
<td>shortlabel</td>
</tr>
</tbody>
</table>

* Either by(groupvar) or by(exp) must be specified.
collect is allowed; see [U] 11.1.10 Prefix commands.

Options

**Main**

by(groupvar | exp) specifies one variable defining the categories or a logical expression that categorizes the observations into two groups.

- If by(groupvar) is specified, groupvar may be a numeric or string variable taking on any values.
- If by(exp) is specified, the expression must evaluate to true (1), false (0), or missing.
- by() is required.

**Options**

generate(stubname) specifies how the new variables are to be named. If generate() is not specified, separate uses the name of the original variable, shortening it if necessary. If generate() is specified, separate uses stubname. If any of the resulting names is too long when the values are suffixed, it is not shortened and an error message is issued.

- sequential specifies that categories be numbered sequentially from 1. By default, separate uses the actual values recorded in the original variable, if possible, and sequential numbers otherwise.
- separate can use the original values if they are all nonnegative integers smaller than 10,000.

missing also creates a variable for the category missing if missing occurs (groupvar takes on the value missing or exp evaluates to missing). The resulting variable is named in the usual manner but with an appended underscore, for example, bp_. By default, separate creates no such variable. The contents of the other variables are unaffected by whether missing is specified.

shortlabel creates a variable label that is shorter than the default. By default, when separate generates the new variable labels, it includes the name of the variable being separated. shortlabel specifies that the variable name be omitted from the new variable labels.
Remarks and examples

➤ Example 1

We have data on the miles per gallon (mpg) and country of manufacture of 74 automobiles. We want to compare the distributions of mpg for domestic and foreign automobiles by plotting the quantiles of the two distributions (see [R] Diagnostic plots).

```
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. separate mpg, by(foreign)
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage</th>
<th>Display</th>
<th>Value</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg0</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td>mpg, foreign == Domestic</td>
</tr>
<tr>
<td>mpg1</td>
<td>byte</td>
<td>%8.0g</td>
<td></td>
<td>mpg, foreign == Foreign</td>
</tr>
</tbody>
</table>

```
. list mpg* foreign
```

```
  mpg  mpg0  mpg1  foreign
  ---  ----  ----  ------
   22   22 .   Domestic
   17   17 .   Domestic
   22   22 .   Domestic
   16   16 .   Domestic
   17   17 .   Domestic
   28   28 .   Domestic
(8 omitted)
   . 25   25 .    Foreign
   17   17 .    Foreign
```

```
. qqplot mpg0 mpg1
```

In our auto dataset, the foreign cars have better gas mileage.
Stored results

`separate` stores the following in `r()`: 

Macros 

`r(varlist)` names of the newly created variables 

Acknowledgment

`separate` was originally written by Nicholas J. Cox of the Department of Geography at Durham University, UK, who is coeditor of the *Stata Journal* and author of *Speaking Stata Graphics*.

Reference

Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. College Station, TX: Stata Press.

Also see

[R] `tabulate oneway` — One-way table of frequencies 
[R] `tabulate twoway` — Two-way table of frequencies 
[R] `tabulate, summarize()` — One- and two-way tables of summary statistics
shell — Temporarily invoke operating system

## Description

`shell` (synonym: “!”) allows you to send commands to your operating system or to enter your operating system for interactive use. Stata will wait for the shell to close or the `operating_system_command` to complete before continuing.

`winexec` allows you to start other programs (such as browsers) from Stata’s command line. Stata will continue without waiting for the program to complete.

`xshell` (Stata for Mac and Unix(GUI) only) brings up an `xterm` window in which the command is to be executed.

## Syntax

```
{ _shell | ! } [ operating_system_command ]
```

```
winexec program_name [ program_args ]
```

```
{ xshell | !! } [ operating_system_command ]
```

Command availability:

<table>
<thead>
<tr>
<th>Command</th>
<th>Stata for . . .</th>
<th>Windows</th>
<th>Mac</th>
<th>Unix(GUI)</th>
<th>Unix(console)</th>
</tr>
</thead>
<tbody>
<tr>
<td>shell</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>winexec</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>xshell</td>
<td>–</td>
<td>X</td>
<td>X</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

## Remarks and examples

Remarks are presented under the following headings:

- Stata for Windows
- Stata for Mac
- Stata for Unix(GUI)
- Stata for Unix(console)
Stata for Windows

`shell`, without arguments, preserves your session and invokes the operating system. Stata’s Command window will disappear, and a Windows command prompt will appear, indicating that you may not continue in Stata until you exit the Windows command prompt. To reenter Stata, type `exit` at your operating system’s prompt. Your Stata session is reestablished just as if you had never left.

Say that you are using Stata for Windows and you suddenly realize you need to do two things. You need to enter your operating system for a few minutes. Rather than exiting Stata, doing what you have to do, and then restarting Stata, you type `shell` in the Command window. A Windows command prompt appears:

```
C:\data>
```

You can now do whatever you need to do in Windows, and Stata will wait until you exit the Windows command prompt before continuing.

Experienced Stata users seldom type out the word `shell`. They type “!”. Also you do not have to enter your operating system, issue a command, and then exit back to Stata. If you want to execute one command, you can type the command right after the word `shell` or the exclamation point:

```
. !rename try15.dta final.dta
```

If you do this, the Windows command prompt will open and close as the command is executed.

Stata for Windows users can also use the `winexec` command, which allows you to launch any Windows application from within Stata. You can think of it as a shortcut for clicking on the Windows Start button, choosing Run..., and typing a command.

Assume that you are working in Stata and decide that you want to run a text editor:

```
. winexec notepad
```

(The Windows application Notepad will start and run at the same time as Stata)

You could even pass a filename to your text editor:

```
. winexec notepad c:\docs\myfile.txt
```

You may need to specify a complete path to the executable that you wish to launch:

```
. winexec c:\windows\notepad c:\docs\myfile.txt
```

The important difference between `winexec` and `shell` is that Stata does not wait for whatever program `winexec` launches to complete before continuing. Stata will wait for the program `shell` launches to complete before performing any further commands.

Stata for Mac

`shell`, with arguments, invokes your operating system, executes one command, and redirects the output to the Results window. The command must complete before you can enter another command in the Command window.

Say that you are using Stata for Mac and suddenly realize that there are two things you have to do. You need to switch to the Finder or enter commands from a terminal for a few minutes. Rather than exiting Stata, doing what you have to do, and then switching back to Stata, you type `shell` and the command in the Command window to execute one command. You then repeat this step for each command that you want to execute from the shell.
Experienced Stata users seldom type out the word `shell`. They type “!”.

```bash
    !mv try15.dta final.dta
```

Be careful not to execute commands, such as `vi`, that require interaction from you. Because all output is redirected to Stata’s Results window, you will not be able to interact with the command from Stata. This will effectively lock up Stata because the command will never complete.

When you type `xshell vi myfile.do`, Stata invokes an `xterm` window (which in turn invokes a `shell`) and executes the command there. Typing `!!vi myfile.do` is equivalent to typing `xshell vi myfile.do`.

### Technical note

On macOS, `xterm` is available when `X11` is installed. To install `X11`, you must first download XQuartz from https://xquartz.macosforge.org/.

Stata for Mac users can also use the `winexec` command, which allows you to launch any native application from within Stata. You may, however, have to specify the absolute path to the application. If the application you wish to launch is a macOS application bundle, you must specify an absolute path to the executable in the bundle.

Assume that you are working in Stata and decide that you want to run a text editor:

```bash
    . winexec /Applications/TextEdit.app/Contents/MacOS/TextEdit
    ( The macOS application TextEdit will start and run at the same time as Stata )
```

You could even pass a filename to your text editor:

```bash
    . winexec /Applications/TextEdit.app/Contents/MacOS/TextEdit
    > /Users/cnguyen/myfile.do
```

If you specify a file path as an argument to the program to be launched, you must specify an absolute path. Also using `~` in the path will not resolve to a home directory. If an application cannot be launched from a terminal window, it cannot be launched by `winexec`.

The important difference between `winexec` and `shell` is that Stata does not wait for whatever program `winexec` launches to complete before continuing. Stata will wait for the program `shell` launches to complete before performing any further commands. `shell` is appropriate for executing shell commands; `winexec` is appropriate for launching applications.

### Stata for Unix(GUI)

`shell`, without arguments, preserves your session and invokes the operating system. The Command window will disappear, and an `xterm` window will appear, indicating that you may not do anything in Stata until you exit the `xterm` window. To reenter Stata, type `exit` at the Unix prompt. Your Stata session is reestablished just as if you had never left.

Say that you are using Stata for Unix(GUI) and suddenly realize that you need to do two things. You need to enter your operating system for a few minutes. Rather than exiting Stata, doing what you have to do, and then restarting Stata, you type `shell` in the Command window. An `xterm` window will appear:

```
mycomputer$ _
```
You can now do whatever you need to do, and Stata will wait until you exit the window before continuing.

Experienced Stata users seldom type out the word `shell`. They type `!`. Also you do not have to enter your operating system, issue a command, and then exit back to Stata. If you want to execute one command, you can type the command right after the word `shell` or the exclamation point:

```
. !mv try15.dta final.dta
```

Be careful because sometimes you will want to type

```
. !!vi myfile.do
```

and in other cases,

```
. winexec xedit myfile.do
```

`!!` is a synonym for `xshell`—a command different from, but related to, `shell`—and `winexec` is a different and related command, too.

Before we get into this, understand that if all you want is a shell from which you can issue Unix commands, type `shell` or `!`:

```
. !
mycomputer$ _
```

When you are through, type `exit` to the Unix prompt, and you will return to Stata:

```
mycomputer$ exit
' _
```

If, on the other hand, you want to specify in Stata the Unix command that you want to execute, you need to decide whether you want to use `shell`, `xshell`, or `winexec`. The answer depends on whether the command you want to execute requires a terminal window or is an X application:

```
... does not need a terminal window:   use shell  ... (synonym:  !...)
... needs a terminal window:          use xshell ... (synonym:  !!...)
... is an X application:              use winexec ... (no synonym)
```

When you type `shell mv try15.dta final.dta`, Stata invokes your shell (`/bin/sh`, `/bin/csh`, etc.) and executes the specified command (`mv` here), routing the standard output and standard error back to Stata. Typing `!mv try15.dta final.dta` is the same as typing `shell mv try15.dta final.dta`.

When you type `xshell vi myfile.do`, Stata invokes an `xterm` window (which in turn invokes a shell) and executes the command there. Typing `!!vi myfile.do` is equivalent to typing `xshell vi myfile.do`.

When you type `winexec xedit myfile.do`, Stata directly invokes the command specified (`xedit` here). No `xterm` window is brought up nor is a shell invoked because, here, `xterm` does not need it. `xterm` is an X application that will create its own window in which to run. You could have typed `!!xedit myfile.do`. That would have brought up an unnecessary `xterm` window from which `xedit` would have been executed, and that would not matter. You could even have typed `!xedit myfile.do`. That would have invoked an unnecessary shell from which `xedit` would have been executed, and that would not matter, either. The important difference, however, is that `shell` and `xshell` wait until the process completes before allowing Stata to continue, and `winexec` does not.
Technical note

You can set Stata global macros to control the behavior of `shell` and `xshell`. The macros are

- `$S_SHELL` defines the shell to be used by `shell` when you type a command following `shell`. The default is something like “/bin/sh -c”, although this can vary, depending on how your Unix environment variables are set.
- `$S_XSHELL` defines shell to be used by `shell` and `xshell` when they are typed without arguments. The default is “xterm”.
- `$S_XSHELL2` defines shell to be used by `xshell` when it is typed with arguments. The default is “xterm -e”.

For instance, if you type in Stata

```
.global S_XSHELL2 "/usr/X11R6/bin/xterm -e"
```

and then later type

```
!vi myfile.do
```

then Stata would issue the command `/usr/X11R6/bin/xterm -e vi myfile.do` to Unix.

If you do make changes, we recommend that you record the changes in your `profile.do` file.

Stata for Unix(console)

`shell`, without arguments, preserves your session and then invokes your operating system. Your Stata session will be suspended until you exit the shell, at which point your Stata session is reestablished just as if you had never left.

Say that you are using Stata and you suddenly realize that you need to do two things. You need to enter your operating system for a few minutes. Rather than exiting Stata, doing what you have to do, and then restarting Stata, you type `shell`. A Unix prompt appears:

```
.shell
(Type exit to return to Stata)
$
```

You can now do whatever you need to do and type `exit` when you finish. You will return to Stata just as if you had never left.

Experienced Stata users seldom type out the word `shell`. They type ‘!’. Also you do not have to enter your operating system, issue a command, and then exit back to Stata. If you want to execute one command, you can type the command right after the word `shell` or the exclamation point. If you want to edit the file `myfile.do`, and if `vi` is the name of your favorite editor, you could type

```
!vi myfile.do

Stata opens your editor.
```

```
When you exit your editor:
```

```
. 
```
Reference


Also see

[D] cd — Change directory
[D] copy — Copy file from disk or URL
[D] dir — Display filenames
[D] erase — Erase a disk file
[D] mkdir — Create directory
[D] rmdir — Remove directory
[D] type — Display contents of a file
**Description**

`snapshot` saves to disk and restores from disk copies of the data in memory. `snapshot`’s main purpose is to allow the Data Editor to save and restore data snapshots during an interactive editing session. A more popular alternative for programmers is `preserve`; see [P] `preserve`.

Snapshots are referred to by a `snapshot#`. If no snapshots currently exist, the next snapshot saved will receive a `snapshot#` of 1. If snapshots do exist, the next snapshot saved will receive a `snapshot#` one greater than the highest existing `snapshot#`.

`snapshot save` creates a temporary file containing a copy of the data currently in memory and attaches an optional label (up to 80 characters) to the saved snapshot. Up to 1,000 snapshots may be saved.

`snapshot label` changes the label on the specified snapshot.

`snapshot restore` replaces the data in memory with the data from the specified snapshot.

`snapshot list` lists specified snapshots.

`snapshot erase` erases specified snapshots.

**Quick start**

Save a temporary copy of the data to disk, and label the snapshot `mylabel1`

```
snapshot save, label(mylabel1)
```

List snapshot numbers and labels

```
snapshot list _all
```

Restore snapshot `mylabel1` with number 1

```
snapshot restore 1
```

Change label of snapshot 1 to `mylabel2`

```
snapshot label 1 "mylabel2"
```

Delete all current snapshots, and begin renumbering new snapshots from 1

```
snapshot erase _all
```

**Menu**

Data > Data Editor > Data Editor (Edit)
Syntax

Save snapshot

    snapshot save [, label("label")]

Change snapshot label

    snapshot label snapshot# "label"

Restore snapshot

    snapshot restore snapshot#

List snapshots

    snapshot list [ _all | numlist ]

Erase snapshots

    snapshot erase _all | numlist

collect is allowed with snapshot save; see [U] 11.1.10 Prefix commands.

Option

    label("label") is for use with snapshot save and allows you to label a snapshot when saving it.

Remarks and examples

    snapshot was created to allow a user using the Data Editor to save and restore snapshots of their data while editing them interactively. It is similar to a checkpoint save in a video game, where after you have made a certain amount of progress, you wish to make sure you will be able to return to that point no matter what may happen in the future.

    snapshot does not overwrite any copies of your data that you may have saved to disk. It saves a copy of the data currently in memory to a temporary file and allows you to later restore that copy to memory.

    snapshot saves the date and time at which you create a snapshot. It is a good idea to also give a snapshot a label so that you will be better able to distinguish between multiple snapshots should you need to restore one.

Technical note

    Although we mention above the use of the Data Editor and we demonstrate below the use of snapshot, we recommend that data cleaning not be done interactively. Instead, we recommend that data editing and cleaning be done in a reproducible manner through the use of do-files; see [U] 16 Do-files.
Example 1

You decide to make some changes to auto.dta. You make a snapshot of the data before you begin making changes, and you make another snapshot after the changes:

```
. use https://www.stata-press.com/data/r17/auto
   (1978 automobile data)
. snapshot save, label("before changes")
   snapshot 1 (before changes) created at 19 Apr 2020 21:32
. generate gpm = 1/mpg
. label variable gpm "Gallons per mile"
. snapshot save, label("after changes")
   snapshot 2 (after changes) created at 19 Apr 2020 21:34
```

You go on to do some analyses, but then, for some reason, you accidentally drop the variable you previously created:

```
. drop gpm
```

Luckily, you made some snapshots of your work:

```
. snapshot list
   snapshot 1 (before changes) created at 19 Apr 2020 21:32
   snapshot 2 (after changes) created at 19 Apr 2020 21:34
. snapshot restore 2
```

```
. describe gpm
   Variable Storage Display Value
   name type format label Variable label
       gpm  float %9.0g Gallons per mile
```

Stored results

snapshot save stores the following in r():

Scalars

   r(snapshot)  sequence number of snapshot saved

Also see

[D] edit — Browse or edit data with Data Editor
[P] preserve — Preserve and restore data
sort — Sort data

Description

sort arranges the observations of the current data into ascending order based on the values of the variables in varlist. There is no limit to the number of variables in varlist. Missing numeric values are interpreted as being larger than any other number, so they are placed last with . < .a < .b < ··· < .z. When you sort on a string variable, however, null strings are placed first and uppercase letters come before lowercase letters.

The dataset is marked as being sorted by varlist unless in range is specified. If in range is specified, only those observations are rearranged. The unspecified observations remain in the same place.

Quick start

Sort dataset in memory by ascending values of v1

```
sort v1
```

As above, and order within v1 by ascending values of v2 and within v2 by v3

```
sort v1 v2 v3
```

As above, and keep observations with the same values of v1, v2, and v3 in the same presort order

```
sort v1 v2 v3, stable
```

Menu

Data > Sort
Syntax

```
sort varlist [in] [, stable]
```

Option

`stable` specifies that observations with the same values of the variables in `varlist` keep the same relative order in the sorted data that they had previously. For instance, consider the following data:

```
x    b
3    1
1    2
1    1
1    3
2    4
```

Typing `sort x` without the `stable` option produces one of the following six orderings:

```
x    b
1    2
1    1
1    3
2    4
3    1
```

Without the `stable` option, the ordering of observations with equal values of `varlist` is randomized. With `sort x, stable`, you will always get the first ordering and never the other five.

If your intent is to have the observations sorted first on `x` and then on `b` within tied values of `x` (the fourth ordering above), you should type `sort x b` rather than `sort x, stable`.

`stable` is seldom used and, when specified, causes `sort` to execute more slowly.

Remarks and examples

Sorting data is one of the more common tasks involved in processing data. Often, before Stata can perform some task, the data must be in a specific order. For the `merge` command to create a new dataset that matches records from two datasets on a common key, both of those datasets must be sorted by that key. Either you will sort the data or `merge` will sort it for you. If you want to use the `by varlist:` prefix, the data must be sorted in order of `varlist`. You even sort data to put it into a more convenient order when using `list`.

Remarks are presented under the following headings:

- Finding the smallest values (and the largest)
- Tracking sort order
- Sorting on multiple variables
- Descending sorts
- Sorting on string variables
- Sorting with ties
Finding the smallest values (and the largest)

Sorting data can be informative. Suppose that we have data on automobiles, and each car’s make and mileage rating (called make and mpg) are included among the variables in the data. We want to list the five cars with the lowest mileage rating in our data:

```
. use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
. keep make mpg weight
. sort mpg, stable
. list make mpg in 1/5
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linc. Continental</td>
<td>12</td>
</tr>
<tr>
<td>Linc. Mark V</td>
<td>12</td>
</tr>
<tr>
<td>Cad. Deville</td>
<td>14</td>
</tr>
<tr>
<td>Cad. Eldorado</td>
<td>14</td>
</tr>
<tr>
<td>Linc. Versailles</td>
<td>14</td>
</tr>
</tbody>
</table>

We can also list the five cars with the highest mileage.

```
. list in -5/1
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toyota Corolla</td>
<td>31</td>
<td>2,200</td>
</tr>
<tr>
<td>Plym. Champ</td>
<td>34</td>
<td>1,800</td>
</tr>
<tr>
<td>Datsun 210</td>
<td>35</td>
<td>2,020</td>
</tr>
<tr>
<td>Subaru</td>
<td>35</td>
<td>2,050</td>
</tr>
<tr>
<td>VW Diesel</td>
<td>41</td>
<td>2,040</td>
</tr>
</tbody>
</table>

Tracking sort order

Stata keeps track of the order of your data. For instance, we just sorted the above data on mpg. When we ask Stata to describe the data in memory, it tells us how the dataset is sorted:
. describe
Contains data from https://www.stata-press.com/data/r17/auto.dta
Observations: 74          1978 automobile data
Variables: 3          13 Apr 2020 17:45
(_dta has notes)

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str18</td>
<td>%-18s</td>
<td>Make and model</td>
<td>Make and model</td>
</tr>
<tr>
<td>mpg</td>
<td>int</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td>Weight (lbs.)</td>
<td>Weight (lbs.)</td>
</tr>
</tbody>
</table>

Sorted by: mpg

Note: Dataset has changed since last saved.

Stata keeps track of changes in sort order. If we were to make a change to the mpg variable, Stata would know that the data are no longer sorted. Remember that the first observation in our data has mpg equal to 12, as does the second. Let’s change the value of the first observation:

```
. replace mpg=13 in 1
(1 real change made)
```

. describe
Contains data from https://www.stata-press.com/data/r17/auto.dta
Observations: 74          1978 automobile data
Variables: 3          13 Apr 2020 17:45
(_dta has notes)

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>str18</td>
<td>%-18s</td>
<td>Make and model</td>
<td>Make and model</td>
</tr>
<tr>
<td>mpg</td>
<td>int</td>
<td>%8.0g</td>
<td>Mileage (mpg)</td>
<td>Mileage (mpg)</td>
</tr>
<tr>
<td>weight</td>
<td>int</td>
<td>%8.0gc</td>
<td>Weight (lbs.)</td>
<td>Weight (lbs.)</td>
</tr>
</tbody>
</table>

Sorted by:

Note: Dataset has changed since last saved.

After making the change, Stata indicates that our dataset is “Sorted by:” nothing. Let’s put the dataset back as it was:

```
. replace mpg=12 in 1
(1 real change made)
```

. sort mpg

Technical note

Stata is limited in how it tracks changes in the sort order and will sometimes decide that a dataset is not sorted when, in fact, it is. For instance, if we were to change the first observation of our automobile dataset from 12 miles per gallon to 10, Stata would decide that the dataset is “Sorted by:” nothing, just as it did above when we changed mpg from 12 to 13. Our change in example 2 did change the order of the data, so Stata was correct. Changing mpg from 12 to 10, however, does not really affect the sort order.

As far as Stata is concerned, any change to the variables on which the data are sorted means that the data are no longer sorted, even if the change actually leaves the order unchanged. Stata may be dumb, but it is also fast. It sorts already-sorted datasets instantly, so Stata’s ignorance costs us little.

Technical note
**Sorting on multiple variables**

Data can be sorted by more than one variable, and in such cases, the sort order is lexicographic. If we sort the data by two variables, for instance, the data are placed in ascending order of the first variable, and then observations that share the same value of the first variable are placed in ascending order of the second variable. Let’s order our automobile data by mpg and within mpg by weight:

```
    . sort mpg weight
    . list in 1/8, sep(4)
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linc. Mark V</td>
<td>12</td>
<td>4,720</td>
</tr>
<tr>
<td>Linc. Continental</td>
<td>12</td>
<td>4,840</td>
</tr>
<tr>
<td>Peugeot 604</td>
<td>14</td>
<td>3,420</td>
</tr>
<tr>
<td>Linc. Versailles</td>
<td>14</td>
<td>3,830</td>
</tr>
<tr>
<td>Cad. Eldorado</td>
<td>14</td>
<td>3,900</td>
</tr>
<tr>
<td>Merc. Cougar</td>
<td>14</td>
<td>4,060</td>
</tr>
<tr>
<td>Merc. XR-7</td>
<td>14</td>
<td>4,130</td>
</tr>
<tr>
<td>Cad. Deville</td>
<td>14</td>
<td>4,330</td>
</tr>
</tbody>
</table>

The data are in ascending order of mpg, and within each mpg category, the data are in ascending order of weight. The lightest car that achieves 14 miles per gallon in our data is the Peugeot 604.

**Technical note**

The sorting technique used by Stata is fast, but the order of variables not included in `varlist` is not maintained. If you wish to maintain the order of additional variables, include them at the end of `varlist`. There is no limit to the number of variables by which you may sort.

**Descending sorts**

Sometimes, you may want to order a dataset by descending sequence of something. Perhaps we wish to obtain a list of the five cars achieving the best mileage rating. The `sort` command orders the data only into ascending sequences. Another command, `gsort`, orders the data in ascending or descending sequences; see [D] `gsort`. You can also create the negative of a variable and achieve the desired result:

```
    . generate negmpg = -mpg
    . sort negmpg
    . list in 1/5
```

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
<th>negmpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>VW Diesel</td>
<td>41</td>
<td>2,040</td>
<td>-41</td>
</tr>
<tr>
<td>Subaru</td>
<td>35</td>
<td>2,050</td>
<td>-35</td>
</tr>
<tr>
<td>Datsun 210</td>
<td>35</td>
<td>2,020</td>
<td>-35</td>
</tr>
<tr>
<td>Plym. Champ</td>
<td>34</td>
<td>1,800</td>
<td>-34</td>
</tr>
<tr>
<td>Toyota Corolla</td>
<td>31</td>
<td>2,200</td>
<td>-31</td>
</tr>
</tbody>
</table>

We find that the VW Diesel tops our list.
Sorting on string variables

`sort` may also be used on string variables. The data are sorted alphabetically:

```
. sort make
. list in 1/5

<table>
<thead>
<tr>
<th>make</th>
<th>mpg</th>
<th>weight</th>
<th>negmpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMC Concord</td>
<td>22</td>
<td>2,930</td>
<td>-22</td>
</tr>
<tr>
<td>AMC Pacer</td>
<td>17</td>
<td>3,350</td>
<td>-17</td>
</tr>
<tr>
<td>AMC Spirit</td>
<td>22</td>
<td>2,640</td>
<td>-22</td>
</tr>
<tr>
<td>Audi 5000</td>
<td>17</td>
<td>2,830</td>
<td>-17</td>
</tr>
<tr>
<td>Audi Fox</td>
<td>23</td>
<td>2,070</td>
<td>-23</td>
</tr>
</tbody>
</table>
```

Technical note

Bear in mind that Stata takes “alphabetically” to mean “in order by byte value”. This means that all uppercase letters come before lowercase letters; for example, Z < a. As far as Stata is concerned, the following list is sorted alphabetically:

```
. list, sep(0)

<table>
<thead>
<tr>
<th>myvar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
</tr>
<tr>
<td>Alpha</td>
</tr>
<tr>
<td>BETA</td>
</tr>
<tr>
<td>Beta</td>
</tr>
<tr>
<td>alpha</td>
</tr>
<tr>
<td>beta</td>
</tr>
</tbody>
</table>
```

For most purposes, this method of sorting is sufficient. It is possible to override Stata’s sort logic. See [U] 12.4.2.5 Sorting strings containing Unicode characters for information about ordering strings in a language-sensitive way. We do not recommend that you do this.

Sorting with ties

Sorting when your list of sort variables does not uniquely identify an observation, that is to say when you have ties, is usually dangerous and should be avoided. Consider using `sort` to find the average `mpg` for the five cars with the smallest gear ratio.

```
. use https://www.stata-press.com/data/r17/auto
   (1978 automobile data)
. sort gear_ratio
. summarize mpg in 1/5

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>5</td>
<td>17</td>
<td>3.674235</td>
<td>14</td>
<td>21</td>
</tr>
</tbody>
</table>
```

So the answer is 17.

We go on and do some other work.
We forgot to write down the answer from earlier, and silly us, we were not logging our results. So we run our commands again:

```
use https://www.stata-press.com/data/r17/auto
(1978 automobile data)
sort gear_ratio
summarize mpg in 1/5
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>5</td>
<td>15.8</td>
<td>2.949576</td>
<td>14</td>
<td>21</td>
</tr>
</tbody>
</table>

So the answer is 15.8.

What happened? The title of this section is a clue. Let’s list some of the data:

```
list gear_ratio mpg in 1/10
```

<table>
<thead>
<tr>
<th>gear_ratio</th>
<th>mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>2.19</td>
</tr>
<tr>
<td>2.</td>
<td>2.24</td>
</tr>
<tr>
<td>3.</td>
<td>2.26</td>
</tr>
<tr>
<td>4.</td>
<td>2.28</td>
</tr>
<tr>
<td>5.</td>
<td>2.41</td>
</tr>
<tr>
<td>6.</td>
<td>2.41</td>
</tr>
<tr>
<td>7.</td>
<td>2.41</td>
</tr>
<tr>
<td>8.</td>
<td>2.43</td>
</tr>
<tr>
<td>9.</td>
<td>2.47</td>
</tr>
<tr>
<td>10.</td>
<td>2.47</td>
</tr>
</tbody>
</table>

The first four observations look fine; each value of *gear_ratio* is unique. But the fifth, sixth, and seventh observations all have a *gear_ratio* of 2.41, whereas the values of *mpg* differ. Do we want *mpg* = 15, *mpg* = 16, or *mpg* = 21 in our mean?

There are not many things we can do after a sort that will produce unique results if the sort itself has observations with ties in `varlist`. The ordering is not unique. You must be sure that the ordering really does not matter. If that is the case, then why did you sort in the first place?

So what are we to do? We could rephrase our question as “What is the lowest possible average *mpg* for five cars with the smallest *gear_ratio*?” Then we would type

```
sort gear_ratio mpg
summarize mpg in 1/5
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>5</td>
<td>15.8</td>
<td>2.949576</td>
<td>14</td>
<td>21</td>
</tr>
</tbody>
</table>

Now we will always get the same answer—15.8.

How do you know your sort variables form a unique ordering? Ask

```
isid gear_ratio mpg
```

variables *gear_ratio* and *mpg* do not uniquely identify the observations r(459);

That is still not a unique ordering. Our analysis does not require a fully unique ordering. Because we are summarizing *mpg*, tied values of *mpg* will give the same answer. Even so, there would be no harm in adding another variable to make the ordering truly unique:
What if we did not want the lowest mpg? What if we preferred a randomized answer where the computer chooses one of the observations with tied gear_ratio? The best approach is to use a good random-number generator to create a new variable with random values that you will also sort on:

```
.set seed 12345
.generate rnd = runiform()
.sort gear_ratio rnd
.summarize mpg in 1/5
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>5</td>
<td>17</td>
<td>3.674235</td>
<td>14</td>
<td>21</td>
</tr>
</tbody>
</table>

Why did we set the seed? So that our randomized result is reproducible. That is not a contradiction.

A benefit of this approach is that regardless of any further transformations or manipulations we make on this dataset we can always recover the ordering by typing

```
sort gear_ratio rnd
```

Well, we cannot change the values of gear_ratio or rnd, and we cannot add or insert observations, but any other manipulations are allowed.

Our example is rather artificial, but there are many cases where you do want a randomized order within tied values of a sorted variable. One such case is creating simulated datasets for panel data or multilevel data.

It turns out that our first results were also randomly ordered. That is true because sort performs a quick randomized jumbling before sorting. We were already getting a randomized order within the ties. Do not use this in practice. The randomization performed by sort is designed solely to make sort faster by preventing any possibility of an initial ordering that defeats the sort algorithm and makes the sorting much slower. If you want a random ordering within ties, then use a random-number generator with good properties like the one implemented in runiform(). For more about the random-number generator, see [R] set seed and the references therein.

If you do not want a random ordering within ties and you also do not want to use other variables from the dataset to define a unique ordering, you can add a sequence variable to the dataset and include it in your sort,

```
generate id = _n
.sort gear_ratio id
```

That sort will still depend on the order of your data when id is created, but you will always be able to recreate the ordering by typing

```
sort gear_ratio id
```

The ordering produced after this sort will be identical to the ordering had we instead typed

```
sort gear_ratio, stable
```

The advantage to creating the id variable is that we can recover this ordering at any time in the future by retyping

```
sort gear_ratio id
```
That cannot be said of

```
.sort gear_ratio, stable
```

The ordering after this sort will depend on the order before the `sort` command. So if we sort on another variable between our two stable sorts, the ordering after those two stable sorts will be different.

One final note. If you ran the commands in this entry, you may have obtained different results from those printed here for the first several `summarize` commands and a different ordering from the first `list` command. That is yet another reminder not to perform order-dependent analyses when your current sort order is not unique. You got different results because the jumbler that `sort` preapplies started from a different point than it did when we ran the commands for this manual entry. Unless you start Stata immediately before running a sort with tied values or you set the state of the jumbler, you will rarely get the same ordering for tied keys. If you want to get the ordering we got in this entry, you should use Stata/SE and type

```
.set sortrngstate 12345
```

That’s what we do so that this entry does not change every time we re-create the manuals. See [P] `set sortrngstate`. This is such an esoteric command that we warn you against using it. Regardless, unless your goal is to write a manual entry that describes how to deal with tied values in sorts, do not use `set sortrngstate` to create reproducible sorts. Think about your problem and sort on variables that create the unique ordering you need. Or decide you want a stable sort of the ties based on the current ordering. Or use the method described above that creates a good random number to randomly order the tied values.

References


Also see

[D] `describe` — Describe data in memory or in file

[D] `gsort` — Ascending and descending sort

[U] 11 Language syntax
**split — Split string variables into parts**

**Description**

`split` splits the contents of a string variable, `strvar`, into one or more parts, using one or more `parse_strings` (by default, blank spaces), so that new string variables are generated. Thus `split` is useful for separating “words” or other parts of a string variable. `strvar` itself is not modified.

**Quick start**

Create variables `v#` for each word of `v` separated by spaces

```
split v
```

As above, but split into words or phrases on commas and generate variables `newv#`

```
split v, parse(,) generate(newv)
```

As above, but do not trim leading or trailing spaces

```
split v, parse(,) generate(newv) notrim
```

Create only `newv1`, `newv2`, and `newv3` regardless of the number of possible new variables

```
split v, generate(newv) limit(3)
```

As above, and convert to numeric type when possible

```
split v, generate(newv) limit(3) destring
```

**Menu**

Data > Create or change data > Other variable-transformation commands > Split string variables into parts
Syntax

```
split strvar [if] [in] [, options]
```

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
<td></td>
</tr>
<tr>
<td><code>generate(stub)</code></td>
<td>begin new variable names with <code>stub</code>; default is <code>strvar</code></td>
</tr>
<tr>
<td><code>parse(parse_strings)</code></td>
<td>parse on specified strings; default is to parse on spaces</td>
</tr>
<tr>
<td><code>limit(#)</code></td>
<td>create a maximum of <code>#</code> new variables</td>
</tr>
<tr>
<td><code>notrim</code></td>
<td>do not trim leading or trailing spaces of original variable</td>
</tr>
<tr>
<td><strong>Destring</strong></td>
<td></td>
</tr>
<tr>
<td><code>destring</code></td>
<td>apply <code>destring</code> to new string variables, replacing initial string variables with numeric variables where possible</td>
</tr>
<tr>
<td><code>ignore(&quot;chars&quot;)</code></td>
<td>remove specified nonnumeric characters</td>
</tr>
<tr>
<td><code>force</code></td>
<td>convert nonnumeric strings to missing values</td>
</tr>
<tr>
<td><code>float</code></td>
<td>generate numeric variables as type <code>float</code></td>
</tr>
<tr>
<td><code>percent</code></td>
<td>convert percent variables to fractional form</td>
</tr>
</tbody>
</table>

`collect` is allowed; see [U] 11.1.10 Prefix commands.

**Options**

- `generate(stub)` specifies the beginning characters of the new variable names so that new variables `stub1`, `stub2`, etc., are produced. `stub` defaults to `strvar`.
- `parse(parse_strings)` specifies that, instead of using spaces, parsing use one or more `parse_strings`. Most commonly, one string that is one punctuation character will be specified. For example, if `parse(,)` is specified, then "1,2,3" is split into "1", "2", and "3".

You can also specify 1) two or more strings that are alternative separators of "words" and 2) strings that consist of two or more characters. Alternative strings should be separated by spaces. Strings that include spaces should be bound by " ". Thus if `parse(, " ")` is specified, "1,2 3" is also split into "1", "2", and "3". Note particularly the difference between, say, `parse(a b)` and `parse(ab)`: with the first, `a` and `b` are both acceptable as separators, whereas with the second, only the string `ab` is acceptable.

- `limit(#)` specifies an upper limit to the number of new variables to be created. Thus `limit(2)` specifies that, at most, two new variables be created.
- `notrim` specifies that the original string variable not be trimmed of leading and trailing spaces before being parsed. `notrim` is not compatible with parsing on spaces, because the latter implies that spaces in a string are to be discarded. You can either specify a parsing character or, by default, allow a `trim`.

- `destring` applies `destring` to the new string variables, replacing the variables initially created as strings by numeric variables where possible. See [D] `destring`.

ignore(), `force`, `float`, `percent`; see [D] `destring`. 
split is used to split a string variable into two or more component parts, for example, “words”. You might need to correct a mistake, or the string variable might be a genuine composite that you wish to subdivide before doing more analysis.

The basic steps applied by split are, given one or more separators, to find those separators within the string and then to generate one or more new string variables, each containing a part of the original. The separators could be, for example, spaces or other punctuation symbols, but they can in turn be strings containing several characters. The default separator is a space.

The key string functions for subdividing string variables and, indeed, strings in general, are `strpos()`, which finds the position of separators, and `substr()`, which extracts parts of the string. (See [FN] String functions.) `split` is based on the use of those functions.

If your problem is not defined by splitting on separators, you will probably want to use `substr()` directly. Suppose that you have a string variable, `date`, containing dates in the form "21011952" so that the last four characters define a year. This string contains no separators. To extract the year, you would use `substr(date,-4,4)`. Again suppose that each woman’s obstetric history over the last 12 months was recorded by a `str12` variable containing values such as "npppppppppbnn", where p, b, and n denote months of pregnancy, birth, and nonpregnancy. Once more, there are no separators, so you would use `substr()` to subdivide the string.

`split` discards the separators, because it presumes that they are irrelevant to further analysis or that you could restore them at will. If this is not what you want, you might use `substr()` (and possibly `strpos()`).

Finally, before we turn to examples, compare `split` with the `egen` function `ends()`, which produces the head, the tail, or the last part of a string. This function, like all `egen` functions, produces just one new variable as a result. In contrast, `split` typically produces several new variables as the result of one command. For more details and discussion, including comments on the special problem of recognizing personal names, see [D] `egen`.

`split` can be useful when input to Stata is somehow misread as one string variable. If you copy and paste into the Data Editor, say, under Windows by using the clipboard, but data are space-separated, what you regard as separate variables will be combined because the Data Editor expects comma- or tab-separated data. If some parts of your composite variable are numeric characters that should be put into numeric variables, you could use `destring` at the same time; see [D] `destring`.

```
. split var1, destring
```

Here no `generate()` option was specified, so the new variables will have names `var11`, `var12`, and so forth. You may now wish to use `rename` to produce more informative variable names. See [D] `rename`.

You can also use `split` to subdivide genuine composites. For example, email addresses such as `tech-support@stata.com` may be split at "@":

```
. split address, p(@)
```

This sequence yields two new variables: `address1`, containing the part of the email address before the "@", such as "tech-support", and `address2`, containing the part after the "@", such as "stata.com". The separator itself, "@", is discarded. Because `generate()` was not specified, the name `address` was used as a stub in naming the new variables. `split` displays the names of new variables created, so you will see quickly whether the number created matches your expectations.

If the details of individuals were of no interest and you wanted only machine names, either

```
. egen machinename = ends(address), tail p(@)
```
or

`. generate machinename = substr(address, strpos(address,"@") + 1,.)`

would be more direct.

Next suppose that a string variable holds names of legal cases that should be split into variables for plaintiff and defendant. The separators could be "V", " V. ", " VS ", and " VS. ". (We assume that any inconsistency in the use of uppercase and lowercase has been dealt with by the string function `strupper();` see [FN] String functions.) Note particularly the leading and trailing spaces in our detailing of separators: the first separator is " V ", for example, not "V", which would incorrectly split "GOLIATH V DAVID" into "GOLIATH ", " DA ", and "ID ". The alternative separators are given as the argument to `parse();`:

`. split case, p(" V " " V. " " VS " " VS. ")`

Again with default naming of variables and recalling that separators are discarded, we expect new variables `case1` and `case2`, with no creation of `case3` or further new variables. Whenever none of the separators specified were found, `case2` would have empty values, so we can check:

`. list case if case2 == ""`

Suppose that a string variable contains fields separated by tabs. For example, `import delimited` leaves tabs unchanged. Knowing that a tab is `char(9)`, we can type

`. split data, p('=char(9)') destring
pchar(9) would not work. The argument to `parse()` is taken literally, but evaluation of functions on the fly can be forced as part of macro substitution.

Finally, suppose that a string variable contains substrings bound in parentheses, such as (1 2 3) (4 5 6). Here we can split on the right parentheses and, if desired, replace those afterward. For example,

`. split data, p(")

`. foreach v in `r(varlist)' {
  replace `v' = `v' + ""*
  }

Stored results

`split` stores the following in `r()`:

Scalars
- `r(nvars)` number of new variables created
- `r(varlist)` names of the newly created variables

Acknowledgments

`split` was written by Nicholas J. Cox of the Department of Geography at Durham University, UK, who is coeditor of the `Stata Journal` and author of `Speaking Stata Graphics`. He in turn thanks Michael Blasnik of Nest Labs for ideas contributed to an earlier jointly written program.
Also see

[D] destring — Convert string variables to numeric variables and vice versa
[D] egen — Extensions to generate
[D] rename — Rename variable
[D] separate — Create separate variables
[FN] String functions
**Description**

`splitsample` splits data into random samples based on a specified number of samples and specified proportions for each sample. Splitting can also be done based on clusters. Sample splitting can also be balanced across specified variables. Balanced splitting can be used for matched treatment assignment.

**Quick start**

Split data into two random samples of equal sizes and generate sample ID variable `svar` with values 1 and 2

`splitsample, generate(svar)`

As above, but with sample ID variable `svar` having values 0 and 1

`splitsample, generate(svar) values(0 1)`

Split data into three random samples of equal sizes and generate sample ID variable `svar` with values 1, 2, and 3

`splitsample, generate(svar) nsplit(3)`

As above, but with sample ID variable `svar` equal to missing (.) whenever any of `y` or `x1-x100` have missing values

`splitsample y x1-x100, generate(svar) nsplit(3)`

Split data into three random samples with the first sample having 25% of the observations, the second having 25%, and the third having 50%

`splitsample, generate(svar) split(0.25 0.25 0.5)`

Same sample split as above, but specify the split using ratios rather than proportions

`splitsample, generate(svar) split(1 1 2)`

As above, but maintain the specified sample-size ratios in each group defined by the variables `agegrp` and `gender`

`splitsample, generate(svar) split(1 1 2) balance(agegrp gender)`

As above, but randomly round sample sizes when samples within an `agegrp` by `gender` group cannot be chosen to satisfy the specified sample-size ratios exactly

`splitsample, generate(svar) split(1 1 2) balance(agegrp gender) rround`

Split data into three samples based on clusters defined by `clustvar`

`splitsample, generate(svar) nsplit(3) cluster(clustvar)`
As above, but maintain the specified sample proportions based on clusters in each group defined by the variables `agegrp` and `gender`, randomly round cluster sample sizes, and display a table showing the cluster sample sizes:

```
splitsample, generate(svar) nsplit(3) cluster(clustvar) ///
   balance(agegrp gender) rround show
```

### Menu

Data > Create or change data > Other variable-creation commands > Split data into random samples

### Syntax

```
splitsample [ varlist ] [ if ] [ in ],
generate(newvar [, replace]) [ options ]
```

`varlist` is checked for missing values, and the sample ID variable `newvar` is set to missing for observations where any variable in `varlist` is missing. `_all` or `*` may be specified for `varlist`.

### Options

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td><strong>generate(newvar [, replace])</strong> create new sample ID variable; optionally replace existing variable</td>
</tr>
<tr>
<td><code>nsplit(#)</code></td>
<td>split into <code>#</code> random samples of equal size</td>
</tr>
<tr>
<td><code>split(numlist)</code></td>
<td>specify <code>numlist</code> of proportions or ratios for the split</td>
</tr>
<tr>
<td><code>rround</code></td>
<td>randomly round sample sizes when an exact split cannot be made</td>
</tr>
<tr>
<td><code>values(numlist)</code></td>
<td>specify <code>numlist</code> of values for sample ID variable</td>
</tr>
<tr>
<td><code>cluster(clustvar)</code></td>
<td>split by clusters defined by <code>clustvar</code>, not observations</td>
</tr>
<tr>
<td><code>balance(balvars)</code></td>
<td>split each group defined by the distinct values of <code>balvars</code> independently based on the specified sample proportions</td>
</tr>
<tr>
<td>Advanced</td>
<td><code>strok</code> evaluate string variables in <code>varlist</code> for missing values; by default, string variables are ignored</td>
</tr>
<tr>
<td><code>rseed(#)</code></td>
<td>specify random-number seed</td>
</tr>
<tr>
<td><code>show</code></td>
<td>display a table showing the sample sizes of the split</td>
</tr>
<tr>
<td><code>percent</code></td>
<td>display percentages in the table showing the split</td>
</tr>
</tbody>
</table>

* `generate()` is required.

Collect is allowed; see [U] 11.1.10 Prefix commands.

### Options

**Main**

- `generate(newvar [, replace])` creates a new variable containing ID values for the random samples. The variable `newvar` is valued 1, 2, ... by default. The option `values(numlist)` can be used to specify different ID values. `generate()` is required.
- `replace` allows any existing variable named `newvar` to be replaced.
**nsplit(#)** splits the data into # random samples of equal size, or as close to equal as possible. If neither **nsplit()** nor **split()** is specified, the data are split into two samples.

**split(numlist)** is an alternative to **nsplit()** for specifying the split. This option splits the data into samples whose sizes are proportional to the values of **numlist**. The values of **numlist** can be any positive number. You can specify proportions that sum to 1, or you can specify integers that define ratios for the sample sizes. Regardless of whether you specify decimals less than 1 or integers, the proportions of the split are given by the values in **numlist** divided by their sum.

**rround** specifies that sample sizes be randomly rounded when an exact split cannot be made. When an exact split can be made, this option does nothing. When **split(numlist)** is specified with **rround**, **numlist** must consist of integers, and the integers should contain no common factors. For instance, use **split(1 1 2)**, not **split(25 25 50)**. See *Methods and formulas* for an explanation.

By default, the sample sizes of the splits are calculated using a deterministic rounding formula. That is, if you repeat the splitting with a different random-number seed, you will get exactly the same sample sizes. Specifying **rround** creates randomly rounded sample sizes such that the expected values of the sample sizes match the specified split proportions exactly.

The option **rround** is designed for use with the **balance()** option when the number of observations in each of the balance groups is small. When group sizes are small (especially when smaller than the number of splits), **rround** ensures that the overall actual sample split proportions closely match the specified split proportions.

**values(numlist)** specifies that **numlist** be used for the values of the sample ID variable rather than the default of 1, 2, ... . The number of values in **numlist** must correspond to the number of samples into which the data are split and must be ascending nonnegative integers.

**cluster(clustvar)** specifies that the data be split by the clusters defined by **clustvar**. That is, all observations in a cluster are kept together in the same split sample. The proportions of the split are based on numbers of clusters, not numbers of observations. **clustvar** can be a numeric or string variable.

**balance(balvars)** specifies that each group defined by the distinct values of **balvars** be split independently based on the specified sample proportions. This ensures a balanced, or roughly balanced, distribution of the **balvars** values across the split samples. When the number of observations (or clusters) in each group is about the same as (or smaller than) the number of split samples, the option **rround** is recommended. **balvars** can be numeric or string variables.

---

**strokr** (applies only when a **varlist** is specified) specifies to check any string variables in **varlist** for missing values. For observations with missing values, the generated sample ID variable is set to missing. By default, string variables in **varlist** are ignored.

**rseed(#)** sets the random-number seed. This option can be used to reproduce results. **rseed(#)** is equivalent to typing **set seed #** prior to running **splitsample**. See **set seed**.

**show** displays a table showing the sample sizes of the split. When **cluster()** is specified, it shows the numbers of clusters in the samples. When **balance(balvars)** is specified, it displays a table in which each row corresponds to a distinct set of values of **balvars** and shown across the columns are the numbers of observations (or clusters) belonging to each split sample for that balance group.

**percent** specifies to display percentages rather than the number of observations (or clusters) in the table. **percent** can only be specified with the option **show**.
Remarks and examples

`splitsample` is useful for dividing data into training, validation, and testing samples for machine learning and automated model-building procedures such as those performed by the `lasso`, `stepwise`, and `nestreg` commands.

`splitsample` with the options `balance()` and `rround` can also be used to do random treatment assignment with matching. See example 3.

Example 1: Splitting by observations

Let's create a dataset with 101 observations and run `splitsample` without any options except the required option giving the name of the sample ID variable to generate. Then we tabulate the newly created variable.

```stata
. set obs 101
Number of observations (n) was 0, now 101.
. splitsample, generate(svar)
. tabulate svar
```

<table>
<thead>
<tr>
<th></th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51</td>
<td>50.50</td>
<td>50.50</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>49.50</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>101</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

By default, `splitsample` splits the data into two samples, with the samples as equal in size as possible.

The option `nsplit(#)` can be used to split the data into as many samples as you want—in this case, three samples.

```stata
. splitsample, generate(svar, replace) nsplit(3)
. tabulate svar
```

<table>
<thead>
<tr>
<th></th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34</td>
<td>33.66</td>
<td>33.66</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>32.67</td>
<td>66.34</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
<td>33.66</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>101</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

The option `split(numlist)` can be specified in place of `nsplit()` to split the data into any proportions you want. Here we specify that we want 25% of the observations in sample 1, 25% in sample 2, and 50% in sample 3.

```stata
. splitsample, generate(svar, replace) split(0.25 0.25 0.50) show
```

<table>
<thead>
<tr>
<th></th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td>24.75</td>
<td>24.75</td>
</tr>
<tr>
<td>2</td>
<td>26</td>
<td>25.74</td>
<td>50.50</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>49.50</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>101</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

It split the data as close as it could to 25% : 25% : 50%. The option `show` displayed the tabulation for us.
Example 2: Splitting by clusters

`splitsample` can also split the data by clusters. Let’s create a cluster variable `clustvar` and split the data into three samples with proportions 25% : 25% : 50% for the numbers of clusters. We also specify the option `show`, which gives a convenient tabulation by numbers of clusters rather than numbers of observations.

```
> set seed 12345
> generate clustvar = runiformint(1, 20)
> splitsample, generate(svar, replace) split(0.25 0.25 0.50) cluster(clustvar) show
  svar | Freq. Percent   Cum.  
 -----|----------------------
       |                 
  1   | 5               25.00   25.00
  2   | 5               25.00   50.00
  3   | 10              50.00   100.00
       |                 
  Total| 20              100.00

Total is number of clusters.
```

Because we had 20 clusters, the split into 25% : 25% : 50% yielded cluster sample sizes that met the specified proportions exactly.

The resulting split by number of observations is, of course, different.

```
. tabulate svar
  svar | Freq. Percent   Cum.  
 -----|----------------------
       |                 
  1   | 34              33.66  33.66
  2   | 21              20.79  54.46
  3   | 46              45.54  100.00
       |                 
  Total| 101             100.00
```

When splitting by clusters, the size of each cluster is ignored.

Example 3: Balanced splitting and treatment assignment

`splitsample` can split the data independently within groups using the option `balance()`. Let’s create two fake categorical variables, one `agegrp` representing eight age–group categories, and a 0/1 variable `gender`.

```
> set seed 12345
> generate agegrp = runiformint(1, 8)
> generate gender = runiformint(0, 1)
```

We want to split the data into four samples, where the first three samples are the same size, and the fourth sample is twice the size of each of the others. We specify `split(1 1 1 2)` using integer ratios. We specify the option `balance(agegrp gender)` to ensure that the distribution of `agegrp` × `gender` is roughly balanced across the four samples. The option `show` is useful for seeing the actual splits of the numbers of observations within each `agegrp` × `gender` group.
. splitsample, generate(svar, replace) split(1 1 1 2)
> balance(agegrp gender) show

note: some groups defined by balance() do not contain every sample value.

<table>
<thead>
<tr>
<th>agegrp</th>
<th>gender</th>
<th>svar 1</th>
<th>svar 2</th>
<th>svar 3</th>
<th>svar 4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

We get a message “some groups defined by balance() do not contain every sample value”. Indeed, all the groups of size three have no observations in sample 2. Because we are splitting the data into four samples, obviously we need at least four observations in a group for every sample to contain at least one observation.

Second, we notice that all groups of the same size are split into the four samples with exactly the same number of observations in each sample. For example, the two groups of size eight (agegrp = 1, gender = 0 and agegrp = 8, gender = 0) both have two observations in each of samples 1 and 3, one observation in sample 2, and three observations in sample 4.

Groups of the same size have exactly the same sample-size splits because, by default, the sample sizes for the splits are calculated using a deterministic formula. If the sizes of the groups vary, this typically would not be an issue. Overall, one would expect the actual split proportions to be close to the specified split proportions. But imagine if all, or almost all, the group sizes were the same. What if the size of each group were eight observations in this example? Every group would be split 2 : 1 : 2 : 3 by observations, yielding actual split proportions of 25% : 12.5% : 25% : 37.5%, which are rather different from the specified split proportions of 20% : 20% : 20% : 40%.
The option `rround` provides a solution for this problem. It randomly rounds the split sample sizes when the split cannot be made exactly.

```
.splitsample, generate(svar, replace) split(1 1 1 2)
> balance(agegrp gender) rround rseed(54321) show
```

We see that the groups of sizes three, eight, and nine now have different splits by numbers of observations. The groups of size five have exactly the same splits by size because they could be divided exactly based on the specified split ratios of $1:1:1:2$.

The option `rround` with `balance()` thus does a “more random” assignment of observations (or clusters), which is important when the sizes of the balance groups are small. When the sizes of the balance groups are large, and the sizes of the groups vary, splits made with or without `rround` will be similar.

Note that `rround` with `balance()` is suitable for random treatment assignment with matching defined by values of the balance variables.

The computational procedure for option `rround` first randomly assigns as many observations to the split samples as it can to match the specified split proportions exactly. Leftover observations are assigned to samples by dividing them randomly based on the specified split ratios. Splitting ratios must be specified as integers to facilitate this method of splitting the leftovers. See *Methods and formulas*.

### Example 4: Missing values

`varlist` can be specified with `splitsample` to handle missing values. Let’s say we want to divide our data into training and validation samples for a lasso or other procedure. Imagine that the variables in the lasso have more than a few missing values. Specifying these variables as `varlist` for
splitsample means that the sample ID variable created will have missing values whenever any of the variables in varlist are missing.

Here’s an illustration. We create a couple of variables with missing values.

. set seed 1234
. generate y = runiform()
. replace y = . if runiform() < 0.1
(11 real changes made, 11 to missing)
. generate x = runiform()
. replace x = . if runiform() < 0.1
(15 real changes made, 15 to missing)

Then split the data specifying these variables to be checked for missing:

. splitsample y x, generate(svar, replace)
. tabulate svar, miss

<table>
<thead>
<tr>
<th>svar</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>37.62</td>
<td>37.62</td>
</tr>
<tr>
<td>2</td>
<td>38</td>
<td>37.62</td>
<td>75.25</td>
</tr>
<tr>
<td>.</td>
<td>25</td>
<td>24.75</td>
<td>100.00</td>
</tr>
</tbody>
</table>

The split was done exactly for the observations without missing values.

Stored results

splitsample stores the following in r():

Scalars
- r(N) total number of observations
- r(N_clust) total number of clusters
- r(n_samples) number of split samples

Macros
- r(clustvar) name of cluster variable
- r(balancevars) names of balance variables
- r(rngstate) random-number state used

Methods and formulas

Let $r_1, r_2, \ldots, r_K$ be the arguments to split(numlist). If the split is specified using nsplit(#), then we set each $r_k = 1$, and the number of split samples is $K = #$. The split sample proportions are

$$p_k = \frac{r_k}{R} \quad \text{where} \quad R = \sum_{i=1}^{K} r_i$$

The cumulative proportions are

$$s_k = \sum_{i=1}^{k} p_i$$
For the default deterministic rounding, we calculate cumulative sample sizes:

\[ M_k = \text{round}(N s_k) \]

where \( N \) is the total number of observations or the number of clusters, and \( \text{round}(\cdot) \) is Stata's \texttt{round()} function. When the option \texttt{balance()} is specified, \( N \) is the number of observations or clusters in a single balance group. The sample sizes \( N_1, N_2, \ldots, N_K \) are given by

\[
\begin{align*}
N_1 &= M_1 \\
N_k &= M_k - M_{k-1} \quad \text{for } k = 2, \ldots, K
\end{align*}
\]

When the option \texttt{rround} is specified for random rounding, we first divide \( N \), the number of observations or clusters, as follows:

\[ N = cR + d \]

where \( R \) is the sum of \( r_1, r_2, \ldots, r_K \); \( c \) is a nonnegative integer; and \( 0 \leq d < R \). In other words, \( cR \) observations can be split into \( K \) samples matching the specified split proportions exactly. We randomly pick \( cR \) observations and assign them to the samples. The leftover \( d \) observations are randomly placed in \( R \) bins without replacement, where the first \( r_1 \) bins represent sample 1, the next \( r_2 \) bins represent sample 2, and so on.

The computational procedure for random rounding thus requires \( r_1, r_2, \ldots, r_K \) to be integers and also requires \( R \leq N \). To reduce the variance of the random rounding, the integers \( r_1, r_2, \ldots, r_K \) should have no common factors.

Also see

[D] \texttt{sample} — Draw random sample
**Description**

`stack` stacks the variables in `varlist` vertically, resulting in a dataset with variables `newvars` and 
\[ N_v \cdot (N_v/N_n) \] observations, where \( N_v \) is the number of variables in `varlist` and \( N_n \) is the number in `newvars`. `stack` creates the new variable `_stack` identifying the groups.

**Quick start**

Replace data in memory with \( v, v2 \) appended to \( v1 \) and identify original variable by order in `_stack`

```
stack v1 v2, into(v)
```

As above, but with \( v1 \) appended to \( v2 \) and do not display warning that data in memory will be replaced

```
stack v2 v1, into(v) clear
```

As above, but save result in \( v2 \)

```
stack v2 v1, group(2) clear
```

Append \( v2 \) to \( v1 \) and \( v4 \) to \( v3 \) and save result in `newv1` and `newv2`

```
stack v1 v3 v2 v4, into(newv1 newv2) clear
```

As above, but save results in \( v1 \) and \( v3 \)

```
stack v1 v3 v2 v4, group(2) clear
```

**Menu**

Data > Create or change data > Other variable-transformation commands > Stack data

**Syntax**

```
stack varlist [if] [in], {into(newvars) | group(#)} [options]
```

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
<td></td>
</tr>
<tr>
<td><em>into(newvars)</em></td>
<td>identify names of new variables to be created</td>
</tr>
<tr>
<td>*group(#)</td>
<td>stack # groups of variables in <code>varlist</code></td>
</tr>
<tr>
<td>clear</td>
<td>clear dataset from memory</td>
</tr>
<tr>
<td>wide</td>
<td>keep variables in <code>varlist</code> that are not specified in <code>newvars</code></td>
</tr>
</tbody>
</table>

* Either `into(newvars)` or `group(#)` is required.
Options

into(newvars) identifies the names of the new variables to be created. into() may be specified using variable ranges (for example, into(v1-v3)). Either into() or group(), but not both, must be specified.

group(#) specifies the number of groups of variables in varlist to be stacked. The created variables will be named according to the first group in varlist. Either group() or into(), but not both, must be specified.

clear indicates that it is okay to clear the dataset in memory. If you do not specify this option, you will be asked to confirm your intentions.

wide includes any of the original variables in varlist that are not specified in newvars in the resulting data.

Remarks and examples

Example 1: Illustrating the concept

This command is best understood by examples. We begin with artificial but informative examples and end with useful examples.

. use https://www.stata-press.com/data/r17/stackxmpl
. list
   +----+----+----+----+
   | a   | b   | c   | d   |
   +----+----+----+----+
   | 1   | 2   | 3   | 4   |
   | 5   | 6   | 7   | 8   |
   +----+----+----+----+
. stack a b c d, into(e f) clear
. list
   +----+----+----+----+
   | _stack | e   | f   |
   +----+----+----+----+
   | 1   | 1   | 2   |
   | 2   | 5   | 6   |
   | 3   | 2   | 3   | 4   |
   | 4   | 2   | 7   | 8   |
   +----+----+----+----+

We formed the new variable e by stacking a and c, and we formed the new variable f by stacking b and d. _stack is automatically created and set equal to 1 for the first (a, b) group and equal to 2 for the second (c, d) group. (When _stack==1, the new data e and f contain the values from a and b. When _stack==2, e and f contain values from c and d.)

There are two groups because we specified four variables in the varlist and two variables in the into list, and 4/2 = 2. If there were six variables in the varlist, there would be 6/2 = 3 groups. If there were also three variables in the into list, there would be 6/3 = 2 groups. Specifying six variables in the varlist and four variables in the into list would result in an error because 6/4 is not an integer.
Example 2: Stacking a variable multiple times

Variables may be repeated in the `varlist`, and the `varlist` need not contain all the variables:

```
use https://www.stata-press.com/data/r17/stackxmpl, clear
list
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```
stack a b a c, into(a bc) clear
list
```

<table>
<thead>
<tr>
<th>_stack</th>
<th>a</th>
<th>bc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

a was stacked on a and called a, whereas b was stacked on c and called bc.

If we had wanted the resulting variables to be called simply a and b, we could have used

```
stack a b a c, group(2) clear
```

which is equivalent to

```
stack a b a c, into(a b) clear
```

Example 3: Keeping the original variables

In this artificial but informative example, the wide option includes the variables in the original dataset that were specified in `varlist` in the output dataset:

```
use https://www.stata-press.com/data/r17/stackxmpl, clear
list
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

```
stack a b c d, into(e f) clear wide
list
```

<table>
<thead>
<tr>
<th>_stack</th>
<th>e</th>
<th>f</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>.</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>.</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
In addition to the stacked e and f variables, the original a, b, c, and d variables are included. They are set to missing where their values are not appropriate.

Example 4: Using wide with repeated variables

This is the last artificial example. When you specify the wide option and repeat the same variable name in both the varlist and the into list, the variable will contain the stacked values:

```
. use https://www.stata-press.com/data/r17/stackxmpl, clear
. list
   +-----+-----+-----+-----+
   | a   | b   | c   | d   |
   +-----+-----+-----+-----+
   | 1   | 2   | 3   | 4   |
   +-----+-----+-----+-----+
   | 5   | 6   | 7   | 8   |
   +-----+-----+-----+-----+
. stack a b a c, into(a bc) clear wide
. list
   +-----+-----+-----+-----+
   | _stack | a | bc | b | c |
   +-----+-----+-----+-----+
   | 1     | 1 | 2 | 2 | . |
   +-----+-----+-----+-----+
   | 1     | 5 | 6 | 6 | . |
   +-----+-----+-----+-----+
   | 2     | 1 | 3 | . | 3 |
   +-----+-----+-----+-----+
   | 2     | 5 | 7 | . | 7 |
   +-----+-----+-----+-----+
```

Example 5: Using stack to make graphs

We want one graph of y against x1 and y against x2. We might be tempted to type scatter y x1 x2, but that would graph y against x2 and x1 against x2. One solution is to type

```
. save mydata
. stack y x1 y x2, into(yy x12) clear
. generate y1 = yy if _stack==1
. generate y2 = yy if _stack==2
. scatter y1 y2 x12
. use mydata, clear
```

The names yy and x12 are supposed to suggest the contents of the variables. yy contains (y,y), and x12 contains (x1,x2). We then make y1 defined at the x1 points but missing at the x2 points—graphing y1 against x12 is the same as graphing y against x1 in the original dataset. Similarly, y2 is defined at the x2 points but missing at x1—graphing y2 against x12 is the same as graphing y against x2 in the original dataset. Therefore, scatter y1 y2 x12 produces the desired graph.

Example 6: Plotting cumulative distributions

We wish to graph y1 against x1 and y2 against x2 on the same graph. The logic is the same as above, but let’s go through it. Perhaps we have constructed two cumulative distributions by using cumul (see [R] cumul):
. use https://www.stata-press.com/data/r17/citytemp
   (City temperature data)
   . cumul tempjan, gen(cjan)
   . cumul tempjuly, gen(cjuly)

We want to graph both cumulatives in the same graph; that is, we want to graph \( cjan \) against \( tempjan \) and \( cjuly \) against \( tempjuly \). Remember that we could graph the \( tempjan \) cumulative by typing

\[
\text{. scatter cjan tempjan, c(l) m(o) sort}
\]

(output omitted)

We can graph the \( tempjuly \) cumulative similarly. To obtain both on the same graph, we must stack the data:

\[
\text{. stack cjuly tempjuly cjan tempjan, into(c temp) clear}
\]

\[
\text{. generate cjan = c if _stack==1}
\]

(958 missing values generated)

\[
\text{. generate cjuly = c if _stack==2}
\]

(958 missing values generated)

\[
\text{. scatter cjan cjuly temp, c(l l) m(o o) sort}
\]

(output omitted)

Alternatively, if we specify the wide option, we do not have to regenerate \( cjan \) and \( cjuly \) because they will be created automatically:

\[
\text{. use https://www.stata-press.com/data/r17/citytemp, clear}
\]

(City temperature data)

\[
\text{. cumul tempjan, gen(cjan)}
\]

\[
\text{. cumul tempjuly, gen(cjuly)}
\]

\[
\text{. stack cjuly tempjuly cjan tempjan, into(c temp) clear wide}
\]

\[
\text{. scatter cjan cjuly temp, c(l l) m(o o) sort}
\]

(output omitted)

Technical note

There is a third way, not using the wide option, that is exceedingly tricky but is sometimes useful:

\[
\text{. use https://www.stata-press.com/data/r17/citytemp, clear}
\]

(City temperature data)

\[
\text{. cumul tempjan, gen(cjan)}
\]

\[
\text{. cumul tempjuly, gen(cjuly)}
\]

\[
\text{. stack cjuly tempjuly cjan tempjan, into(c temp) clear}
\]

\[
\text{. sort _stack temp}
\]

\[
\text{. scatter c temp, c(L) m(o)}
\]

(output omitted)

Note the use of connect’s capital L rather than lowercase l option. c(L) connects points only from left to right; because the data are sorted by \_stack temp, temp increases within the first group (cjuly vs. tempjuly) and then starts again for the second (cjan vs. tempjan); see \{G-4\} connectstyle.
Reference

Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. College Station, TX: Stata Press.

Also see

[D] **contract** — Make dataset of frequencies and percentages

[D] **reshape** — Convert data from wide to long form and vice versa

[D] **xpose** — Interchange observations and variables
Description

statsby collects statistics from command across a by list. Typing

```
.statsby exp_list, by(varname): command
```

executes command for each group identified by varname, building a dataset of the associated values from the expressions in exp_list. The resulting dataset replaces the current dataset, unless the saving() option is supplied. varname can refer to a numeric or a string variable.

command defines the statistical command to be executed. Most Stata commands and user-written programs can be used with statsby, as long as they follow standard Stata syntax and allow the if qualifier; see [U] 11 Language syntax. The by prefix cannot be part of command.

exp_list specifies the statistics to be collected from the execution of command. If no expressions are given, exp_list assumes a default depending upon whether command changes results in e() and r(). If command changes results in e(), the default is _b. If command changes results in r() (but not e()), the default is all the scalars posted to r(). It is an error not to specify an expression in exp_list otherwise.

Quick start

Replace data in memory with estimates of the coefficient of x and constant for each value of catvar

```
statsby, by(catvar): regress y x
```

As above, but name new variables b and cons

```
statsby b=_b[x] cons=_b[_cons], by(catvar): regress y x
```

Add standard errors of the estimates and use default variable names

```
statsby _b _se, by(catvar): regress y x
```

As above, but retain data in memory and save estimates to myest.dta

```
statsby _b _se, by(catvar) saving(myest): regress y x
```

As above, and include estimate for entire dataset

```
statsby _b _se, by(catvar) saving(myest) total: regress y x
```

Note: Any command that accepts the statsby prefix may be substituted for regress above.

Menu

Statistics > Other > Collect statistics for a command across a by list
Statsby — Collect statistics for a command across a by list

Syntax

```
statsby [exp_list] [ , options]: command
```

**options**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main</strong></td>
</tr>
<tr>
<td><code>*by(varlist [, missing])</code></td>
</tr>
<tr>
<td><strong>Options</strong></td>
</tr>
<tr>
<td><code>clear</code></td>
</tr>
<tr>
<td><code>saving(filename,...)</code></td>
</tr>
<tr>
<td><code>total</code></td>
</tr>
<tr>
<td><code>subsets</code></td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
</tr>
<tr>
<td><code>nodots</code></td>
</tr>
<tr>
<td><code>dots(#)</code></td>
</tr>
<tr>
<td><code>noisily</code></td>
</tr>
<tr>
<td><code>trace</code></td>
</tr>
<tr>
<td><code>nolegend</code></td>
</tr>
<tr>
<td><code>verbose</code></td>
</tr>
<tr>
<td><strong>Advanced</strong></td>
</tr>
<tr>
<td><code>basepop(exp)</code></td>
</tr>
<tr>
<td><code>force</code></td>
</tr>
<tr>
<td><code>forcedrop</code></td>
</tr>
</tbody>
</table>

* `by()` is required on the dialog box because `statsby` is useful to the interactive user only when using `by()`. All weight types supported by `command` are allowed except `pweights`; see [U] 11.1.6 weight.

**exp_list** contains

- `(name: elist)`
- `elist`
- `eexp`

**elist** contains

- `newvarname = (exp)`
- `(exp)`

**eexp** is

- `specname`
- `[eqno]specname`

**specname** is

- `_b`
- `_b[]`
- `_se`
- `_se[]`

**eqno** is

- `##`
- `name`

**exp** is a standard Stata expression; see [U] 13 Functions and expressions.

Distinguish between `[]`, which are to be typed, and `[]`, which indicate optional arguments.
by(varlist [, missing]) specifies a list of existing variables that would normally appear in the by varlist: section of the command if you were to issue the command interactively. By default, statsby ignores groups in which one or more of the by() variables is missing. Alternatively, missing causes missing values to be treated like any other values in the by-groups, and results from the entire dataset are included with use of the subsets option. If by() is not specified, command will be run on the entire dataset. varlist can contain both numeric and string variables.

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

saving(filename[, suboptions]) creates a Stata data file (.dta file) consisting of (for each statistic in exp_list) a variable containing the replicates.

double specifies that the results for each replication be stored as doubles, meaning 8-byte reals. By default, they are stored as floats, meaning 4-byte reals.

every(#) specifies that results be written to disk every #th replication. every() should be specified in conjunction with saving() only when command takes a long time for each replication. This will allow recovery of partial results should your computer crash. See [P] postfile.

total specifies that command be run on the entire dataset, in addition to the groups specified in the by() option.

subsets specifies that command be run for each group defined by any combination of the variables in the by() option.

nodots and dots(#) specify whether to display replication dots. By default, one dot character is displayed for each by-group. A red ‘x’ is displayed if command returns an error or if any value in exp_list is missing. You can also control whether dots are printed using set dots; see [R] set.

nodots suppresses display of the replication dots.

dots(#) displays dots every # replications. dots(0) is a synonym for nodots.

noisily causes the output of command to be displayed for each by-group. This option implies the nodots option.

trace causes a trace of the execution of command to be displayed. This option implies the noisily option.

nolegend suppresses the display of the table legend, which identifies the rows of the table with the expressions they represent.

verbose requests that the full table legend be displayed. By default, coefficients and standard errors are not displayed.

basepop(exp) specifies a base population that statsby uses to evaluate the command and to set up for collecting statistics. The default base population is the entire dataset, or the dataset specified by any if or in conditions specified on the command.
One situation where basepop() is useful is collecting statistics over the panels of a panel dataset by using an estimator that works for time series, but not panel data, for example,

```
.statsby, by(mypanels) basepop(mypanels==2): arima ...
```

force suppresses the restriction that command not be a svy command. statsby does not perform subpopulation estimation for survey data, so it should not be used with svy. statsby reports an error when it encounters svy in command if the force option is not specified. This option is seldom used, so use it only if you know what you are doing.

forcedrop forces statsby to drop all observations except those in each by-group before calling command for the group. This allows statsby to work with user-written programs that completely ignore if and in but do not return an error when either is specified. forcedrop is seldom used.

**Remarks and examples**

Remarks are presented under the following headings:

- Collecting coefficients and standard errors
- Collecting stored results
- All subsets

**Collecting coefficients and standard errors**

➢ **Example 1**

We begin with an example using auto2.dta. In this example, we want to collect the coefficients from a regression in which we model the price of a car on its weight, length, and mpg. We want to run this model for both domestic and foreign cars. We can do this easily by using statsby with the extended expression _b.

```plaintext
.use https://www.stata-press.com/data/r17/auto2
(1978 automobile data)
.statsby _b, by(foreign) verbose nodots: regress price weight length mpg
```

Command: regress price weight length mpg

```
  _b_weight: _b[weight]
  _b_length: _b[length]
  _b_mpg: _b[mpg]
  _b_cons: _b[_cons]
  By: foreign

.list

<table>
<thead>
<tr>
<th>foreign</th>
<th>_b_weight</th>
<th>_b_length</th>
<th>_b_mpg</th>
<th>_b_cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domestic</td>
<td>6.767233</td>
<td>-109.9518</td>
<td>142.7663</td>
<td>2359.475</td>
</tr>
<tr>
<td>Foreign</td>
<td>4.784841</td>
<td>13.39052</td>
<td>-18.4072</td>
<td>-6497.49</td>
</tr>
</tbody>
</table>
```

If we were interested only in the coefficient of a particular variable, such as mpg, we would specify that particular coefficient; see [U] 13.5 Accessing coefficients and standard errors.
. use https://www.stata-press.com/data/r17/auto2, clear
(1978 automobile data)

. statsby mpg=_b[mpg], by(foreign) nodots: regress price weight length mpg
   Command: regress price weight length mpg
            mpg: _b[mpg]
            By: foreign

. list

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Domestic</td>
<td>142.7663</td>
</tr>
<tr>
<td>2.</td>
<td>Foreign</td>
<td>-18.4072</td>
</tr>
</tbody>
</table>

The extended expression _se indicates that we want standard errors.

. use https://www.stata-press.com/data/r17/auto2, clear
(1978 automobile data)

. statsby _se, by(foreign) verbose nodots: regress price weight length mpg
   Command: regress price weight length mpg
            _se_weight: _se[weight]
            _se_length: _se[length]
            _se_mpg: _se[mpg]
            _se_cons: _se[_cons]
            By: foreign

. list

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Domestic</td>
<td>1.226326</td>
<td>39.48193</td>
<td>134.7221</td>
</tr>
<tr>
<td>2.</td>
<td>Foreign</td>
<td>1.670006</td>
<td>50.70229</td>
<td>59.37442</td>
</tr>
</tbody>
</table>

Example 2

For multiple-equation estimations, we can use [eqno]_b ([eqno]_se) to get the coefficients (standard errors) of a specific equation or use _b (_se) to get the coefficients (standard errors) of all the equations. To demonstrate, we use heckman and a slightly different dataset.

. use https://www.stata-press.com/data/r17/statsby, clear
. statsby _b, by(group) verbose nodots: heckman price mpg, sel(trunk)
   Command: heckman price mpg, sel(trunk)
            price_b_mpg: [price]_b[mpg]
            price_b_cons: [price]_b[_cons]
            select_b_tr-k: [select]_b[trunk]
            select_b_cons: [select]_b[_cons]
            _eq3_b_athrho: [/]_b[athrho]
            _eq3_b_insi-a: [/]_b[lnsigma]
            By: group
To collect the coefficients of the first equation only, we would specify \[\text{price}\] \_b instead of \_b.

```
use https://www.stata-press.com/data/r17/statsby, clear
statsby \[\text{price}\] \_b, by(group) verbose nodots: heckman price mpg, sel(trunk)
```

Technical note

If `command` fails on one or more groups, `statsby` will capture the error messages and ignore those groups.

Collecting stored results

Many Stata commands store results of calculations; see [U] 13.6 Accessing results from Stata commands. `statsby` can collect the stored results and expressions involving these stored results, too. Expressions must be bound in parentheses.
Example 3

Suppose that we want to collect the mean and the median of `price`, as well as their ratios, and we want to collect them for both domestic and foreign cars. We might type

```
. use https://www.stata-press.com/data/r17/auto2, clear
(1978 automobile data)
. statsby mean=r(mean) median=r(p50) ratio=(r(mean)/r(p50)), by(foreign) nodots:
> summarize price, detail
```

```
Command: summarize price, detail
   mean: r(mean)
   median: r(p50)
   ratio: r(mean)/r(p50)
By: foreign
```

```
. list

<table>
<thead>
<tr>
<th>foreign</th>
<th>mean</th>
<th>median</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domestic</td>
<td>6072.423</td>
<td>4782.5</td>
<td>1.269717</td>
</tr>
<tr>
<td>Foreign</td>
<td>6384.682</td>
<td>5759</td>
<td>1.108644</td>
</tr>
</tbody>
</table>
```

Technical note

In `exp_list`, `newvarname` is not required. If no new variable name is specified, `statsby` names the new variables `_stat_1`, `_stat_2`, and so forth.

All subsets

Example 4

When there are two or more variables in `by(varlist)`, we can execute `command` for any combination, or subset, of the variables in the `by()` option by specifying the `subsets` option.

```
. use https://www.stata-press.com/data/r17/auto2, clear
(1978 automobile data)
. statsby mean=r(mean) median=r(p50) n=r(N), by(foreign rep78) subsets nodots:
> summarize price, detail
```

```
Command: summarize price, detail
   mean: r(mean)
   median: r(p50)
   n: r(N)
By: foreign rep78
```
In the above dataset, observation 6 is for domestic cars, regardless of the repair record; observation 10 is for foreign cars, regardless of the repair record; observation 11 is for both foreign cars and domestic cars given that the repair record is 1; and the last observation is for the entire dataset.

Technical note

To see the output from command for each group identified in the by() option, we can use the noisily option.

```
use https://www.stata-press.com/data/r17/auto2, clear
(1978 automobile data)
statsby mean=r(mean) se=(r(sd)/sqrt(r(N))), by(foreign) noisily nodots:
> summarize price
```

```
Variable | Obs | Mean | Std. dev. | Min | Max
---------|-----|------|-----------|-----|-----
price | 74 | 6165.257 | 2949.496 | 3291 | 15906
```

Statsby legend:

```
Command: summarize price
  mean: r(mean)
  se: r(sd)/sqrt(r(N))
  By: foreign
Statsby groups
running (summarize price) on group 1
```
. summarize price

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>52</td>
<td>6072.423</td>
<td>3097.104</td>
<td>3291</td>
<td>15906</td>
</tr>
</tbody>
</table>

running (summarize price) on group 2

. summarize price

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>22</td>
<td>6384.682</td>
<td>2621.915</td>
<td>3748</td>
<td>12990</td>
</tr>
</tbody>
</table>

. list

<table>
<thead>
<tr>
<th>foreign</th>
<th>mean</th>
<th>se</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domestic</td>
<td>6072.423</td>
<td>429.4911</td>
</tr>
<tr>
<td>Foreign</td>
<td>6384.682</td>
<td>558.9942</td>
</tr>
</tbody>
</table>

Acknowledgment

Speed improvements in \texttt{statsby} were based on code written by Michael Blasnik of Nest Labs.

References


Also see

[D] \texttt{by} — Repeat Stata command on subsets of the data

[D] \texttt{collapse} — Make dataset of summary statistics

[P] \texttt{postfile} — Post results in Stata dataset

[R] \texttt{bootstrap} — Bootstrap sampling and estimation

[R] \texttt{jackknife} — Jackknife estimation

[R] \texttt{permute} — Monte Carlo permutation tests
sysuse — Use shipped dataset

Description

sysuse filename loads the specified Stata-format dataset that was shipped with Stata or that is stored along the ado-path. If filename is specified without a suffix, .dta is assumed.

sysuse dir lists the names of the datasets shipped with Stata plus any other datasets stored along the ado-path.

Quick start

List example datasets installed with Stata
sysuse dir

Use auto.dta example dataset installed with Stata
sysuse auto

As above, but clear current dataset from memory first
sysuse auto, clear

Menu

File  >  Example datasets...
Syntax

Use example dataset installed with Stata

```
sysuse "filename", clear
```

List example Stata datasets installed with Stata

```
sysuse dir [, all]
```

Options

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

all specifies that all datasets be listed, even those that include an underscore (\_) in their name. By default, such datasets are not listed.

Remarks and examples

Remarks are presented under the following headings:

- Typical use
- A note concerning shipped datasets
- Using user-installed datasets
- How sysuse works

Typical use

A few datasets are included with Stata and are stored in the system directories. These datasets are often used in the help files to demonstrate a certain feature.

Typing

```
.sysuse dir
```

lists the names of those datasets. One such dataset is `lifeexp.dta`. If you simply type `use lifeexp`, you will see

```
use lifeexp
file lifeexp.dta not found
r(601);
```

Type `sysuse`, however, and the dataset is loaded:

```
.sysuse lifeexp
(Life expectancy, 1998)
```

The datasets shipped with Stata are stored in different folders (directories) so that they do not become confused with your datasets.
A note concerning shipped datasets

Not all the datasets used in the manuals are shipped with Stata. To obtain the other datasets, see [D] webuse.

The datasets used to demonstrate Stata are often fictional. If you want to know whether a dataset is real or fictional, and its history, load the dataset and type

```
notes
```

A few datasets have no notes. This means that the datasets are believed to be real, but that they were created so long ago that information about their original source has been lost. Treat such datasets as if they were fictional.

Using user-installed datasets

Any datasets you have installed using net or ssc (see [R] net and [R] ssc) can be listed by typing sysuse dir and can be loaded using sysuse filename.

Any datasets you store in your personal ado folder (see [P] sysdir) are also listed by sysuse dir and can be loaded using sysuse filename.

How sysuse works

sysuse simply looks across the ado-path for .dta files; see [P] sysdir.

By default, sysuse dir does not list a dataset that contains an underscore (_) in its name. By convention, such datasets are used by ado-files to achieve their ends and probably are not of interest to you. If you type sysuse dir, all, then all datasets are listed.

Stored results

sysuse dir stores in the macro r(files) the list of dataset names.

sysuse filename stores in the macro r(fn) the filename, including the full path specification.

Also see

[D] webuse — Use dataset from Stata website
[D] use — Load Stata dataset
[P] findfile — Find file in path
[P] sysdir — Query and set system directories
[R] net — Install and manage community-contributed additions from the Internet
[R] ssc — Install and uninstall packages from SSC
type — Display contents of a file

Description

type lists the contents of a file stored on disk. This command is similar to the Windows type command and the Unix more(1) or pg(1) commands.

In Stata for Mac and Stata for Unix, cat is a synonym for type.

On all platforms, Stata understands a leading “~” as an abbreviation for the home directory.

Quick start

Display contents of myfile.txt in the Results window

```
type myfile.txt
```

As above, but display myfile.txt saved in ~/mydir/mysubdir using Stata for Windows

```
type ~/mydir/mysubdir/myfile.txt
```

As above, but using Stata for Mac or Unix

```
type ~/mydir/mysubdir/myfile.txt
```

Display contents of my file.txt

```
type "my file.txt"
```

Display the first 20 lines of myfile.txt

```
type myfile.txt, lines(20)
```

Syntax

```
type ["] filename[" [ , options ]
```

Note: Double quotes must be used to enclose filename if the name contains blanks.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asis</td>
<td>show file as is; default is to display files with suffix .smcl or .sthlp as SMCL</td>
</tr>
<tr>
<td>smcl</td>
<td>display file as SMCL; default for files with suffix .smcl or .sthlp</td>
</tr>
<tr>
<td>showtabs</td>
<td>display tabs as &lt;T&gt; rather than being expanded</td>
</tr>
<tr>
<td>starbang</td>
<td>list lines in the file that begin with “*!”</td>
</tr>
<tr>
<td>lines(#)</td>
<td>list first # lines</td>
</tr>
</tbody>
</table>
Options

asis specifies that the file be shown exactly as it is. The default is to display files with the suffix .smcl or .sthlp as SMCL, meaning that the SMCL directives are interpreted and properly rendered. Thus type can be used to look at files created by the log using command.

smcl specifies that the file be displayed as SMCL, meaning that the SMCL directives are interpreted and properly rendered. This is the default for files with the suffix .smcl or .sthlp.

showtabs requests that any tabs be displayed as <T> rather than being expanded.

starbang lists only the lines in the specified file that begin with the characters “*!””. Such comment lines are typically used to indicate the version number of ado-files, class files, etc. starbang may not be used with SMCL files.

lines(#) lists the first # lines of a file. lines() is ignored if the file is displayed as SMCL or if # is less than or equal to 0.

Remarks and examples

Example 1

We have raw data containing the level of Lake Victoria Nyanza and the number of sunspots during the years 1902–1921 stored in a file called sunspots.raw. We want to read this dataset into Stata by using infile, but we cannot remember the order in which we entered the variables. We can find out by using the type command:

    . type sunspots.raw
    1902 -10  5  1903   13  24  1904   18  42
    1905   15   63  1906   29  54  1907   21  62
    1908   10   49  1909    8  44  1910    1  19
    1911    7    6  1912  -11  4    1913    3   1
    1914   -2    6  1915    4  47  1916   15  57
    1917   35  104  1918   27  81  1919    8  64
    1920    3    6  1921   -5  25

Looking at this output, we now remember that the variables are entered year, level, and number of sunspots. We can read this dataset by typing infile year level spots using sunspots.

If we wanted to see the tabs in sunspots.raw, we could type

    . type sunspots.raw, showtabs
    1902 -10  5<T>1903  13 24<T>1904  18 42
    1905  15 63<T>1906  29 54<T>1907  21 62
    1908  10 49<T>1909  8 44<T>1910  1 19
    1911  7 6<T>1912  11 4<T>1913  3 1
    1914  2 10<T>1915  4 47<T>1916  15 57
    1917 35 104<T>1918  27 81<T>1919  8 64
    1920  3 38<T>1921  5 25
Example 2

In a previous Stata session, we typed `log using myres` and created `myres.smcl`, containing our results. We can use `type` to list the log:

```
. type myres.smcl

name: <unnamed>
log: /work/peb/dof/myres.smcl
log type: smcl
opened on: 20 Jan 2021, 15:37:48
```

```
use lbw
(Hosmer & Lemeshow data)
.logistic low age lwt i.race smoke ptl ht ui
Logistic regression
   Number of obs = 189
   LR chi2(8)    = 33.22
   Prob > chi2  = 0.0001
Log likelihood = -100.724  Pseudo R2 = 0.1416
(output omitted)
.estat gof
Logistic model for low, goodness-of-fit test
(output omitted)
```

```
. log close
name: <unnamed>
log: /work/peb/dof/myres.smcl
log type: smcl
closed on: 20 Jan 2021, 15:38:30
```

We could also use `view` to look at the log; see `[R] view`.

Also see

[D] cd — Change directory
[D] copy — Copy file from disk or URL
[D] dir — Display filenames
[D] erase — Erase a disk file
[D] mkdir — Create directory
[D] rmdir — Remove directory
[D] shell — Temporarily invoke operating system
[P] viewsource — View source code
[R] translate — Print and translate logs
[R] view — View files and logs
[U] 11.6 Filenaming conventions
**Description**

The `unicode` command provides utilities to help you work with Unicode strings in your data. If you have only plain ASCII characters in your data (a–z, A–Z, 0–9, and typical punctuation characters), you can stop reading now. Otherwise, continue with Remarks and examples below.

**Remarks and examples**

We recommend that you start with some overview documentation. First, you should read **[U] 12.4.2 Handling Unicode strings**, which will explain the difference between ASCII and Unicode and provide detailed advice on working with Unicode strings in Stata. In that section, you will learn about locales, encodings, sorting, and Unicode-specific string functions. For a general overview of Unicode-specific advice, see `help unicode advice`.

Second, if you have datasets, do-files, ado-files, or other files that you used with Stata 13 or earlier and those files contain characters other than plain ASCII such as accented characters, Chinese, Japanese, or Korean (CJK) characters, Cyrillic characters, and the like, you should read **[D] unicode translate**.

`unicode` provides the following utilities:

- **[D] unicode translate** — Translate files to Unicode
- **[D] unicode encoding** — Unicode encoding utilities
- **[D] unicode locale** — Unicode locale utilities
- **[D] unicode collator** — Language-specific Unicode collators
- **[D] unicode convertfile** — Low-level file conversion between encodings

You may also find `help encodings` useful if you need to choose an encoding when converting a string from extended ASCII to Unicode.

**Also see**

- **[D] unicode collator** — Language-specific Unicode collators
- **[D] unicode convertfile** — Low-level file conversion between encodings
- **[D] unicode encoding** — Unicode encoding utilities
- **[D] unicode locale** — Unicode locale utilities
- **[D] unicode translate** — Translate files to Unicode
- **[U] 12.4.2 Handling Unicode strings**
Title

- **unicode collator** — Language-specific Unicode collators

Description

`unicode collator list` lists the subset of locales that have language-specific collators for the Unicode string comparison functions: `ustrcompare()`, `ustrcompareex()`, `ustrsortkey()`, and `ustrsortkeyex()`.

Syntax

```
unicode collator list [pattern]
```

`pattern` is one of `_all`, `*`, `_name*`, `_name`, or `name*`. If you specify nothing, `_all`, or `*`, then all results will be listed. `_name*` lists all results containing `name`; `_name` lists all results ending with `name`; and `name*` lists all results starting with `name`.

Remarks and examples

Remarks are presented under the following headings:

- **Overview of collation**
- **The role of locales in collation**
- **Further controlling collation**

Overview of collation

Collation is the process of comparing and sorting Unicode character strings as a human might logically order them. We call this ordering strings in a language-sensitive manner. To do this, Stata uses a Unicode tool known as the Unicode collation algorithm, or UCA.

To perform language-sensitive string sorts, you must combine `ustrsortkey()` or `ustrsortkeyex()` with `sort`. It is a complicated process and there are several issues about which you need to be aware. For details, see [U] 12.4.2.5 Sorting strings containing Unicode characters. To perform language-sensitive string comparisons, you can use `ustrcompare()` or `ustrcompareex()`.

For details about the UCA, see http://www.unicode.org/reports/tr10/.

The role of locales in collation

During collation, Stata can use the default collator or it can perform language-sensitive string comparisons or sorts that require knowledge of a locale.

A locale identifies a community with a certain set of preferences for how their language should be written; see [U] 12.4.2.4 Locales in Unicode. For example, in English, the uppercase letter of the Latin small letter “i” is the Latin capital letter “I”. However, in Turkish, the uppercase letter is “I” with a dot above it (Unicode \u0130); hence, the case mapping is locale-sensitive.
Collation in Stata involves the locale-sensitive functions `ustrcompare()`, `ustrcompareex()`, `ustrsortkey()`, and `ustrsortkeyex()`. If you specify a locale with one of these functions or if you have set the locale globally (see `[P] set locale_functions`), then collation may be performed using a language-specific collator.

Because a locale is simply an identifier to locate the resources for specific services, there is no validation of the locale. For example, specifying “klingon” is as valid as specifying “en” when calling `ustrcompare()` or the other functions discussed here. If the collation data for the “klingon” locale is found, then the locale is populated; otherwise, fallback rules are followed. For more information, see `Default locale and locale fallback` in `[D] unicode locale`.

Stata supports hundreds of locales, but only about 100 have a language-specific collator. `unicode collator list` lets you determine whether your locale (or language) has its own collator. For example, Stata supports two locales for the Zulu language: `zu` is a general locale and `zu_ZA` is Zulu specific to South Africa. Only `zu` has a language-specific collator.

Further controlling collation

`ustrcompare()` and `ustrsort()` use the default collation algorithm for the locale. However, you can exercise finer control over the collation algorithm if you use `ustrcompareex()` or `ustrsortkeyex()`.

An International Components for Unicode (ICU) locale may contain up to five subtags in the following order: language, script, country, variant, and keywords. Stata usually uses only the language and country tags. However, collation keywords may be used in the `ustrcompareex()` and `ustrsortkeyex()` functions.

The collation keyword specifies the string sort order of the locale. For example, “pinyin” and “stroke” for Chinese language produce different string sort orders. In most cases, it is not necessary to specify a collation keyword; the default collator (either for Stata or for the language) provides sufficient control. However, some programmers may wish to specify a specific value. If you do not know the value of the collation keyword, you can obtain a list of valid collation values and their meanings in XML format at http://unicode.org/repos/cldr/trunk/common/bcp47/collation.xml.

If you are comparing or sorting Unicode strings that have come from different data sources, then you may need to normalize the strings before ordering them. See `ustrnormalize()` for details on normalization, and note the `norm` parameter in `ustrcompareex()` and `ustrsortkeyex()`.

Also see

[D] unicode — Unicode utilities
[D] unicode locale — Unicode locale utilities
[U] 12.4.2 Handling Unicode strings
[U] 12.4.2.5 Sorting strings containing Unicode characters
Title

**unicode convertfile —** Low-level file conversion between encodings

Description  Syntax  Options  Remarks and examples  Also see

**Description**

`unicode convertfile` converts text files from one encoding to another encoding. It is a low-level utility that will feel familiar to those of you who have used the Unix command `iconv` or the similar International Components for Unicode (ICU)-based command `uconv`. If you need to convert Stata datasets (*.dta) or text files commonly used with Stata such as do-files, ado-files, help files, and CSV (*.csv) files, you should use the `unicode translate` command; see [D] `unicode translate`. If you wish to convert individual strings or string variables in your dataset, use the `ustrfrom()` and `ustrto()` functions.

**Syntax**

```
unicode convertfile srcfilename destfilename [, options]
```

*`srcfilename`* is a text file that is to be converted from a given encoding and *`destfilename`* is the destination text file that will use a different encoding.

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>srcencoding()</code></td>
<td>encoding of the source file; UTF-8 if not specified</td>
</tr>
<tr>
<td><code>dstencoding()</code></td>
<td>encoding of the destination file; UTF-8 if not specified</td>
</tr>
<tr>
<td><code>srccallback()</code></td>
<td>what to do if source file contains invalid byte sequence(s)</td>
</tr>
<tr>
<td><code>dstcallback()</code></td>
<td>what to do if destination encoding does not support characters in the source file</td>
</tr>
<tr>
<td><code>replace</code></td>
<td>replace the destination file if it exists</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stop</code></td>
<td>specify that <code>unicode convertfile</code> stop with an error if an invalid character is encountered; the default</td>
</tr>
<tr>
<td><code>skip</code></td>
<td>specify that <code>unicode convertfile</code> skip invalid characters</td>
</tr>
<tr>
<td><code>substitute</code></td>
<td>specify that <code>unicode convertfile</code> substitute invalid characters with the destination encoding’s substitute character during conversion; the substitute character for Unicode encodings is \ufffd</td>
</tr>
<tr>
<td><code>escape</code></td>
<td>specify that <code>unicode convertfile</code> replace any Unicode characters not supported in the destination encoding with an escaped string of the hex value of the Unicode code point. The string is in 4-hex-digit form \uhhhh for a code point less than or equal to \uffff. The string is in 8-hex-digit form \Uhhhhhhhh for code points greater than \uffff. <code>escape</code> may only be specified when converting from a Unicode encoding such as UTF-8.</td>
</tr>
</tbody>
</table>

842
Options

srcencoding([string]) specifies the source file encoding. See help encodings for a list of common encodings and advice on choosing an encoding.

dstencoding([string]) specifies the destination file encoding. See help encodings for a list of common encodings and advice on choosing an encoding.

srccallback(method) specifies the method for handling characters in the source file that cannot be converted.

dstcallback(method) specifies the method for handling characters that are not supported in the destination encoding.

replace permits unicode convertfile to overwrite an existing destination file.

Remarks and examples

Remarks are presented under the following headings:

Conversion between encodings
Invalid and unsupported characters
Examples

Conversion between encodings

unicode convertfile is a utility to convert strings from one encoding to another. Encoding is the method by which text is stored in a computer. It maps a character to a nonnegative integer, called a code point, and then maps that integer to a single byte or a sequence of bytes. Common encodings are ASCII, UTF-8, and UTF-16. Stata uses UTF-8 encoding for storing text. Unless otherwise noted, the terms “Unicode string” and “Unicode character” in Stata refer to a UTF-8 encoded Unicode string or character. For more information about encodings, see [U] 12.4.2.3 Encodings. See help encodings for a list of common encodings, and see [D] unicode encoding for a utility to find all available encodings.

If you are using unicode convertfile to convert a file to UTF-8 format, the string encoding using by Stata, you only need to specify the encoding of the source file. By default, UTF-8 is selected as the encoding for the destination file. You can also use unicode convertfile to convert files from UTF-8 encoding to another encoding. Although conversion to or from UTF-8 is the most common usage, you can use unicode convertfile to convert files between any pair of encodings.

Be aware that some characters may not be shared across encodings. The next section explains options for dealing with unsupported characters.

Invalid and unsupported characters

Unsupported characters generally occur in two ways: the bytes used to encode a character in the source encoding are not valid in the destination encoding such as UTF-8 (called an invalid sequence); or the character from the source encoding does not exist in the destination encoding.

It is common to encounter inconvertible characters when converting from a Unicode encoding such as UTF-8 to some other encoding. UTF-8 supports more than 100,000 characters. Depending on the characters in your file and the destination encoding you select, it is possible that not all characters will be supported. For example, ASCII only supports 128 characters, so all Unicode characters with code points greater than 127 are unsupported in ASCII encoding.
Examples

Convert file from Latin1 encoding to UTF-8 encoding

`. unicode convertfile data.csv data_utf8.csv, srcencoding(ISO-8859-1)`

Convert file from UTF-32 encoding to UTF-16 encoding, skipping any invalid sequences in the source file

`. unicode convertfile utf32file.txt utf16file.txt, srcencoding(UTF-32) > dstencoding(UTF-16) srccallback(skip)`

Also see

[D] `unicode` — Unicode utilities
[D] `unicode translate` — Translate files to Unicode
[U] 12.4.2 Handling Unicode strings
[U] 12.4.2.6 Advice for users of Stata 13 and earlier
Description

unicode encoding list and unicode encoding alias list encodings that are available in Stata. See help encodings for advice on choosing an encoding and a list of the most common encodings. unicode encoding list provides a list of all encodings and their aliases or those that meet specified criteria. unicode encoding alias provides a list of alternative names that may be used to refer to a specific encoding.

unicode encoding set sets an encoding to be used with the unicode translate command; see [D] unicode translate for documentation for unicode encoding set.

Syntax

List encodings

    unicode encoding list [pattern]

List all aliases of an encoding

    unicode encoding alias name

Set an encoding for use with unicode translate

    unicode encoding set name

pattern is one of the following: *, _all, *name*, *name, or name*. Specifying nothing, _all, or * lists all results. Specifying *name* lists all results containing name. Specifying *name lists all results ending with name. Specifying name* lists all results starting with name.

Remarks and examples

Encoding is the method by which text is stored in a computer. It maps a character to a nonnegative integer, called a code point, then maps that integer to a single byte or a sequence of bytes. Common encodings are ASCII (for which there are many variants), UTF-8, and UTF-16. Stata uses UTF-8 encoding for storing text and UTF-16 to encode the GUI on Microsoft Windows and macOS. For more information about encodings, see [U] 12.4.2.3 Encodings.

The most common reason you will need to specify an encoding is when converting a dataset, do-file, ado-file, or some other file used with Stata 13 or earlier (which was not Unicode aware) for use with Stata 17. See [D] unicode translate for help with this, and see help encodings for advice on choosing an encoding and a list of common encodings.
Some commands and functions require that you specify one or more encodings. Often you will need to use only common encodings. However, you may not know how to specify these to Stata. For example, suppose that we are using `unicode translate` to convert a do-file from Stata 13 that contains extended ASCII characters for use in Stata 17. If we are working on a Windows machine, the most likely encoding is Windows-1252. If we want to check that this is how it should be specified as we use `unicode translate`, we can type

```
. unicode encoding list Windows-1252
```

Stata returns all encodings for which the encoding name or an alias exactly matches `Windows-1252`. Capitalization does not matter.

If we wanted to search for all encodings and aliases that have `windows` anywhere in their name, we could type

```
. unicode encoding list *windows*
```

and see a long list of matches.

If we are told that a text file is encoded with `ibm-913_P100-2000` and we want to see by what other names that encoding is known (perhaps because we just do not want to type out such a long string when using Stata’s functions that need an encoding), we can use

```
. unicode encoding alias ibm-913_P100-2000
```

and we find that there are many synonyms, including some that are much easier to type.

You may not know the exact encoding that you need and wish to browse the full list of available encodings. To do this, you can just type `unicode encoding list` without specifying a pattern.

**Also see**

help encodings

[D] `unicode` — Unicode utilities

[D] `unicode translate` — Translate files to Unicode

[U] 12.4.2 Handling Unicode strings

[U] 12.4.2.3 Encodings
Description

unicode locale list lists all available locales or those locales that meet the specified criteria. Any of these locale codes may be specified in Stata or Mata functions that accept a locale as an argument, such as ustrcompare() and ustrupper(), or in the set locale_functions setting.

unicode uipackage list lists all localization packages that are available for the graphics user interface (GUI). Any of the listed locales may be specified in the set locale_ui setting to change the language of the text that is displayed in GUI elements such as the menus and dialog boxes.

Syntax

List locales

    unicode locale list [pattern]

List user interface (UI) localization packages

    unicode uipackage list

pattern is one of _all, *, *name*, *name, or name*. If you specify nothing, _all, or *, then all results will be listed. *name* lists all results containing name; *name lists all results ending with name; and name* lists all results starting with name.

Remarks and examples

Remarks are presented under the following headings:

Overview
Default locale and locale fallback

Overview

A locale identifies a user community with a certain preference for how their language should be written; see [U] 12.4.2.4 Locales in Unicode. A locale can be as general as a certain language (for example, “en” for English) or can be more specific to a country or region (for example, “en_US” for U.S. English or “en_HK” for Hong Kong English. Stata uses International Components for Unicode’s (ICU’s) locale format. See http://userguide.icu-project.org/locale for full information about ICU. Note that ICU differs from the POSIX locale identifiers used by Linux systems.

Locales use tags to define how specific they are to language variants. An ICU locale may contain up to five subtags in the following order: language, script, country, variant, and keywords. Typically, the language is required and the other tags are optional. In most cases, Stata uses only the language and country tags. For example, “en_US” specifies the language as English and the country as the USA.
Many language-specific operations require the locale to perform their task. This kind of operation is called locale-sensitive. For example, in English, the uppercase letter of the Latin small letter “i” is the Latin capital letter “I”. However, in Turkish, the uppercase letter is “İ” with a dot above it (Unicode \u0130); hence, the case mapping is locale-sensitive.

The following functions are locale-sensitive: `ustrupper()`, `ustrlower()`, `ustrtitle()`, `ustrword()`, `ustrwordcount()`, `ustrcompare()`, `ustrcompareex()`, `ustrsortkey()`, and `ustrsortkeyex()`.

Although Stata usually uses only the language and country tags, collation keywords may also be used in functions `ustrcompare()` and `ustrsortkey()` to affect ordering of Unicode strings. The collation keyword affects the string sort order of the locale. For example, “pinyin” and “stroke” for Chinese language produce different string sort orders. In most cases, it is not necessary to specify a collation keyword; the default collator (either for Stata or for the language) provides sufficient control. However, some programmers may wish to specify a specific value. If you do not know the value of the collation keyword, you can obtain a list of valid collation values and their meanings in XML format at http://unicode.org/repos/cldr/trunk/common/bcp47/collation.xml.

**Default locale and locale fallback**

Because a locale is simply an identifier to locate the resources for specific services, there is no validation of the locale. For example, specifying “klingon” is as valid as specifying “en” when calling `ustrcompare()` or the other functions discussed here. If the collation data for the “klingon” locale is found, then the locale is populated; otherwise, a fallback search process starts.

The fallback process proceeds as follows:

1. The variant is removed if there is one.
2. The country is removed if there is one.
3. The script is removed if there is one.
4. Steps 1–3 are repeated on the default locale.
5. If a locale cannot be found after following the previous steps, the ICU “Root”, or built-in fallback, locale is used.

The process stops at any point if the desired information is found. The ICU default locale is usually the system locale on the machine, which you can change. Note that on macOS, the ICU default locale is usually “en_US_posix”, which does not change even if you change the system locale from the operating system’s “Language” setting. To see the ICU default locale, you can type

```
. display c(locale_icudflt)
```

You can also find it under the Unicode settings in the output of `creturn list` along with two other locale-related settings: `locale_ui` and `locale_functions`. See [P] set locale_ui and [P] set locale_functions for details.

`set locale_functions` affects the functions `ustrupper()`, `ustrlower()`, `ustrtitle()`, `ustrword()`, `ustrwordcount()`, `ustrcompare()`, `ustrcompareex()`, `ustrsortkey()`, and `ustrsortkeyex()` when no locale is specified. If `locale_functions` is not set, the default ICU locale `c(locale_icudflt)` is used.

For example, if your operating system is Microsoft Windows English version, the system locale is most likely “en”. It is “en_US” if you chose the country to be USA during installation of the operating system. If `locale_functions` is not set or is set to default, then `ustrupper("istanbul")` is equivalent to `ustrupper("istanbul", "en_US")`, which returns ISTANBUL.
However, if `locale_functions` is set to `tr` for Turkish, then `ustrupper("istanbul")` is equivalent to `ustrupper("istanbul", "tr")`, which returns İSTANBUL with a dot over the capital I. Although ICU does not validate locales, Stata validates that the language subtag of the `locale_functions` setting is a valid ISO-639-2 language code. (See the ISO-639-2 list at http://www.loc.gov/standards/iso639-2/.) Hence, `set locale_functions klingon` will produce an error.

With the fallback rules, the effective locale can be very different from the locale you specified, depending on the operation being performed. Currently, `ustrword()` and `ustrwordcount()`, which use ICU’s word break iterator service, and `ustrcompare()`, `ustrcompareex()`, `ustrsortkey()`, and `ustrsortkeyex()`, which use ICU’s collation service, are affected by this. You may use the functions `wordbreaklocale()` and `collatorlocale()` to find the effective locale from the requested locale.

Also see

[D] `unicode` — Unicode utilities

[P] `set locale_functions` — Specify default locale for functions

[P] `set locale_ui` — Specify a localization package for the user interface

[U] 12.4.2 Handling Unicode strings

[U] 12.4.2.4 Locales in Unicode
**Title**

| **unicode translate** — Translate files to Unicode |

**Description**

`unicode translate` translates files containing extended ASCII to Unicode (UTF-8).

Extended ASCII is how people got accented Latin characters such as “á” and “à” and got characters from other languages such as “ɲ”, “Θ”, and “わたし” before the advent of Unicode or, in this context, before Stata became Unicode aware.

If you have do-files, ado-files, .dta files, etc., from Stata 13 or earlier—and those files contain extended ASCII—you need to use the `unicode translate` command to translate the files from extended ASCII to Unicode.

The `unicode translate` command is also useful if you have text files containing extended ASCII that you wish to read into Stata.

**Syntax**

**Analyze files to be translated**

```
unicode analyze filespec [, redo nodata]
```

**Set encoding to be used during translation**

```
unicode encoding set ["]encoding["]
```

**Translate or retranslate files**

```
unicode translate filespec [, invalid(escape|mark|ignore)]
    transutf8 nodata]
```

```
unicode retranslate filespec [, invalid(escape|mark|ignore)]
    transutf8 replace nodata]
```

**Restore backups of translated files**

```
unicode restore filespec [, replace]
```

**Delete backups of translated files**

```
unicode erasebackups, badidea
```
filespec is a single filename or a file specification containing * and ? specifying one or more files, such as

* .dta
* .do
* *
*
myfile.*
year??data.dta

unicode analyzes and translates .dta files and text files. It assumes that filenames with suffix .dta contain Stata datasets and that all other suffixes contain text. Those other suffixes are .ado, .do, .mata, .txt, .csv, .sthlp, .class, .dlg, .idl, .ihlp, .smcl, and .stbcal.

Files with suffixes other than those listed are ignored. Thus “*.dta” would ignore any .docx files or files with other suffixes. If such files contain text, they can be analyzed and translated by specifying the suffix explicitly, such as info.README and *.README.

Options

redo is allowed with unicode analyze. unicode analyze remembers results from one run to the next so that it does not repeat results for files that have been previously analyzed and determined not to need translation. Thus unicode analyze’s output focuses on the files that remain to be translated. redo specifies that unicode analyze show the analysis for all files specified.

nodata is used with unicode analyze, translate, and retranslate. It specifies that the contents of the str# and strL variables in .dta files are not to be translated. The contents of the variables are to be left as is. The default behavior is to translate if necessary.

If option nodata is specified, only the metadata—variable names, dataset label, variable labels, value labels, and characteristics—are analyzed and perhaps translated.

This option is provided for two reasons.

nodata is included for those who do not trust automated software to modify the most vital part of their datasets, the data themselves. We emphasize to those users that unicode backs up files, and so translated files are easily restored to their original status.

The other reason nodata is included is for those datasets that include string variables in which some variables (observations) use one encoding and other variables (observations) use another. Such datasets are rare and called mixed-encoding datasets. One could arise if dataset result.dta was the result of merging input1.dta and input2.dta, and input1.dta encoded its string variables using ISO-8859-1, whereas input2.dta used JIS-X-0208. Such datasets are rare because if this had occurred, you would have noticed when you produced result.dta. The two extended ASCII encodings are simply not compatible, and one group or another of characters would have displayed incorrectly.

invalid and invalid() are allowed with unicode translate and retranslate. They specify how invalid characters are to be handled. Invalid characters are not supposed to arise, and when they do, it is a sign that you have set the wrong extended ASCII encoding. So let’s assume that you have indeed set the right encoding and that still one or a few invalid characters do arise. The stories on how this might happen are long and technical, and all of them involve you playing sophisticated font games, or they involve you using a proprietary extended ASCII encoding that is no longer available, and so you are using an encoding that is close to the actual encoding used.
By default, `unicode` will not translate files containing invalid characters. `unicode` instead warns you so that you can specify the correct extended ASCII encoding.

`invalid` specifies the invalid characters are to be shown with an escape sequence. If a string contained “A@B”, where @ indicates an invalid character, after translation, the string might contain “A%XCDB”, which is say, %XCD was substituted for @. In general, invalid characters are replaced with %X##, where ## is the invalid character's hex value. The substitution is admittedly ugly, but it ensures that distinct strings remain distinct, which is important if the string is used as an identifier when you use the data.

`invalid(escape)` is a synonym for `invalid`.

`invalid(mark)` specifies that the official Unicode replacement character be substituted for invalid characters. That official character is \ufffd in Unicode speak and how it looks varies across operating systems. On Windows, the Unicode replacement character looks like a square; on Mac and Unix, it looks like a question mark in a hexagon.

`invalid(ignore)` indicates that the invalid character simply be removed. “A@B” becomes “AB”.

`transutf8` is allowed with `unicode translate` and `retranslate`. `transutf8` specifies that characters that look as if they are UTF-8 already should nonetheless be translated according to the extended ASCII encoding. Do not specify this option unless `unicode` suggests it when you translate the file without the option, and even then, specify the option only after you have examined the translated file and determined that you agree.

For most of us, this issue arises when two extended ASCII characters appear next to each other, such as a German word containing “"ß”, or a French word containing “¨ö”. Even when extended ASCII characters are adjacent, that is not necessarily sufficient to mimic valid UTF-8 characters, but some combinations do mimic UTF-8.

Adjacent UTF-8 characters that mimic UTF-8 characters are actually likely when you are using a CJK extended ASCII encoding. CJK stands for Chinese, Japanese, and Korean.

In any case, if `unicode analyze` reports when valid UTF-8 strings appear and if the file needs translating because it is not all ASCII plus UTF-8, you may need to specify `transutf8` when you translate the file. If you are unsure, proceed by translating the file without specifying `transutf8`, inspect the result, and retranslate if necessary.

`replace` has nothing to do with translation and is allowed with `unicode retranslate` and `restore`. It has to do with the restoration of original, untranslated files from the backups that `unicode translate` and `retranslate` make. Option `replace` should not be specified unless `unicode` suggests it.

`unicode` keeps backups of your originals. When you restore the originals or retranslate files (which involves restoring the originals), `unicode` checks that the previously translated file is unchanged from when `unicode` last translated it. It does this because if you modified the translated file since translation, those changes might be important to you and because if `unicode` restored the original from the backup, you would lose those changes. `replace` specifies that it is okay to change the previously translated file even though it has changed.

`badidea` is used with `unicode erasebackups` and is not optional. Erasing the backups of original files is usually a bad idea. We recommend you keep them for six months or so. Eventually, however, you will want to delete the backups. You are required to specify option `badidea` to show that you realize that erasing the backups is a bad idea if done too soon.
Remarks and examples

Remarks are presented under the following headings:

What is this about?
Do I need to translate my files?
Overview of the process
How to determine the extended ASCII encoding
Use of unicode analyze
Use of unicode translate: Overview
Use of unicode translate: A word on backups
Use of unicode translate: Output
Translating binary strLs

What is this about?

Stata 14 and later use UTF-8, a form of Unicode, to encode strings. Stata 13 and earlier used ASCII. Datasets, do-files, ado-files, help files, and the like may need translation to display properly in Stata 17.

Files containing strings using only plain ASCII do not need translation. Plain ASCII provides the following characters:

- Latin letters: A–Z, a–z
- Digits: 0–9
- Symbols: ! ” # $ % ’ ( ) * + , − . / : ; < = > ? @ [ \ ] ` ^ _ { | } ~

If the variable names, variable labels, value labels, and string variables in your .dta files and the lines in your do-files, ado-files, and other Stata text files contain only the characters above, there is nothing you need to do.

On the other hand, if your .dta files, do-files, ado-files, etc., contain accented characters such as á à ë ô ù ý ... or symbols such as £ ¥ ... or characters from other alphabets, 3HаТб こんなちは

then the files do need translating so that the characters display correctly.

unicode analyze will tell you whether you have such files, and unicode translate will translate them.

You first use unicode analyze. It may turn out that no files need translating, and in that case, you are done.

If you do have files that need translating, you will use unicode translate. unicode translate makes a backup of your file before translating it.

If you do have files that need translating, unicode translate will translate them. Before you can use unicode translate, you must set the extended ASCII encoding that your files used. You do this with unicode encoding set. Encodings go by names such as ISO-8859-1, Windows-1252,
Big5, ISO-2022-KR, and about a thousand other names. However, there are only 231 encodings. Most of the names are aliases (synonyms). ISO-8859-1, for instance, is also known as ISO-Latin1, Latin1, and other names.

See `help encodings` for more information on encodings. Some of you will find the appropriate encoding name immediately. Others will be able only to narrow down the alternatives. Even so, all is not lost. `unicode translate` makes it easy to translate and retranslate a file over and over again until you find the encoding that works best. Once you find that encoding, it is likely that all of your files are using the same encoding.

**Do I need to translate my files?**

*Can I ignore the issue?*

If you are asking whether you can close your eyes and ignore the issue, the answer is maybe and maybe not.

If you have files using extended ASCII, they will not display correctly in Stata 17. We view that as a significant problem, but let's assume that does not concern you. If you used extended ASCII for variable names, you may find it difficult or impossible to type the untranslated name. That would be a problem. Other than that, you are probably okay, or more accurately, we cannot think of a problem even though we have tried. We have tried because if we could think of a problem, we would have fixed it. Stata's data management routines have been modified and certified to work with UTF-8. If they receive extended ASCII, they can mightily mess up what is displayed, but beyond that, they should produce results equivalent to what previous Statas produced.

Our advice is, for safety's sake, do not ignore the problem.

However, you do not need to analyze and translate all of your files today. One day, you will use a dataset and results will look odd when you `describe` or `list` the data. You will see unprintable characters and probably mutter a few unprintable words yourself, but having discovered the problem, you can then turn to solving it using `unicode analyze` and `unicode translate`.

However, we recommend that you learn to use `unicode translate` today. Take some files you are working with, determine whether you have a problem, and fix them if you do.

**Do my files need translation?**

If you are asking whether you have files that contain extended ASCII in hopes that you do not, here is our answer:

If you live and work in an English-speaking country, you probably do not have files containing extended ASCII.

If you live and work outside an English-speaking country but you have limited yourself to the unadorned Latin alphabet, you probably do not have files containing extended ASCII.

Otherwise, you probably do have files containing extended ASCII.

*How will I know what to do?*

`unicode analyze` will tell you whether you have files containing extended ASCII. `unicode analyze` can look at single files, or it can look at all the files in a directory. And if you do have files containing extended ASCII, `unicode translate` will fix the files.
Overview of the process

You will analyze your files and, if necessary, translate them. You can do this one file at a time by typing

. unicode analyze myfile.dta
. unicode encoding set encoding
. unicode translate myfile.dta

or you can do this with all of your files at once by typing

. unicode analyze *
. unicode encoding set encoding
. unicode translate *

Shockingly, we are going to advise you that analyzing and even translating all of your files at once is perfectly safe! That is because

1. `unicode analyze` by default ignores files that are not Stata related.
2. `unicode analyze` reads your files and reports on them; it does not change them.
3. `unicode analyze` might report that no files need translating. In that case, you are done.
4. if you do have files that need translating, before you can use `unicode translate`, you must set the extended ASCII encoding. How you determine the encoding is the topic of the next section.
5. `unicode translate`, just like `unicode analyze`, ignores by default files that are not Stata related. Typing `unicode translate *` is safe.
6. `unicode translate` does not modify files that do not need translation. This does not hinge on your having run `unicode analyze`. Typing `unicode translate *` is safe.
7. `unicode translate` does not modify files in which the translation goes poorly; it discards the translation. Typing `unicode translate *` is safe.
8. `unicode translate` makes backups of the original of any file it does translate successfully. At any time, you can type

   . unicode restore *

   and the files in your directory are back to being just as they were when you started. Typing `unicode translate *` is safe.

In the rest of this manual entry, we could discuss what might happen when you run `unicode analyze` and `unicode translate` and offer advice on what you might do about it.

`unicode analyze` and `unicode translate`, however, produce a ream of output, especially if you run them on a group of files. That output is tailored to your files and your situation. That output states what did happen and offers advice. Read it.

How to determine the extended ASCII encoding

We are getting ahead of ourselves because we have not yet determined that any of your files do need translating. Whether translation is necessary can be determined without knowing the extended ASCII encoding.

Determining the encoding can be more difficult than you would wish. Back in the day when the experts were still trying to make the extended ASCII solution work, the cleverest among them went to a lot of effort to hide the encoding from you, and they did a good job.
When the time comes to type

. unicode encoding set encoding

see help encodings. We have advice. In the meantime, allow us to predict how this process will transpire:

Some of you will not be able to determine the encoding your files are using, but you will be able to make guesses and narrow the choices down to a few of them. Then you will experiment to see which works best. We say “see” because that is literally how you are going to do it. You will guess, you will translate, and you will look at the result. And then you will repeat the process with a different encoding. The unicode command will make the translation and retranslation part easy.

Many of you will discover the single encoding that works for all of your files. Some of you will discover that one encoding works for most of your files but that there are one or two other encodings that you have to use with other files.

And then there is the issue of mixed UTF-8 and extended ASCII. This will affect only a few of you.

1. **unicode translate** will warn you when a file is a mix of UTF-8 and extended ASCII. It warns you because 1) the file could be exactly what it appears to be, a mix of encodings, or 2) the file is all extended ASCII and a few extended ASCII strings are merely masquerading as UTF-8.

2. By default, **unicode translate** assumes that the file really is a mix. It does not translate the UTF-8 strings; it translates just the strings that are extended ASCII.

   *Technical note:* Here is how this works. A variable label appearing to be UTF-8 already is not translated, whereas another variable label containing extended ASCII is translated even if a part of it appears to be UTF-8. **unicode translate** assumes that each variable label follows a single encoding. This same logic applies to **str** and **strL** variables in the data. The variable is assumed to use the same encoding in all observations.

3. The default assumption may be incorrect; the file could be entirely extended ASCII. The default assumption is more likely to be incorrect in the CJK case. You can determine whether the default assumption is correct by looking at the file after translation. If some parts of it look like memory junk, then use **unicode retranslate**, **transutf8** to retranslate the file, and if you do not like that result, use **unicode retranslate** without **transutf8** to return to the previous result. Or you could use **unicode restore** to return to the original file and start all over again, perhaps with a different encoding.

   *Technical note:* There is no difference between using **unicode restore** followed by **unicode translate** and using **unicode retranslate**. So if you want to try a different encoding, you can restore, set the new encoding, and translate, or you can set the new encoding and retranslate.

**Use of unicode analyze**

If the files you want to examine are not in the current directory, change to the appropriate directory:

. cd wherever

**unicode analyze** and all the rest of the **unicode** commands described in this entry look at files in the current directory and only files in the current directory. **unicode** does not even look in subdirectories of the current directory.

Analyze the file.

. unicode analyze myfile.dta
**unicode analyze** will report whether the file needs translation and provide other information, too. The output looks something like this:

```
. unicode analyze myfile.dta
File summary (before starting):
  1 file(s) specified
  1 file(s) to be examined ...
File myfile.dta (Stata dataset)

File does not need translation

File summary:
  all files okay
```

Or it might look like this:

```
. unicode analyze myfile.dta
File summary (before starting):
  1 file(s) specified
  1 file(s) to be examined ...
File myfile.dta (Stata dataset)
  3 variable names need translation
  2 variable labels need translation
  1 str# variable needs translation

File needs translation.
Use **unicode translate** on this file

File summary:
  1 file needs translation
```

If you were to now rerun the analysis in the case where the file does not need translation, you would see something like this:

```
. unicode analyze myfile.dta
File summary (before starting):
  1 file(s) specified
  1 file(s) already known to be ASCII in previous runs
  0 file(s) to be examined ...
(none to do)
```

If you want to see the detailed output, type **unicode analyze myfile.dta, redo**.

The primary purpose of **unicode analyze** is to get the files that do not need translating out of the way. **unicode analyze** does not change your files; it just dismisses the ones that need no further work.

You can run **unicode analyze** on multiple files, and we recommend that you do that.

```
. unicode analyze *
  30 file(s) specified
  6 file(s) not Stata
  1 file(s) already known to be ASCII in previous runs
  1 file(s) already known to be UTF-8 in previous runs
  22 files(s) to be examined
```

There is more to the output, but before we look at that, note that **unicode analyze** reported that 6 files were not Stata. **unicode analyze** and **unicode translate** ignore non-Stata files unless you explicitly specify them, say, by typing **unicode analyze README** or **unicode analyze *.README**.
Let’s now return to the remaining output from `unicode analyze *`:

```
File filename (filetype)
  notes about elements that need translating
    recommendations

File filename (filetype)
  notes about elements that need translating
    recommendations

Files matching * that need translation:
  list of files

File summary:
  2 file(s) skipped (known okay from previous runs)
  8 file(s) need translation
```

`unicode analyze` produced a lot of output. If you are like us, you will want a log of the output and perhaps want to look at it in the Viewer. It is not too late, just remember to specify the `redo` option:

```
. log using output
. unicode analyze *, redo
  (output omitted)
. log close
. view output.smcl
```

If you are really like us, you will instead want a file you can edit in Stata’s Do-file Editor:

```
. log using output.log
. unicode analyze *, redo
  (output omitted)
. log close
. doedit output.log
```

Now, you can edit the output to make a to-do list for yourself. We go through the output and delete the parts with which we agree, such as the following:

```
File myfile.do (text file)
  40 line(s) in file

File does not need translation.
```
Buried in the output, however, may be something like this:

File german.dta (Stata dataset)

File does not need translation, except ...  
The file appears to be UTF-8 already. Sometimes, files that need translating can look like UTF-8. Look at these examples:
variable name "länge"
 variable label "Kofferraumvolumen (Kubikfuß)"
value-label contents "Ausländisch"
contents of str# variable marke
Do they look okay to you?
If not, the file needs translating or retranslating with the transutf8 option. Type

    . unicode translate "bill_utf8.dta", transutf8
    . unicode retranslate "bill_utf8.dta", transutf8

This file, too, is marked as not needing translation, and we agree based on the evidence presented, but we might not have agreed. Assume that the file was named japan.dta and that the examples did not look like Japanese but looked like memory junk. We would want to add this file to our list to translate and remind ourselves to specify option transutf8 when translating.

It is unlikely that any file that unicode analyze reports as purely UTF-8 needs translating unless the file is short, and then you must look at it to determine whether the file really is UTF-8.

Here is a different example. The file, according to unicode analyze, needs translation, but it also includes UTF-8:

File filter.do (text file)  
40 line(s) in file
33 line(s) ASCII
1 line(s) UTF-8
6 line(s) need translation

File needs translation. Use unicode translate on this file.  
There are three possibilities.
1) The file is exactly what it appears to be, a mix of extended ASCII and UTF-8. Use unicode translate.
2) The UTF-8 lines are extended ASCII masquerading as UTF-8. Use unicode translate, transutf8.
3) The file is UTF-8 with some invalid characters. Set the encoding to utf8 and then use unicode translate, invalid().

unicode analyze thinks this file needs translation and speculates about how it should be translated. Read the output. Possibility 3) did not even occur to us. Even so, and even without looking at the file, we would favor possibility 2) because there is only one UTF-8 line and there are 6 lines known to need translation.

You will learn that running unicode analyze is optional. The advantage of running unicode analyze is that it offers advice.

You can analyze files repeatedly. If you type unicode analyze without the redo option, the output reappears, but files are skipped that unicode analyze previously determined as not needing translation. Specify redo and you will see all the files.

unicode analyze remembers results from previous runs. Five years from now, unicode analyze will remember the files it has examined and determined do not need translation, and it will even know whether the file has changed in the intervening five years and so needs reexamination.

unicode analyze remembers from one run to the next by creating a directory named bak.stunicode, where it can put its notes. Ignore the directory and its subdirectories. When we tell you about unicode translate, you will learn that bak.stunicode is also where backups of unmodified original files
are stored. Now that you know that, you might be tempted to restore originals from the backups by copying the files. Do not do that because you will confuse `unicode`. Use `unicode restore` to restore originals. We will get to that.

The purpose of `unicode analyze` is to dismiss all the files that do not have problems so you can focus on those that do. When you later use `unicode translate`, it will also skip over files that do not need translating. Using `unicode analyze` is optional, and even if you do not use it, `unicode translate` will never translate a file that does not need it; `unicode translate` runs `unicode analyze` in secret if it needs to.

**Use of `unicode translate`: Overview**

Let’s assume that we have used `unicode analyze` and learned that the following files need translating:

```
  myfile.dta
  anotherfile.do
```

Before we can translate the files, we must set the extended ASCII encoding. See help `encodings` when you are translating your files.

Let’s just assume right now that we know the encoding for the files is ISO-8859-1, and then we will assume that we were wrong and show you how we get out of that situation.

Step 1. Inform `unicode` of the encoding by typing

```
  . unicode encoding set ISO-8859-1
```

Step 2. Translate the files, one at a time by typing

```
  . unicode translate myfile.dta
  . unicode translate anotherfile.do
```

or both in one command by typing

```
  . unicode translate *
```

Specifying `*` or `*.dta` or `*.dta` or `m*.dta` or `*.dta` or `m*.dta` or any other file specification is perfectly safe. `unicode translate` ignores irrelevant files just as `unicode analyze` does. `unicode translate` also ignores files that do not need translating, and it ignores files that have already been translated. `unicode translate` does not depend on your having run `unicode analyze` previously.

`unicode translate` has another great feature: it makes backups of the files it modifies. If, after translation, you decide you do not like the translation, you can restore the original by typing

```
  . unicode restore myfile.dta
```

You can even type

```
  . unicode restore *
```

if you want all of your files restored.

You do not have to restore the original just to retranslate it. Use `unicode retranslate` instead:

```
  . unicode retranslate myfile.dta
  . unicode retranslate *
```

The only reason to run `unicode retranslate`, however, is if you want to specify different options or try a different encoding:

```
  . unicode encoding set some_other_encoding
  . unicode retranslate *
```

And if you do not like that result, you can still `unicode restore`. 
Use of `unicode translate`: A word on backups

`unicode translate` and `retranslate` automatically make backups when they modify a file and a backup does not already exist. `unicode` calculates and keeps track of checksums calculated on the original and translated files, so it knows whether the files are subsequently changed. `unicode` is thoroughly tested. What could possibly go wrong?

If you are like us, you trust nobody with regard to your files. We do not even trust ourselves. Trust us on this. Make your own back up in whatever way you know before using `unicode translate`. Backup the entire directory. We would make a zip file of it, but if nothing else, just copy all the files to a new, out-of-the-way directory. We predict you will not need the copies, but one never knows for sure.

Even if `unicode` is perfect, the subsequent validity of the backups depends on the `bak.standard` subdirectory not being corrupted by another process or even by you. More than once, we have ourselves damaged files in haste.

After you have translated your files, keep the backups for a while. Eventually, however, there will come a day when the backups are no longer needed. The command to delete the backups of your originals is

```
. unicode erasebackups, badidea
```

You must specify option `badidea`. Think of `badidea` as an abbreviation for `badideaifdonestoosoon`: what you are doing in specifying the option is stating that it is not too soon.

Use of `unicode translate`: Output

`unicode translate`’s output looks just like `unicode analyze`’s output except that the content varies:

```
. unicode translate *
 30  file(s) specified
  6  file(s) not Stata
  6  file(s) already known to be ASCII in previous runs
  4  file(s) already known to be UTF-8 in previous runs
 14  files(s) to be examined

File filename (filetype)
  notes about the translation
```

result message

File badfile.ado (textfile)
  40 lines in file
 16 lines ASCII
  2 lines translated
 22 lines w/ invalid chars not translated

File not translated because it contains untranslatable characters;
   you need to specify a different encoding or, if you are sure that you have the correct encoding, use `unicode translate` with the `invalid()` option

...
File filename (filetype)
notes about the translation
notes about elements that need translating

result message

Files matching * that still need translation:
badfile.ado

File summary:
  10  file(s) skipped (known okay from previous runs)
  13  file(s) successfully translated
  1  files(s) not translated because they contain untranslatable characters
      you need to specify a different encoding or, if you are sure that you have the correct encoding, use unicode translate with the invalid() option

One file still needs translation according to the output. How can files still need translation? The output explains. We had untranslatable characters. The output even says what to do about it. We should specify a different encoding—the fact that we had untranslatable characters is evidence that we are using the wrong encoding—or we should accept that there are invalid characters in our file and tell unicode translate how to handle them. It will help us make the decision if we scan up from the file-summary message to find the detailed output for badfile.ado:

File badfile.ado (textfile)
  40 lines in file
  16 lines ASCII
  2 lines translated
  22 lines w/ invalid chars not translated

File not translated because it contains untranslatable characters;
      you need to specify a different encoding or, if you are sure that you have the correct encoding, use unicode translate with the invalid() option

You can read about the invalid() option under Options, but this looks like a case where the file needs a different encoding; 2 lines translated with the current encoding, and 22 did not. If we had instead seen that 22 lines translated and that 2 lines had invalid characters, we would be less sure about needing a different encoding. Assume the output had been

File badfile.ado (textfile)
  40 lines in file
  38 lines ASCII
  2 lines w/ invalid chars not translated

File not translated because it contains untranslatable characters;
      you need to specify a different encoding or, if you are sure that you have the correct encoding, use unicode translate with the invalid() option

That an ado-file is mostly ASCII does not surprise us. The fact that no lines could be translated (given the encoding) speaks volumes. We need a different encoding.
Most of our files were translated. For successful translations, the detailed output for .dta files will be something like the following:

File trees.dta (Stata dataset)
  9 variable names okay, ASCII
  3 variable names translated
  all data labels okay, ASCII
  8 variable labels okay, ASCII
  4 variable labels translated
  all value-label names okay, ASCII
  all value-label contents translated
  all characteristic names okay, ASCII
  all characteristic contents okay, ASCII
  all str# variables okay, ASCII

File successfully translated

The detailed output for text files might look like the following:

File runjob.do (textfile)
  120 lines in file
  101 lines ASCII
  19 lines translated

File successfully translated

Here is an example of a file that translated successfully but produced a lot of output:

File northwest.dta (Stata dataset)
  all variable names okay, ASCII
  all data labels okay, ASCII
  all variable labels okay, ASCII
  all value-label names okay, ASCII
  all value-label contents okay, ASCII
  all characteristic names okay, ASCII
  all characteristic contents okay, ASCII
  1 strL variable okay, ASCII
  1 strL variable(s) have binary values
    This concerns strL variable diagnotes.
    StrL variables that contain binary values in even one observation are not translated by unicode. Translating binary values is inappropriate. Rarely, however, "binary" values are just text or the variable contains binary values in some observations and nonbinary values in others. You translate such variables using generate or replace; see translating binary strLs.
  1 strL variable translated
  2 str# variables okay, ASCII
  1 str# variable translated

File successfully translated

The extra output concerns a strL variable that was not translated. The output states that the variable is binary and that translating binary strLs is inappropriate, but maybe not. This is the topic of the next section.
Translating binary strLs

`unicode translate` does not translate binary strLs. That is probably the right decision. StrLs are sometimes used in Stata to record documents, images, and other binary files, and modifying binary files is never a good idea.

Stata marks strL variables as binary on an observation-by-observation basis. As far as `unicode translate` is concerned, however, if there is just one observation in which the strL is marked as binary, it treats all observations as binary and does not translate them. The thinking is that variables hold different realizations of the same underlying type of thing, and if the strL is binary in one observation, it is probably truly binary in all observations.

Perhaps you know differently in your specific application and wish to translate the variable’s nonbinary observations or all of its observations. Here is how you do that.

You use string function `ustrfrom()` to obtain a translated string. Assuming the existing strL variable is named `myvar`, you type

```
. generate strL newvar = ustrfrom(myvar, "encoding", #)
```

Specify encoding just as you would with `unicode encoding set`. `encoding` might be ISO-8859-1, Windows-1252, Big5, ISO-2022-KR, or any other extended ASCII encoding. Whatever string you specify for `encoding`, make sure it is valid and spelled correctly. Testing the string with `unicode encoding set` is one way to do that.

# is specified as 1, 2, 3, or 4 and determines how invalid characters are to be handled. Three of the four values correspond to `unicode`’s `invalid()` option:

<table>
<thead>
<tr>
<th>#</th>
<th>equivalent to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>invalid(mark)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>invalid(escape)</code></td>
</tr>
<tr>
<td>3</td>
<td><code>invalid(escape)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>invalid(escape)</code></td>
</tr>
</tbody>
</table>

The remaining code, 3, specifies that the function return “ ” if invalid characters are encountered.

So one way of translating all the values of `myvar` would be

```
. generate strL try = ustrfrom(myvar, "ISO-8859-1", 1)
. browse newvar           // review result
. replace newvar = try
. drop try
```

If you want to translate only the nonbinary values of `myvar`, you could type

```
. gen strL try = ustrfrom(myvar, "ISO-8859-1", 1) if !_strisbinary(myvar)
. replace try = myvar if _strisbinary(myvar)
```

That would use Stata’s definition of binary, which is difficult to explain. Another good definition of binary is that the string not contain binary 0:

```
. gen strL try = ustrfrom(myvar, "ISO-8859-1", 1) if !strpos(myvar, char(0))
. replace try = myvar if strpos(myvar, char(0))
```

Also see

[D] `unicode` — Unicode utilities

[U] 12.4.2 Handling Unicode strings

[U] 12.4.2.6 Advice for users of Stata 13 and earlier
use — Load Stata dataset

Description

use loads into memory a Stata-format dataset previously saved by save. If filename is specified without an extension, .dta is assumed. If your filename contains embedded spaces, remember to enclose it in double quotes.

In the second syntax for use, a subset of the data may be read.

Quick start

Load Stata-format dataset mydata.dta into memory from current directory
   use mydata

As above, but load data from the mysubdir subdirectory in current directory and clear current data from memory first
   use mysubdir/mydata, clear

Load only variables v1, v2, and v3 from mydata.dta
   use v1 v2 v3 using mydata

As above, and further restrict to the first 100 observations
   use v1 v2 v3 in 1/100 using mydata

Load observations from mydata.dta where catvar = 2
   use if catvar==2 using mydata

Menu

File  >  Open...
Syntax

Load Stata-format dataset

```
use filename [, clear nolabel]
```

Load subset of Stata-format dataset

```
use [varlist] [if] [in] using filename [, clear nolabel]
```

Options

clear specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

nolabel prevents value labels in the saved data from being loaded. It is unlikely that you will ever want to specify this option.

Remarks and examples

Example 1

We have no data in memory. In a previous session, we issued the command `save hiway` to save the Minnesota Highway Data that we had been analyzing. We retrieve it now:

```
. use hiway
(Minnesota Highway Data, 1973)
```

Stata loads the data into memory and shows us that the dataset is labeled “Minnesota Highway Data, 1973”.

Example 2

We continue to work with our `hiway` data and find an error in our data that needs correcting:

```
. replace spdlimit=70 in 1
(1 real change made)
```

We remember that we need to forward some information from another dataset to a colleague. We use that other dataset:

```
. use accident
no; dataset in memory has changed since last saved
r(4);
```

Stata refuses to load the data because we have not saved the `hiway` data since we changed it.

```
. save hiway, replace
file hiway.dta saved
. use accident
(Minnesota accident data)
```
After we save our hiway data, Stata lets us load our accident dataset. If we had not cared whether our changed hiway dataset were saved, we could have typed `use accident, clear` to tell Stata to load the accident data without saving the changed dataset in memory.

---

### Technical note

In example 2, you saved a revised `hiway.dta` dataset, which you forward to your colleague. Your colleague issues the command

```stata
use hiway
```

and gets the message

```
file hiway.dta not Stata format
r(610);
```

Your colleague is using a version of Stata older than Stata 14. If your colleague is using Stata 11, 12, or 13, you can save the dataset in Stata 11, 12, or 13 format by using the `saveold` command; see `[D] save`.

Newer versions of Stata can always read datasets created by older versions of Stata. Stata/MP and Stata/SE can read datasets created by Stata/BE. Stata/BE can read datasets created by Stata/MP and Stata/SE if those datasets conform to Stata/BE’s limits; see `[R] Limits`.

---

### Example 3

If you are using a dataset that is too large for the amount of memory on your computer, you could load only some of the variables:

```stata
use ln_wage grade age tenure race using
> https://www.stata-press.com/data/r17/nlswork
(National Longitudinal Survey. Young Women 14-26 years of age in 1968)

describe
```

```
Contains data from https://www.stata-press.com/data/r17/nlswork.dta
Observations: 28,534
(National Longitudinal Survey. Young Women 14-26 years of age in 1968)
Variables: 5
27 Nov 2020 08:14

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>byte</td>
<td>%8.0g</td>
<td>age in current year</td>
<td></td>
</tr>
<tr>
<td>race</td>
<td>byte</td>
<td>%8.0g</td>
<td>race</td>
<td></td>
</tr>
<tr>
<td>grade</td>
<td>byte</td>
<td>%8.0g</td>
<td>current grade completed</td>
<td></td>
</tr>
<tr>
<td>tenure</td>
<td>float</td>
<td>%9.0g</td>
<td>job tenure, in years</td>
<td></td>
</tr>
<tr>
<td>ln_wage</td>
<td>float</td>
<td>%9.0g</td>
<td>ln(wage/GNP deflator)</td>
<td></td>
</tr>
</tbody>
</table>

Sorted by:
```

Stata successfully loaded the five variables.
Example 4

You are new to Stata and want to try working with a Stata dataset that was used in example 1 of [XT] xtlogit. You load the dataset:

```
. use https://www.stata-press.com/data/r17/union
(NLS Women 14-24 in 1968)
```

The dataset is successfully loaded, but it would have been shorter to type

```
. webuse union
(NLS Women 14-24 in 1968)
```

`webuse` is a synonym for `use https://www.stata-press.com/data/r17/`; see [D] webuse.

Also see

[D] compress — Compress data in memory

[D] datasignature — Determine whether data have changed

[D] import — Overview of importing data into Stata

[D] save — Save Stata dataset

[D] sysuse — Use shipped dataset

[D] webuse — Use dataset from Stata website

[U] 11.6 Filenaming conventions

[U] 22 Entering and importing data
varmanage — Manage variable labels, formats, and other properties

Description

varmanage opens the Variables Manager. The Variables Manager allows for the sorting and filtering of variables for the purpose of setting properties on one or more variables at a time. Variable properties include the name, label, storage type, format, value label, and notes. The Variables Manager also can be used to create varlists for the Command window.

Menu

Data > Variables Manager

Syntax

varmanage

Remarks and examples

A tutorial discussion of varmanage can be found in [GS] 7 Using the Variables Manager (GSM, GSU, or GSW).

Also see

[D] drop — Drop variables or observations
[D] edit — Browse or edit data with Data Editor
[D] format — Set variables’ output format
[D] label — Manipulate labels
[D] notes — Place notes in data
[D] rename — Rename variable
vl stands for variable list. It is a suite of commands for creating and managing named variable lists. Lists are intended especially to be used as arguments to estimation commands.

In particular, the suite is designed to help divide variables into two groups: one group that will be treated as factor variables and another group that will be treated as continuous or interval variables.

vl creates two types of named variable lists: system-defined variable lists, created automatically by \texttt{vl set}, and user-defined variable lists, created by \texttt{vl create}. You will usually use \texttt{vl set} to create system-defined variable lists first, and then create your own variable lists from them with \texttt{vl create}.

After creating a variable list called \texttt{vl\_username}, the expression \$\texttt{vl\_username} can be used in Stata anywhere a \texttt{varlist} is allowed. Variable lists are actually \texttt{global macros}, and the \texttt{vl} commands are a convenient way to create and manipulate them.

Variable lists are saved with the dataset.

Remarks and examples

Remarks are presented under the following headings:

- \texttt{Introduction}
- \texttt{vl set and system-defined variable lists}
- \texttt{Classification criteria for system-defined variable lists}
- \texttt{Moving variables into another classification}
- \texttt{vl create and user-defined variable lists}
- \texttt{vl list}
- \texttt{vl substitute and factor-variable operators}
- \texttt{Exploring data with vl set}
- \texttt{Changing the cutoffs for classification}
- \texttt{Moving variables from one classification to another}
- \texttt{Dropping variables and rebuilding variable lists}
- \texttt{Changing variables and updating variable lists}
- \texttt{Saving and using datasets with variable lists}
- \texttt{User-defined variable lists and factor-variable operators}
- \texttt{Updating variable lists created by vl substitute}
Introduction

The `vl` commands are the following:

**System only**
- **`vl set`** initializes the system-defined variable lists based on the number of levels and other characteristics of a variable
- **`vl move`** moves variables from one system-defined variable list to another

**User only**
- **`vl create`** creates user-defined variable lists
- **`vl modify`** adds or removes variables from user-defined variable lists
- **`vl label`** adds a label to a user-defined variable list
- **`vl substitute`** creates a user-defined variable list using factor-variable operators

**System or user**
- **`vl list`** lists the contents of variable lists, either system or user
- **`vl dir`** displays the defined variable lists, either system or user
- **`vl drop`** deletes variable lists or removes variables from multiple variable lists
- **`vl clear`** deletes all variable lists
- **`vl rebuild`** restores variable lists

The first thing to note is that some `vl` commands only work with system-defined variable lists, some only work with user-defined variable lists, and others work with both.

`vl set` is typically used first. It initializes the system-defined variable lists. By default, it classifies all the numeric variables in your dataset. Or you can specify `varlist` and have it classify only those variables.

When we are discussing the `vl` commands and say “variable list”, we mean a named variable list created by `vl set` or `vl create`. A traditional Stata list of variables, that is, `varlist`, we will call `varlist`. Variable lists contain `varlists`.

`vl create` allows you to create your own variable lists, either starting with system-defined variable lists or with `varlists` you specify. There is no need to run `vl set` and create system-defined variable lists. You can create your own from scratch. If you are familiar with the variables in your dataset and know which ones you want treated as factor variables and which as continuous variables, you may want to create only user-defined variable lists.

`vl rebuild` restores all the `vl`-generated variable lists after loading a dataset that previously had variable lists. Stata saves variable lists when you `save` your data, but when you `use` the saved data file, they are not automatically restored.

We will explain how to use `vl` with a series of examples.

**`vl set` and system-defined variable lists**

We will first show examples using Stata’s automobile dataset because it only has a small number of variables and the output will not be too lengthy. We will do that even though you are unlikely to want to use `vl` with this small dataset. `vl` is intended for use with dozens or even thousands of variables.
. sysuse auto
(1978 automobile data)

Typing `vl set` without `varlist` classifies all the numeric variables in the data.

```
. vl set
```

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>2</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>2</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>7</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

Notes

1. Review contents of `vlcategorical` and `vlcontinuous` to ensure they are correct. Type `vl list vlcategorical` and type `vl list vlcontinuous`.
2. If there are any variables in `vluncertain`, you can reallocate them to `vlcategorical`, `vlcontinuous`, or `vlother`. Type `vl list vluncertain`.
3. Use `vl move` to move variables among classifications. For example, type `vl move (x50 x80) vlcontinuous` to move variables `x50` and `x80` to the continuous classification.
4. `vlnames` are global macros. Type the `vlname` without the leading dollar sign (`$`) when using `vl` commands. Example: `vlcategorical not $vlcategorical`. Type the dollar sign with other Stata commands to get a varlist.

By default, all numeric variables are put into one of four system-defined variable lists: `vlcategorical`, `vlcontinuous`, `vluncertain`, or `vlother`.

`vlcategorical` is intended for variables that are to be used as factor variables. `vlcontinuous` is intended for variables that are to be treated as continuous. `vluncertain` is intended for variables that may be categorical or may be continuous. `vlother` is a garbage classification intended for variables you want to ignore. `vl set` only puts constants and variables that are always missing into `vlother`, but you can move other variables there—more on that later.

Classification criteria for system-defined variable lists

Division into `vlcategorical`, `vlcontinuous`, or `vluncertain` is determined by several criteria.

First, if the variable contains any noninteger values, it goes in `vlcontinuous`.

Second, if the variable has negative values, it goes in `vlcontinuous` because factor variables in Stata must be nonnegative. If you have a variable that has values $-1$ and $1$, you must recode it as $0$ and $1$ (or $1$ and $2$ or any other two distinct nonnegative integers) before you can use it as a factor variable.

Third, values of factor variables must be smaller than $2^{31} = 2,147,483,648$, so a variable with any values $\geq 2^{31}$ goes in `vlcontinuous`.

Fourth, constants, even when nonnegative integers, go in `vlother`.

For the remaining variables containing nonnegative integers, where they are placed is determined by two cutoffs, which can be specified by the options `categorical(#)` and `uncertain(#)`. 
When the number of levels (distinct values), \( L \), is
\[
2 \leq L \leq \text{categorical(\#)}
\]
the variable goes in \( \text{vlcategorical} \). When
\[
\text{categorical(\#)} < L \leq \text{uncertain(\#)}
\]
the variable goes in \( \text{vluncertain} \). When
\[
L > \text{uncertain(\#)}
\]
the variable goes in \( \text{vlcontinuous} \).

The defaults are \text{categorical(10)} and \text{uncertain(100)}, which are admittedly arbitrary. They were chosen because they are easy-to-remember round numbers. In many cases, you will want to use different cutoffs. See the next section, where we reset \text{categorical(\#)} and \text{uncertain(\#)}.

### Moving variables into another classification

\text{vl list} will show how each variable was classified and why.

```
. vl list, minimum maximum observations
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
<th>Min</th>
<th>Max</th>
<th>Obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep78</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>69</td>
</tr>
<tr>
<td>foreign</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>74</td>
</tr>
<tr>
<td>headroom</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td>1.5</td>
<td>5</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td>2.19</td>
<td>3.89</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>price</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>74</td>
<td>3291</td>
<td>15906</td>
<td>74</td>
</tr>
<tr>
<td>mpg</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>21</td>
<td>12</td>
<td>41</td>
<td>74</td>
</tr>
<tr>
<td>trunk</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>18</td>
<td>5</td>
<td>23</td>
<td>74</td>
</tr>
<tr>
<td>weight</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>64</td>
<td>1760</td>
<td>4840</td>
<td>74</td>
</tr>
<tr>
<td>length</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>47</td>
<td>142</td>
<td>233</td>
<td>74</td>
</tr>
<tr>
<td>turn</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>18</td>
<td>31</td>
<td>51</td>
<td>74</td>
</tr>
<tr>
<td>displacement</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>31</td>
<td>79</td>
<td>425</td>
<td>74</td>
</tr>
</tbody>
</table>

We specified options \text{minimum}, \text{maximum}, and \text{observations} to display the minimum and maximum values of each variable and the number of nonmissing observations.

\text{vl set} does not use the minimum and maximum to determine whether the variable goes in \text{vlcategorical}, \text{vlcontinuous}, or \text{vluncertain}. If the variable is a nonnegative integer, only the number of levels matters to \text{vl set}. A variable with levels 1,000,000 and 2,000,000 is classified the same as a variable with levels 0 and 1. The minimum and maximum can be displayed because you might want to use them to reclassify the variables.

In our example, we look at the number of levels and the minimum and maximum of the variables in \text{vluncertain}, and we decide we want to treat them all as continuous. We use \text{vl move} to move them into \text{vlcontinuous}.

```
. vl move vluncertain vlcontinuous
note: 7 variables specified and 7 variables moved.
```

<table>
<thead>
<tr>
<th>Macro</th>
<th># Added/Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vlcategorical</td>
<td>0</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>7</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>-7</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
</tr>
</tbody>
</table>
When variables are moved into a different system-defined variable list, they are moved out of their current list.

Moving on, variable `rep78`, which gives the vehicle repair record, is worth some thought.

```
. tabulate rep78

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2.90</td>
<td>2.90</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>11.59</td>
<td>14.49</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>43.48</td>
<td>57.97</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>26.09</td>
<td>84.06</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>15.94</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>69</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>
```

rep78 could be considered categorical and used as a factor variable or could be considered as an interval variable and treated as a continuous variable.

Let's say we want to move it into `vlcontinuous`. To specify variable names directly, you specify them in parentheses. We move `rep78`.

```
. vl move (rep78) vlcontinuous
```

```
Macro # Added/Removed
$vlcategorical -1
$vlcontinuous 1
$vluncertain 0
$vlother 0
```

**vl create and user-defined variable lists**

`vl set` and `vl move` are a first-pass classification of your variables. Next you will likely want to create specialized variable lists for use as independent variables for an estimation command.

You can create variable lists based on a specific set of variables. Use `vl create` and specify a `varlist` enclosed in parentheses, `()`.  

```
. vl create power = (gear_ratio displacement weight)
   note: $power initialized with 3 variables.
```

```
. vl create nonpower = (turn length rep78)
   note: $nonpower initialized with 3 variables.
```

We want to model `mpg`. We created the variable list `power`, containing variables we think are related to power, and another variable list `nonpower`, containing variables that are not related to power but might be predictive of `mpg`.

After creating these variable lists, we decide the variable `length` belongs in `power` instead of `nonpower`. So we add it to `power` by using the `vl modify` command.

```
. vl modify power = power + (length)
   note: 1 variable added to $power.
```

`vl create` and `vl modify` are like `generate` and `replace` in Stata. `vl create` creates new variable lists. `vl modify` modifies existing variable lists.
We can use `vl list` to see the variable lists to which the variable `length` belongs.

```
. vl list (length), user
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>$nonpower</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
<tr>
<td>length</td>
<td>$power</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
</tbody>
</table>

We used `vl list` with `varlist` enclosed in parentheses. We specified option `user` to list only the user-defined variable lists.

If we do not want `length` in `nonpower`, we must explicitly move it out.

```
. vl modify nonpower = nonpower - (length)
note: 1 variable removed from $nonpower.
```

In this way, `vl modify` differs from `vl move`. `vl move` moves a variable out of its current system-defined variable list when the variable is moved into a new one. `vl modify` only modifies the specified variable list.

We can create new user-defined variable lists from existing variable lists, whether user or system defined.

```
. vl create xvars = power + nonpower
note: $xvars initialized with 6 variables.
```
Using (*) to specify the *varlist* for *vl list* gives a listing ordered by variable name first and then variable-list name.

```
. vl list (*)
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>74</td>
</tr>
<tr>
<td>price</td>
<td>not in vluser</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>mpg</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>21</td>
</tr>
<tr>
<td>mpg</td>
<td>not in vluser</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>rep78</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>rep78</td>
<td>$nonpower</td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>rep78</td>
<td>$xvars</td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>headroom</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>headroom</td>
<td>not in vluser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trunk</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>trunk</td>
<td>not in vluser</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>weight</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>64</td>
</tr>
<tr>
<td>weight</td>
<td>$power</td>
<td>integers &gt;=0</td>
<td>64</td>
</tr>
<tr>
<td>weight</td>
<td>$xvars</td>
<td>integers &gt;=0</td>
<td>64</td>
</tr>
<tr>
<td>length</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
<tr>
<td>length</td>
<td>$power</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
<tr>
<td>length</td>
<td>$xvars</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
<tr>
<td>turn</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>turn</td>
<td>$nonpower</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>turn</td>
<td>$xvars</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>displacement</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>31</td>
</tr>
<tr>
<td>displacement</td>
<td>$power</td>
<td>integers &gt;=0</td>
<td>31</td>
</tr>
<tr>
<td>displacement</td>
<td>$xvars</td>
<td>integers &gt;=0</td>
<td>31</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$power</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$xvars</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
</tr>
<tr>
<td>foreign</td>
<td>not in vluser</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See [D] *vl list* for all the different ways it can list variable lists and variables.

### vl substitute and factor-variable operators

Factor-variable operators can be used with variable lists using *vl substitute*. Here is an example:

```
. vl substitute indepvars = i.vlcategorical##c.xvars
```

See [U] 11.4.3 Factor variables.

To see what is in *indepvars*, we use the global macro syntax with a $ in front of its name and use *display* to view its contents.

```
. display "$indepvars"
```

```
i.foreign gear_ratio displacement weight length turn rep78 i.foreign#c.gear_ratio i > .foreign#c.displacement i.foreign#c.weight i.foreign#c.length i.foreign#c.turn i. > foreign#c.rep78
```
To use variable lists with other Stata commands, we do the same thing. We treat the list name like the global macro it is and put a $ in front of it.

\[
. \text{regress mpg} \; $\text{indepvars}$
\]

<table>
<thead>
<tr>
<th>Source</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
<th>Number of obs = 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>1945.54632</td>
<td>13</td>
<td>149.657409</td>
<td>\ F(13, 55) = 20.86</td>
</tr>
<tr>
<td>Residual</td>
<td>394.656577</td>
<td>55</td>
<td>7.17557413</td>
<td>\ Prob &gt; F = 0.0000</td>
</tr>
<tr>
<td>Total</td>
<td>2340.2029</td>
<td>68</td>
<td>34.4147485</td>
<td>\ Adj R-squared = 0.7915</td>
</tr>
</tbody>
</table>

| mpg | Coefficient | Std. err. | t    | P>|t|   | [95% conf. interval] |
|-----|-------------|-----------|------|-------|-------------------|
| foreign | -32.65519 | 24.36955  | -1.34 | 0.186 | -81.49286 -16.18248 |
| gear_ratio | -.0847818 | 1.959716 | -0.04 | 0.966 | -4.012141 3.842577 |
| foreign#c.rep78 | 4.480624 | 1.10794  | 4.04 | 0.000 | 2.260263 6.700985 |
| _cons | 50.52293 | 8.553643 | 5.91 | 0.000 | 33.38104 67.66481 |

Just like all the other user-defined variable lists, variable lists created by \textit{vl substitute} are saved with the data. See \textbf{[D] vl rebuild}.

**Exploring data with vl set**

Consider a bigger dataset. It is fictitious data, designed to mimic real questionnaire data.

\[
. \text{use https://www.stata-press.com/data/r17/questionnaire, clear}
\]

\textit{vl} can be used to explore your data. It is a bit like \textit{codebook} except that \textit{codebook} provides more information. \textit{vl set}, however, is much faster. \textit{vl set} is even speedy with datasets containing millions of observations and thousands of variables.
We run `vl set` with the `list()` option, which is equivalent to using the `vl list` command. We also specify the option `nonotes` to suppress the notes at the end of the table.

```
. vl set, list(min max obs) nonotes
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
<th>Min</th>
<th>Max</th>
<th>Obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>gender</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,058</td>
</tr>
<tr>
<td>age</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>47</td>
<td>2</td>
<td>64</td>
<td>1,058</td>
</tr>
<tr>
<td>q1</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>40</td>
<td>1</td>
<td>47</td>
<td>1,048</td>
</tr>
<tr>
<td>q2</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1,046</td>
</tr>
<tr>
<td>q3</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,049</td>
</tr>
<tr>
<td>q4</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,042</td>
</tr>
<tr>
<td>q5</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,048</td>
</tr>
<tr>
<td>q6</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1,046</td>
</tr>
<tr>
<td>q7</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,047</td>
</tr>
<tr>
<td>q8</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,046</td>
</tr>
<tr>
<td>q9</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,051</td>
</tr>
<tr>
<td>q10</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,047</td>
</tr>
<tr>
<td>q11</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,046</td>
</tr>
<tr>
<td>q12</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1,052</td>
</tr>
<tr>
<td>q13</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,045</td>
</tr>
<tr>
<td>q14</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,047</td>
</tr>
<tr>
<td>q15</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>36</td>
<td>0</td>
<td>37</td>
<td>1,040</td>
</tr>
<tr>
<td>q16</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1,046</td>
</tr>
<tr>
<td>q17</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,054</td>
</tr>
<tr>
<td>q18</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1,048</td>
</tr>
<tr>
<td>q19</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1,043</td>
</tr>
<tr>
<td>q20</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>30</td>
<td>1</td>
<td>30</td>
<td>1,048</td>
</tr>
<tr>
<td>check1</td>
<td>$vlcontinuous</td>
<td>negative</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1,050</td>
</tr>
<tr>
<td>q21</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>39</td>
<td>2</td>
<td>40</td>
<td>1,048</td>
</tr>
<tr>
<td>q22</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>32</td>
<td>3</td>
<td>36</td>
<td>1,050</td>
</tr>
<tr>
<td>q23</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1,050</td>
</tr>
<tr>
<td>q24</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td>8.7</td>
<td>69.9</td>
<td>1,045</td>
<td></td>
</tr>
<tr>
<td>check8</td>
<td>$vlcontinuous</td>
<td>all missing</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
</tbody>
</table>

(output omitted)

| q45      | $vlcontinuous | noninteger      | 8.7    | 69.9 | 1,045|
| check8   | $vlcontinuous | all missing     | .      | .    | .    | 0    |

(output omitted)

| q60      | $vlcontinuous | all missing     | .      | .    | .    | 0    |

(output omitted)

| q76      | $vlcontinuous | integers >=0    | >100   | 84   | 287  | 1,051|

(output omitted)

| q161     | $vlcategorical | 0 and 1         | 2      | 0    | 1    | 1,047|

(output omitted)

| check8   | $vlcontinuous | constant        | 1      | 1    | 1    | 1,058|

Summary

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>138</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>3</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>21</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>9</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

From the summary table, we see that most of the variables were put in `$vlcategorical`. The default cutoff for the number of levels for `$vlcategorical` is 10, so these 138 variables all have 10 levels or less.
Three variables were put in `vlcontinuous`. One, `q24`, has negative values. Its values are actually only $-1$ and $1$. So it is integer with only two levels, yet it is classified as continuous. Factor variables must be nonnegative, so any variable with negative values is put into `vlcontinuous`. We need to recode `q24` as 0/1 (or 1/2, etc.) to use it as a factor variable.

The variable `q45` was put in `vlcontinuous` because it contains noninteger values.

The variable `q76` was put in `vlcontinuous` because, although it is a nonnegative integer, it has over 100 levels. The default cutoff is 100 for determining whether variables are put in `vlcontinuous` or `vluncertain`. Note that the output does not say exactly how many levels, just that the number is greater than 100.

The variable list `vluncertain` contains 21 variables. These are nonnegative integers with the number of levels $> 10$ and $\leq 100$.

The variable list `vlother` contains nine variables. These variables are either constants or all missing—variables not suitable for any statistical analyses.

### Changing the cutoffs for classification

The default classification produced by `vl set` was not very useful in this case. `vl set` put too many variables in `vlcategorical`, and it put too many in `vluncertain`. Most of the variables in `vluncertain` are integer-valued scales, and we want those in `vlcontinuous`.

We will fix this. We run `vl set` again to re-create the classifications, and this time, we specify `categorical(4)` and `uncertain(19)`, meaning that variables in `vlcategorical` can have up to 4 levels and variables with 5 to 19 levels are placed in `vluncertain`. We also specify the option `dummy` to tell `vl set` to smarten up and put all the 0/1 variables in their own classification. Finally, we specify option `clear` to clear the old classifications. See `[D] vl set`.

```
  . vl set, categorical(4) uncertain(19) dummy clear nonotes
```

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vldummy</td>
<td>99</td>
<td>0/1 variables</td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>16</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>21</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>26</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>9</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

We did not really need to create the `vldummy` variable list. Had we wanted to treat the dummy variables as factor variables, we could have let `vl set` put them in `vlcategorical`, as it would by default. Note that `vldummy` contains only 0/1 variables. A 1/2 variable is still put in `vlcategorical`.

### Moving variables from one classification to another

At this point, we are happy with the variables that are in `vlcategorical` and `vlcontinuous`. We are unhappy with having variables in `vluncertain`, and we have 26 of them! Those variables have between 5 and 19 levels. Let’s list the variables and categorize them by hand.
Many of the variables have seven levels. Let’s tabulate one of them.

```
. tabulate q18
```

<table>
<thead>
<tr>
<th>Question 18</th>
<th>Freq.</th>
<th>Percent</th>
<th>Cum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>very strongly disagree</td>
<td>136</td>
<td>12.98</td>
<td>12.98</td>
</tr>
<tr>
<td>strongly disagree</td>
<td>148</td>
<td>14.12</td>
<td>27.10</td>
</tr>
<tr>
<td>disagree</td>
<td>144</td>
<td>13.74</td>
<td>40.84</td>
</tr>
<tr>
<td>neither agree nor disagree</td>
<td>146</td>
<td>13.93</td>
<td>54.77</td>
</tr>
<tr>
<td>agree</td>
<td>173</td>
<td>16.51</td>
<td>71.28</td>
</tr>
<tr>
<td>strongly agree</td>
<td>146</td>
<td>13.93</td>
<td>85.21</td>
</tr>
<tr>
<td>very strongly agree</td>
<td>155</td>
<td>14.79</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>1,048</td>
<td>100.00</td>
<td></td>
</tr>
</tbody>
</table>

This variable contains a Likert scale and, because of that, we want to treat the variable as continuous. In fact, all the variables with seven levels are Likert scales. We move them all into vlcontinuous.

```
. vl move (q18 q35 q63 q93 q103 q111 q112 q120 q132 q157) vlcontinuous
```

<table>
<thead>
<tr>
<th>Macro</th>
<th># Added/Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vldummy</td>
<td>0</td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>0</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>10</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>-10</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
</tr>
</tbody>
</table>
Now we can list the remaining `vluncertain` variables.

```
. vl list vluncertain
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>q12</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q23</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>10</td>
</tr>
<tr>
<td>q27</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>8</td>
</tr>
<tr>
<td>q28</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>15</td>
</tr>
<tr>
<td>q39</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q54</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>10</td>
</tr>
<tr>
<td>q66</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q80</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q81</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q92</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q99</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q119</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>8</td>
</tr>
<tr>
<td>q124</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>14</td>
</tr>
<tr>
<td>q127</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>q135</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>10</td>
</tr>
<tr>
<td>q141</td>
<td><code>$vluncertain</code></td>
<td>integers &gt;=0</td>
<td>12</td>
</tr>
</tbody>
</table>

You can decide for yourself where they go and use `vl move` to place them.

**Dropping variables and rebuilding variable lists**

We have variables in `vlother`.

```
. vl list vlother
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>check1</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>check2</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>q60</td>
<td><code>$vlother</code></td>
<td>all missing</td>
<td></td>
</tr>
<tr>
<td>check3</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>check4</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>check5</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>check6</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>check7</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>check8</td>
<td><code>$vlother</code></td>
<td>constant</td>
<td>1</td>
</tr>
</tbody>
</table>

We could use `vl drop` to remove them from the `vl` system classification. But we do not want them in our dataset, so we drop them.

```
. drop $vlother
```

Now if we run

```
. vl list
```

```
variable check1 not found
Run vl rebuild to rebuild vl macros.
```

```
r(111);
```

we get an error! `vl` keeps track of all the variables put into variable lists, and whenever a `vl` command is run, it first checks that everything is okay. It discovered missing variables and needs confirmation that this is intentional. If it is, we `vl rebuild` the system.
. vl rebuild
Rebuilding vl macros ...

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vldummy</td>
<td>99</td>
<td>0/1 variables</td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>16</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>31</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>16</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

### Changing variables and updating variable lists

If you change the values of a variable, you need to `vl set` the variable again to update its statistics. You can update its statistics leaving its classification unchanged or tell `vl set` to redo the classification as well.

We noticed that `age` had a suspiciously low minimum.

```
. vl list (age), min max obs
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
<th>Min</th>
<th>Max</th>
<th>Obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>&gt;19</td>
<td>2</td>
<td>64</td>
<td>1,058</td>
</tr>
</tbody>
</table>

We do not believe a two-year-old took our questionnaire. Let’s find the ID of this subject.

```
. list id age if age == 2
```

<table>
<thead>
<tr>
<th>id</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>543.</td>
<td>05034558 2</td>
</tr>
</tbody>
</table>

We check our original data source and discover that the subject was 20 years old. We correct the value of `age`.

```
. replace age = 20 if id == "05034558" & age == 2
```

(1 real change made)

Now the minimum of `age` stored by `vl` is wrong. We could ignore it, or we could fix it by using the `update` option of `vl set`. The option `update` does not change the classification of a variable; it only updates the stored statistics.
. vl set age, update list(min max obs) nonotes

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
<th>Min</th>
<th>Max</th>
<th>Obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>$vlcontinuous</td>
<td>integers &gt;=0</td>
<td>47</td>
<td>18</td>
<td>64</td>
<td>1,058</td>
</tr>
</tbody>
</table>

Summary

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vldummy</td>
<td>99</td>
<td>0/1 variables</td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>16</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>31</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>16</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

If we wanted to redo the classification of *age* and update its statistics, we would type

. vl set age, redo

(output omitted)

**Saving and using datasets with variable lists**

When we save our data, the *vl* system is saved.

. save quest_with_vl
file *quest_with_vl.dta* saved

However, when we use our data, the *vl* system is not automatically restored.

. use quest_with_vl
(Fictitious Questionnaire Data)

Type *vl rebuild* to bring the system back to life.

. vl rebuild
Rebuilding *vl* macros ...

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vldummy</td>
<td>99</td>
<td>0/1 variables</td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>16</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>31</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>16</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

See [D] *vl rebuild* for other instances when you need to run *vl rebuild*. 
User-defined variable lists and factor-variable operators

We continue with our previous example using fictitious questionnaire data.

The system-defined variable lists are good for organizing variables. We would not use them, however, to specify `varlists` for estimation commands if for no other reason than we do not want to use all the variables in the dataset. For this purpose, we need to create user-defined variable lists.

Here is a variable list containing demographic variables we want to use for model fitting.

```
.vl create demographics = (gender q3 q4 q5)
note: $demographics initialized with 4 variables.
```

We are going to create two more variable lists: `factors`, containing variables we want to treat as factor variables, and `control_scales`, containing variables we want to treat as continuous.

```
.vl create factors = vldummy + vlcategorical
note: $factors initialized with 115 variables.
.vl create control_scales = (q15 q20 q21 q22)
note: $control_scales initialized with 4 variables.
```

This is the real power of `vl`. We created `factors` from `vldummy` plus `vlcategorical`. But `factors` contains variables in `demographics`, and we want to handle the `demographics` variables differently. So we remove them from `factors`. We also remove some other variables we do not want in our model.

```
.vl modify factors = factors - demographics
note: 4 variables removed from $factors.
.vl modify factors = factors - (q155 q156 q158)
note: 3 variables removed from $factors.
```

We are going to fit a `poregress` model, and our variables of interest (ones for which we want to do inference) are the categorical variables `q7`, `q13`, and `q16`, and the continuous variable `q35`.

We create a variable list with the categorical ones, and remove them from `factors`.

```
.vl create fvofointerest = (q7 q13 q16)
note: $fvofointerest initialized with 3 variables.
.vl modify factors = factors - fvofointerest
note: 3 variables removed from $factors.
```

Now we use `vl substitute` to create a variable list that contains factor variables.

```
.vl substitute interest = i.fvofointerest q35
```

Notice that we tucked the continuous variable `q35` in at the end. `vl substitute` lets you specify variable lists and variables by using factor-variable operators—or not—in a natural way.

If you want to see the contents of a variable list created using `vl substitute`, you can `display` it:

```
.display "$interest"
  i.q7 i.q13 i.q16 q35
```

The one thing to remember about `vl substitute` is that it is a one-shot deal. Once the variable list is created, you cannot modify it. If you want to change it, you must delete it using `vl drop` and then re-create it using `vl substitute`.

We are going to go nuts and create a variable list consisting of bushels of interactions.

```
.vl substitute controlvars = i.demographics i.factors##c.control_scales
```
The interest variable list contains our variables of interest for `poregress`. The controlvars variable list contains control variables for the model.

```
.poregress q1 $interest, controls($controlvars)
```

Estimating lasso for q1 using plugin
Estimating lasso for 1bn.q7 using plugin
Estimating lasso for 1bn.q13 using plugin
Estimating lasso for 2bn.q16 using plugin
Estimating lasso for 3bn.q16 using plugin
Estimating lasso for q35 using plugin

Partialing-out linear model

| Number of obs = 339 |
| Number of controls = 1,137 |
| Number of selected controls = 12 |
| Wald chi2(5) = 12.89 |
| Prob > chi2 = 0.0244 |

| q1 | Robust Coefficient | std. err. | z | P>|z| | [95% conf. interval] |
|-----|--------------------|-----------|---|-----|--------------------------|
| q7  | yes                | -1.333003 | .7441531 | -1.79 | 0.073 | -2.791516 | .1255107 |
| q13 | yes                | .4321797 | .684376 | 0.63 | 0.528 | -.9091725 | 1.773532 |
| q16 | 2                  | .6905278 | .8355682 | 0.83 | 0.409 | -.9471559 | 2.328211 |
|     | 3                  | 2.497944 | .8572828 | 2.91 | 0.004 | .8177008 | 4.178188 |
|     | q35                | -.1238627 | .1833827 | -0.68 | 0.499 | -.4832861 | .2355608 |

Note: Chi-squared test is a Wald test of the coefficients of the variables of interest jointly equal to zero. Lassos select controls for model estimation. Type lassoinfo to see number of selected variables in each lasso.

Using vl, we can specify huge `varlists` in a succinct notation. If we were to list the expanded estimation command, it would take half a page!

**Updating variable lists created by vl substitute**

What is especially convenient about variable lists is how easy they are to modify. Suppose we decide we do not want `q13` in our model. We cannot explicitly change `interest` because it was created by `vl substitute`, but we can change `fvofinterest`.

```
.vl modify fvofofinterest = fvofofinterest - (q13)
note: 1 variable removed from $fvofofinterest.
```
We now update interest using vl rebuild.

```
    . vl rebuild
    Rebuilding vl macros ...
```

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vldummy</td>
<td>99</td>
<td>0/1 variables</td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>16</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>31</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>16</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
<tr>
<td>User</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$demographics</td>
<td>4</td>
<td>variables</td>
</tr>
<tr>
<td>$factors</td>
<td>105</td>
<td>variables</td>
</tr>
<tr>
<td>$control_scales</td>
<td>4</td>
<td>variables</td>
</tr>
<tr>
<td>$fvofofinterest</td>
<td>2</td>
<td>variables</td>
</tr>
<tr>
<td>$interest</td>
<td></td>
<td>factor-variable list</td>
</tr>
<tr>
<td>$controlvars</td>
<td></td>
<td>factor-variable list</td>
</tr>
</tbody>
</table>

And we see that q13 is gone from our variable list.

```
    . display "$interest"
    i.q7 i.q16 q35
```

Also see

[D] vl create — Create and modify user-defined variable lists
[D] vl drop — Drop variable lists or variables from variable lists
[D] vl list — List contents of variable lists
[D] vl rebuild — Rebuild variable lists
[D] vl set — Set system-defined variable lists
vl create — Create and modify user-defined variable lists

Description

vl create creates user-defined variable lists.

vl modify modifies existing user-defined variable lists.

vl substitute creates a variable list using factor-variable operators operating on variable lists.

After creating a variable list called vlusername, the expression $vlusername can be used in Stata anywhere a varlist is allowed. Variable lists are actually global macros, and the vl commands are a convenient way to create and manipulate them. They are saved with the dataset. See [D] vl rebuild.

For an introduction to the vl commands, see [D] vl.

Quick start

Create a variable list

vl create demographics = (age_cat gender)

Add variables to a variable list

vl modify demographics = demographics + (educ_cat income_cat)

Add the variables in the variable list named othervars to the existing variable list called myxvars

vl modify myxvars = myxvars + othervars

Remove the variable x8 from the variable list

vl modify myxvars = myxvars - (x8)

Apply factor-variable operator i. to all the variables in a variable list

vl substitute idemographics = i.demographics

Create interactions between the levels of the variables in the variable list demographics and the continuous variables in the variable list vlcontinuous

vl substitute myinteractions = i.demographics#c.vlcontinuous

Run a regression specifying the independent variables using variable lists

regress y $idemographics $myxvars $myinteractions
Syntax

Create user-defined variable lists

vl create vlusername = (varlist)
vl create vlusername = vname + | - (varlist)
vl create vlusername = vname1 [ + | - vname2]

Modify user-defined variable lists

vl modify vlusername = (varlist)
vl modify vlusername = vname + | - (varlist)
vl modify vlusername = vname1 [ + | - vname2]

Apply factor-variable operators to variable-list names

vl substitute vlusername = i.vname
vl substitute vlusername = i.vname1##c.vname2

Label a user-defined variable-list name

vl label vlusername [ "label"]

vname is an existing user-defined variable-list name or a system-defined variable-list name. When specifying varlist, it is always enclosed in parentheses: (varlist). See [D] vl.

Remarks and examples

Remarks are presented under the following headings:
vl create
vl modify
Using variable lists with other Stata commands
vl substitute

vl create

vl create creates a new variable list. It can be created from a list of variables:

. vl create myxvars = (x1-x100)

In the above, note that the varlist is enclosed in parentheses. varlists must always be enclosed in parentheses.

When we are discussing the vl commands and say “variable list,” we mean a named variable list created by vl create or vl set. In this case, we created the variable list myxvars. A traditional Stata list of variables, that is, a varlist, we will call a varlist.

A new variable list also can be created from an existing variable list:

. vl create indepvars = myxvars
vl modify

vl modify is the same as vl create, except that vl modify cannot create new variables lists, and vl create cannot modify existing lists.

The operator + can be used to take the union of two variable lists with duplicates removed.

. vl modify indepvars = myxvars + othervars

The operator - can be used to obtain the difference of two variable lists.

. vl modify indepvars = myxvars - othervars

Now indepvars contains the variables that are in myxvars excluding any that are in othervars. If there are variables in othervars that are not in myxvars, it is not an error. These variables are simply ignored.

The + and - operators can be used with varlists as well.

. vl modify indepvars = myxvars + (w1 w2 w3)

(varlist) must be specified after + or -, never before.

To list the variables in a variable list, use vl list. To see a directory of variable lists that have been created, type vl dir. See [D] vl list for details on these two commands.

vl label attaches a label to the variable list that is displayed by vl dir.

. vl label indepvars "My brilliant choice of variables"

To delete indepvars, type

. vl drop indepvars

vl drop has other uses too; see [D] vl drop.

Using variable lists with other Stata commands

To use variable lists with other Stata commands, type $ in front of the variable-list name. Remember: With the vl commands, do not use $. With other Stata commands, use $.

. display "$indepvars"
. summarize $indepvars
. regress y $indepvars

If you know Stata, you will have already figured out that variable lists are global macros. But the vl system is more than another way to create global macros. For instance, variable lists are saved with the dataset. Global macros are not. Both variable lists and other vl system information are saved. To make the vl system come back to life in the state we last had it, after we use a dataset, we type

. vl rebuild

See [D] vl rebuild.

vl substitute

Factor-variable operators can be used with variable lists. There are two ways to do this.

The first is to use factor-variable operators on the global macro form of the variable list like so:

. regress y i.($myfactors)#c.($mycontinuous)
Here myfactors is a user-defined variable list containing variables you want treated as factors. mycontinuous are variables you want treated as continuous. Specifying i(...)##c(...) means you want main effects of the factors plus interactions of all their levels with the continuous variables. Note that the parentheses, (), are required.

A second way to use factor-variable operators with variable lists is with the command vl substitute. For example,

```
. vl substitute myinteractions = i.myfactors##c.mycontinuous
. regress y $myinteractions
```

would produce the same result as the previous command. However, using vl substitute has the advantage that the variable lists it creates will be saved with your dataset, just like any other variable list.

See [U] 11.4.3 Factor variables.

You can mix variable names with names of variable lists:

```
. vl substitute myinteractions = i.gender##c.(mycontinuous x100)
```

Here gender and x100 are variable names and mycontinuous is a variable list.

Be careful when mixing variable names and names of variable lists. vl substitute first assumes names are names of variable lists. Then it looks for variable names. For example, if you have both a variable named x and a variable list named x, and you specify

```
. vl substitute myinteractions = i.gender##c.(mycontinuous x)
```

then vl substitute will assume x is the variable list.

Using vl substitute to create a user-defined variable list is a one-shot deal. These variable lists cannot be modified after they are created. If you want to change them, first drop them,

```
. vl drop myinteractions
```

and then define them again:

```
. vl substitute myinteractions = i.myfactors##c.mycontinuous
```

For examples using vl create, vl modify, and vl substitute, see [D] vl.

Also see

[D] vl — Manage variable lists
[D] vl drop — Drop variable lists or variables from variable lists
[D] vl list — List contents of variable lists
[D] vl rebuild — Rebuild variable lists
[D] vl set — Set system-defined variable lists
vl drop — Drop variable lists or variables from variable lists

### Description

vl drop *vlusername* deletes user-defined variable lists.

vl drop *vlsysname* zeros system-defined variable lists. They still exist but are empty.

vl drop *(varlist)* removes variables from all variable lists.

vl clear deletes all variable lists and removes all traces of the vl system.

For an introduction to the vl commands, see [D] vl.

### Quick start

Delete the user-defined variable list *myfav*

```
vl drop myfav
```

Zero the system-defined variable list *vluncertain*

```
vl drop vluncertain
```

Drop the variables *x1* and *x2* from all variable lists

```
vl drop (x1 x2)
```

As above, but only drop them from user-defined variable lists

```
vl drop (x1 x2), user
```

Delete all variable lists and all traces of the vl system

```
vl clear
```

Delete all user-defined variable lists

```
vl clear, user
```

Delete all system-defined variable lists and the stored variable statistics

```
vl clear, system
```
Syntax

Drop variable lists
   vl drop vlnamelist [, system user]

Drop variables from variable lists
   vl drop (varlist) [, system user]

Clear all variable lists
   vl clear [, system user]

vlnamelist is a list of variable-list names. (_all) or (*) can be used to specify all numeric variables in the dataset.

Options

system when specified with vl drop (varlist), drops the variables in varlist only from system-defined variable lists. By default, variables are dropped from all variable lists, both system-defined and user-defined.

When specified with vl clear, only the system-defined variable lists are deleted. By default, both the system-defined and user-defined variable lists are deleted, and all traces of the vl system are gone.

user when specified with vl drop (varlist), drops the variables in varlist only from user-defined variable lists.

When specified with vl clear, only the user-defined variable lists are deleted.

Remarks and examples

When given one or more names of user-defined variable lists, vl drop deletes them. That is, typing

   . vl drop myname

deltes the user-defined variable list myname. It is as if myname was never created. A new variable list called myname can now be created using vl create.

When given one or more names of system-defined variable lists, vl drop zeros them. That is, typing

   . vl drop vluncertain

zeros the system-defined variable list vluncertain. It still exists but is empty. A single system-defined variable list cannot be deleted.

All system-defined variable lists can be deleted using

   . vl clear, system
All system-defined variable lists are now gone. Also deleted are the stored variable statistics, namely, the number of levels, minimum and maximum values, and the number of nonmissing observations. It is as if \texttt{vl set} was never run.

Typing

\texttt{. vl clear}

deletes all variable lists and all traces of the \texttt{vl} system.

Typing

\texttt{. vl drop (varlist)}

removes the variables in \texttt{varlist} from all variable lists.

Say we only want to remove variable \texttt{x8} from the user-defined variable list \texttt{mylist}. To do this, we type

\texttt{. vl modify mylist = mylist - (x8)}

Note the parentheses around \texttt{x8}; see \texttt{[D] vl create}.

Say you want to remove variable \texttt{x8} from the system-defined variable list \texttt{v1categorical}. System-defined variable lists are disjoint, so a variable is only in one of them. Thus, we can remove it by typing

\texttt{. vl drop (x8), system}

Rather than drop it, we could have moved it to the system-defined variable list \texttt{vlother}.

\texttt{. vl move (x8) vlother}

See \texttt{[D] vl set}.

\textbf{Also see}

\texttt{[D] vl} — Manage variable lists

\texttt{[D] vl create} — Create and modify user-defined variable lists

\texttt{[D] vl list} — List contents of variable lists

\texttt{[D] vl rebuild} — Rebuild variable lists

\texttt{[D] vl set} — Set system-defined variable lists
Title

vl list — List contents of variable lists

Description

vl list shows the contents of variable lists when given names of variable lists. When given names of variables, it shows the variable lists to which each variable belongs.

vl dir shows the names of all variable lists.

For an introduction to the vl commands, see [D] vl.

Quick start

Show the contents of all variable lists
    vl list

Show the contents of the system-defined variable list vlcategorical
    vl list vlcategorical

Show the contents of the user-defined variable list myfav
    vl list myfav

Show the variable lists to which x1-x100 belong
    vl list (x1-x100)

Show the variable lists to which every numeric variable belongs
    vl list (*)

Show the contents of all system-defined variable lists
    vl list, system

Show the contents of all user-defined variable lists
    vl list, user

Show the contents of all variable lists, and show the minimum value, maximum value, and number of nonmissing values for each variable
    vl list, minimum maximum observations

Show the contents of all variable lists, ordered by variable list and then alphabetically by variable name
    vl list, sort

Show the variable lists to which every numeric variable belongs, ordered alphabetically by variable name and then by variable list
    vl list (*), sort
Syntax

Show the contents of variable lists

```
vl list [vlnamelist] [, options]
```

Show the variable lists to which variables belong

```
vl list (varlist) [, options]
```

Show names of all variable lists

```
vl dir [, system user]
```

`vlnamelist` is a list of variable-list names.

(`_all`) or (*) can be used to specify all numeric variables in the dataset.

<table>
<thead>
<tr>
<th><code>options</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>system</code></td>
<td>show only system-defined variable lists</td>
</tr>
<tr>
<td><code>user</code></td>
<td>show only user-defined variable lists</td>
</tr>
<tr>
<td><code>minimum</code></td>
<td>show minimum value of each variable</td>
</tr>
<tr>
<td><code>maximum</code></td>
<td>show maximum value of each variable</td>
</tr>
<tr>
<td><code>observations</code></td>
<td>show number of nonmissing observations of each variable</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>order by variable list and then alphabetically by variable name when <code>vlnamelist</code> is specified; order alphabetically by variable name and then by variable list when <code>(varlist)</code> is specified</td>
</tr>
<tr>
<td><code>strok</code></td>
<td>allow string variables when <code>(varlist)</code> is specified</td>
</tr>
<tr>
<td><code>nolstretch</code></td>
<td>do not stretch the width of the table to accommodate long names</td>
</tr>
</tbody>
</table>

`collect` is allowed with `vl list` and `vl dir`; see [U] 11.1.10 Prefix commands.

Options

`system` specifies that only system-defined variable lists be shown. By default, both system-defined and user-defined variable lists are shown.

`user` specifies that only user-defined variable lists be shown.

`minimum` specifies that the minimum value of each variable be displayed.

`maximum` specifies that the minimum value of each variable be displayed.

`observations` specifies that number of nonmissing observations of each variable be displayed.

`sort` specifies that the listing be sorted. When `vlnamelist` is specified, the listing is ordered by variable list and then alphabetically by variable name. By default in this case, variables are listed in the order in which they were added to the variable list.

When `(varlist)` is specified, the listing is ordered alphabetically by variable name and then by variable list. By default in this case, variables are listed in the order in which they appear in `varlist`.

`strok` specifies that string variables be included in the listing when `(varlist)` is specified. By default, specifying string variables in `varlist` gives an error message. Specifying `strok` prevents this error message and lists any string variables.
nolstretch specifies that the width of the table not be automatically widened to accommodate long variable and variable-list names. When nolstretch is specified, names are abbreviated to make the table width no more than 79 characters. The default, lstretch, is to automatically widen the table up to the width of the Results window. To change the default, use set lstretch off.

Remarks and examples

vl list produces two types of listings. The first lists by variable-list name and then by variable name. The second is the reverse; it lists by variable name and then by variable-list name.

Typing

   . vl list

produces the first type of listing. This listing is useful when you want to see the contents of each variable list.

Typing

   . vl list (*)

or

   . vl list (x1-x100)

produces the second type of listing. This listing is useful when you want to see all variable lists to which a variable belongs.

System-defined variable lists are disjoint, so a variable can only belong to one of them. There is no such restriction on user-defined variable lists. Variables can belong to more than one user-defined variable list.

Typing

   . vl dir

shows all the variable lists, both system-defined and user-defined. The options system and user work with both vl list and vl dir to restrict the output accordingly.

Example 1: Showing the contents of variable lists

We show examples using Stata’s automobile dataset because it has only a small number of variables and the output will not be too lengthy.

   . sysuse auto
   (1978 automobile data)

We run vl set with the option nonotes to suppress the notes at the end of the output.

   . vl set, nonotes

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>2</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>2</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>7</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>
Let’s list the contents of the variable lists.

. vl list

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values Description</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep78</td>
<td>$vlcategorical</td>
<td>integers &gt;=0</td>
<td>5</td>
</tr>
<tr>
<td>foreign</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
</tr>
<tr>
<td>headroom</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>74</td>
</tr>
<tr>
<td>mpg</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>21</td>
</tr>
<tr>
<td>trunk</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>weight</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>64</td>
</tr>
<tr>
<td>length</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
<tr>
<td>turn</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>displacement</td>
<td>$vluncertain</td>
<td>integers &gt;=0</td>
<td>31</td>
</tr>
</tbody>
</table>

We decide to treat all the variables in $vluncertain as continuous, so we move them to $vlcontinuous. Then we run vl dir to confirm that $vluncertain is empty.

. vl move $vluncertain $vlcontinuous

note: 7 variables specified and 7 variables moved.

<table>
<thead>
<tr>
<th>Macro</th>
<th># Added/Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vlcategorical</td>
<td>0</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>7</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>-7</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
</tr>
</tbody>
</table>

. vl dir

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>2</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>9</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>0</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
</tbody>
</table>

Let’s create two user-defined variable lists.

. vl create power = (gear_ratio weight displacement)

note: $power initialized with 3 variables.

. vl create other = (price turn length)

note: $other initialized with 3 variables.
Let's do a listing ordered by variable list. We specify options to see the minimum and maximum values and the number of nonmissing observations for each variable.

```
. vl list, minimum maximum observations
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
<th>Min</th>
<th>Max</th>
<th>Obs</th>
</tr>
</thead>
<tbody>
<tr>
<td>rep78</td>
<td>$vlcategorical</td>
<td>integers $&gt;=0</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>69</td>
</tr>
<tr>
<td>foreign</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>74</td>
</tr>
<tr>
<td>headroom</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td>1.5</td>
<td>5</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td>2.19</td>
<td>3.89</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>price</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>74</td>
<td>3291</td>
<td>15906</td>
<td>74</td>
</tr>
<tr>
<td>mpg</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>21</td>
<td>12</td>
<td>41</td>
<td>74</td>
</tr>
<tr>
<td>trunk</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>18</td>
<td>5</td>
<td>23</td>
<td>74</td>
</tr>
<tr>
<td>weight</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>64</td>
<td>1760</td>
<td>4840</td>
<td>74</td>
</tr>
<tr>
<td>length</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>47</td>
<td>142</td>
<td>233</td>
<td>74</td>
</tr>
<tr>
<td>turn</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>18</td>
<td>31</td>
<td>51</td>
<td>74</td>
</tr>
<tr>
<td>displacement</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>31</td>
<td>79</td>
<td>425</td>
<td>74</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$power</td>
<td>noninteger</td>
<td>2.19</td>
<td>3.89</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>weight</td>
<td>$power</td>
<td>integers $&gt;=0</td>
<td>64</td>
<td>1760</td>
<td>4840</td>
<td>74</td>
</tr>
<tr>
<td>displacement</td>
<td>$power</td>
<td>integers $&gt;=0</td>
<td>31</td>
<td>79</td>
<td>425</td>
<td>74</td>
</tr>
<tr>
<td>price</td>
<td>$other</td>
<td>integers $&gt;=0</td>
<td>74</td>
<td>3291</td>
<td>15906</td>
<td>74</td>
</tr>
<tr>
<td>turn</td>
<td>$other</td>
<td>integers $&gt;=0</td>
<td>18</td>
<td>31</td>
<td>51</td>
<td>74</td>
</tr>
<tr>
<td>length</td>
<td>$other</td>
<td>integers $&gt;=0</td>
<td>47</td>
<td>142</td>
<td>233</td>
<td>74</td>
</tr>
</tbody>
</table>

Specifying (*) means that we want a listing ordered by variable name.

```
. vl list (*)
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>price</td>
<td>$other</td>
<td>integers $&gt;=0</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>mpg</td>
<td>not in vluser</td>
<td></td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>$vlcategorical</td>
<td>integers $&gt;=0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>rep78</td>
<td>not in vluser</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>headroom</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>headroom</td>
<td>not in vluser</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trunk</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>trunk</td>
<td>not in vluser</td>
<td></td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>$power</td>
<td>integers $&gt;=0</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>$other</td>
<td>integers $&gt;=0</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>turn</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>turn</td>
<td>$other</td>
<td>integers $&gt;=0</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>displacement</td>
<td>$vlcontinuous</td>
<td>integers $&gt;=0</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>displacement</td>
<td>$power</td>
<td>integers $&gt;=0</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$vlcontinuous</td>
<td>noninteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$power</td>
<td>noninteger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>$vlcategorical</td>
<td>0 and 1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>not in vluser</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Variables are listed multiple times showing all the variable lists to which each belongs. We can restrict the listing to user-defined variable lists.

```
.vl list (*), user
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Values</th>
<th>Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>price</td>
<td>$other</td>
<td>integers &gt;=0</td>
<td>74</td>
</tr>
<tr>
<td>mpg</td>
<td>not in vluser</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>rep78</td>
<td>not in vluser</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>headroom</td>
<td>not in vluser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trunk</td>
<td>not in vluser</td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>weight</td>
<td>$power</td>
<td>integers &gt;=0</td>
<td>64</td>
</tr>
<tr>
<td>length</td>
<td>$other</td>
<td>integers &gt;=0</td>
<td>47</td>
</tr>
<tr>
<td>turn</td>
<td>$other</td>
<td>integers &gt;=0</td>
<td>18</td>
</tr>
<tr>
<td>displacement</td>
<td>$power</td>
<td>integers &gt;=0</td>
<td>31</td>
</tr>
<tr>
<td>gear_ratio</td>
<td>$power</td>
<td>noninteger</td>
<td></td>
</tr>
<tr>
<td>foreign</td>
<td>not in vluser</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

See the lines “not in vluser”? They are omitted if you run `vl list, user`.

Let’s use `vl substitute` with factor-variable operators to create interactions between the variables in the system-defined variable list, `$vlcategorical`, and the variables in our user-defined variable list, `$mycontinuous`.

```
.vl substitute indepvars = i.$vlcategorical##c.(power other)
```

The factor-variable list `indepvars` shows up when we run `vl dir`.

```
.vl dir
```

<table>
<thead>
<tr>
<th>Macro</th>
<th># Vars</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$vlcategorical</td>
<td>2</td>
<td>categorical variables</td>
</tr>
<tr>
<td>$vlcontinuous</td>
<td>9</td>
<td>continuous variables</td>
</tr>
<tr>
<td>$vluncertain</td>
<td>0</td>
<td>perhaps continuous, perhaps categorical variables</td>
</tr>
<tr>
<td>$vlother</td>
<td>0</td>
<td>all missing or constant variables</td>
</tr>
<tr>
<td>User</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$power</td>
<td>3</td>
<td>variables</td>
</tr>
<tr>
<td>$other</td>
<td>3</td>
<td>variables</td>
</tr>
<tr>
<td>$indepvars</td>
<td></td>
<td>factor-variable list</td>
</tr>
</tbody>
</table>

Factor-variable lists do not work with `vl list`. But you can display their contents because variable lists are global macros. You can list the contents of a variable list by typing

```
.display "$indepvars"
```

```
i.rep78 i.foreign gear_ratio weight displacement price turn length i.rep78#c.gear_ratio
> atio i.rep78#c.weight i.rep78#c.displacement i.rep78#c.price i.rep78#c.turn i.rep78
> 78#c.length i.foreign#c.gear_ratio i.foreign#c.weight i.foreign#c.displacement i.foreign#c.price i.foreign#c.turn i.foreign#c.length
```
Stored results

vl list stores the following in \( r() \):

Scalars
- \( r(k) \)  
  - number of variables listed
- \( r(\text{system}) \)  
  - number of variables listed in system-defined variable lists
- \( r(\text{not\_system}) \)  
  - number of variables listed not in system-defined variable lists
- \( r(\text{vlcategorical}) \)  
  - number of variables listed in \( \text{vlcategorical} \)
- \( r(\text{vlcontinuous}) \)  
  - number of variables listed in \( \text{vlcontinuous} \)
- \( r(\text{vluncertain}) \)  
  - number of variables listed in \( \text{vluncertain} \)
- \( r(\text{vlother}) \)  
  - number of variables listed in \( \text{vlother} \)
- \( r(\text{vldummy}) \)  
  - number of variables listed in \( \text{vldummy} \) when defined
- \( r(\text{not\_user}) \)  
  - number of variables listed not in user-defined variable lists
- \( r(\text{vlusername}) \)  
  - number of variables listed in \( \text{vlusername} \)
- \( r(\text{string}) \)  
  - number of string variables listed when \( \text{strok} \) specified

Macros
- \( r(\text{vlsysnames}) \)  
  - names of all system-defined variable lists
- \( r(\text{vlusernames}) \)  
  - names of all user-defined variable lists

vl dir stores the following in \( r() \):

Scalars
- \( r(\text{system}) \)  
  - number of variables in system-defined variable lists
- \( r(\text{vlcategorical}) \)  
  - number of variables in \( \text{vlcategorical} \)
- \( r(\text{vlcontinuous}) \)  
  - number of variables in \( \text{vlcontinuous} \)
- \( r(\text{vluncertain}) \)  
  - number of variables in \( \text{vluncertain} \)
- \( r(\text{vlother}) \)  
  - number of variables in \( \text{vlother} \)
- \( r(\text{vldummy}) \)  
  - number of variables in \( \text{vldummy} \) when defined
- \( r(\text{user}) \)  
  - number of variables in user-defined variable lists
- \( r(\text{vlusername}) \)  
  - number of variables in \( \text{vlusername} \)
- \( r(\text{string}) \)  
  - number of string variables listed when \( \text{strok} \) specified

Macros
- \( r(\text{vlsysnames}) \)  
  - names of system-defined variable lists
- \( r(\text{vlusernames}) \)  
  - names of user-defined variable lists

Also see

[D] vl — Manage variable lists
[D] vl create — Create and modify user-defined variable lists
[D] vl drop — Drop variable lists or variables from variable lists
[D] vl rebuild — Rebuild variable lists
[D] vl set — Set system-defined variable lists
Description

vl rebuild restores system-defined and user-defined variable lists. After loading a dataset with use, run vl rebuild.

After using merge or append, run vl rebuild to merge variable lists. You only need to run vl rebuild when the using dataset has variable lists.

After dropping variables with drop, run vl rebuild to remove the dropped variables from all variable lists.

After modifying variable lists with vl modify or vl move, run vl rebuild to update variable lists created by vl substitute.

And if you are confused, know that it never hurts to run vl rebuild.

For an introduction to the vl commands, see [D] vl.

Quick start

Restore variable lists after loading a dataset with use

vl rebuild

After running merge when the using dataset has variable lists, merge its variable lists into those in the master dataset

vl rebuild

After dropping variables with drop, remove the dropped variables from all variable lists

vl rebuild

Update a variable list created by vl substitute after modifying any of its component variable lists

vl rebuild

Syntax

vl rebuild

collect is allowed; see [U] 11.1.10 Prefix commands.
Remarks and examples

Remarks are presented under the following headings:

- Reloading datasets
- Merging datasets
- Dropping variables
- vl substitute and vl rebuild
- Characteristics

Reloading datasets

System-defined and user-defined variable lists are saved with the dataset. However, they are not automatically restored when you reload the data. Just type `vl rebuild` to restore them.

```plaintext
. use ...
. vl rebuild
```

Merging datasets

Another time when `vl rebuild` is needed is when a `merge` is done and the `using` dataset has variable lists.

```plaintext
. merge ... using filename
. vl rebuild
```

Only when `filename` has variable lists is it necessary to run `vl rebuild`. When both the master dataset in memory and `filename` have variable lists, `vl rebuild` merges them. When the master dataset has variable lists but `filename` does not, there is no need to run `vl rebuild`. However, running `vl rebuild` is always harmless.

Dropping variables

When you drop variables from the data in memory using `drop`, the dropped variables are not automatically removed from variable lists. They can be explicitly removed by using `vl drop`.

```plaintext
. drop varlist
. vl drop (varlist)
```

Instead of running `vl drop` with the list of variables that were dropped, you can simply type

```plaintext
. vl rebuild
```

It will do the same thing, and you do not have to remember the names of the variables that were dropped.

If you drop or add observations or change any of the values of variables in variable lists, `vl rebuild` does not update the stored variable statistics, namely, the number of levels, the minimum and maximum values, and the number of nonmissing observations. If you want to update these statistics without changing the system-defined classifications, type

```plaintext
. vl set, update
```

If you want to update the statistics and redo the system-defined classifications for all variables, type

```plaintext
. vl set, clear
```

See [D] vl set.
vl substitute and vl rebuild

vl rebuild has another important use. It will update variable lists created by vl substitute. For example, we created two user-defined variable lists:

```
. vl create myfactors = (x1 x2 x3)
. vl create mycontinuous = (c1 c2 c3 c4 c5)
```

Then we created a variable list using factor-variable operators:

```
. vl substitute myinteraction = i.myfactors##c.mycontinuous
```

If we modify mycontinuous,

```
. vl modify mycontinuous = mycontinuous - (c3)
```

then the global macro `$myinteraction` for the variable list `myinteraction` remains unchanged. Running

```
. vl rebuild
```

updates the global macro `$myinteraction`.

Again, if you make any changes to your data or to your variable lists, and you want to make sure everything is set properly and up to date, just type

```
. vl rebuild
```

Characteristics

Advanced Stata users will likely guess how variable lists and variable statistics are stored with the dataset. They are stored as characteristics. If you want to see them, type

```
. char list
```

See `[P] char`. 

Stored results

vl rebuild stores the following in `r()`:

Scalars

- `r(k_system)` number of variables in system-defined variable lists
- `r(k_vlcategorical)` number of variables in vl_categorical
- `r(k_vlcontinuous)` number of variables in vl_continuous
- `r(k_vluncertain)` number of variables in vl_uncertain
- `r(k_vlother)` number of variables in vl_other
- `r(k_vldummy)` number of variables in vl_dummy when defined
- `r(k_user)` number of variables in user-defined variable lists
- `r(k_vlusername)` number of variables in vl_username

Macros

- `r(vlsysnames)` names of system-defined variable lists
- `r(vlusernames)` names of user-defined variable lists
Also see

[D] vl — Manage variable lists
[D] vl create — Create and modify user-defined variable lists
[D] vl drop — Drop variable lists or variables from variable lists
[D] vl list — List contents of variable lists
[D] vl set — Set system-defined variable lists
vl set — Set system-defined variable lists

Description

vl set is designed to identify variables that are to be treated as factor variables in Stata’s estimation commands.

vl set creates the system-defined variable lists vlcategorical, vlcontinuous, vluncertain, and vlother. Variables are placed in them based on their values (integer or noninteger, all nonnegative, etc.) and default or user-specified cutoffs for the number of levels in a variable.

vl move moves variables from one classification to another.

Variable lists are actually global macros, and they are saved with the dataset. See [D] vl rebuild.

For an introduction to the vl commands, see [D] vl.

Quick start

Classify all numeric variables in the dataset

vl set

As above, and include a vldummy classification for 0/1 variables

vl set, dummy

Classify all numeric variables in the dataset, and list each variable as it is classified

vl set, list

Put nonnegative integer variables with 6 or fewer categories into vlcategorical; put nonnegative integer variables with 7–20 categories into vluncertain; put nonnegative integer variables with more than 20 categories into vlcontinuous

vl set, categorical(6) uncertain(20)

Classify only the variables x1-x100

vl set x1-x100

Discard the existing classifications, and classify all numeric variables again

vl set, clear

Redo the classification of the variable age

vl set age, redo

Update the stored statistics for the variable age, but do not change its classification

vl set age, update

Move the variables x8 and x20 out of their current classification and into vlcategorical

vl move (x8 x20) vlcategorical
Move all the variables in vluncertain into vlcontinuous

\[ \text{vl move vluncertain vlcontinuous} \]

### Syntax

**Create system-defined variable lists**

\[
\text{vl set} \ [\text{varlist}] [, \text{options}] 
\]

**Move variables from their current system-defined variable list to another**

\[
\text{vl move} (\text{varlist}) \text{ vlsysname} 
\]

**Move all variables in one system-defined variable list to another**

\[
\text{vl move vlsysname1 vlsysname2} 
\]

**varlist** contains only numeric variables. If not specified, then all numeric variables in the dataset are classified.

<table>
<thead>
<tr>
<th><strong>options</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>categorical(#)</strong></td>
<td>upper limit for the number of categories in vlcategorical</td>
</tr>
<tr>
<td><strong>uncertain(#)</strong></td>
<td>upper limit for the number of categories in vluncertain</td>
</tr>
<tr>
<td><strong>dummy</strong></td>
<td>create variable list vldummy containing 0/1 variables</td>
</tr>
<tr>
<td><strong>list([list_options])</strong></td>
<td>list variables as they are classified</td>
</tr>
<tr>
<td><strong>clear</strong></td>
<td>discard all existing classifications and make new classifications</td>
</tr>
<tr>
<td><strong>redo</strong></td>
<td>redo classifications for variables in varlist</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>update stored statistics for variables in varlist, but do not change their classification</td>
</tr>
<tr>
<td><strong>nonotes</strong></td>
<td>suppress the notes below the summary table</td>
</tr>
</tbody>
</table>

**collect** is allowed with vl set; see [U] 11.1.10 Prefix commands.

### Options

**categorical(#)** specifies that variables containing nonnegative integers be put into the vlcategorical variable list when the number of levels is between 2 and # inclusive. Variables with only one level (that is, constants) are put into the vlother variable list. The default is categorical(10).

**categorical(.)** can be specified to set the upper limit effectively to infinity. That is, all variables containing nonnegative integers (whose values are less than \(2^{31} = 2,147,483,648\)) are put into vlcategorical. Setting # to . or a large value can slow computation time considerably when the number of observations is extremely large.

**uncertain(#)** specifies that variables containing nonnegative integers be put into the vluncertain variable list when the number of levels are between categorical(#) + 1 and # inclusive. The default is uncertain(100).

# must be \(\geq\) categorical(#). To omit the vluncertain classification, set # = categorical(#) or specify uncertain(0).
uncertain(.) can be specified to set the upper limit effectively to infinity. That is, all variables containing nonnegative integers (whose values are less than $2^{31} = 2,147,483,648$) with more than categorical(#) levels are put into vluncertain. Setting # to . or a large value can slow computation time considerably when the number of observations is extremely large.

dummy specifies that a vldummy variable list be created containing 0/1 variables. By default, 0/1 variables are put into vlcategorical.

list[(list_options)] lists variables as they are classified. The classification is shown as well as the number of levels for variables in vlcategorical and vluncertain. list_options are as follows:

- **minimum** shows the minimum value of each variable;
- **maximum** shows the maximum value of each variable; and
- **observations** shows the number of nonmissing values of each variable.

The same listing can be obtained using vl list after running vl set.

clear specifies that all the system-defined variable lists (if any) be dropped and the classifications redone. It is equivalent to running vl clear, system and then running vl set.

redo specifies that the classifications be redone for the variables in varlist. It is equivalent to running vl drop (varlist), system and then running vl set varlist.

update specifies that all statistics (number of levels, minimum value, maximum value, and number of nonmissing observations) that are saved for the variables in varlist be updated but the classifications of the variables not be changed. update is intended for use when observations are added to or dropped from the data and you want the classifications to remain unchanged.

nonotes specifies that the notes at the bottom of the summary table not be displayed. By default, the notes are shown.

Remarks and examples

- vl set creates the system-defined variable lists vlcategorical, vlcontinuous, vluncertain, and vlother.

The vlcategorical variable list is intended for variables that will be used as factor variables in estimation commands.

The vlcontinuous variable list is intended for variables that will be used as continuous variables in estimation commands.

The vluncertain variable list is intended for variables that we may want to treat as factors or as continuous, and we will decide which on a case-by-case basis. As we decide, we use vl move to move them out of vluncertain and into vlcategorical or vlcontinuous. For example, we decide we want variable q31, currently in vluncertain, to be a factor variable. We type

```
   . vl move (q31) vl_categorical
```

In the above, note that q31 is enclosed in parentheses. varlists must always be enclosed in parentheses in vl move.

When q31 is moved into vl_categorical, it is automatically moved out of vluncertain. The system-defined variable lists are always kept as disjoint sets. That is, a variable can only appear in one system-defined variable list. User-defined variable lists can be made to be overlapping. See [D] vl create and [D] vl.
Suppose we look at the remaining variables in `vluncertain`, and we decide that they all should be treated as continuous. We type

`. vl move vluncertain vlcategorical`

Suppose we look at the remaining variables in `vluncertain`, and we decide we do not want any of them in any of the estimation commands we wish to run. We could move them to `vlother`.

`. vl move vluncertain vlother`

`vlother` is intended to be a garbage classification for variables you do not want to use in estimation commands. `vl set` puts variables that are constant and variables that are missing for all observations into `vlother`.

Suppose, however, we simply want some variables gone from the system-defined variable lists. We do not want them shown when we do a `vl list`. To make them gone, gone, gone, use `vl drop`.

`. vl drop (varlist), system`

This removes the variables in `varlist` from the system-defined variable lists.

We can also

`. vl drop vluncertain`

This removes all the variables in `vluncertain`. `vluncertain` still exists, but it is empty. We can still move other variables into it if we want. System-defined variable lists always exist although they may be empty. They cannot be renamed. If you do not like this behavior, you can create your own variable lists using `vl create`. For example,

`. vl create mycat = vlcategorical
  . vl create mycont = vlcontinuous`

If you are done using the system-defined variable lists and do not want them around, you can remove them by typing

`. vl clear, system`

The system-defined variable lists will be gone, but user-defined variable lists will remain. When you clear the system-defined variable lists, you also erase the statistics that are stored with each variable in the system.

When `vl set` runs, it calculates the minimum, maximum, and number of nonmissing observations for each variable. It also computes the number of levels for the variables in `vlcategorical` and `vluncertain`. It does not compute the number of levels for other variables. That is why `vl set` is so fast even when there are millions of observations.

Computing the exact number of levels when there are thousands of levels can be time consuming. You can have `vl set` compute the number of levels for more variables by specifying the option `uncertain(#)` and setting # to a large number or missing (.). But expect it to be much slower when there are lots of observations.

To use variable lists with other Stata commands, type `$` in front of the variable-list name. Remember: With the `vl` commands, do not use `. With other Stata commands, use `$.`

`. display "$vlcategorical"
  . summarize $vlcontinuous
  . regress y i.($vlcategorical) $vlcontinuous`

If you know Stata, you will have already sensed that variable lists are global macros.
In this example, we used \texttt{i.($vl\text{categorical}$)} to turn the variables in \texttt{vl\text{categorical}} into factor variables. More likely, however, you will want to create your own variable lists based on the system-defined variable lists, and then apply factor-variable operators. The \texttt{vl create}, \texttt{vl modify}, and \texttt{vl substitute} commands were designed for this purpose. See \texttt{[D] vl create}.

Variable lists are saved with the dataset. Not only are variable lists saved but also all the \texttt{vl} system information and variable statistics are saved. To make the \texttt{vl} system come back to life in the state we last had it, after we \texttt{use} a dataset, we type

\begin{verbatim}
. vl rebuild
\end{verbatim}

See \texttt{[D] vl rebuild}.

For examples of using \texttt{vl set} and its options, see \texttt{[D] vl}.

**Stored results**

\texttt{vl set} stores the following in \texttt{r()}:  

**Scalars**  
\begin{itemize}
\item \texttt{r(k\_system)}: number of variables in system-defined variable lists
\item \texttt{r(k\_vl\text{categorical})}: number of variables in \texttt{vl\text{categorical}}
\item \texttt{r(k\_vl\text{continuous})}: number of variables in \texttt{vl\text{continuous}}
\item \texttt{r(k\_vl\text{uncertain})}: number of variables in \texttt{vl\text{uncertain}}
\item \texttt{r(k\_vl\text{other})}: number of variables in \texttt{vl\text{other}}
\item \texttt{r(k\_vl\text{dummy})}: number of variables in \texttt{vl\text{dummy}} when defined
\end{itemize}

**Macros**  
\begin{itemize}
\item \texttt{r(vlsysnames)}: names of system-defined variable lists
\end{itemize}

**Also see**  

\begin{itemize}
\item \texttt{[D] vl} — Manage variable lists
\item \texttt{[D] vl create} — Create and modify user-defined variable lists
\item \texttt{[D] vl drop} — Drop variable lists or variables from variable lists
\item \texttt{[D] vl list} — List contents of variable lists
\item \texttt{[D] vl rebuild} — Rebuild variable lists
\end{itemize}
**webuse — Use dataset from Stata website**

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Menu</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>webuse</strong> filename loads the specified dataset, obtaining it over the web. By default, datasets are obtained from <a href="https://www.stata-press.com/data/r17/">https://www.stata-press.com/data/r17/</a>. If filename is specified without a suffix, .dta is assumed.</td>
<td><strong>webuse nlswork.dta</strong> dataset from default Stata Press website</td>
<td><strong>Menu</strong></td>
<td><strong>Syntax</strong></td>
</tr>
<tr>
<td><strong>webuse query</strong> reports the URL from which datasets will be obtained.</td>
<td>As above, but clear current dataset from memory first</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>webuse set</strong> allows you to specify the URL to be used as the source for datasets. <strong>webuse set</strong> without arguments resets the source to <a href="https://www.stata-press.com/data/r17/">https://www.stata-press.com/data/r17/</a>.</td>
<td>Change URL for data downloads to <a href="http://www.myuniversity.edu/mycourse">http://www.myuniversity.edu/mycourse</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Quick start**

Load example nlswork.dta dataset from default Stata Press website
webuse nlswork

As above, but clear current dataset from memory first
webuse nlswork, clear

Change URL for data downloads to http://www.myuniversity.edu/mycourse
webuse set www.myuniversity.edu/mycourse

Reset source for datasets to Stata Press
webuse set

Report current URL from which datasets will be obtained
webuse query

**Menu**

File > Example datasets...
webuse — Use dataset from Stata website

Syntax

Load dataset over the web

```
webuse ["filename"] [ , clear ]
```

Report URL from which datasets will be obtained

```
webuse query
```

Specify URL from which dataset will be obtained

```
webuse set [https://]url[/]
```

Reset URL to default

```
webuse set
```

Option

```
clear
```

specifies that it is okay to replace the data in memory, even though the current data have not been saved to disk.

Remarks and examples

Remarks are presented under the following headings:

```
Typical use
A note concerning example datasets
Redirecting the source
```

Typical use

In the examples in the Stata manuals, we see things such as

```
.use https://www.stata-press.com/data/r17/lifeexp
```

The above is used to load—in this instance—the dataset lifeexp.dta. You can type that, and it will work:

```
.use https://www.stata-press.com/data/r17/lifeexp
(Life expectancy, 1998)
```

Or you may simply type

```
.webuse lifeexp
(Life expectancy, 1998)
```

webuse is a synonym for use https://www.stata-press.com/data/r17/.
A note concerning example datasets

The datasets used to demonstrate Stata are often fictional. If you want to know whether a dataset is real or fictional, and its history, load the dataset and type

```
.notes
```

A few datasets have no notes. This means that the datasets are believed to be real but that they were created so long ago that information about their original source has been lost. Treat such datasets as if they were fictional.

Redirecting the source

By default, `webuse` obtains datasets from https://www.stata-press.com/data/r17/, but you can change that. Say that the site http://www.zzz.edu/users/~sue/ has several datasets that you wish to explore. You can type

```
.webuse set http://www.zzz.edu/users/~sue
```

`webuse` will become a synonym for `use http://www.zzz.edu/users/~sue/` for the rest of the session or until you give another `webuse` command.

When you set the URL, you may omit the trailing slash (as we did above), or you may include it:

```
.webuse set http://www.zzz.edu/users/~sue/
```

You may also omit `https://` or `http://`:

```
.webuse set www.zzz.edu/users/~sue
```

If you type `webuse set` without arguments, the URL will be reset to the default, https://www.stata-press.com/data/r17/:

```
.webuse set
```

Also see

[D] `sysuse` — Use shipped dataset

[D] `use` — Load Stata dataset

[U] 1.2.2 Example datasets
Description

`xpose` transposes the data, changing variables into observations and observations into variables. All new variables—that is, those created by the transposition—are made the default storage type. Thus any original variables that were strings will result in observations containing missing values. (If you transpose the data twice, you will lose the contents of string variables.)

Quick start

Replace dataset in memory with transposed variables and observations

```bash
xpose, clear
```

Add `_varname` containing the original variable names

```bash
xpose, clear varname
```

Use the most compact data type that preserves accuracy in the transposed data

```bash
xpose, clear promote
```

Menu

Data > Create or change data > Other variable-transformation commands > Interchange observations and variables
Syntax

\texttt{xpose, clear [options]}

<table>
<thead>
<tr>
<th>options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*clear</td>
<td>reminder that untransposed data will be lost if not previously saved</td>
</tr>
<tr>
<td>format</td>
<td>use largest numeric display format from untransposed data</td>
</tr>
<tr>
<td>format(%fmt)</td>
<td>apply specified format to all variables in transposed data</td>
</tr>
<tr>
<td>varname</td>
<td>add variable _varname containing original variable names</td>
</tr>
<tr>
<td>promote</td>
<td>use the most compact data type that preserves numeric accuracy</td>
</tr>
</tbody>
</table>

\* clear is required.

Options

clear is required and is supposed to remind you that the untransposed data will be lost (unless you have saved the data previously).

format specifies that the largest numeric display format from your untransposed data be applied to the transposed data.

format(\%fmt) specifies that the specified numeric display format be applied to all variables in the transposed data.

varname adds the new variable \_varname to the transposed data containing the original variable names. Also, with or without the varname option, if the variable \_varname exists in the dataset before transposition, those names will be used to name the variables after transposition. Thus transposing the data twice will (almost) yield the original dataset.

promote specifies that the transposed data use the most compact numeric data type that preserves the original data accuracy.

If your data contain any variables of type double, all variables in the transposed data will be of type double.

If variables of type float are present, but there are no variables of type double or long, the transposed variables will be of type float. If variables of type long are present, but there are no variables of type double or float, the transposed variables will be of type long.

Remarks and examples

\textgt; Example 1

We have a dataset on something by county and year that contains

\texttt{. use https://www.stata-press.com/data/r17/xposexmpl}
\texttt{. list}

<table>
<thead>
<tr>
<th>county</th>
<th>year1</th>
<th>year2</th>
<th>year3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>57.2</td>
<td>11.3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>12.5</td>
<td>8.2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>18</td>
<td>14.2</td>
</tr>
</tbody>
</table>
Each observation reflects a county. To change this dataset so that each observation reflects a year, type

```
. xpose, clear varname
. list
```

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>_varname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>county</td>
</tr>
<tr>
<td>57.2</td>
<td>12.5</td>
<td>18</td>
<td>year1</td>
</tr>
<tr>
<td>11.3</td>
<td>8.2</td>
<td>14.2</td>
<td>year2</td>
</tr>
<tr>
<td>19.5</td>
<td>28.9</td>
<td>33.2</td>
<td>year3</td>
</tr>
</tbody>
</table>

We would now have to drop the first observation (corresponding to the previous county variable) to make each observation correspond to one year. Had we not specified the `varname` option, the variable `_varname` would not have been created. The `_varname` variable is useful, however, if we want to transpose the dataset back to its original form.

```
. xpose, clear
. list
```

<table>
<thead>
<tr>
<th>county</th>
<th>year1</th>
<th>year2</th>
<th>year3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>57.2</td>
<td>11.3</td>
<td>19.5</td>
</tr>
<tr>
<td>2</td>
<td>12.5</td>
<td>8.2</td>
<td>28.9</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>14.2</td>
<td>33.2</td>
</tr>
</tbody>
</table>

Reference

Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. College Station, TX: Stata Press.

Also see

[D] reshape — Convert data from wide to long form and vice versa
[D] stack — Stack data
Title

zipfile — Compress and uncompress files and directories in zip archive format

<table>
<thead>
<tr>
<th>Description</th>
<th>Quick start</th>
<th>Syntax</th>
<th>Option for zipfile</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Option for unzipfile</td>
<td>Remarks and examples</td>
<td></td>
</tr>
</tbody>
</table>

Description

zipfile compresses files and directories into a zip file that is compatible with Zip64, WinZip, PKZIP 2.04g, and other applications that use the zip archive format.

unzipfile extracts files and directories from a file in zip archive format into the current directory. unzipfile can open zip files created by Zip64, WinZip, PKZIP 2.04g, and other applications that use the zip archive format.

Quick start

Compress mydata.dta and save as myproject.zip
zipfile mydata.dta, saving(myproject)

As above, but also compress mydofile.do and mylog.smcl
zipfile mydata.dta mydofile.do mylog.smcl, saving(myproject)

Replace myproject.zip if it already exists
zipfile mydata.dta mydofile.do mylog.smcl, ///
saving(myproject, replace)

Compress all files in the myproject subdirectory of the current directory
zipfile myproject/*, saving(myproject)

Extract files and directories from myzip.zip to the current directory
unzipfile myzip

As above, but replace any file or directory in the current directory that has the same name as a file or directory in the zip file
unzipfile myzip, replace
Syntax

Add files or directories to a zip file

\[
\text{zipfile } \text{file} | \text{directory} \ [\text{file} | \text{directory}] \ldots, \text{saving} (\text{zipfilename}[ , \text{replace}])
\]

Extract files or directories from a zip file

\[
\text{unzipfile } \text{zipfilename} [ , \text{replace}]
\]

Note: Double quotes must be used to enclose file and directory if the name or path contains blanks. file and directory may also contain the \? and \* wildcard characters.

Option for zipfile

saving(\text{zipfilename}[ , \text{replace}]) is required. It specifies the filename to be created or replaced. If zipfilename is specified without an extension, .zip will be assumed.

Option for unzipfile

replace overwrites any file or directory in the current directory with the files or directories in the zip file that have the same name.

Remarks and examples

\textbf{Example 1: Creating a zip file}

Suppose that we would like to zip all the .dta files in the current directory into the file myfiles.zip. We would type

\[
\text{. zipfile } *.*.dta, \text{ saving}(\text{myfiles})
\]

But we notice that we did not want the files in the current directory; instead, we wanted the files in the dta, abc, and eps subdirectories. We can easily zip all the .dta files from all three-character subdirectories of the current directory and overwrite the file myfiles.zip if it exists by typing

\[
\text{. zipfile } ???/**/*.dta, \text{ saving}(\text{myfiles, replace})
\]

\textbf{Example 2: Unzipping a zip file}

Say, for example, we send myfiles.zip to a colleague, who now wants to unzip the file in the current directory, overwriting any files or directories that have the same name as the files or directories in the zip file. The colleague should type

\[
\text{. unzipfile myfiles, replace}
\]
Glossary

ASCII. ASCII stands for American Standard Code for Information Interchange. It is a way of representing text and the characters that form text in computers. It can be divided into two sections: plain, or lower ASCII, which includes numbers, punctuation, plain letters without diacritical marks, whitespace characters such as space and tab, and some control characters such as carriage return; and extended ASCII, which includes letters with diacritical marks as well as other special characters.

Before Stata 14, datasets, do-files, ado-files, and other Stata files were encoded using ASCII.

binary 0. Binary 0, also known as the null character, is traditionally used to indicate the end of a string, such as an ASCII or UTF-8 string.

Binary 0 is obtained by using `char(0)` and is sometimes displayed as `\0`. See [U] 12.4.10 strL variables and binary strings for more information.

binary string. A binary string is, technically speaking, any string that does not contain text. In Stata, however, a string is only marked as binary if it contains binary 0, or if it contains the contents of a file read in using the `fileread()` function, or if it is the result of a string expression containing a string that has already been marked as binary.

In Stata, strL variables, string scalars, and Mata strings can store binary strings.

See [U] 12.4.10 strL variables and binary strings for more information.

byte. Formally, a byte is eight binary digits (bits), the units used to record computer data. Each byte can also be considered as representing a value from 0 through 255. Do not confuse this with Stata’s byte variable storage type, which allows values from −127 to 100 to be stored. With regard to strings, all strings are composed of individual characters that are encoded using either one byte or several bytes to represent each character.

For example, in UTF-8, the encoding system used by Stata, byte value 97 encodes “a”. Byte values 195 and 161 in sequence encode “á”.

characteristics. Characteristics are one form of metadata about a Stata dataset and each of the variables within the dataset. They are typically used in programming situations. For example, the xt commands need to know the name of the panel variable and possibly the time variable. These variable names are stored in characteristics within the dataset. See [U] 12.8 Characteristics for an overview and [P] char for a technical description.

code pages. A code page maps extended ASCII values to a set of characters, typically for a specific language or set of languages. For example, the most commonly used code page is Windows-1252, which maps extended ASCII values to characters used in Western European languages. Code pages are essentially encodings for extended ASCII characters.

code point. A code point is the numerical value or position that represents a single character in a text system such as ASCII or Unicode. The original ASCII encoding system contains only 128 code points and thus can represent only 128 characters. Historically, the 128 additional bytes of extended ASCII have been encoded in many different and inconsistent ways to provide additional sets of 128 code points. The formal Unicode specification has 1,114,112 possible code points, of which roughly 250,000 have been assigned to actual characters. Stata uses UTF-8 encoding for Unicode. Note that the UTF-8–encoded version of a code point does not have the same numeric value as the code point itself.

display column. A display column is the space required to display one typical character in the fixed-width display used by Stata’s Results window and Viewer. Some characters are too wide for one display column. Each character is displayed in one or two display columns.
All plain ASCII characters (for example, “M” and “9”) and many UTF-8 characters that are not plain ASCII (for example, “é”) require the same space when using a fixed-width font. That is to say, they all require a single display column.

Characters from non-Latin alphabets, such as Chinese, Cyrillic, Japanese, and Korean, may require two display columns.

See [U] 12.4.2.2 Displaying Unicode characters for more information.

display format. The display format for a variable specifies how the variable will be displayed by Stata. For numeric variables, the display format would indicate to Stata how many digits to display, how many decimal places to display, whether to include commas, and whether to display in exponential format. Numeric variables can also be formatted as dates. For strings, the display format indicates whether the variable should be left-aligned or right-aligned in displays and how many characters to display. Display formats may be specified by the format command. Display formats may also be used with individual numeric or string values to control how they are displayed. Distinguish display formats from storage types.

encodings. An encoding is a way of representing a character as a byte or series of bytes. Examples of encoding systems are ASCII and UTF-8. Stata uses UTF-8 encoding.

For more information, see [U] 12.4.2.3 Encodings.

extended ASCII. Extended ASCII, also known as higher ASCII, is the byte values 128 to 255, which were not defined as part of the original ASCII specification. Various code pages have been defined over the years to map the extended ASCII byte values to many characters not supported in the original ASCII specification, such as Latin letters with diacritical marks, such as “â” and “Á”; non-Latin alphabets, such as Chinese, Cyrillic, Japanese, and Korean; punctuation marks used in non-English languages, such as “<”, complex mathematical symbols such as “±”, and more.

Although extended ASCII characters are stored in a single byte in ASCII encoding, UTF-8 stores the same characters in two to four bytes. Because each code page maps the extended ASCII values differently, another distinguishing feature of extended ASCII characters is that their meaning can change across fonts and operating systems.

frames. Frames, also known as data frames, are in-memory areas where datasets are analyzed. Stata can hold multiple datasets in memory, and each dataset is held in a memory area called a frame. A variety of commands exist to manage frames and manipulate the data in them. See [D] frames.

hexadecimal. The hexadecimal number system, or simply hex, is a base-16 number system represented by digits 0 through 9 and letters A through F.

higher ASCII. See extended ASCII.

locale. A locale is a code that identifies a community with a certain set of rules for how their language should be written. A locale can refer to something as general as an entire language (for example, “en” for English) or something as specific as a language in a particular country (for example, “en_HK” for English in Hong Kong).

A locale specifies a set of rules that govern how the language should be written. Stata uses locales to determine how certain language-specific operations are carried out. For more information, see [U] 12.4.2.4 Locales in Unicode.

long format and wide format. Think of a dataset as having an ID variable, $i$, and a variable, $j$, whose values denote a subobservation. For instance, a person might be the $i$ variable, and a year might be the $j$ variable, so you have information about a set of people across several years. If this information is organized such that the $j$ variable is explicitly specified, then the data are in long format; otherwise, they are in wide format. For instance,
are in long format because the $j$ variable, `year`, is explicitly specified. In the following, the data are in wide format:

```
  id  income1980  income1981  income1982
  1    10000        12000        11000
  2    15000        14000        17000
```

See [D] `reshape` for how to go between long and wide format.

**lower ASCII.** See *plain ASCII*.

**null-terminator.** See *binary 0*.

**numlist.** A numlist is a list of numbers. That list can be one or more arbitrary numbers or can use certain shorthands to indicate ranges, such as `5/9` to indicate integers 5, 6, 7, 8, and 9. Ranges can be ascending or descending and can include an optional increment or decrement amount, such as `10.5(-2)4.5` to indicate 10.5, 8.5, 6.5, and 4.5. See [U] 11.1.8 `numlist` for a list of shorthands to indicate ranges.

**plain ASCII.** We use plain ASCII as a nontechnical term to refer to what computer programmers call lower ASCII. These are the plain Latin letters “a” to “z” and “A” to “Z”; numbers “0” through “9”; many punctuation marks, such as “!”; simple mathematical symbols, such as “+”; and whitespace and control characters such as space (“ ”), tab, and carriage return.

Each plain ASCII character is stored as a single byte with a value between 0 and 127. Another distinguishing feature is that the byte values used to encode plain ASCII characters are the same across different operating systems and are common between ASCII and UTF-8.

Also see *ASCII* and *encodings*.

**prefix command.** A prefix command is a command in Stata that prefixes other Stata commands. For example, by `varlist:`. The command by `region:` `summarize marriage_rate divorce_rate` would summarize `marriage_rate` and `divorce_rate` for each region separately. See [U] 11.1.10 `Prefix commands`.

**storage types.** A storage type is how Stata stores a variable. The numeric storage types in Stata are `byte`, `int`, `long`, `float`, and `double`. There is also a string storage type. The storage type is specified before the variable name when a variable is created. See [U] 12.2.2 `Numeric storage types`, [U] 12.4 `Strings`, and [D] `Data types`. Distinguish storage types from display formats.

**str1, str2, ..., str2045.** See `strL`.

**strL.** `strL` is a storage type for string variables. The full list of string storage types is `str1, str2, ..., str2045, and strL`.

`str1, str2, ..., str2045` are fixed-length storage types. If variable `mystr` is `str8`, then 8 bytes are allocated in each observation to store `mystr`’s value. If you have 2,000 observations, then 16,000 bytes in total are allocated.

Distinguish between storage length and string length. If `myvar` is `str8`, that does not mean the strings are 8 characters long in every observation. The maximum length of strings is 8 characters. Individual observations may have strings of length 0, 1, ..., 8. Even so, every string requires 8 bytes of storage.
You need not concern yourself with the storage length because string variables are automatically promoted. If `myvar` is `str8`, and you changed the contents of `myvar` in the third observation to “Longer than 8”, then `myvar` would automatically become `str13`.

If you changed the contents of `myvar` in the third observation to a string longer than 2,045 characters, `myvar` would become `strL`.

`strL` variables are not necessarily longer than 2,045 characters; they can be longer or shorter than 2,045 characters. The real difference is that `strL` variables are stored as varying length. Pretend that `myothervar` is a `strL` and its third observation contains “this”. The total memory consumed by the observation would be $64 + 4 + 1 = 69$ bytes. There would be 64 bytes of tracking information, 4 bytes for the contents (there are 4 characters), and 1 more byte to terminate the string. If the fifth observation contained a 2,000,000-character string, then $64 + 2,000,000 + 1 = 2,000,069$ bytes would be used to store it.

Another difference between `str1`, `str2`, ..., `str2045`, and `strLs` is that the `str#` storage types can store only ASCII strings. `strL` can store ASCII or binary strings. Thus a `strL` variable could contain, for instance, the contents of a Word document or a JPEG image or anything else.

`strL` is pronounce `sturl`.

titlecase, title-cased string, and Unicode title-cased string. In grammar, titlecase refers to the capitalization of the key words in a phrase. In Stata, titlecase refers to (a) the capitalization of the first letter of each word in a string and (b) the capitalization of each letter after a nonletter character. There is no judgment of the word’s importance in the string or whether the letter after a nonletter character is part of the same word. For example, “it’s” in titlecase is “It’S”.

A title-cased string is any string to which the above rules have been applied. For example, if we used the `strproper()` function with the book title `Zen and the Art of Motorcycle Maintenance`, Stata would return the title-cased string `Zen And The Art Of Motorcycle Maintenance`.

A Unicode title-cased string is a string that has had Unicode title-casing rules applied to Unicode words. This is almost, but not exactly, like capitalizing the first letter of each Unicode word. Like capitalization, title-casing letters is locale-dependent, which means that the same letter might have different titlecase forms in different locales. For example, in some locales, capital letters at the beginning of words are not supposed to have accents on them, even if that capital letter by itself would have an accent.

If you do not have characters beyond plain ASCII and your locale is English, there is no distinction in results. For example, `ustrtitle()` with an English `locale` locale also would return the title-cased string `Zen And The Art Of Motorcycle Maintenance`.

Use the `ustrtitle()` function to apply the appropriate capitalization rules for your language (locale).

Unicode. Unicode is a standard for encoding and dealing with text written in almost any conceivable living or dead language. Unicode specifies a set of encoding systems that are designed to hold (and, unlike extended ASCII, to keep separate) characters used in different languages. The Unicode standard defines not only the characters and encodings for them, but also rules on how to perform various operations on words in a given language (locale), such as capitalization and ordering. The most common Unicode encodings are `mUTF-8`, `UTF-16`, and `UTF-32`. Stata uses `UTF-8`.

Unicode character. Technically, a Unicode character is any character with a Unicode encoding. Colloquially, we use the term to refer to any character other than the plain ASCII characters.
Unicode normalization. Unicode normalization allows us to use a common representation and therefore compare Unicode strings that appear the same when displayed but could have more than one way of being encoded. This rarely arises in practice, but because it is possible in theory, Stata provides the `ustrnormalize()` function for converting between different normalized forms of the same string.

For example, suppose we wish to search for “˜n” (the lowercase n with a tilde over it from the Spanish alphabet). This letter may have been encoded with the single code point U+00F1. However, the sequence U+006E (the Latin lowercase “n”) followed by U+0303 (the tilde) is defined by Unicode to be equivalent to U+00F1. This type of visual identicalness is called canonical equivalence. The one-code-point form is known as the canonical composited form, and the multiple-code-point form is known as the canonical decomposed form. Normalization modifies one or the other string to the opposite canonical equivalent form so that the underlying byte sequences match. If we had strings in a mixture of forms, we would want to use this normalization when sorting or when searching for strings or substrings.

Another form of Unicode normalization allows characters that appear somewhat different to be given the same meaning or interpretation. For example, when sorting or indexing, we may want the code point U+FB00 (the typographic ligature “ff”) to match the sequence of two Latin “f” letters encoded as U+0066 U+0066. This is called compatible equivalence.

Unicode title-cased string. See `titlecase`, `title-cased string`, and `Unicode title-cased string`.

UTF-8. UTF-8 stands for Universal character set + Transformation Format—8-bit. It is a type of Unicode encoding system that was designed for backward compatibility with ASCII and is used by Stata 14.

value label. A value label defines a mapping between numeric data and the words used to describe what those numeric values represent. So, the variable disease might have a value label status associated with it that maps 1 to positive and 0 to negative. See [U] 12.6.3 Value labels.

varlist. A varlist is a list of variables that observe certain conventions: variable names may be abbreviated; the asterisk notation can be used as a shortcut to refer to groups of variables, such as `income*` or `*1995` to refer to all variable names beginning with `income` or all variable names ending in 1995, respectively; and a dash may be used to indicate all variables stored between the two listed variables, for example, `mpg-weight`. See [U] 11.4 varname and varlists.

wide format. See `long` and `wide format`. 
Subject and author index

See the combined subject index and the combined author index in the Stata Index.