infix (fixed format) — Import text data in fixed format

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*$_2$ 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
```

Same as above, but read `v1` as a string variable

```
infix str v1 1-6 v2 7 using mydata
```

Same 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
```

Same as above, but for `mydata.txt`

```
infix 2 lines 1: v1 1-6 2: v2 1 using mydata.txt
```

Same as above, but with data beginning on line 3

```
infix 3 firstlineoffile 2 lines 1: v1 1-6 2: v2 1 using mydata.txt
```

Same 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

`infix using` *dfilename* [*if*] [*in*] [`, using(`*filename*$_2$`) clear`]

`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

begin dictionary file

```
infix dictionary [using filename] {
        * comments preceded by asterisk may appear freely
        specifications
}
(your data might appear here)
```

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.

specifications is

`firstlineoffile`
`lines`
#`:`
`/`
[`byte` | `int` | `float` | `long` | `double` | `str`] *varlist* [#:]#[-#]

Options

Main

`using(`*filename*$_2$`)` 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*$_2$`)` 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.

`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, `infix` will refuse to read new data if data are already in memory. `clear` allows `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 to 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 `byte`s.

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

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:

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

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
```

	acc_rate	spdlimit	acc_pts
1.	4.58	55	.46
2.	2.86	60	4.4
3.	1.61	.	2.2
4.	3.02	60	4.7

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

begin emp2.dct

```
infix dictionary using emp2.raw {
    2 lines
    1:
        id        1-6
        str name  7-36
    2:
        age       1-2
        str sex     4
}
```

end emp2.dct

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

begin mydata1.dct

```
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
}
```

end mydata1.dct

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
```

	id	income	educ	sex	age	rcode	q1	q2	q3	q4	q5
1.	1024	25000	HS	Male	28	1	1	9	5	0	3
2.	1025	27000	C	Female	24	0	2	2	1	1	3

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,

begin mydata3.dct

```
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 `infix` does not really jump to record 2, then record 3, then record 1 again, etc. `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.

◁

Reading subsets of observations

If you wanted to read only the information about males from some raw data file, you might type

```
. infix id 1-6  str name 7-36  age 38-39  str sex 41  using employee.raw
> if sex=="M"
```

If your specification was instead recorded in a dictionary, you could type

```
. infix using employee.dct if sex=="M"
```

In another dataset, if you wanted to read just the first 100 observations, you could type

```
. infix 2 lines  1: id 1-6  str name 7-36  2: age 1-2  str sex 4  using emp2.raw
> in 1/100
```

or if the specification was instead recorded in a dictionary and you wanted observations 101–573, you could type

```
. infix using emp2.dct in 101/573
```

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