12 Data

Contents

12.1 Data and datasets

Data form a rectangular table of numeric and string values in which each row is an observation on all the variables and each column contains the observations on one variable. Variables are designated by *variable names*. Observations are numbered sequentially from `1` to `_N`. The following example of data contains the first five odd and first five even positive integers, along with a string variable:

```
     odd  even   name
1.     1     2   Bill
2.     3     4   Mary
3.     5     6    Pat
4.     7     8  Roger
5.     9    10   Sean
```

The observations are numbered `1` to `5`, and the variables are named `odd`, `even`, and `name`. Observations are referred to by number, and variables by name.

A *dataset* is *data* plus labelings, formats, notes, and characteristics.

All aspects of *data* and *datasets* are defined here. Long (2009) offers a long-time Stata user's hard-won advice on how to manage data in Stata to promote accurate, replicable research. Mitchell (2010) provides many examples on data management in Stata.

12.2 Numbers

A *number* may contain a sign, an integer part, a decimal point, a fraction part, an `e` or `E`, and a signed integer exponent. Numbers may *not* contain commas; for example, the number 1,024 must be typed as `1024` (or `1024.` or `1024.0`). The following are examples of valid numbers:

```
5
-5
5.2
.5
5.2e+2
5.2e-2
```

❑ Technical note

Stata also allows numbers to be represented in a hexadecimal/binary format, defined as

[+|-]`0.0`[⟨*zeros*⟩]{`X`|`x`}`-3ff`

or

[+|-]`1.`⟨*hexdigit*⟩[⟨*hexdigits*⟩]{`X`|`x`}{`+`|`-`}⟨*hexdigit*⟩[⟨*hexdigits*⟩]

The lead digit is always 0 or 1; it is 0 only when the number being expressed is zero. A maximum of 13 digits to the right of the hexadecimal point are allowed. The power ranges from `-3ff` to `+3ff`. The number is expressed in hexadecimal (base 16) digits; the number a`X+`b means $a \times 2^b$. For instance, `1.0X+3` is 2^3 or 8. `1.8X+3` is 12 because 1.8_{16} is $1 + 8/16 = 1.5$ in decimal and the number is thus $1.5 \times 2^3 = 1.5 \times 8 = 12$.

Stata can also display numbers using this format; see [U] **12.5.1 Numeric formats**. For example,

```
. display 1.81x+2
6.015625
. display %21x 6.015625
+1.8100000000000X+002
```

This hexadecimal format is of special interest to numerical analysts.

❑

12.2.1 Missing values

A number may also take on the special value *missing*, denoted by a period (`.`). You specify a missing value anywhere that you may specify a number. Missing values differ from ordinary numbers in one respect: any arithmetic operation on a missing value yields a missing value.

In fact, there are 27 missing values in Stata: '`.`', the one just discussed, as well as `.a`, `.b`, ..., and `.z`, which are known as extended missing values. The missing value '`.`' is known as the default or system missing value. Some people use extended missing values to indicate why a certain value is unknown—the question was not asked, the person refused to answer, etc. Other people have no use for extended missing values and just use '`.`'.

Stata's default or system missing value will be returned when you perform an arithmetic operation on missing values or when the arithmetic operation is not defined, such as division by zero, or the logarithm of a nonpositive number.

```
. display 2/0
.
. list
```

	a
1.	.b
2.	.
3.	.a
4.	3
5.	6

```
. generate x = a + 1
(3 missing values generated)
. list
```

	a	x
1.	.b	.
2.	.	.
3.	.a	.
4.	3	4
5.	6	7

Numeric missing values are represented by "large positive values". The ordering is

$$\text{all numbers} < \texttt{.} < \texttt{.a} < \texttt{.b} < \cdots < \texttt{.z}$$

Thus the expression

$$\texttt{age} > 60$$

is true if variable `age` is greater than 60 or is missing. Similarly,

$$\texttt{gender} \neq 0$$

is true if `gender` is not zero or is missing.

To exclude missing values, you must ask whether the value is less than '.'; to detect missing values, you must ask whether the value is greater than or equal to '.'. For instance,

```
. list if age>60 & age<.
. generate agegt60 = 0 if age<=60
. replace agegt60 = 1 if age>60 & age<.
. generate agegt60 = (age>60) if age<.
```

❑ Technical note

Before Stata 8, Stata had only one representation for missing values, the period (.).

To ensure that old programs and do-files continue to work properly, when `version` is set less than 8, all missing values are treated as being the same. Thus `. == .a == .b == .z`, and so '*exp*`==.`' and '*exp*`!=.`' work just as they previously did.

❑

▷ Example 1

We have data on the income of husbands and wives recorded in the variables `hincome` and `wincome`, respectively. Typing the `list` command, we see that your data contain

```
. use http://www.stata-press.com/data/r15/gxmpl3
. list
```

	hincome	wincome
1.	32000	0
2.	35000	34000
3.	47000	.b
4.	.z	50000
5.	.a	.

The values of `wincome` in the third and fifth observations are *missing*, as distinct from the value of `wincome` in the first observation, which is known to be zero.

If we use the `generate` command to create a new variable, `income`, that is equal to the sum of `hincome` and `wincome`, three missing values would be produced.

```
. generate income = hincome + wincome
(3 missing values generated)
. list
```

	hincome	wincome	income
1.	32000	0	32000
2.	35000	34000	69000
3.	47000	.b	.
4.	.z	50000	.
5.	.a	.	.

`generate` produced a warning message that 3 missing values were created, and when we list the data, we see that 47,000 plus *missing* yields *missing*.

◁

❑ Technical note

Stata stores numeric missing values as the largest 27 numbers allowed by the particular storage type; see [U] **12.2.2 Numeric storage types**. There are two important implications. First, if you `sort` on a variable that has missing values, the missing values will be placed last, and the sort order of any missing values will follow the rule regarding the properties of missing values stated above.

```
. sort wincome
. list wincome
```

	wincome
1.	0
2.	34000
3.	50000
4.	.
5.	.b

The second implication concerns relational operators and missing values. Do not forget that a missing value will be larger than any numeric value.

```
. list if wincome > 40000
```

	hincome	wincome	income
3.	.z	50000	.
4.	.a	.	.
5.	47000	.b	.

Observations 4 and 5 are listed because '`.`' and '`.b`' are both missing and thus are greater than 40,000. Relational operators are discussed in detail in [U] **13.2.3 Relational operators**.

❑

▷ Example 2

In producing statistical output, Stata ignores observations with missing values. Continuing with the example above, if we request summary statistics on `hincome` and `wincome` by using the `summarize` command, we obtain

```
. summarize hincome wincome
```

Variable	Obs	Mean	Std. Dev.	Min	Max
hincome	3	38000	7937.254	32000	47000
wincome	3	28000	25534.29	0	50000

Some commands discard the entire observation (known as *casewise deletion*) if one of the variables in the observation is missing. If we use the `correlate` command to obtain the correlation between `hincome` and `wincome`, for instance, we obtain

```
. correlate hincome wincome
(obs=2)
```

	hincome	wincome
hincome	1.0000	
wincome	1.0000	1.0000

The correlation coefficient is calculated over two observations.

◁

12.2.2 Numeric storage types

Numbers can be stored in one of five variable types: `byte`, `int`, `long`, `float` (the default), or `double`. `byte`s are, naturally, stored in 1 byte. `int`s are stored in 2 bytes, `long`s and `float`s in 4 bytes, and `double`s in 8 bytes. The table below shows the minimum and maximum values for each storage type.

Storage type	Minimum	Maximum	Closest to 0 without being 0	Bytes
`byte`	−127	100	±1	1
`int`	−32,767	32,740	±1	2
`long`	−2,147,483,647	2,147,483,620	±1	4
`float`	$-1.70141173319 \times 10^{38}$	$1.70141173319 \times 10^{38}$	$\pm 10^{-38}$	4
`double`	$-8.9884656743 \times 10^{307}$	$+8.9884656743 \times 10^{307}$	$\pm 10^{-323}$	8

Do not confuse the term *integer*, which is a characteristic of a number, with `int`, which is a storage type. For instance, the number 5 is an integer, no matter how it is stored; thus, if you read that an argument must be an integer, that does not mean that it must be stored as an `int`.

12.3 Dates and times

Stata has nine date, time, and date-and-time numeric encodings known collectively as `%t` variables or values. They are

`%tC`	calendar date and time, adjusted for leap seconds
`%tc`	calendar date and time, ignoring leap seconds
`%td`	calendar date
`%tw`	week
`%tm`	calendar month
`%tq`	financial quarter
`%th`	financial half-year
`%ty`	calendar year
`%tb`	business calendars

All except `%ty` and `%tb` are based on 0 = beginning of January 1960. `%tc` and `%tC` record the number of milliseconds since then. `%td` records the number of days. The others record the numbers of weeks, months, quarters, or half-years. `%ty` simply records the year, and `%tb` records a user-defined business calendar format.

For a full discussion of working with dates and times, see **[U] 24 Working with dates and times**.

12.4 Strings

This section describes the treatment of strings by Stata. The section is divided into the following subsections:

12.4.1 Overview

A string is a sequence of characters.

```
Samuel Smith
California
U.K.
```

Usually—but not always—strings are enclosed in double quotes.

```
"Samuel Smith"
"California"
"U.K."
```

Strings typed in quotes are called *string literals*.

Strings can be stored in Stata datasets in string variables.

```
. use http://www.stata-press.com/data/r15/auto, clear
(1978 Automobile Data)
. describe make
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
make            str18   %-18s                 Make and Model
```

The string-variable storage types are `str1`, `str2`, . . . , `str2045`, and `strL`. For example, variable `make` is a `str18` variable. It can contain strings of up to 18 characters long. The strings are not all 18 characters long.

```
. list make in 1/2
     +-------------+
     | make        |
     |-------------|
  1. | AMC Concord |
  2. | AMC Pacer   |
     +-------------+
```

`str18` means that the variable cannot hold a string longer than 18 bytes, and even that is an unimportant detail, because Stata automatically promotes `str#` variables to be longer when required.

```
. replace make = "Mercedes Benz Gullwing" in 1
variable make was str18 now str22
(1 real change made)
```

Strings in Stata can also be stored in labels and notes that let you see information about your dataset. See **[U] 12.6 Dataset, variable, and value labels** and **[U] 12.7 Notes attached to data**. Strings in Stata programs can be stored in string scalars, macros, characteristics, and in stored results.

Stata provides a suite of string functions, such as `strlen()`, `substr()`.

```
. generate len = strlen(make)
. generate str first5 = substr(make, 1,5)
. list make len first5 in 1/2
```

	make	len	first5
1.	Mercedes Benz Gullwing	22	Merce
2.	AMC Pacer	9	AMC P

Many Stata commands can use string variables.

```
. generate str brand = word(make, 1)
. tabulate brand
```

brand	Freq.	Percent	Cum.
AMC	2	2.70	2.70
Audi	2	2.70	5.41
BMW	1	1.35	6.76
Buick	7	9.46	16.22
Cad.	3	4.05	20.27
Chev.	6	8.11	28.38
Datsun	4	5.41	33.78
Dodge	4	5.41	39.19
Fiat	1	1.35	40.54
Ford	2	2.70	43.24
Honda	2	2.70	45.95
Linc.	3	4.05	50.00
Mazda	1	1.35	51.35
Merc.	6	8.11	59.46
Mercedes	1	1.35	60.81
Olds	7	9.46	70.27
Peugeot	1	1.35	71.62
Plym.	5	6.76	78.38
Pont.	6	8.11	86.49
Renault	1	1.35	87.84
Subaru	1	1.35	89.19
Toyota	3	4.05	93.24
VW	4	5.41	98.65
Volvo	1	1.35	100.00
Total	74	100.00	

Beginning in Stata 14, text in Stata strings can include Unicode characters and is encoded as UTF-8. This means that you can use plain ASCII characters (also known as "lower ASCII" and stored as 0–127 on computers) like those shown above. You can also use the remaining Latin characters, as well as characters from the Chinese, Cyrillic, and Japanese alphabets, among others. However, if

you have characters other than ASCII in your datasets, do-files, or ado-files, you may need to take special steps. See [U] **12.4.2 Handling Unicode strings**.

12.4.2 Handling Unicode strings

If you do not have Unicode characters beyond the plain ASCII characters, you do not need to use any special steps to work with your data. In many cases, the same is true even if you do have other Unicode characters. While it is impossible to provide a rule for every situation, there are some general guidelines that you should be aware of.

The fundamental concept to understand is the difference between characters and bytes. Characters are what you see. For example, "a", "Z", and "@" are characters. Bytes are used to encode characters, which are stored on a computer.

For plain ASCII characters, there is a one-to-one mapping between the number of bytes and the number of characters. By contrast, UTF-8 encoded Unicode characters require two, three, or four bytes. For this reason, strings containing Unicode characters require string functions that recognize whole characters; see [U] **12.4.2.1 Unicode string functions**. Some characters from older Stata files, known as extended ASCII characters, will not display correctly and can cause unexpected results. To avoid this, you must properly convert your older datasets and text files, such as do-files, if they contain extended ASCII. See [U] **12.4.2.6 Advice for users of Stata 13 and earlier**.

If you do have characters in your data other than plain ASCII characters, or if you write commands for others to use, you should read the following sections.

12.4.2.1 Unicode string functions

Some of Stata's string functions exist in Unicode-aware versions so they can understand the string as a sequence of Unicode characters rather than as a sequence of bytes. At times, you will need to use one of these Unicode-aware functions to return accurate results. For example, suppose that our data on `make` included a car manufactured by Clénet Coachworks.

If we wanted to know the correct string length, we would use `ustrlen()`, not `strlen()`. The former will give you the answer you expect, 17, while the latter will return the number of bytes used to store that string, 18.

There are other Unicode-aware functions. For example, to change Unicode characters to uppercase, lowercase, or titlecase, use functions `ustrupper()`, `ustrlower()`, or `ustrtitle()`. If you want to see if there is a Unicode variant of the string function you want to use, check [FN] **String functions**.

Note that Unicode-aware functions are not required just because a variable contains UTF-8 characters beyond the plain ASCII range. For example, suppose that rather than wanting the string length, we wanted to replace "Mercedes" with "Merc.". We could use `subinstr()` instead of `usubinstr()` because neither "Mercedes" nor "Merc." contains UTF-8 characters.

Other Unicode-aware functions address the display columns. These functions are primarily of interest to programmers. See [U] **12.4.2.2 Displaying Unicode characters**.

If you are in doubt, or if you are writing code to be used in a general way by others, you should use the Unicode-aware version of a string function, if it exists. The Unicode-aware functions generally have the same names as the regular string functions, but with "u" as a prefix. See [FN] **String functions**.

12.4.2.2 Displaying Unicode characters

Stata has a concept called a display column to ensure that the fixed-width output in Stata's Results and Viewer windows continues to align properly. Stata automatically displays each character in one or two display columns.

Most users, even users with UTF-8 characters beyond the ASCII range, will find that there is no distinction between the number of characters and the number of display columns because most characters are displayed in one column. Some wider characters, however, such as Chinese, Japanese, and Korean (CJK) characters, occupy two display columns.

You may occasionally wish to account for the number of display columns that a string occupies. Just as some Stata functions understand Unicode characters, some functions understand display columns. These functions are prefixed with "ud". For example, you can obtain the number of display columns for a string with `udstrlen(`*string*`)`. If you want to extract a subset of characters from the beginning of a string and make sure it fits within 10 display columns, use `udsubstr(`*string*`,1,10)`. See [FN] **String functions** for more information.

12.4.2.3 Encodings

An encoding is the way a computer stores a given string of text. ASCII and UTF-8, which is how Stata stores all text, are examples of encodings. Plain ASCII characters are stored as a single byte, each with a value between 0 and 127. "a", "Z", and "@" are all examples of plain ASCII characters, and their respective byte values are 97, 90, and 64.

The letter "á" is also a character. In UTF-8 encoding, that single character is stored as two bytes: 195 and 161. All Unicode characters beyond the plain ASCII range are stored as two or more bytes, and each of those bytes has a value between 128 and 255. Some characters in UTF-8 encoding take three or even four bytes to store.

Not every possible combination of bytes represents a valid Unicode character. Because two or more bytes are required to encode a Unicode character, any single byte between 128 and 255 is not a valid Unicode character. Invalid Unicode characters are most likely to occur if you have extended ASCII characters in a file from a previous version of Stata; see **[U] 12.4.2.6 Advice for users of Stata 13 and earlier**.

If you have text in other encodings, including text in Stata files, you must convert it to UTF-8 for it to display properly and for some of Stata's string functions to work properly. To convert a file to UTF-8, you must know the original encoding. The most common encoding is Windows-1252. To obtain a list of other common encodings as well as a list of all possible encodings, see `unicode encoding list` and `unicode encoding alias` in [D] **unicode encoding**.

The `unicode analyze` and `unicode translate` commands help to convert text files and Stata datasets. See [D] **unicode translate** for more information. Also see **[U] 12.4.2.6 Advice for users of Stata 13 and earlier**.

12.4.2.4 Locales in Unicode

A locale identifies a community with a certain set of rules for how their language should be written. A locale can be as general as a certain language, such as "en" for English, or it can be specific to a country or region, such as "en_US" for U.S. English and "en_HK" for Hong Kong English.

Locales use *tags* to define how specific they are to language variants; these *tags* include 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.

Certain language-specific operations require a locale to be properly carried out. For example, in English, the uppercase version of "i" is "I". In Turkish, the uppercase version of "i" is an "İ" [that is, an "I" with a dot above it (Unicode character \u0130)]. To specify how to properly convert a letter to uppercase, you can specify the locale in the `ustrupper()` function, for example, `ustrupper("i", "en_US")`.

The following Stata functions are locale-dependent: `ustrupper()`, `ustrlower()`, `ustrtitle()`, `ustrword()`, `ustrwordcount()`, `ustrcompare()`, `ustrcompareex()`, `ustrsortkey()`, and `ustrsortkeyex()`.

If you do not explicitly specify a locale when using these functions, the current Stata `locale_functions` setting will be used. You can see the current setting by typing

```
. display c(locale_functions)
```

and

```
. unicode locale list
```

to see a list of supported locales. It is unlikely, however, that you will ever need to change the `set locale_functions` setting.

See [P] **set locale_functions** for more information about setting the locale, including information about the how default value is determined.

12.4.2.5 Sorting strings containing Unicode characters

This section deals with *collation*, sorting strings that contain Unicode characters, and the special rules that apply when you do. Many users will find that they can skip this section.

If you do not have Unicode characters beyond the plain ASCII range, you can skip this section. You can also skip this section if you are interested in using `sort` only so that you can use another command or prefix. For example, suppose you have the variable `id` that contains Unicode characters and you want to type

```
. statsby id: regress y x1 x2
```

If your aim is to group the coefficients by `id` only and the exact order of `id` does not matter, then the advice in this section does not apply to you. The usual `sort` command will be sufficient.

The steps described here also do not apply to commands that require the data to be sorted or grouped. For example, suppose that you wish to perform a one-to-one `merge` for two datasets using `id` as the key variable. You can just type

```
. merge 1:1 id using ...
```

Finally, you can skip this section if you do not want to apply language-specific rules to the Unicode characters in your data. For example, if you do not particularly care that "café" is sorted before or after "cafe", but only that the two words are distinguished, then this section is not for you.

For users who wish to sort or compare strings as a human might, there are four rules that you should keep in mind.

1. Sorting is locale-specific.
2. You must generate a sort key. You cannot sort by the variable itself.
3. There are multiple options for controlling the order of Unicode strings.
4. Concatenation is required to sort by *varlist*.

Rules 1 and 3 also apply to string comparisons. We explain each of these rules in more detail below. But first, it may be helpful to review how sorting works in general.

Stata's `sort` command and Stata's logical operators $>$ and $<$ order strings based on the byte values of the characters. For example, the byte value for "a" is 97 and the byte value for "A" is 65, so "a" $>$ "A". Similarly, the byte value for "Z" is 90, so "a" $>$ "Z". This means that words starting with "Z" come before "a", which might surprise you because, in an English dictionary, words starting with "Z" would certainly come after words starting with "a".

For example, suppose we have the following data:

```
. list mystr

     +--------+
     | mystr  |
     |--------|
  1. | Quick  |
  2. | quick  |
  3. | brown  |
  4. |   Fox  |
  5. |  jump  |
     +--------+
```

If we `sort` these data and then `list` them, we see

```
. sort mystr
. list

     +--------+
     | mystr  |
     |--------|
  1. |   Fox  |
  2. | Quick  |
  3. | brown  |
  4. |  jump  |
  5. | quick  |
     +--------+
```

This probably is not the order you would have placed these values in.

To sort the values of `mystr` in a more human fashion, you can use a Unicode tool, known as the Unicode collation algorithm (UCA), for comparing and sorting strings in a language-aware manner. Given knowledge of a locale and perhaps some optional instructions about whether to consider things like case and diacritical marks, the UCA can order Unicode strings as a human (or a dictionary) would.

Stata and Mata provide access to the UCA via the `ustrcompare()`, `ustrcompareex()`, and `ustrsortkey()`, `ustrsortkeyex()` functions. Stata also provides access via the `collatorlocale()` and `collatorversion()` functions.

See http://www.unicode.org/reports/tr10/ for the formal specification of the UCA.

Rule 1: Sorting is locale-dependent.

The ordering of strings in Unicode depends on the specified language and any optional tags and keywords that are specified with the locale.

For the `ustrcompare()` and `ustrsortkey()` functions, the default rules for ordering by language (and country, if specified) are used. You can use the current Stata `locale_functions` setting or specify a different locale with these each of these functions. See [U] **12.4.2.4 Locales in Unicode** for more information about locales, and see [D] **unicode collator** for information about locale-specific collation.

For advanced control of ordering, use the `ustrcompareex()` and `ustrsortkeyex()` functions. These functions allow you to specify a collation keyword, which is used for finer control for ordering, such as whether case-sensitivity and diacritical marks matter. For example, "pinyin" and "stroke"

for the Chinese language produce different sort orders. A list of valid collation keywords and their meanings may be found http://unicode.org/repos/cldr/trunk/common/bcp47/collation.xml.

Rule 2: You must generate a sort key.

To appropriately sort your data with all the rules of the locale applied, you must generate a *sort key*. A sort key is a string created by the UCA that can be used to sort Unicode strings. You sort on the sort key rather than the Unicode string variable. The sort key is not a variable we would ever want to use for any purpose other than data management because it is not human-readable.

You can generate a sort key using either `ustrsortkey()` or `ustrsortkeyex()`. You then `sort` your data by the new variable. The following example illustrates the difference between `sort` and Unicode collation using the above functions:

```
. generate sortkey = ustrsortkey(mystr, "en")
. sort sortkey
. list mystr
```

	mystr
1.	brown
2.	Fox
3.	jump
4.	quick
5.	Quick

It is important to note that the Stata dataset is sorted by `sortkey` and not by `mystr`, even though `mystr` appears to be sorted correctly. Stata is aware of sorting only by `sortkey`. This means that if you need to perform an operation that relies on the sort order, such as `by`, you should use `sortkey` rather than `mystr`, such as

```
. by sortkey: ...
```

Also note that sort keys generated from one locale or one set of advanced options in `ustrsortkeyex()` are usually not compatible or comparable with sort keys generated from another locale or another set of options. For example, you should not compare the sort keys generated from the `"en"` locale with those generated from the `"fr"` locale.

❑ Technical note

The effective locale may be different from the requested locale. Thus, the sort keys obtained on a different machine, or even on a different user account on the same machine, may be different unless the locale is specified. You can retrieve the effective locale with the function `collatorlocale()` and then use that effective locale in future calls to the Unicode ordering functions.

❑

❑ Technical note

The Unicode standard is constantly adding more characters, and language rules are constantly changing, which means that sort keys produced by the current version of the UCA may not be compatible with sort keys of the same strings produced by future versions of the UCA.

You can use function `collatorversion()` to retrieve the current version of the collation routine and then store the result (for example, in a variable characteristic) with any saved sort keys if those keys are intended for future use.

If the current version is different from the saved sort key, then you should regenerate the sort key variables if you want them to be up-to-date with the new language rules or if you want to compare them with newly generated sort keys.

❑

Rule 3: There are multiple options for controlling the order of Unicode strings.

This may appear straightforward, but some finer points of the UCA could surprise you. Consider an example of string comparisons.

```
. display ustrcompare("café","cafe","fr")
1
```

Here we asked Stata to compare the string "café" to the string "cafe" using the French locale (`"fr"`). Stata reported `1`, which means that in this case "café" is considered to be greater than "cafe". If we were sorting our data, this means "café" would be sorted after "cafe".

Now consider

```
. display ustrcompare("café du monde","cafe new york","fr")
-1
```

It might surprise you that the result is `-1`, which means that in this case "café du monde" is considered to be less than "cafe new york", even though we already established that "café" is greater than "cafe".

The reason is that the difference between "d" and "n" in the second word of each string is considered by the UCA to be a *primary difference*, whereas the difference between "é" and "e" in the first word of each string is a diacritical mark which is considered to be a *secondary difference*. The primary difference outweighs the secondary difference even though it occurs later in the string.

The default behavior of `ustrcompare()` and `ustrsortkey()` should be sufficient for most comparison and sorting needs. For advanced control over how Unicode strings are ordered, including whether the ordering should be based on differences from primary to quaternary, use `ustrcompareex()` and `ustrsorkeyex()`. See [FN] **String functions**.

Rule 4: Concatenation is required to sort by a varlist.

An important implication of Rule 3 arises when creating sort keys for Unicode strings. Ordinarily, if you want to sort on two string variables, you can simply type

```
. sort string1 string2
```

However, to take full advantage of the UCA while sorting two or more strings, you should first concatenate them and then sort the result.

```
. generate string3 = string1 + string2
. generate sortkey = ustrsortkey(sting3, "fr")
. sort sortkey
```

If you do not do this, then primary differences that might arise in `string2` will not override any secondary differences in `string1`.

12.4.2.6 Advice for users of Stata 13 and earlier

In this section, we discuss how to use your older Stata files in modern Stata and also points you should consider when sharing your modern Stata files with users of Stata 13 and earlier.

In Stata 13 and earlier, Unicode characters were not supported. If you have only plain ASCII characters in your datasets, do-files, and ado-files, then you do not need to take any special steps to continue using these files with modern Stata. You can use `saveold` to share your dataset with users of older versions of Stata. Your do-files and ado-files can be shared directly.

If files you used with Stata 13 or earlier contain strings with extended ASCII characters, you should convert those strings to Unicode UTF-8 encoding so they will work properly with modern Stata. The `unicode analyze` command will check your files to see if they need conversion, and if so, the `unicode translate` command will convert them to UTF-8 encoding. See [D] **unicode translate**. To convert a single variable, use `ustrfrom()`.

If you have Unicode characters in your dataset and you wish to share it with a user of Stata 13 or earlier, be aware that while they can load a dataset created with the `saveold` command, their copy of Stata is not Unicode-aware and will not display Unicode characters properly. Before you use `saveold`, you can convert your string variables from the UTF-8 encoding to an extended ASCII encoding by using `ustrto()`. We recommend that you `generate` a new variable when using `ustrfrom()` or `ustrto()` so that you can review the results and make sure you are satisfied before you `replace` your existing variable. `ustrfrom()` and `ustrto()` may also be used with Mata string matrices.

12.4.3 Strings containing identifying data

String variables often contain identifying information, such as the patient's name or the name of the city or state. Such strings are typically listed but are not used directly in statistical analysis, although the data might be sorted on the string or datasets might be merged on the basis of one or more string variables.

12.4.4 Strings containing categorical data

Strings sometimes contain information to be used directly in analysis, such as the patient's sex, which might be coded "male" or "female". Stata shows a decided preference for such information to be numerically encoded and stored in numeric variables. Stata's statistical routines treat string variables as if every observation records a numeric missing value. Stata provides two commands for converting string variables into numeric codes and back again: `encode` and `decode`. See [U] **23.2 Categorical string variables** and [U] **11.4.3 Factor variables**.

12.4.5 Strings containing numeric data

If a string variable contains the character representation of a number, say, `myvar` contains "1", "1.2", and "−5.2", you can convert the string into a numeric value by using the `real()` function or the `destring` command. For example,

```
. generate newvar = real(myvar)
```

To convert a numeric variable to its string representation, you can use the `string()` function or the `tostring` command. For example,

```
. generate as_str = string(numvar)
```

See [FN] **String functions** and [D] **destring**.

12.4.6 String literals

A string literal is a sequence of printable characters enclosed in quotes. The quotes are not considered part of the string; they merely serve to delimit the beginning and end of the string. The following are examples of string literals:

```
"Hello, world"
"String"
"string"
" string"
"string "
""
"x/y+3"
"1.2"
```

All the strings above are distinct. Capitalization matters, as do leading and trailing spaces. Also note that `"1.2"` is a string and not a number because it is enclosed in quotes.

There is never a circumstance in which a string cannot be delimited with quotes, but there are instances where strings do not have to be delimited by quotes, such as when inputting data. In those cases, nondelimited strings are stripped of their leading and trailing spaces. Delimited strings are always accepted as is.

The list above could also be written as

```
‘"Hello, world"’
‘"String"’
‘"string"’
‘" string"’
‘"string "’
‘""’
‘"x/y+3"’
‘"1.2"’
```

`‘"` and `"’` are called compound double quotes.

Use of compound double quotes can help solve the problem of typing strings that themselves contain double quotes.

```
‘"Bob said, "Wow!" and promptly fainted."’
```

Strings in compound quotes can themselves contain compound quotes.

```
‘"The compound quotes characters are ‘" and "’"’
```

12.4.7 str1–str2045 and str

`str` is something `generate` understands. We will get to that.

`str1`–`str2045` are known as Stata's fixed-length string storage types.

They are called that because, in your dataset, if a variable is stored as a `str#`, then each observation requires # bytes to store the contents of the variable. You obviously do not want # to be longer than necessary. Stata's `compress` command will shorten `str#` strings that are unnecessarily long.

```
. use http://www.stata-press.com/data/r15/auto, clear
(1978 Automobile Data)
. compress
  variable mpg was int now byte
  variable rep78 was int now byte
  variable trunk was int now byte
  variable turn was int now byte
  variable make was str18 now str17
  (370 bytes saved)
```

In [U] 12.4.1 Overview, we used `str` with `generate`:

```
. generate str brand = word(make, 1)
```

`str` is something `generate` understands and tells `generate` to create a `str#` variable of the minimum required length. Although you cannot tell from the output, `generate` created variable `brand` as a `str7`.

Stata commands automatically promote `str#` storage types when necessary:

```
. replace make = "Mercedes Benz Gullwing" in 1
variable make was str17 now str22
(1 real change made)
```

In fact, if the string to be stored is longer than 2,045 bytes, `generate` and `replace` will even promote to `strL`. We discuss `strL`s in the next section.

12.4.8 strL

`strL` variables can be 0 to 2-billion bytes long.

The "L" stands for long, and `strL` is often pronounced *sturl*.

`strL` variables are not required to be longer than 2,045 bytes.

`str#` variables can store strings of up to 2,045 bytes, so `strL` and `str#` overlap. This overlap is comparable to the overlap of the numeric types `int` and `float`. Any number that can be stored as an `int` can be stored as a `float`. Similarly, any string that can be stored as a `str#`, can be stored as a `strL`. The reverse is not true. In addition, `strL` variables can hold binary strings, whereas `str#` variables can only hold text strings. Thus the analogy between `str#`/`strL` and `int`/`float` is exact. There will be occasions when you will want to use `strL` variables in preference to `str#` variables, just as there are occasions when you will want to use `float` variables in preference to `int` variables.

`strL` variables work just like `str#` variables. Below we repeat what we did in [U] 12.4.1 Overview using a `strL` variable.

```
. use http://www.stata-press.com/data/r15/auto, clear
(1978 Automobile Data)
. generate strL mymake = make
. describe mymake
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
mymake          strL    %9s
. list mymake in 1/2

     +-------------+
     |      mymake |
     |-------------|
  1. | AMC Concord |
  2. |   AMC Pacer |
     +-------------+
```

We can replace strL values just as we can replace str# values:

```
. replace mymake = "Mercedes Benz Gullwing" in 1
(1 real change made)
```

We can use string functions with strL variables just as we can with str# variables:

```
. generate len = strlen(mymake)
. generate strL first5 = substr(mymake, 1, 5)
. list mymake len first5 in 1/2
```

	mymake	len	first5
1.	Mercedes Benz Gullwing	22	Merce
2.	AMC Pacer	9	AMC P

We can even make tabulations:

```
. generate strL brand = word(mymake, 1)
. tabulate brand
```

brand	Freq.	Percent	Cum.
AMC	2	2.70	2.70
Audi	2	2.70	5.41
BMW	1	1.35	6.76
(*output omitted*)			
Volvo	1	1.35	100.00
Total	74	100.00	

The only limitations are the following:

1. You cannot use strL variables as the matching (key) variables in a match merge of two datasets.
2. strL variables cannot be used with fillin.

strL variables are stored differently from str# variables. str# variables require # bytes per observation. strL variables require the actual number of bytes per string per observation, which means strLs require even less memory than str# when the value being stored is less than # bytes long. Most strLs, however, have an 80-byte overhead per value stored; the exception is strLs containing empty strings, in which case the overhead is 8 bytes.

Whether strL or str# requires less memory for storing the same string values depends on the string values themselves. compress can be used to figure that out:

```
. compress
  variable mpg was int now byte
  variable rep78 was int now byte
  variable trunk was int now byte
  variable turn was int now byte
  variable len was float now byte
  variable make was str18 now str17
  variable mymake was strL now str22
  variable first5 was strL now str5
  variable brand was strL now str8
  (12,420 bytes saved)
```

compress decided to demote all of our strL variables to str# to save memory.

compress, however, never promotes a str# variable to a strL, even if that would save memory. It does not do this because, as we mentioned, there are a few things you can do with str# variables that you cannot do with strL variables.

You can use recast to promote str# to strL:

```
. * variable make is currently str17
. recast strL make
. describe make
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
make            strL    %-9s                  Make and Model
. compress make
  variable make was strL now str17
  (5,607 bytes saved)
```

12.4.9 strL variables and duplicated values

You would never know it, but when strL variables have the same values across observations, Stata stores only one copy of each value. This is called coalescing, and it saves memory.

Stata mostly coalesces strL variables automatically as they are created, but sometimes duplicate values escape its attention. When you type compress, however, Stata looks for coalescing opportunities. You might see

```
. compress x
x is strL now coalesced
(11,301,687 bytes saved)
```

We recommend that you type compress occasionally when strL variables are present.

12.4.10 strL variables and binary strings

strLs can hold binary strings. A binary string is, technically speaking, any string that contains binary 0. Here is an example:

```
. use http://www.stata-press.com/data/r15/auto, clear
(1978 Automobile Data)
. replace make = "a" + char(0) + "b" in 1
variable make was str18 now strL
(1 real change made)
. list make in 1
```

	make
1.	a\0b

list displays binary zeros as \0.

If we did this same experiment with a str# variable and include the nopromote option to prevent promotion, we would see something different:

```
. use http://www.stata-press.com/data/r15/auto, clear
(1978 Automobile Data)
. replace make = "a" + char(0) + "b" in 1, nopromote
(1 real change made)
. list make in 1

     +------+
     | make |
     |------|
  1. |    a |
     +------+
```

For str# strings, binary 0 indicates the end of the string, and thus the variable really does contain "a" in the first observation.

str# variables cannot contain binary 0; strL variables can.

compress knows this. If we typed compress in the first example, we would discover that compress would not demote make to be a str#. It would not do this because one of the values could not be stored in a str# variable. This is no different from compress not demoting a float variable to an int because one of the values is 1.5.

12.4.11 strL variables and files

strLs can be used to hold the contents of files. We have data on 10 patients. Some of the data have been coded from doctor notes, and those notes are stored in notes_2217.xyz, notes_2221.xyz, notes_2222.xyz, and so on. We could do the following:

```
. generate strL notes = fileread("notes_2217.xyz") in 1
. replace notes = fileread("notes_2221.xyz") in 2
. replace notes = fileread("notes_2222.xyz") in 3
. ...
```

It would be even easier for us to type

```
. generate str fname = "notes_" + string(patid) + ".xyz"
. generate strL notes = fileread(fname)
```

The original files can be re-created from the copies stored in Stata. To re-create all the files, we could type

```
. generate len = filewrite(fname, notes)
```

If we want to know whether the phrase "Diabetes Mellitus Type 1" appears in the notes and whether doctors recorded the disease as T1DM, we can type

```
. generate t2dm = (strpos("notes", "T1DM")) != 0
```

Of course, that depends on the notes_*.xyz files being either text or text-like enough so that the T1DM would show up as "T1DM".

Note that strpos() and all of Stata's string functions also work with binary strings.

12.4.12 String display formats

The format for strings is %[-]#s, such as %18s and %-18s. # may be up to 2,045. # indicates the width of the field. %#s specifies that the string be displayed right-aligned in the field, and %-#s specifies that the string is displayed left-aligned.

Stata sets good default formats for str# variables. The default format is %#s, so if a variable is str18, its default format is %18s.

Stata sets poor default formats for strL variables. Stata uses %9s in all cases. Because strL variables can be so long, there is no good choice for the format; the question is merely how much of the string you want to see.

When the format is too short for the length of the string, whether the string is str# or strL, Stata usually displays # − 2 characters of the string and adds two dots at the end. We say "usually" because a few commands are able to do something better than that.

12.4.13 How to see the full contents of a strL or a str# variable

By default, the list command shows only the first part of long strings, followed by two dots. How much list shows is determined by the width of your Results window.

list will show the first 2,045 bytes of long strings, whether stored as strLs or str#s, if you add the notrim option.

```
. list, notrim
```
(*output omitted*)

```
. list mystr, notrim
```
(*output omitted*)

```
. list mystr in 5, notrim
```
(*output omitted*)

Another way to display long strings is to use the display command. With display, you can see the entire contents. To display the fifth observation of the variable mystr, you type

```
. display _asis mystr[5]
```
(*output omitted*)

That one command can produce a lot of output if the string is long, even hundreds of thousands of pages! Remember that you can press *Break* to stop the listing.

To see the first 5,000 characters of the string, you type

```
. display _asis usubstr(mystr[5], 1, 5000)
```

For detailed information about displaying Unicode characters beyond plain ASCII characters, see **[U] 12.4.2.2 Displaying Unicode characters**.

Very rarely, a string variable might contain SMCL output. SMCL is Stata's text markup language. A variable might contain SMCL if you used fileread() to read a Stata log file into it. In that case, you can see the text correctly formatted by typing

```
. display as txt mystr[1]
```
(*output omitted*)

To learn more about other features of display, see [R] **display**.

12.4.14 Notes for programmers

The maximum length of macros is shorter than that of `strLs`. This means the following:

1. You can use macros in string expressions without fear that results will be truncated.
2. You can enclose expanded macros in quotes—`‘"‘macname’"’`—to form string literals without fear of truncation.
3. Macros cannot hold binary strings. If you are working with binary strings, use string scalars, which are also implemented as `strLs`. See [P] **scalar**.
4. You should not assume that the result of a string expression will fit into a macro. If you are sure it will, go ahead and store the result into a macro. If you are not sure, use a string scalar, which can hold a `strL`.
5. You should not assume that the contents of a `strL` variable will fit into a macro. Use string scalars.
6. In programming, use string scalars just as you would use numeric scalars.

```
program ...
        version 15.0
        ...
        tempname mystr
        ...
        scalar ‘mystr’ = ...
        ...
        generate ... = ...‘mystr’...
        ...
end
```

`mystr` in the above code is a macro containing a temporary name. Thus `‘mystr’` is a reference, not an expansion, of the contents of the string scalar.

12.5 Formats: Controlling how data are displayed

Formats describe how a number or string is to be presented. For instance, how is the number 325.24 to be presented? As `325.2`, or `325.24`, or `325.240`, or `3.2524e+02`, or `3.25e+02`, or some other way? The *display format* tells Stata exactly how to present such data. You do not have to specify display formats because Stata always makes reasonable assumptions about how to display a variable, but you always have the option.

12.5.1 Numeric formats

A Stata numeric format is formed by

first type	`%`	to indicate the start of the format
then optionally type	`-`	if you want the result left-aligned
then optionally type	`0`	if you want to retain leading zeros (1)
then type	a number w	stating the width of the result
then type	`.`	
then type	a number d	stating the number of digits to follow the decimal point
then type		
either	`e`	for scientific notation, e.g., 1.00e+03
or	`f`	for fixed format, e.g., 1000.0
or	`g`	for general format; Stata chooses based on the number being displayed
then optionally type	`c`	to indicate comma format (not allowed with `e`)

(1) Specifying `0` to mean “include leading zeros” will be honored only with the `f` format.

For example,

```
%9.0g    general format, 9 columns wide
                 sqrt(2) =  1.414214
                   1,000 =       1000
              10,000,000 =  1.00e+07
%9.0gc   general format, 9 columns wide, with commas
                 sqrt(2) =  1.414214
                   1,000 =      1,000
              10,000,000 =  1.00e+07
%9.2f    fixed format, 9 columns wide, 2 decimal places
                 sqrt(2) =       1.41
                   1,000 =    1000.00
              10,000,000 = 10000000.00
%9.2fc   fixed format, 9 columns wide, 2 decimal places, with commas
                 sqrt(2) =       1.41
                   1,000 =   1,000.00
              10,000,000 = 10,000,000.00
%9.2e    exponential format, 9 columns wide
                 sqrt(2) =  1.41e+00
                   1,000 =  1.00e+03
              10,000,000 =  1.00e+07
```

Stata has three numeric format types: `e`, `f`, and `g`. The formats are denoted by a leading percent sign (%) followed by the string *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 *w*. Finally, a character denotes the format type (`e`, `f`, or `g`), and a `c` may optionally be appended to that to indicate that commas are to be included in the result (`c` is not allowed with `e`.)

By default, every numeric variable is given a %*w*`.0g` format, where *w* is large enough to display the largest number of the variable's type. The %*w*`.0g` format is a set of formatting rules that present the values in as readable a fashion as possible without sacrificing precision. The `g` format changes the number of decimal places displayed whenever it improves the readability of the current value.

The default formats for each of the numeric variable types are

```
byte     %8.0g
int      %8.0g
long    %12.0g
float    %9.0g
double  %10.0g
```

You can change the format of a variable by using the `format` *varname* %*fmt* command.

In addition to %*w*`.0g`, %*w*`.0gc` is also allowed and displays numbers with commas. "One thousand" is displayed as 1000 in %`9.0g` format and as 1,000 in %`9.0gc` format.

In addition to using %*w*`.0g` and %*w*`.0gc`, you can use %*w*.*d*`g` and %*w*.*d*`gc`, $d > 0$. For example, %`9.4g` and %`9.4gc`. The 4 means to display approximately four significant digits. For instance, the number 3.14159265 in %`9.4g` format is displayed as 3.142, 31.4159265 as 31.42, 314.159265 as 314.2, and 3141.59265 as 3142. The format is not exactly a significant digit format because 31415.9265 is displayed as 31416, not as 3.142e+04.

Under the `f` format, values are always displayed with the same number of decimal places, even if this results in a loss in the displayed precision. Thus the `f` format is similar to the C `f` format. Stata's `f` format is also similar to the Fortran `F` format, but, unlike the Fortran `F` format, it switches to `g` whenever a number is too large to be displayed in the specified `f` format.

In addition to %*w*.*d*`f`, the format %*w*.*d*`fc` can display numbers with commas.

The `e` format is similar to the C `e` and the Fortran `E` format. Every value is displayed as a leading digit (with a minus sign, if necessary), followed by a decimal point, the specified number of digits, the letter `e`, a plus sign or a minus sign, and the power of 10 (modified by the preceding sign) that multiplies the displayed value. When the `e` format is specified, the width must exceed the number of digits that follow the decimal point by at least seven to accommodate the leading sign and digit, the decimal point, the `e`, and the signed power of 10.

▷ Example 3

Below we have a five-observation dataset with three variables: `e_fmt`, `f_fmt`, and `g_fmt`. All three variables have the same values stored in them; only the display format varies. `describe` shows the display format to the right of the variable type.

```
. use http://www.stata-press.com/data/r15/format, clear
. describe
Contains data from http://www.stata-press.com/data/r15/format.dta
  obs:             5
 vars:             3                          12 Mar 2016 15:18
 size:            60
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
e_fmt           float   %9.2e
f_fmt           float   %10.2f
g_fmt           float   %9.0g
-------------------------------------------------------------------------------
Sorted by:
```

The formats for each of these variables were set by typing

```
. format e_fmt %9.2e
. format f_fmt %10.2f
```

It was not necessary to set the format for the `g_fmt` variable because Stata automatically assigned it the `%9.0g` format. Nevertheless, we could have typed `format g_fmt %9.0g`. Listing the data results in

```
. list
```

	e_fmt	f_fmt	g_fmt
1.	2.80e+00	2.80	2.801785
2.	3.96e+06	3962322.50	3962323
3.	4.85e+00	4.85	4.852834
4.	-5.60e-06	-0.00	-5.60e-06
5.	6.26e+00	6.26	6.264982

◁

❑ Technical note

The discussion above is incomplete. There is one other format available that will be of interest to numerical analysts. The `%21x` format displays base 10 numbers in a hexadecimal (base 16) format. The number is expressed in hexadecimal (base 16) digits; the number a`X+`b means $a \times 2^b$. For example,

```
. display %21x 1234.75
+1.34b0000000000X+00a
```

Thus the base 10 number 1,234.75 has a base 16 representation of `1.34bX+0a`, meaning

$$\left(1 + 3 \cdot 16^{-1} + 4 \cdot 16^{-2} + 11 \cdot 16^{-3}\right) \times 2^{10}$$

Remember, the hexadecimal–decimal equivalents are

hexadecimal	decimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
a	10
b	11
c	12
d	13
e	14
f	15

See [U] **12.2 Numbers**.

❑

12.5.2 European numeric formats

The three numeric formats `e`, `f`, and `g` will use '`,`' to indicate the decimal symbol if you specify their width and depth as *w*`,`*d* rather than *w*`.`*d*. For instance, the format `%9,0g` will display what Stata would usually display as 1.5 as `1,5`.

If you use the European specification with `fc` or `gc`, the comma will be presented as a period. For instance, `%9,0gc` would display what Stata would usually display as 1,000.5 as `1.000,5`.

If this way of presenting numbers appeals to you, consider using Stata's `set dp comma` command. `set dp comma` tells Stata to interpret nearly all `%`*w*`.`*d*{`g`|`f`|`e`} formats as `%`*w*`,`*d*{`g`|`f`|`e`} formats. Most of Stata is written using a period to represent the decimal symbol, and that means that even if you set the appropriate `%`*w*`,`*d*{`g`|`f`|`e`} format for your data, it will affect only displays of the data. For instance, if you type `summarize` to obtain summary statistics or `regress` to obtain regression results, the decimal will still be shown as a period.

`set dp comma` changes that and affects all of Stata. With `set dp comma`, it does not matter whether your data are formatted `%`*w*`.`*d*{`g`|`f`|`e`} or `%`*w*`,`*d*{`g`|`f`|`e`}. All results will be displayed using a comma as the decimal character.

```
. use http://www.stata-press.com/data/r15/auto
(1978 Automobile Data)
. set dp comma
. summarize mpg weight foreign
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         74    21,2973    5,785503         12         41
      weight |         74    3019,459    777,1936       1760       4840
     foreign |         74    ,2972973    ,4601885          0          1
. regress mpg weight foreign
      Source |       SS           df       MS      Number of obs   =        74
-------------+----------------------------------   F(2, 71)        =     69,75
       Model |  1619,2877         2  809,643849   Prob > F        =    0,0000
    Residual |  824,171761        71   11,608053   R-squared       =    0,6627
-------------+----------------------------------   Adj R-squared   =    0,6532
       Total |  2443,45946        73  33,4720474   Root MSE        =    3,4071

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -,0065879   ,0006371   -10,34   0,000    -,0078583   -,0053175
     foreign |  -1,650029   1,075994    -1,53   0,130     -3,7955    ,4954422
       _cons |    41,6797   2,165547    19,25   0,000     37,36172    45,99768
------------------------------------------------------------------------------
```

You can switch the decimal character back to a period by typing `set dp period`.

❑ 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` and then find that the program does not work and instead presents some sort of syntax error.

If, when using any program, you do get an unanticipated error, try setting `dp` back to `period`. See [D] **format** for more information.

Also understand that `set dp comma` affects how Stata outputs numbers, not how it inputs them. You must still use the period to indicate the decimal point on all input. Even with `set dp comma`, you type

```
. replace x=1.5 if x==2
```

❑

12.5.3 Date and time formats

Date and time formats are really a numeric format because Stata stores dates as the number of milliseconds, days, weeks, months, quarters, half-years, or years from 01jan1960; see [U] **24 Working with dates and times**.

The syntax of the `%t` format is

first type	`%`	to indicate the start of the format
then optionally type	`-`	if you want the result left-aligned
then type	`t`	
then type	*character*	to indicate the units
then optionally type	*other characters*	to indicate how the date/time is to be displayed

The letter you type to specify the units is

`C`	milliseconds from 01jan1960, adjusted for leap seconds
`c`	milliseconds from 01jan1960, ignoring leap seconds
`d`	days from 01jan1960
`w`	weeks from 1960-w1
`m`	calendar months from jan1960
`q`	quarters from 1960-q1
`h`	half years from 1960-h1

There are many codes you can type after that to specify exactly how the date/time is to be displayed, but usually, you do not. Most users use the default `%tc` for date/times and `%td` for dates. See [D] **datetime display formats** for details.

12.5.4 String formats

The syntax for a string format is

first type	`%`	to indicate the start of the format
then optionally type	`-`	if you want the result left-aligned
then type	a number	indicating the width of the result
then type	`s`	

For instance, `%10s` represents a string format with a width of 10 display columns; see [U] **12.4.2.2 Displaying Unicode characters**.

For `str`w, the default format is `%`w`s` or `%9s`, whichever is wider. For example, a `str10` variable receives a `%10s` format. Strings are displayed right-justified in the field, unless the minus sign is coded; `%-10s` would display the string left-aligned.

▷ Example 4

Our automobile data contain a string variable called `make`.

```
. use http://www.stata-press.com/data/r15/auto
(1978 Automobile Data)
. describe make
              storage   display    value
variable name   type    format     label      variable label
------------------------------------------------------------------------------
make            str18   %-18s                 Make and Model
. list make in 63/67

     +-------------------+
     |               make |
     |-------------------|
 63. |        Mazda GLC  |
 64. |      Peugeot 604  |
 65. |  Renault Le Car   |
 66. |        Subaru     |
 67. |    Toyota Celica  |
     +-------------------+
```

These values are left-aligned because `make` has a display format of `%-18s`. If we want to right-align the values, we could change the format.

```
. format %18s make
. list make in 63/67

     +-------------------+
     |              make |
     |-------------------|
 63. |       Mazda GLC |
 64. |    Peugeot 604 |
 65. |  Renault Le Car |
 66. |           Subaru |
 67. |   Toyota Celica |
     +-------------------+
```

◁

12.6 Dataset, variable, and value labels

Labels are strings used to label elements in Stata, such as labels for datasets, variables, and values.

12.6.1 Dataset labels

Associated with every dataset is an 80-character *dataset label*, which is initially set to blanks. You can use the `label data "`*text*`"` command to define the dataset label.

▷ Example 5

We have just entered 1980 state data on marriage rates, divorce rates, and median ages. The `describe` command will describe the data in memory:

```
. describe
Contains data
  obs:            50
 vars:             4
 size:         1,200
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
state           str8    %9s
median_age      float   %9.0g
marriage_rate   long    %12.0g
divorce_rate    long    %12.0g
-------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

`describe` shows that there are 50 observations on 4 variables named `state`, `median_age`, `marriage_rate`, and `divorce_rate`. `state` is stored as a `str8`; `median_age` is stored as a `float`; and `marriage_rate` and `divorce_rate` are both stored as `long`s. Each variable's display format (see **[U] 12.5 Formats: Controlling how data are displayed**) is shown. Finally, the data are not in any particular sort order, and the dataset has changed since it was last saved on disk.

We can label the data by typing `label data "1980 state data"`. We type this and then type `describe` again.

```
. label data "1980 state data"
. describe
Contains data
  obs:            50                          1980 state data
 vars:             4
 size:         1,200
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
state           str8    %9s
median_age      float   %9.0g
marriage_rate   long    %12.0g
divorce_rate    long    %12.0g
-------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

◁

The dataset label is displayed by the `describe` and `use` commands.

12.6.2 Variable labels

In addition to the name, every variable has associated with it an 80-character *variable label*. The variable labels are initially set to blanks. You use the `label variable` *varname* "*text*" command to define a new variable label.

▷ Example 6

We have entered data on four variables: `state`, `median_age`, `marriage_rate`, and `divorce_rate`. `describe` portrays the data we entered.

```
. describe
Contains data from states.dta
  obs:            50                          1980 state data
 vars:             4
 size:         1,200
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
state           str8    %9s
median_age      float   %9.0g
marriage_rate   long    %12.0g
divorce_rate    long    %12.0g
-------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

We can associate labels with the variables by typing

```
. label variable median_age "Median Age"
. label variable marriage_rate "Marriages per 100,000"
. label variable divorce_rate "Divorces per 100,000"
```

From then on, the result of `describe` will be

```
. describe
Contains data
  obs:            50                          1980 state data
 vars:             4
 size:         1,200
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
state           str8    %9s
median_age      float   %9.0g                 Median Age
marriage_rate   long    %12.0g                Marriages per 100,000
divorce_rate    long    %12.0g                Divorces per 100,000
-------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

◁

Whenever Stata produces output, it will use the variable labels rather than the variable names to label the results if there is room.

12.6.3 Value labels

Value labels define a correspondence or mapping between numeric data and the words used to describe what those numeric values represent. Mappings are named and defined by the `label define` *lblname # "string" # "string"...* command. The maximum length for the *lblname* is 32 characters. *#* must be an integer or an extended missing value (`.a`, `.b`, ..., `.z`). The maximum length of *string* is 32,000 bytes. Named mappings are associated with variables by the `label values` *varname lblname* command.

▷ Example 7

The definition makes value labels sound more complicated than they are in practice. We create a dataset on individuals in which we record a person's sex, coding 0 for males and 1 for females. If our dataset also contained an employee number and salary, it might resemble the following:

```
. use http://www.stata-press.com/data/r15/gxmpl4
(2007 Employee data)
. describe
Contains data from http://www.stata-press.com/data/r15/gxmpl4.dta
  obs:             7                          2007 Employee data
 vars:             3                          11 Feb 2016 15:31
 size:            84
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
empno           float   %9.0g                 Employee number
sex             float   %9.0g                 Sex
salary          float   %8.0fc                Annual salary, exclusive of bonus
-------------------------------------------------------------------------------
Sorted by:
```

```
. list
```

	empno	sex	salary
1.	57213	0	34,000
2.	47229	1	37,000
3.	57323	0	34,000
4.	57401	0	34,500
5.	57802	1	37,000
6.	57805	1	34,000
7.	57824	0	32,500

We could create a mapping called `sexlabel` defining 0 as "Male" and 1 as "Female", and then associate that mapping with the variable `sex` by typing

```
. label define sexlabel 0 "Male" 1 "Female"
. label values sex sexlabel
```

From then on, our data would appear as

```
. describe
Contains data from http://www.stata-press.com/data/r15/gxmpl4.dta
  obs:             7                          2007 Employee data
 vars:             3                          11 Feb 2016 15:31
 size:            84
```

variable name	storage type	display format	value label	variable label
empno	float	%9.0g		Employee number
sex	float	%9.0g	sexlabel	Sex
salary	float	%8.0fc		Annual salary, exclusive of bonus

```
Sorted by:
. list
```

	empno	sex	salary
1.	57213	Male	34,000
2.	47229	Female	37,000
3.	57323	Male	34,000
4.	57401	Male	34,500
5.	57802	Female	37,000
6.	57805	Female	34,000
7.	57824	Male	32,500

Notice not only that the value label is used to produce words when we `list` the data, but also that the association of the variable `sex` with the value label `sexlabel` is shown by the `describe` command.

◁

❑ Technical note

Value labels and variables may share the same name. For instance, rather than calling the value label `sexlabel` in the example above, we could just as well have named it `sex`. We would then type `label values sex sex` to associate the value label named `sex` with the variable named `sex`.

❑

▷ Example 8

Stata's `encode` and `decode` commands provide a convenient way to go from string variables to numerically coded variables and back again. Let's pretend that, in the example above, rather than coding 0 for males and 1 for females, we created a string variable recording either `"male"` or `"female"`.

```
. use http://www.stata-press.com/data/r15/gxmpl5
(2007 Employee data)
. describe
Contains data from http://www.stata-press.com/data/r15/gxmpl5.dta
  obs:             7                          2007 Employee data
 vars:             3                          11 Feb 2016 15:37
 size:            98
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
empno           float   %9.0g                 Employee number
sex             str6    %9s                   Sex
salary          float   %8.0fc                Annual salary, exclusive of bonus
-------------------------------------------------------------------------------
Sorted by:
. list
```

	empno	sex	salary
1.	57213	male	34,000
2.	47229	female	37,000
3.	57323	male	34,000
4.	57401	male	34,500
5.	57802	female	37,000
6.	57805	female	34,000
7.	57824	male	32,500

We now want to create a numerically encoded variable—we will call it `gender`—from the string variable. We want to do this, say, because we typed `anova salary sex` to perform a one-way ANOVA of salary on sex, and we were told that there were "no observations". We then remembered that all of Stata's statistical commands treat string variables as if they contain nothing but missing values. The statistical commands work only with numerically coded data.

```
. encode sex, generate(gender)
. describe
Contains data from http://www.stata-press.com/data/r15/gxmpl5.dta
  obs:             7                          2007 Employee data
 vars:             4                          11 Feb 2016 15:37
 size:           126
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
empno           float   %9.0g                 Employee number
sex             str6    %9s                   Sex
salary          float   %8.0fc                Annual salary, exclusive of bonus
gender          long    %8.0g      gender     Sex
-------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

`encode` adds a new `long` variable called `gender` to the data and defines a new value label called `gender`. The value label `gender` maps 1 to the string `male` and 2 to `female`, so if we were to `list` the data, we could not tell the difference between the `gender` and `sex` variables. However, they are different. Stata's statistical commands know how to deal with `gender` but do not understand the `sex` variable. See **[U] 23.2 Categorical string variables**.

❑

❑ Technical note

Perhaps rather than employee data, our data are on persons undergoing sex-change operations. There would, therefore, be two sex variables in our data: sex before the operation and sex after the operation. Assume that the variables are named `presex` and `postsex`. We can associate the *same* value label to each variable by typing

```
. label define sexlabel 0 "Male" 1 "Female"
. label values presex sexlabel
. label values postsex sexlabel
```

❑

❑ Technical note

Stata's input commands (`input` and `infile`) can switch from the words in a value label back to the numeric codes. Remember that `encode` and `decode` can translate a string to a numeric mapping and vice versa, so we can map strings to numeric codes either at the time of input or later.

For example,

```
. label define sexlabel 0 "Male" 1 "Female"
. input empno sex:sexlabel salary, label
        empno        sex     salary
  1. 57213 Male 34000
  2. 47229 Female 37000
  3. 57323 0 34000
  4. 57401 Male 34500
  5. 57802 Female 37000
  6. 57805 Female 34000
  7. 57824 Male 32500
  8. end
```

The `label define` command defines the value label `sexlabel`. `input empno sex:sexlabel salary, label` tells Stata to input three variables from the keyboard (`empno`, `sex`, and `salary`), attach the value label `sexlabel` to the `sex` variable, and look up any words that are typed in the value label to try to convert them to numbers. To demonstrate, we `list` the data that we recently entered:

```
. list
```

	empno	sex	salary
1.	57213	Male	34000
2.	47229	Female	37000
3.	57323	Male	34000
4.	57401	Male	34500
5.	57802	Female	37000
6.	57805	Female	34000
7.	57824	Male	32500

Compare the information we typed for observation 3 with the result listed by Stata. We typed `57323 0 34000`. Thus the value of `sex` in the third observation is `0`. When Stata listed the observation, it indicated that the value is `Male` because we told Stata in our `label define` command that zero is equivalent to `Male`.

Let's now add one more observation to our data:

```
. input, label
        empno        sex     salary
  8. 67223 FEmale 33000
'FEmale' cannot be read as a number
  8. 67223 Female 33000
  9. end
```

At first we typed `67223 FEmale 33000`, and Stata responded with "'FEmale' cannot be read as a number". Remember that Stata always respects case, so `FEmale` is not the same as `Female`. Stata prompted us to type the line again, and we did so, this time correctly.

❑

❑ Technical note

Coupled with the `automatic` option, Stata not only can go from words to numbers but can also create the mapping. Let's input the data again, but this time, rather than typing the data, let's read the data from a file. Assume that we have a text file named `employee.raw` stored on our disk that contains

```
57213 Male 34000
47229 Female 37000
57323 Male 34000
57401 Male 34500
57802 Female 37000
57805 Female 34000
57824 Male 32500
```

The `infile` command can read these data and create the mapping automatically.

```
. label list sexlabel
value label sexlabel not found
r(111);

. infile empno sex:sexlabel salary using employee, automatic
(7 observations read)
```

Our first command, `label list sexlabel`, is only to prove that we had not previously defined the value label `sexlabel`. Stata `infile`d the data without complaint. We now have

```
. list
```

	empno	sex	salary
1.	57213	Male	34000
2.	47229	Female	37000
3.	57323	Male	34000
4.	57401	Male	34500
5.	57802	Female	37000
6.	57805	Female	34000
7.	57824	Male	32500

Of course, `sex` is just another numeric variable; it does not actually take on the values `Male` and `Female`—it takes on numeric codes that have been automatically mapped to `Male` and `Female`. We can find out what that mapping is by using the `label list` command.

```
. label list sexlabel
sexlabel:
           1 Male
           2 Female
```

We discover that Stata attached the codes 1 to `Male` and 2 to `Female`. Anytime we want to see what our data really look like, ignoring the value labels, we can use the `nolabel` option.

```
. list, nolabel
```

	empno	sex	salary
1.	57213	1	34000
2.	47229	2	37000
3.	57323	1	34000
4.	57401	1	34500
5.	57802	2	37000
6.	57805	2	34000
7.	57824	1	32500

❑

12.6.4 Labels in other languages

A dataset can contain labels—data, variable, and value—in up to 100 languages. To discover the languages available for the dataset in memory, type `label language`. You will see

```
. label language
Language for variable and value labels
    In this dataset, value and variable labels have been defined in only one
    language:  default
    To create new language:          . label language <name>, new
    To rename current language:      . label language <name>, rename
```

or something like the following:

```
. label language
Language for variable and value labels
    Available languages:
            de
            en
            sp
    Currently set is:                . label language sp
    To select different language:    . label language <name>
    To create new language:          . label language <name>, new
    To rename current language:      . label language <name>, rename
```

Right now, the example dataset is set with `sp` (Spanish) labels:

```
. describe
Contains data
  obs:            74                          Automóviles, 1978
 vars:            12                          3 Oct 2016 13:53
 size:         3,478
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
make            str18   %-18s                 Marca y modelo
price           int     %8.0gc                Precio
mpg             int     %8.0g                 Consumo de combustible
rep78           int     %8.0g                 Historia de reparaciones
headroom        float   %6.1f                 Cabeza adelante
trunk           int     %8.0g                 Volumen del maletero
weight          int     %8.0gc                Peso
length          int     %8.0g                 Longitud
turn            int     %8.0g                 Radio de giro
displacement    int     %8.0g                 Cilindrada
gear_ratio      float   %6.2f                 Relación de cambio
foreign         byte    %8.0g                 Extranjero
-------------------------------------------------------------------------------
Sorted by: foreign
```

To create labels in more than one language, you set the new language and then define the labels in the standard way; see [D] **label language**.

12.7 Notes attached to data

A dataset may contain notes, which are nothing more than little bits of text that you define and review with the `notes` command. Typing `note`, a colon, and the text defines a note:

```
. note:  Send copy to Bob once verified.
```

You can later display whatever notes you have previously defined by typing `notes`:

```
. notes
_dta:
  1.  Send copy to Bob once verified.
```

Notes are saved with the data, so once you save your dataset, you can replay this note in the future, too.

You can add more notes:

```
. note: Mary wants a copy, too.
. notes
_dta:
  1.  Send copy to Bob once verified.
  2.  Mary wants a copy, too.
```

The notes you have added so far are attached to the data generically, which is why Stata prefixes them with `_dta` when it lists them. You can attach notes to variables:

```
. note state: verify values for Nevada.
. note state: what about the two missing values?
. notes
_dta:
  1.  Send copy to Bob once verified.
  2.  Mary wants a copy, too.
state:
  1.  verify values for Nevada.
  2.  what about the two missing values?
```

When you `describe` your data, you can see whether notes are attached to the dataset or to any of the variables:

```
. describe
Contains data from states.dta
  obs:            50                          1980 state data
 vars:             4
 size:         1,200                          (_dta has notes)
-------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------------------
state           str8    %9s                 *
median_age      float   %9.0g                 Median Age
marriage_rate   long    %12.0g                Marriages per 100,000
divorce_rate    long    %12.0g                Divorces per 100,000
                                             * indicated variables have notes
-------------------------------------------------------------------------------
Sorted by:
     Note: Dataset has changed since last saved.
```

See [D] **notes** for a complete description of this feature.

12.8 Characteristics

Characteristics are an arcane feature of Stata but are of great use to Stata programmers. In fact, the `notes` command described above was implemented using characteristics.

The dataset itself and each variable within the dataset have associated with them a set of characteristics. Characteristics are named and referred to as *varname*`[`*charname*`]`, where *varname* is the name of a variable or `_dta`. The characteristics contain text and are stored with the data in the Stata-format `.dta` dataset, so they are recalled whenever the data are loaded.

How are characteristics used? The [XT] **xt** commands need to know the name of the panel variable, and some of these commands also need to know the name of the time variable. `xtset` is used to specify the panel variable and optionally the time variable. Users need `xtset` their data only once. Stata then remembers this information, even from a different Stata session. Stata does this with characteristics: `_dta[iis]` contains the name of the panel variable and `_dta[tis]` contains the name of the time variable. When an `xt` command is issued, the command checks these characteristics to obtain the panel and time variables' names. If this information is not found, then the data have not previously been `xtset` and an error message is issued. This use of characteristics is hidden from the user—no mention is made of how the commands remember the identity of the panel variable and the time variable.

As a Stata user, you need understand only how to set and clear a characteristic for the few commands that explicitly reveal their use of characteristics. You set a variable *varname*'s characteristic *charname* to *x* by typing

```
. char varname[charname] x
```

You set the data's characteristic *charname* to be *x* by typing

```
. char _dta[charname] x
```

You clear a characteristic by typing

```
. char varname[charname]
```

where *varname* is either a variable name or `_dta`. You can clear a characteristic, even if it has never been set.

The most important feature of characteristics is that Stata remembers them from one session to the next; they are saved with the data.

❑ Technical note

Programmers will want to know more. A technical description is found in [P] **char**, but for an overview, you may refer to *varname*'s *charname* characteristic by embedding its name in single quotes and typing `'`*varname*`[`*charname*`]'`; see [U] **18.3.13 Referring to characteristics**.

You can fetch the names of all characteristics associated with *varname* by typing

```
. local macname : char varname[]
```

The maximum length of the contents of a characteristic is 67,784 bytes for Stata/IC, Stata/SE, and Stata/MP. The association of names with characteristics is by convention. If you, as a programmer, wish to create new characteristics for use in your ado-files, do so, but include at least one capital letter in the characteristic name. The current convention reserves all lowercase names for "official" Stata.

❑

12.9 Data Editor and Variables Manager

We have spent most of this chapter writing about data management performed from Stata's command line. However, Stata provides two powerful features in its interface to help you examine and manage your data: the Data Editor and the Variables Manager.

The Data Editor is a spreadsheet-style data editor that allows you to enter new data, edit existing data, safely browse your data in a read-only mode, and perform almost any data-management task you desire in a reproducible manner using a graphical interface. To open the Data Editor, select **Data > Data Editor > Data Editor (Edit)** or **Data > Data Editor > Data Editor (Browse)**. See [GS] **6 Using the Data Editor** (GSM, GSU, or GSW) for a tutorial discussion of the Data Editor. See [D] **edit** for technical details.

The Variables Manager is a tool that lists and allows you to manage all the properties of the variables in your data. Variable properties include the name, label, storage type, format, value label, and notes. The Variables Manager allows you to sort and filter your variables; this is something that you will find to be very useful if you work with datasets containing many variables. The Variables Manager also can be used to create varlists for the Command window. To open the Variables Manager, select **Data > Variables Manager**. See [GS] **7 Using the Variables Manager** (GSM, GSU, or GSW) for a tutorial discussion of the Variables Manager.

Both the Data Editor and Variables Manager submit commands to Stata to perform any changes that you request. This lets you see a log of what changes were made, and it also allows you to work interactively while still building a list of commands that you can execute later to reproduce your analysis.

12.10 References

Cox, N. J. 2006. Stata tip 33: Sweet sixteen: Hexadecimal formats and precision problems. *Stata Journal* 6: 282–283.

——. 2010a. Stata tip 84: Summing missings. *Stata Journal* 10: 157–159.

——. 2010b. Stata tip 85: Looping over nonintegers. *Stata Journal* 10: 160–163.

Long, J. S. 2009. *The Workflow of Data Analysis Using Stata*. College Station, TX: Stata Press.

Longest, K. C. 2014. *Using Stata for Quantitative Analysis*. 2nd ed. Thousand Oaks, CA: Sage.

Mitchell, M. N. 2010. *Data Management Using Stata: A Practical Handbook*. College Station, TX: Stata Press.

Rising, W. R. 2010. Stata tip 86: The missing() function. *Stata Journal* 10: 303–304.