18 Programming Stata

Contents

Stata programming is an advanced topic. Some Stata users live productive lives without ever programming Stata. After all, you do not need to know how to program Stata to import data, create new variables, and fit models. On the other hand, programming Stata is not difficult—at least if the problem is not difficult—and Stata's programmability is one of its best features. The real power of Stata is not revealed until you program it.

Stata has two programming languages. One, known informally as "ado", is the focus of this chapter. It is based on Stata's commands, and you can write scripts and programs to automate reproducible analyses and to add new features to Stata.

The other language, Mata, is a byte-compiled language with syntax similar to C/C++, but with extensive matrix capabilities. The two languages can interact with each other. You can call Mata functions from ado-programs, and you can call ado-programs from Mata functions. You can learn all about Mata in the *Mata Reference Manual*.

Stata also has a Project Manager to help you manage large collections of Stata scripts, programs, and other files. See [P] **Project Manager**.

If you are uncertain whether to read this chapter, we recommend that you start reading and then bail out when it gets too arcane for you. You will learn things about Stata that you may find useful even if you never write a Stata program.

If you want even more, we offer courses over the Internet on Stata programming; see **[U] 3.6.2 NetCourses**. Baum (2016) provides a wealth of practical knowledge related to Stata programming.

18.1 Description

When you type a command that Stata does not recognize, Stata first looks in its memory for a program of that name. If Stata finds it, Stata executes the program.

There is no Stata command named `hello`,

```
. hello
command hello is unrecognized
r(199);
```

but there could be if you defined a program named `hello`, and after that, the following might happen when you typed `hello`:

```
. hello
hi there
. _
```

This would happen if, beforehand, you had typed

```
. program hello
  1. display "hi there"
  2. end
. _
```

That is how programming works in Stata. A program is defined by

```
program progname
        Stata commands
end
```

and it is executed by typing *progname* at Stata's dot prompt.

18.2 Relationship between a program and a do-file

Stata treats programs the same way it treats do-files. Below we will discuss passing arguments, consuming results from Stata commands, and other topics, but everything we say applies equally to do-files and programs.

Programs and do-files differ in the following ways:

1. You invoke a do-file by typing `do` *filename*. You invoke a program by simply typing the program's name.
2. Programs must be defined (loaded) before they are used, whereas all that is required to run a do-file is that the file exist. There are ways to make programs load automatically, however, so this difference is of little importance.
3. When you type `do` *filename*, Stata displays the commands it is executing and the results. When you type *progname*, Stata shows only the results, not the display of the underlying commands. This is an important difference in outlook: in a do-file, how it does something is as important as what it does. In a program, the how is no longer important. You might think of a program as a new feature of Stata.

Let's now mention some of the similarities:

1. Arguments are passed to programs and do-files in the same way.
2. Programs and do-files both contain Stata commands. Any Stata command you put in a do-file can be put in a program.
3. Programs may call other programs. Do-files may call other do-files. Programs may call do-files (this rarely happens), and do-files may call programs (this often happens). Stata allows programs (and do-files) to be nested up to 64 deep.

Now here is the interesting thing: programs are typically defined in do-files (or in a variant of do-files called ado-files; we will get to that later).

You can define a program interactively, and that is useful for pedagogical purposes, but in real applications, you will compose your program in a text editor and store its definition in a do-file.

You have already seen your first program:

```
program hello
        display "hi there"
end
```

You could type those commands interactively, but if the body of the program were more complicated, that would be inconvenient. So instead, suppose that you typed the commands into a do-file:

begin hello.do

```
program hello
        display "hi there"
end
```

end hello.do

Now returning to Stata, you type

```
. do hello

. program hello
  1.         display "hi there"
  2. end

.
end of do-file
```

Do you see that typing `do hello` did nothing but load the program? Typing `do hello` is the same as typing out the program's definition because that is all the do-file contains. The do-file was executed, but the statements in the do-file only defined the program `hello`; they did not execute it. Now that the program is loaded, we can execute it interactively:

```
. hello
hi there
```

So, that is one way you could use do-files and programs together. If you wanted to create new commands for interactive use, you could

1. Write the command as a `program` ... `end` in a do-file.
2. `do` the do-file before you use the new command.
3. Use the new command during the rest of the session.

There are more convenient ways to do this that would automatically load the do-file, but put that aside. The above method would work.

Another way we could use do-files and programs together is to put the definition of the program and its execution together into a do-file:

begin hello.do

```
program hello
        display "hi there"
end
hello
```

end hello.do

Here is what would happen if we executed this do-file:

```
. do hello

. program hello
  1.         display "hi there"
  2. end

. hello
hi there

.
end of do-file
```

Do-files and programs are often used in such combinations. Why? Say that program `hello` is long and complicated and you have a problem where you need to do it twice. That would be a good reason to write a program. Moreover, you may wish to carry forth this procedure as a step of your analysis and, being cautious, do not want to perform this analysis interactively. You never intended program `hello` to be used interactively—it was just something you needed in the midst of a do-file—so you defined the program and used it there.

Anyway, there are many variations on this theme, but few people actually sit in front of Stata and interactively type `program` and then compose a program. They instead do that in front of their text editor. They compose the program in a do-file and then execute the do-file.

There is one other (minor) thing to know: once a program is defined, Stata does not allow you to redefine it:

```
. program hello
program hello already defined
r(110);
```

Thus, in our most recent do-file that defines and executes `hello`, we could not rerun it in the same Stata session:

```
. do hello
. program hello
program hello already defined
r(110);
end of do-file
r(110);
```

That problem is solved by typing `program drop hello` before redefining it. We could do that interactively, or we could modify our do-file:

begin hello.do

```
program drop hello
program hello
        display "hi there"
end
hello
```

end hello.do

There is a problem with this solution. We can now rerun our do-file, but the first time we tried to run it in a Stata session, it would fail:

```
. do hello
. program drop hello
hello not found
r(111);
end of do-file
r(111);
```

The way around this conundrum is to modify the do-file:

begin hello.do

```
capture program drop hello
program hello
        display "hi there"
end
hello
```

end hello.do

`capture` in front of a command makes Stata indifferent to whether the command works; see [P] **capture**. In real do-files containing programs, you will often see `capture program drop` before the program's definition.

To learn about the `program` command itself, see [P] **program**. It manipulates programs. `program` can define programs, drop programs, and show you a directory of programs that you have defined.

A program can contain any Stata command, but certain Stata commands are of special interest to program writers; see the *Programming* heading in the subject table of contents in the *Stata Index*.

18.3 Macros

Before we can begin programming, we must discuss macros, which are the variables of Stata programs.

A *macro* is a string of characters, called the *macroname*, that stands for another string of characters, called the *macro contents*.

Macros can be local or global. We will start with local macros because they are the most commonly used, but nothing really distinguishes one from the other at this stage.

18.3.1 Local macros

Local macro names can be up to 31 (not 32) characters long.

One sets the contents of a local macro with the `local` command. In fact, we can do this interactively. We will begin by experimenting with macros in this way to learn about them. If we type

```
. local shortcut "myvar thisvar thatvar"
```

then `‘shortcut’` is a synonym for "`myvar thisvar thatvar`". Note the single quotes around `shortcut`. We said that sentence exactly the way we meant to because

if you type	`‘shortcut’`,
i.e.,	left-single-quote `shortcut` right-single-quote,
Stata hears	`myvar thisvar thatvar`.

To access the contents of the macro, we use a left single quote (located at the upper left on most keyboards), the macro name, and a right single quote (located under the " on the right side of most keyboards).

The single quotes bracketing the macroname `shortcut` are called the macro-substitution characters. `shortcut` means `shortcut`. `‘shortcut’` means `myvar thisvar thatvar`.

So, if you typed

```
. list ‘shortcut’
```

the effect would be exactly as if you typed

```
. list myvar thisvar thatvar
```

Macros can be used anywhere in Stata. For instance, if we also defined

```
. local cmd "list"
```

we could type

```
. ‘cmd’ ‘shortcut’
```

to mean `list myvar thisvar thatvar`.

For another example, consider the definitions

```
. local prefix "my"
. local suffix "var"
```

Then

```
. ‘cmd’ ‘prefix’‘suffix’
```

would mean `list myvar`.

One other important note is on the way we use left and right single quotes within Stata, which you will especially deal with when working with macros (see [U] **18.3 Macros**). Single quotes (and double quotes, for that matter) may look different on your keyboard, your monitor, and our printed documentation, making it difficult to determine which key to press on your keyboard to replicate what we have shown you.

For the left single quote, we use the grave accent, which occupies a key by itself on most computer keyboards. On U.S. keyboards, the grave accent is located at the top left, next to the numeral 1. On some non-U.S. keyboards, the grave accent is produced by a dead key. For example, pressing the grave accent dead key followed by the letter a would produce à; to get the grave accent by itself, you would press the grave accent dead key followed by a space. This accent mark appears in our printed documentation as `‘`.

For the right single quote, we use the standard single quote, or apostrophe. On U.S. keyboards, the single quote is located on the same key as the double quote, on the right side of the keyboard next to the *Enter* key.

18.3.2 Global macros

Let’s put aside why Stata has two kinds of macros—local and global—and focus right now on how global macros work.

Global macros can have names that are up to 32 (not 31) characters long. You set the contents of a global macro by using the `global` rather than the `local` command:

```
. global shortcut "alpha beta"
```

You obtain the contents of a global macro by prefixing its name with a dollar sign: `$shortcut` is equivalent to “`alpha beta`”.

In the previous section, we defined a local macro named `shortcut`, which is a different macro. `‘shortcut’` is still “`myvar thisvar thatvar`”.

Local and global macros may have the same names, but even if they do, they are unrelated and are still distinguishable.

Global macros are just like local macros except that you set their contents with `global` rather than `local`, and you substitute their contents by prefixing them with a `$` rather than enclosing them in `‘’`.

18.3.3 The difference between local and global macros

The difference between local and global macros is that local macros are private and global macros are public.

Say that you have written a program

```
program myprog
        code using local macro alpha
end
```

The local macro `alpha` in `myprog` is private in that no other program can modify or even look at `alpha`’s contents. To make this point absolutely clear, assume that your program looks like this:

```
program myprog
        code using local macro alpha
        mysub
        more code using local macro alpha
end
program mysub
        code using local macro alpha
end
```

myprog calls mysub, and both programs use a local macro named alpha. Even so, the local macros in each program are different. mysub's alpha macro may contain one thing, but that has nothing to do with what myprog's alpha macro contains. Even when mysub begins execution, its alpha macro is different from myprog's. It is not that mysub's inherits myprog's alpha macro contents but is then free to change it. It is that myprog's alpha and mysub's alpha are entirely different things.

When you write a program using local macros, you need not worry that some other program has been written using local macros with the same names. Local macros are just that: local to your program.

Global macros, on the other hand, are available to all programs. If both myprog and mysub use the global macro beta, they are using the same macro. Whatever the contents of $beta are when mysub is invoked, those are the contents when mysub begins execution, and, whatever the contents of $beta are when mysub completes, those are the contents when myprog regains control.

18.3.4 Macros and expressions

From now on, we are going to use local and global macros according to whichever is convenient; whatever is said about one applies to the other.

Consider the definitions

```
. local one 2+2
. local two = 2+2
```

(which we could just as well have illustrated using the global command). In any case, note the equal sign in the second macro definition and the lack of the equal sign in the first. Formally, the first should be

```
. local one "2+2"
```

but Stata does not mind if we omit the double quotes in the local (global) statement.

local one 2+2 (with or without double quotes) copies the string 2+2 into the macro named one.

local two = 2+2 evaluates the expression 2+2, producing 4, and stores 4 in the macro named two.

That is, you type

local *macname contents*

if you want to copy *contents* to *macname*, and you type

local *macname* = *expression*

if you want to evaluate *expression* and store the result in *macname*.

In the second form, *expression* can be numeric or string. 2+2 is a numeric expression. As an example of a string expression,

```
. local res = substr("this",1,2) + "at"
```

stores that in res.

Because the expression can be either numeric or string, what is the difference between the following statements?

```
. local a "example"
. local b = "example"
```

Both statements store `example` in their respective macros. The first does so by a simple copy operation, whereas the second evaluates the expression `"example"`, which is a string expression because of the double quotes that, here, evaluates to itself. You could put a more complicated expression to be evaluated on the right-hand side of the second syntax.

There are some other issues of using macros and expressions that look a little strange to programmers coming from other languages, at least the first time they see them. Say that the macro `‘i’` contains `5`. How would you increment `i` so that it contains $5 + 1 = 6$? The answer is

```
local i = ‘i’ + 1
```

Do you see why the single quotes are on the right but not the left? Remember, `‘i’` refers to the contents of the local macro named `i`, which, we just said, is `5`. Thus, after expansion, the line reads

```
local i = 5 + 1
```

which is the desired result.

There is a another way to increment local macros that will be more familiar to some programmers, especially C programmers:

```
local ++i
```

As C programmers would expect, `local ++i` is more efficient (executes more quickly) than `local i = i+1`, but in terms of outcome, it is equivalent. You can decrement a local macro by using

```
local --i
```

`local --i` is equivalent to `local i = i-1` but executes more quickly. Finally,

```
local i++
```

will *not* increment the local macro `i` but instead redefines the local macro `i` to contain `++`. There is, however, a context in which `i++` (and `i--`) do work as expected; see [U] **18.3.7 Macro increment and decrement functions**.

18.3.5 Double quotes

Consider another local macro, `‘answ’`, which might contain `yes` or `no`. In a program that was supposed to do something different on the basis of `answ`’s content, you might code

```
if "‘answ’" == "yes" {
        ...
}
else {
        ...
}
```

Note the odd-looking `"‘answ’"`, and now think about the line after substitution. The line reads either

```
if "yes" == "yes" {
```

or

```
if "no" == "yes" {
```

either of which is the desired result. Had we omitted the double quotes, the line would have read

```
if no == "yes" {
```

(assuming `‘answ’` contains `no`), and that is not at all the desired result. As the line reads now, `no` would not be a string but would be interpreted as a variable in the data.

The key to all of this is to think of the line after substitution.

Double quotes are used to enclose strings: `"yes"`, `"no"`, `"my dir\my file"`, `"‘answ’"` (meaning that the contents of local macro `answ`, treated as a string), and so on. Double quotes are used with macros,

```
local a "example"
if "‘answ’" == "yes" {
        ...
}
```

and double quotes are used by many Stata commands:

```
. regress lnwage age ed if sex=="female"
. generate outa = outcome if drug=="A"
. use "person file"
```

Do not omit the double quotes just because you are using a “quoted” macro:

```
. regress lnwage age ed if sex=="‘x’"
. generate outa = outcome if drug=="‘firstdrug’"
. use "‘filename’"
```

Stata has two sets of double-quote characters, of which `""` is one. The other is `‘"’`. They both work the same way:

```
. regress lnwage age ed if sex==‘"female"’
. generate outa = outcome if drug==‘"A"’
. use ‘"person file"’
```

No rational user would use `‘"’` (called compound double quotes) instead of `""` (called simple double quotes), but smart programmers do use them:

```
local a ‘"example"’
if ‘"‘answ’"’ == ‘"yes"’ {
        ...
}
```

Why is `‘"example"’` better than `"example"`, `‘"‘answ’"’` better than `"‘answ’"`, and `‘"yes"’` better than `"yes"`? The answer is that only `‘"‘answ’"’` is better than `"‘answ’"`; `‘"example"’` and `‘"yes"’` are no better—and no worse—than `"example"` and `"yes"`.

`‘"‘answ’"’` is better than `"‘answ’"` because the macro `answ` might itself contain (simple or compound) double quotes. The really great thing about compound double quotes is that they nest. Say that `‘answ’` contained the string “`I "think" so`”. Then,

Stata would find	`if "‘answ’"=="yes"`
confusing because it would expand to	`if "I "think" so"=="yes"`
Stata would not find	`if ‘"‘answ’"’==‘"yes"’`
confusing because it would expand to	`if ‘"I "think" so"’==‘"yes"’`

Open and close double quote in the simple form look the same; open quote is " and so is close quote. Open and close double quote in the compound form are distinguishable; open quote is ‘" and close quote is "’, and so Stata can pair the close with the corresponding open double quote. ‘"I "think" so"’ is easy for Stata to understand, whereas "I "think" so" is a hopeless mishmash. (If you disagree, consider what "A"B"C" might mean. Is it the quoted string A"B"C, or is it quoted string A, followed by B, followed by quoted string C?)

Because Stata can distinguish open from close quotes, even nested compound double quotes are understandable: ‘"I ‘"think"’ so"’. (What does "A"B"C" mean? Either it means ‘"A‘"B"’C"’ or it means ‘"A"’B‘"C"’.)

Yes, compound double quotes make you think that your vision is stuttering, especially when combined with the macro substitution ‘’ characters. That is why we rarely use them, even when writing programs. You do not have to use exclusively one or the other style of quotes. It is perfectly acceptable to code

```
local a "example"
if ‘"‘answ’"’ == "yes" {
        ...
}
```

using compound double quotes where it might be necessary (‘"‘answ’"’) and using simple double quotes in other places (such as "yes"). It is also acceptable to use simple double quotes around macros (for example, "‘answ’") if you are certain that the macros themselves do not contain double quotes or (more likely) if you do not care what happens if they do.

Sometimes careful programmers should use compound double quotes. Later you will learn that Stata's syntax command interprets standard Stata syntax and so makes it easy to write programs that understand things like

```
. myprog mpg weight if strpos(make,"VW")!=0
```

syntax works—we are getting ahead of ourselves—by placing the if *exp* typed by the user in the local macro if. Thus ‘if’ will contain "if strpos(make,"VW")!=0" here. Now say that you are at a point in your program where you want to know whether the user specified an if *exp*. It would be natural to code

```
if ‘"‘if’"’ != "" {
        // the if exp was specified
        ...
}
else {
        // it was not
        ...
}
```

We used compound double quotes around the macro ‘if’. The local macro ‘if’ might contain double quotes, so we placed compound double quotes around it.

18.3.6 Macro functions

In addition to allowing =*exp*, local and global provide *macro functions*. The use of a macro function is denoted by a colon (:) following the macro name, as in

```
local       lbl : variable label myvar
local filenames : dir "." files "*.dta"
local        xi : word ‘i’ of ‘list’
```

Some macro functions access a piece of information. In the first example, the variable label associated with variable `myvar` will be stored in macro `lbl`. Other macro functions perform operations to gather the information. In the second example, macro `filenames` will contain the names of all the `.dta` datasets in the current directory. Still other macro functions perform an operation on their arguments and return the result. In the third example, `xi` will contain the `‘i’`th word (element) of `‘list’`. See [P] **macro** for a list of the macro functions.

Another useful source of information is `c()`, documented in [P] **creturn**:

```
local  today "‘c(current_date)’"
local curdir "‘c(pwd)’"
local   newn = c(N)+1
```

`c()` refers to a prerecorded list of values, which may be used directly in expressions or which may be quoted and the result substituted anywhere. `c(current_date)` returns today's date in the form "*dd MON yyyy*". Thus the first example stores in macro `today` that date. `c(pwd)` returns the current directory, such as `C:\data\proj`. Thus the second example stores in macro `curdir` the current directory. `c(N)` returns the number of observations of the data in memory. Thus the third example stores in macro `newn` that number, plus one.

Note the use of quotes with `c()`. We could just as well have coded the first two examples as

```
local  today = c(current_date)
local curdir = c(pwd)
```

`c()` is a Stata function in the same sense that `sqrt()` is a Stata function. Thus we can use `c()` directly in expressions. It is a special property of macro expansion, however, that you may use the `c()` function inside macro-expansion quotes. The same is not true of `sqrt()`.

In any case, whenever you need a piece of information, whether it be about the dataset or about the environment, look in [P] **macro** and [P] **creturn**. It is likely to be in one place or the other, and sometimes, it is in both. You can obtain the current directory by using

```
local curdir = c(pwd)
```

or by using

```
local curdir : pwd
```

When information is in both, it does not matter which source you use.

18.3.7 Macro increment and decrement functions

We mentioned incrementing macros in [U] **18.3.4 Macros and expressions**. The construct

```
command that makes reference to ‘i’
local ++i
```

occurs so commonly in Stata programs that it is convenient (and faster when executed) to collapse both lines of code into one and to increment (or decrement) `i` at the same time that it is referred to. Stata allows this:

```
while (‘++i’ < 1000) {
        ...
}
while (‘i++’ < 1000) {
        ...
}
while (‘--i’ > 0) {
        ...
}
while (‘i--’ > 0) {
        ...
}
```

Above we have chosen to illustrate this by using Stata's `while` command, but `++` and `--` can be used anyplace in any context, just so long as it is enclosed in macro-substitution quotes.

When the `++` or `--` appears before the name, the macro is first incremented or decremented, and then the result is substituted.

When the `++` or `--` appears after the name, the current value of the macro is substituted and then the macro is incremented or decremented.

❑ Technical note

Do not use the inline `++` or `--` operators when a part of the line might not be executed. Consider

```
if (‘i’==0) local j = ‘k++’
```

versus

```
if (‘i’==0) {
        local j = ‘k++’
}
```

The first will not do what you expect because macros are expanded before the line is interpreted. Thus the first will result in `k` always being incremented, whereas the second increments `k` only when `‘i’==0`.

❑

18.3.8 Macro expressions

Typing

command that makes reference to `‘=`*exp*`’`

is equivalent to

`local` *macroname* `=` *exp*
command that makes reference to `‘`*macroname*`’`

although the former runs faster and is easier to type. When you use `‘=`*exp*`’` within some larger command, *exp* is evaluated by Stata's expression evaluator, and the results are inserted as a literal string into the larger command. Then the command is executed. For example,

```
summarize u4
summarize u‘=2+2’
summarize u‘=4*(cos(0)==1)’
```

all do the same thing. *exp* can be any valid Stata expression and thus may include references to variables, matrices, scalars, or even other macros. In the last case, just remember to enclose the submacros in quotes:

```
replace `var' = `group'[`=`j'+1']
```

Also, typing

command that makes reference to `` `:`*macro function*`' ``

is equivalent to

`local` *macroname* `:` *macro function*
command that makes reference to `` ``*macroname*'' ``

Thus one might code

```
format y `:format x'
```

to assign to variable `y` the same format as the variable `x`.

❑ Technical note

There is another macro expansion operator, `.` (called dot), which is used in conjunction with Stata's class system; see [P] **class** for more information.

There is also a macro expansion function, `macval()`, which is for use when expanding a macro—`` `macval(`*name*`)' ``—which confines the macro expansion to the first level of *name*, thereby suppressing the expansion of any embedded references to macros within *name*. Only a few Stata users have or will ever need this, but, if you suspect you are one of them, see [P] **macro** and then see [P] **file** for an example.

❑

18.3.9 Advanced local macro manipulation

This section is really an aside to help test your understanding of macro substitution. The tricky examples illustrated below sometimes occur in real programs.

1. Say that you have macros `x1`, `x2`, `x3`, and so on. Obviously, `` `x1' `` refers to the contents of `x1`, `` `x2' `` to the contents of `x2`, etc. What does `` `x`i'' `` refer to? Suppose that `` `i' `` contains `6`.

   The rule is to expand the inside first:

`` `x`i'' ``	expands to	`` `x6' ``
`` `x6' ``	expands to	the contents of local macro `x6`

   So, there you have a vector of macros.

2. We have already shown adjoining expansions: `` `alpha'`beta' `` expands to `myvar` if `` `alpha' `` contains `my` and `` `beta' `` contains `var`. What does `` `alpha'`gamma'`beta' `` expand to when `gamma` is undefined?

   Stata does not mind if you refer to a nonexistent macro. A nonexistent macro is treated as a macro with no contents. If local macro `gamma` does not exist, then

`` `gamma' ``	expands to	nothing

   It is not an error. Thus `` `alpha'`gamma'`beta' `` expands to `myvar`.

3. You clear a local macro by setting its contents to nothing:

   `local` *macname*
   or `local` *macname* `""`
   or `local` *macname* `= ""`

18.3.10 Advanced global macro manipulation

Global macros are rarely used, and when they are used, it is typically for communication between programs. You should never use a global macro where a local macro would suffice.

1. Constructions like `$x$i` are expanded sequentially. If `$x` contained `this` and `$i 6`, then `$x$i` expands to `this6`. If `$x` was undefined, then `$x$i` is just `6` because undefined global macros, like undefined local macros, are treated as containing nothing.
2. You can nest macro expansion by including braces, so if `$i` contains `6`, `${x$i}` expands to `${x6}`, which expands to the contents of `$x6` (which would be nothing if `$x6` is undefined).
3. You can mix global and local macros. Assume that local macro `j` contains `7`. Then, `${x‘j’}` expands to the contents of `$x7`.
4. You also use braces to force the contents of global macros to run up against the succeeding text. For instance, assume that the macro `drive` contains “`b:`”. If `drive` were a local macro, you could type

   ```
   ‘drive’myfile.dta
   ```

   to obtain `b:myfile.dta`. Because `drive` is a global macro, however, you must type

   ```
   ${drive}myfile.dta
   ```

   You could not type

   ```
   $drive myfile.dta
   ```

   because that would expand to `b: myfile.dta`. You could not type

   ```
   $drivemyfile.dta
   ```

   because that would expand to `.dta`.
5. Because Stata uses `$` to mark global-macro expansion, printing a real `$` is sometimes tricky. To display the string $22.15 with the `display` command, you can type `display "\$22.15"`, although you can get away with `display "$22.15"` because Stata is rather smart. Stata would not be smart about `display "$this"` if you really wanted to display `$this` and not the contents of the macro `this`. You would have to type `display "\$this"`. Another alternative would be to use the SMCL code for a dollar sign when you wanted to display it: `display "{c S|}this"`; see [P] **smcl**.
6. Real dollar signs can also be placed into the contents of macros, thus postponing substitution. First, let’s understand what happens when we do not postpone substitution; consider the following definitions:

   ```
   global baseset "myvar thatvar"
   global bigset "$baseset thisvar"
   ```

   `$bigset` is equivalent to “`myvar thatvar thisvar`”. Now say that we redefine the macro `baseset`:

   ```
   global baseset "myvar thatvar othvar"
   ```

   The definition of `bigset` has not changed—it is still equivalent to “`myvar thatvar thisvar`”. It has not changed because `bigset` used the definition of `baseset` that was current at the time it was defined. `bigset` no longer knows that its contents are supposed to have any relation to `baseset`.

Instead, let's assume that we had defined `bigset` as

```
global bigset "\$baseset thisvar"
```

at the outset. Then `$bigset` is equivalent to "`$baseset thisvar`", which in turn is equivalent to "`myvar thatvar othvar thisvar`". Because `bigset` explicitly depends upon `baseset`, anytime we change the definition of `baseset`, we will automatically change the definition of `bigset` as well.

18.3.11 Constructing Windows filenames by using macros

Stata uses the `\` character to tell its parser not to expand macros.

Windows uses the `\` character as the directory path separator.

Mostly, there is no problem using a `\` in a filename. However, if you are writing a program that contains a Windows path in macro `path` and a filename in `fname`, do not assemble the final result as

```
`path'\`fname'
```

because Stata will interpret the `\` as an instruction to not expand `` `fname' ``. Instead, assemble the final result as

```
`path'/`fname'
```

Stata understands `/` as a directory separator on all platforms.

18.3.12 Accessing system values

Stata programs often need access to system parameters and settings, such as the value of π, the current date and time, or the current working directory.

System values are accessed via Stata's c-class values. The syntax works much the same as if you were referring to a local macro. For example, a reference to the c-class value for π, `` `c(pi)' ``, will expand to a literal string containing `3.141592653589793` and could be used to do

```
. display sqrt(2*`c(pi)')
2.5066283
```

You could also access the current time

```
. display "`c(current_time)'"
11:34:57
```

C-class values are designed to provide one all-encompassing way to access system parameters and settings, including system directories, system limits, string limits, memory settings, properties of the data currently in memory, output settings, efficiency settings, network settings, and debugging settings.

See [P] **creturn** for a detailed list of what is available. Typing

```
. creturn list
```

will give you the list of current settings.

18.3.13 Referring to characteristics

Characteristics—see [U] 12.8 Characteristics—are like macros associated with variables. They have names of the form *varname*[*charname*]—such as `mpg[comment]`—and you quote their names just as you do macro names to obtain their contents:

To substitute the value of *varname*[*charname*], type	‘*varname*[*charname*]’
For example,	‘`mpg[comment]`’

You set the contents using the `char` command:

`char` *varname*[*charname*] [["]*text*["]]

This is similar to the `local` and `global` commands, except that there is no `=`*exp* variation. You clear a characteristic by setting its contents to nothing just as you would with a macro:

Type	`char` *varname*[*charname*]
or	`char` *varname*[*charname*] `""`

What is unique about characteristics is that they are saved with the data, meaning that their contents survive from one session to the next, and they are associated with variables in the data, so if you ever drop a variable, the associated characteristics disappear, too. (Also, `_dta[`*charname*`]` is associated with the data but not with any variable in particular.)

All the standard rules apply: characteristics may be referred to by quotation in any context, and the characteristic's contents are substituted for the quoted characteristic name. As with macros, referring to a nonexistent characteristic is not an error; it merely substitutes to nothing.

18.4 Program arguments

When you invoke a program or do-file, what you type following the program or do-file name are the arguments. For instance, if you have a program called `xyz` and type

```
. xyz mpg weight
```

then `mpg` and `weight` are the program's arguments, `mpg` being the first argument and `weight` the second.

Program arguments are passed to programs via local macros:

Macro	Contents
‘`0`’	what the user typed exactly as the user typed it, odd spacing, double quotes, and all
‘`1`’	the first argument (first word of ‘`0`’)
‘`2`’	the second argument (second word of ‘`0`’)
‘`3`’	the third argument (third word of ‘`0`’)
...	...
‘`*`’	the arguments ‘`1`’, ‘`2`’, ‘`3`’, ..., listed one after the other and with one blank in between; similar to but different from ‘`0`’ because odd spacing and double quotes are removed

That is, what the user types is passed to you in three different ways:

1. It is passed in `‘0’` exactly as the user typed it, meaning quotes, odd spacing, and all.
2. It is passed in `‘1’`, `‘2’`, ... broken out into arguments on the basis of blanks (but with quotes used to force binding; we will get to that).
3. It is passed in `‘*’` as "`‘1’ ‘2’ ‘3’` ...", which is a crudely cleaned up version of `‘0’`.

You will probably not use all three forms in one program.

We recommend that you ignore `‘*’`, at least for receiving arguments; it is included so that old Stata programs will continue to work.

Operating directly with `‘0’` takes considerable programming sophistication, although Stata's `syntax` command makes interpreting `‘0’` according to standard Stata syntax easy. That will be covered in **[U] 18.4.4 Parsing standard Stata syntax** below.

The easiest way to receive arguments, however, is to deal with the positional macros `‘1’`, `‘2’`, ...

At the start of this section, we imagined an `xyz` program invoked by typing `xyz mpg weight`. Then `‘1’` would contain `mpg`, `‘2’` would contain `weight`, and `‘3’` would contain nothing.

Let's write a program to report the correlation between two variables. Of course, Stata already has a command that can do this—`correlate`—and, in fact, we will implement our program in terms of `correlate`. It is silly, but all we want to accomplish right now is to show how Stata passes arguments to a program.

Here is our program:

```
program xyz
        correlate ‘1’ ‘2’
end
```

Once the program is defined, we can try it:

```
. use https://www.stata-press.com/data/r18/auto
(1978 automobile data)
. xyz mpg weight
(obs=74)
             |      mpg   weight
-------------+------------------
         mpg |   1.0000
      weight |  -0.8072   1.0000
```

See how this works? We typed `xyz mpg weight`, which invoked our `xyz` program with `‘1’` being `mpg` and `‘2’` being `weight`. Our program gave the command `correlate ‘1’ ‘2’`, and that expanded to `correlate mpg weight`.

Stylistically, this is not a good example of the use of positional arguments, but realistically, there is nothing wrong with it. The stylistic problem is that if `xyz` is really to report the correlation between two variables, it ought to allow standard Stata syntax, and that is not a difficult thing to do. Realistically, the program works.

Positional arguments, however, play an important role, even for programmers who care about style. When we write a subroutine—a program to be called by another program and not intended for direct human use—we often pass information by using positional arguments.

Stata forms the positional arguments `‘1’`, `‘2’`, ... by taking what the user typed following the command (or do-file), parsing it on white space with double quotes used to force binding, and stripping the quotes. The arguments are formed on the basis of words, but double-quoted strings are kept together as one argument but with the quotes removed.

Let's create a program to illustrate these concepts. Although we would not normally define programs interactively, this program is short enough that we will:

```
. program listargs
  1.  display "The 1st argument you typed is:  `1'"
  2.  display "The 2nd argument you typed is:  `2'"
  3.  display "The 3rd argument you typed is:  `3'"
  4.  display "The 4th argument you typed is:  `4'"
  5.  end
```

The `display` command simply types the double-quoted string following it; see [P] **display**.

Let's try our program:

```
. listargs
The 1st argument you typed is:
The 2nd argument you typed is:
The 3rd argument you typed is:
The 4th argument you typed is:
```

We type `listargs`, and the result shows us what we already know—we typed nothing after the word `listargs`. There are no arguments. Let's try it again, this time adding `this is a test`:

```
. listargs this is a test
The 1st argument you typed is:  this
The 2nd argument you typed is:  is
The 3rd argument you typed is:  a
The 4th argument you typed is:  test
```

We learn that the first argument is '`this`', the second is '`is`', and so on. Blanks always separate arguments. You can, however, override this feature by placing double quotes around what you type:

```
. listargs "this is a test"
The 1st argument you typed is:  this is a test
The 2nd argument you typed is:
The 3rd argument you typed is:
The 4th argument you typed is:
```

This time we typed only one argument, '`this is a test`'. When we place double quotes around what we type, Stata interprets whatever we type inside the quotes to be one argument. Here `` `1' `` contains '`this is a test`' (the double quotes were removed).

We can use double quotes more than once:

```
. listargs "this is" "a test"
The 1st argument you typed is:  this is
The 2nd argument you typed is:  a test
The 3rd argument you typed is:
The 4th argument you typed is:
```

The first argument is '`this is`' and the second argument is '`a test`'.

18.4.1 Named positional arguments

Positional arguments can be named: in your code, you do not have to refer to `` `1' ``, `` `2' ``, `` `3' ``, ...; you can instead refer to more meaningful names, such as `n`, `a`, and `b`; `numb`, `alpha`, and `beta`; or whatever else you find convenient. You want to do this because programs coded in terms of `` `1' ``, `` `2' ``, ... are hard to read and therefore are more likely to contain errors.

You obtain better-named positional arguments by using the `args` command:

```
program progname
        args argnames
        ...
end
```

For instance, if your program received four positional arguments and you wanted to call them `varname`, `n`, `oldval`, and `newval`, you would code

```
program progname
        args varname n oldval newval
        ...
end
```

`varname`, `n`, `oldval`, and `newval` become new local macros, and `args` simply copies `‘1’`, `‘2’`, `‘3’`, and `‘4’` to them. It does not change `‘1’`, `‘2’`, `‘3’`, and `‘4’`—you can still refer to the numbered macros if you wish—and it does not verify that your program receives the right number of arguments. If our example above were invoked with just two arguments, `‘oldval’` and `‘newval’` would contain nothing. If it were invoked with five arguments, the fifth argument would still be out there, stored in local macro `‘5’`.

Let's make a command to create a dataset containing n observations on `x` ranging from a to b. Such a command would be useful, for instance, if we wanted to graph some complicated mathematical function and experiment with different ranges. It is convenient if we can type the range of `x` over which we wish to make the graph rather than concocting the range by hand. (In fact, Stata already has such a command—`range`—but it will be instructive to write our own.)

Before writing this program, we had better know how to proceed, so here is how you could create a dataset containing n observations with `x` ranging from a to b:

1. `clear`
   to clear whatever data are in memory.
2. `set obs` n
   to make a dataset of n observations on no variables; if n were 100, we would type `set obs 100`.
3. `gen x = (_n-1)/(`n`-1)*(`b`-`a`)+`a
   because the built-in variable `_n` is 1 in the first observation, 2 in the second, and so on; see **[U] 13.4 System variables (_variables)**.

So, the first version of our program might read

```
program rng                                  // arguments are n a b
        clear
        set obs ‘1’
        generate x = (_n-1)/(_N-1)*(‘3’-‘2’)+‘2’
end
```

The above is just a direct translation of what we just said. `‘1’` corresponds to n, `‘2’` corresponds to a, and `‘3’` corresponds to b. This program, however, would be far more understandable if we changed it to read

```
program rng
        args n a b
        clear
        set obs ‘n’
        generate x = (_n-1)/(_N-1)*(‘b’-‘a’)+‘a’
end
```

18.4.2 Incrementing through positional arguments

Some programs contain k arguments, where k varies, but it does not much matter because the same thing is done to each argument. One such program is `summarize`: type `summarize mpg` to obtain summary statistics on `mpg`, and type `summarize mpg weight` to obtain first summary statistics on `mpg` and then summary statistics on `weight`.

```
program ...
        local i = 1
        while "`i'" != "" {
                logic stated in terms of `i'
                local ++i
        }
end
```

Equivalently, if the logic that uses `` `i' `` contains only one reference to `` `i' ``,

```
program ...
        local i = 1
        while "`i'" != "" {
                logic stated in terms of `i++'
        }
end
```

Note the tricky construction `` `i' ``, which then itself is placed in double quotes—`"`i'"`—for the `while` loop. To understand it, say that `i` contains 1 or, equivalently, `` `i' `` is 1. Then `` `i' `` is `` `1' `` is the name of the first variable. `"`i'"` is the name of the first variable in quotes. The `while` asks if the name of the variable is nothing and, if it is not, executes. Now `` `i' `` is 2, and `"`i'"` is the name of the second variable, in quotes. If that name is not `""`, we continue. If the name is `""`, we are done.

Say that you were writing a subroutine that was to receive k variables, but the code that processes each variable needs to know (while it is processing) how many variables were passed to the subroutine. You need first to count the variables (and so derive k) and then, knowing k, pass through the list again.

```
program progname
        local k = 1                 // count the number of arguments
        while "`k'" != "" {
                local ++k
        }
        local --k                   // k contains one too many
                                    // now pass through again
        local i = 1
        while `i' <= `k' {
                code in terms of `i' and `k'
                local ++i
        }
end
```

In the above example, we have used `while`, Stata's all-purpose looping command. Stata has two other looping commands, `foreach` and `forvalues`, and they sometimes produce code that is more readable and executes more quickly. We direct you to read [P] **foreach** and [P] **forvalues**, but at this point, there is nothing they can do that `while` cannot do. Above we coded

```
        local i = 1
        while `i' <= `k' {
                code in terms of `i' and `k'
                local ++i
        }
```

to produce logic that looped over the values `‘i’` = 1 to `‘k’`. We could have instead coded

```
forvalues i = 1(1)‘k’ {
        code in terms of ‘‘i’’ and ‘k’
}
```

Similarly, at the beginning of this subsection, we said that you could use the following code in terms of `while` to loop over the arguments received:

```
program ...
        local i = 1
        while "‘‘i’’" != "" {
                logic stated in terms of ‘‘i’’
                local ++i
        }
end
```

Equivalent to the above would be

```
program ...
        foreach x of local 0 {
                logic stated in terms of ‘x’
        }
end
```

See [P] **foreach** and [P] **forvalues**.

You can combine `args` and incrementing through an unknown number of positional arguments. Say that you were writing a subroutine that was to receive `varname`, the name of some variable; `n`, which is some sort of count; and at least one and maybe 20 variable names. Perhaps you are to sum the variables, divide by `n`, and store the result in the first variable. What the program does is irrelevant; here is how we could receive the arguments:

```
program progname
        args varname n
        local i 3
        while "‘‘i’’" != "" {
                logic stated in terms of ‘‘i’’
                local ++i
        }
end
```

18.4.3 Using macro shift

Another way to code the repeat-the-same-process problem for each argument is

```
program ...
        while "‘1’" != "" {
                logic stated in terms of ‘1’
                macro shift
        }
end
```

`macro shift` shifts `‘1’`, `‘2’`, `‘3’`, ..., one to the left: what was `‘1’` disappears, what was `‘2’` becomes `‘1’`, what was `‘3’` becomes `‘2’`, and so on.

The outside `while` loop continues the process until macro `‘1’` contains nothing.

`macro shift` is an older construct that we no longer advocate using. Instead, we recommend that you use the techniques described in the previous subsection, that is, references to `‘‘i’’` and `foreach` or `forvalues`.

There are two reasons we make this recommendation: `macro shift` destroys the positional macros `‘1’`, `‘2’`, which must then be reset using `tokenize` should you wish to pass through the argument list again, and (more importantly) if the number of arguments is large (which in Stata/MP and Stata/SE is more likely), `macro shift` can be extremely slow.

❑ Technical note

`macro shift` can do one thing that would be difficult to do by other means.

`‘*’`, the result of listing the contents of the numbered macros one after the other with one blank between, changes with `macro shift`. Say that your program received a list of variables and that the first variable was the dependent variable and the rest were independent variables. You want to save the first variable name in `‘lhsvar’` and all the rest in `‘rhsvars’`. You could code

```
program progname
        local lhsvar "‘1’"
        macro shift 1
        local rhsvars "‘*’"
        ...
end
```

Now suppose that one macro contains a list of variables and you want to split the contents of the macro in two. Perhaps `‘varlist’` is the result of a `syntax` command (see **[U] 18.4.4 Parsing standard Stata syntax**), and you now wish to split `‘varlist’` into `‘lhsvar’` and `‘rhsvars’`. `tokenize` will reset the numbered macros:

```
program progname
        ...
        tokenize ‘varlist’
        local lhsvar "‘1’"
        macro shift 1
        local rhsvars "‘*’"
        ...
end
```

❑

18.4.4 Parsing standard Stata syntax

Let’s now switch to `‘0’` from the positional arguments `‘1’`, `‘2’`, ...

You can parse `‘0’` (what the user typed) according to standard Stata syntax with one command. Remember that standard Stata syntax is

[by *varlist*:] *command* [*varlist*] [*=exp*] [`using` *filename*] [*if*] [*in*] [*weight*]
[, *options*]

See **[U] 11 Language syntax**.

The `syntax` command parses standard syntax. You code what amounts to the syntax diagram of your command in your program, and then `syntax` looks at `‘0’` (it knows to look there) and compares what the user typed with what you are willing to accept. Then one of two things happens: either `syntax` stores the pieces in an easily processable way or, if what the user typed does not match what you specified, `syntax` issues the appropriate error message and stops your program.

Consider a program that is to take two or more variable names along with an optional `if` *exp* and `in` *range*. The program would read

```
program ...
        syntax varlist(min=2) [if] [in]
        ...
 end
```

You will have to read [P] **syntax** to learn how to specify the syntactical elements, but the command is certainly readable, and it will not be long until you are guessing correctly about how to fill it in. And yes, the square brackets really do indicate optional elements, and you just use them with `syntax` in the natural way.

The one `syntax` command you code encompasses the parsing process. Here, if what the user typed matches "two or more variables and an optional if and in", `syntax` defines new local macros:

`` `"varlist"' ``	the two or more variable names
`` `"if"' ``	the `if` *exp* specified by the user (or nothing)
`` `"in"' ``	the `in` *range* specified by the user (or nothing)

To see that this works, experiment with the following program:

```
program tryit
        syntax varlist(min=2) [if] [in]
        display "varlist now contains |`"varlist"'|"
        display `""if now contains |`"if"'|"'
        display "in now contains |`"in"'|"
 end
```

Below we experiment:

```
. tryit mpg weight
varlist now contains |mpg weight|
if now contains ||
in now contains ||
. tryit mpg weight displ if foreign==1
varlist now contains |mpg weight displ|
if now contains |if foreign==1|
in now contains ||
. tryit mpg wei in 1/10
varlist now contains |mpg weight|
if now contains ||
in now contains |in 1/10|
. tryit mpg
too few variables specified
r(102);
```

In our third try we abbreviated the weight variable as `wei`, yet, after parsing, `syntax` unabbreviated the variable for us.

If this program were next going to step through the variables in the varlist, the positional macros `` `"1"' ``, `` `"2"' ``, ... could be reset by coding

```
tokenize `"varlist"'
```

See [P] **tokenize**. `tokenize` `` `"varlist"' `` resets `` `"1"' `` to be the first word of `` `"varlist"' ``, `` `"2"' `` to be the second word, and so on.

18.4.5 Parsing immediate commands

Immediate commands are described in [U] **19 Immediate commands**—they take numbers as arguments. By convention, when you name immediate commands, you should make the last letter of the name *i*. Assume that `mycmdi` takes as arguments two numbers, the first of which must be a positive integer, and allows the options `alpha` and `beta`. The basic structure is

```
program mycmdi
        gettoken n 0 : 0, parse(" ,")          /* get first number */
        gettoken x 0 : 0, parse(" ,")          /* get second number */
        confirm integer number `n'             /* verify first is integer */
        confirm number `x'                     /* verify second is number */
        if `n'<=0 error 2001                   /* check that n is positive */
        place any other checks here
        syntax [, Alpha Beta]                  /* parse remaining syntax */
        make calculation and display output
end
```

See [P] **gettoken**.

18.4.6 Parsing nonstandard syntax

If you wish to interpret nonstandard syntax and positional arguments are not adequate for you, you know that you face a formidable programming task. The key to the solution is the `gettoken` command.

`gettoken` can pull one token from the front of a macro according to the parsing characters you specify and, optionally, define another macro or redefine the initial macro to contain the remaining (unparsed) characters. That is,

Say that `` `0' `` contains	"`this is what the user typed`"
After `gettoken`,	
new macro `` `token' `` could contain	"`this`"
and `` `0' `` could still contain	"`this is what the user typed`"
or	
new macro `` `token' `` could contain	"`this`"
and new macro `` `rest' `` could contain	"` is what the user typed`"
and `` `0' `` could still contain	"`this is what the user typed`"
or	
new macro `` `token' `` could contain	"`this`"
and `` `0' `` could contain	"` is what the user typed`"

A simplified syntax of `gettoken` is

`gettoken` *emname1* [*emname2*] `:` *emname3* [`,` `parse(`*pchars*`)` `quotes` `match(`*lmacname*`)` `bind`]

where *emname1*, *emname2*, *emname3*, and *lmacname* are the names of local macros. (Stata provides a way to work with global macros, but in practice that is seldom necessary; see [P] **gettoken**.)

`gettoken` pulls the first token from *emname3* and stores it in *emname1*, and if *emname2* is specified, stores the remaining characters from *emname3* in *emname2*. Any of *emname1*, *emname2*, and *emname3* may be the same macro. Typically, `gettoken` is coded

`gettoken` *emname1* `: 0` [`,` *options*]
`gettoken` *emname1* `0 : 0` [`,` *options*]

because `‘0’` is the macro containing what the user typed. The first coding is used for token lookahead, should that be necessary, and the second is used for committing to taking the token.

`gettoken`'s options are

`parse("`*string*`")`	for specifying parsing characters the default is `parse(" ")`, meaning to parse on white space it is common to specify `parse(‘"" "’)`, meaning to parse on white space and double quote (`‘"" "’` is the string double-quote-space in compound double quotes)
`quotes`	to specify that outer double quotes *not* be stripped
`match(`*lmacname*`)`	to bind on parentheses and square brackets *lmacname* will be set to contain "`(`", "`[`", or nothing, depending on whether *emname1* was bound on parentheses or brackets or if `match()` turned out to be irrelevant *emname1* will have the outside parentheses or brackets removed

`gettoken` binds on double quotes whenever a (simple or compound) double quote is encountered at the beginning of *emname3*. Specifying `parse(‘"" "’)` ensures that double-quoted strings are isolated.

`quote` specifies that double quotes not be removed from the source in defining the token. For instance, in parsing "`"this is" a test`", the next token is "`this is`" if `quote` is not specified and is "`"this is"`" if `quote` is specified.

`match()` specifies that parentheses and square brackets be matched in defining tokens. The outside level of parentheses or brackets is stripped. In parsing "`(2+3)/2`", the next token is "`2+3`" if `match()` is specified. In practice, `match()` might be used with expressions, but it is more likely to be used to isolate bound varlists and time-series varlists.

18.5 Scalars and matrices

In addition to macros, scalars and matrices are provided for programmers; see **[U] 14 Matrix expressions**, [P] **scalar** and [P] **matrix**.

As far as scalar calculations go, you can use macros or scalars. Remember, macros can hold numbers. Stata's scalars are, however, slightly faster and are a little more accurate than macros. The speed issue is so slight as to be nearly immeasurable. Macros are accurate to a minimum of 12 decimal digits, and scalars are accurate to roughly 16 decimal digits. Which you use makes little difference except in iterative calculations.

Scalars can hold strings, and, in fact, can hold longer strings than macros can. Scalars can also hold binary "strings". See **[U] 12.4.14 Notes for programmers**.

Stata has a serious matrix programming language called Mata, which is the subject of another manual. Mata can be used to write subroutines that are called by Stata programs. See the *Mata Reference Manual*, and in particular, [M-1] **Ado**.

18.6 Temporarily destroying the data in memory

It is sometimes necessary to modify the data in memory to accomplish a particular task. A well-behaved program, however, ensures that the user's data are always restored. The `preserve` command makes this easy:

```
code before the data need changing
preserve
code that changes data freely
```

When you use the `preserve` command, Stata/MP and Stata/SE make a copy of the user's data in memory. Stata/BE makes a copy on disk. There is a setting, `max_preservemem`, to control how much memory Stata/MP will use for such copies before falling back to disk. See [P] **preserve**. When your program terminates—no matter how—Stata restores the data and erases the temporary file.

An alternative to `preserve` is to use frames to make a copy of the data that need changing, manipulate the data in the newly copied frame, and then drop that frame afterward. See *Example of use in programs* in [D] **frame prefix**.

18.7 Temporary objects

If you write a substantial program, it will invariably require the use of temporary variables in the data, or temporary scalars, matrices, or files. Temporary objects are necessary while the program is making its calculations, and once the program completes they are discarded.

Stata provides three commands to create temporary objects: `tempvar` creates names for variables in the dataset, `tempname` creates names for scalars and matrices, and `tempfile` creates names for files. All are described in [P] **macro**, and all have the same syntax:

{ `tempvar` | `tempname` | `tempfile` } *macname* [*macname* ...]

The commands create local macros containing names you may use.

18.7.1 Temporary variables

Say that, in making a calculation, you need to add variables `sum_y` and `sum_z` to the data. You might be tempted to code

```
...
generate sum_y = ...
generate sum_z = ...
...
```

but that would be poor because the dataset might already have variables named `sum_y` and `sum_z` in it and you will have to remember to drop the variables before your program concludes. Better is

```
...
tempvar sum_y
generate `sum_y' = ...
tempvar sum_z
generate `sum_z' = ...
...
```

or

```
...
tempvar sum_y sum_z
generate `sum_y' = ...
generate `sum_z' = ...
...
```

It is not necessary to explicitly `drop `sum_y'` and `sum_z'` when you are finished, although you may if you wish. Stata will automatically drop any variables with names assigned by `tempvar`. After issuing the `tempvar` command, you must refer to the names with the enclosing quotes, which signifies macro expansion. Thus, after typing `tempvar sum_y`—the one case where you do not put single quotes around the name—refer thereafter to the variable `sum_y`, with quotes. `tempvar` does not create temporary variables. Instead `tempvar` creates names that may later be used to create new variables that will be temporary, and `tempvar` stores that name in the local macro whose name you provide.

A full description of `tempvar` can be found in [P] **macro**.

18.7.2 Temporary scalars and matrices

`tempname` works just like `tempvar`. For instance, a piece of your code might read

```
tempname YXX XXinv
matrix accum `YXX' = price weight mpg
matrix `XXinv' = invsym(`YXX'[2..., 2...])
tempname b
matrix `b' = `XXinv'*`YXX'[1..., 1]
```

The above code solves for the coefficients of a regression on `price` on `weight` and `mpg`; see **[U] 14 Matrix expressions** and [P] **matrix** for more information on the matrix commands.

As with temporary variables, temporary scalars and matrices are automatically dropped at the conclusion of your program.

18.7.3 Temporary files

In cases where you ordinarily might think you need temporary files, you may not because of Stata's ability to preserve and automatically restore the data in memory; see **[U] 18.6 Temporarily destroying the data in memory** above.

For more complicated programs, Stata does provide temporary files. A code fragment might read

```
preserve                          /* save original data */
tempfile males females
keep if sex==1
save "`males'"
restore, preserve                 /* get back original data */
keep if sex==0
save "`females'"
```

As with temporary variables, scalars, and matrices, it is not necessary to delete the temporary files when you are through with them; Stata automatically erases them when your program ends.

18.7.4 Temporary frames

You might want a program to temporarily create an additional dataset in memory without disturbing the dataset in the current frame. You can obtain a temporary name for a frame, copy or load data into it, and perform manipulations on those data. When your program is done, that frame and the data in it will automatically be removed from memory. For example, some code might read

```
tempname frname
frame copy default `frname'
frame `frname' {
    commands which modify the data in frame `frname'
}
...
```

When your program exits, successfully or not, any temporary frames it created will automatically be removed from memory.

18.8 Accessing results calculated by other programs

Stata commands that report results also store the results where they can be subsequently used by other commands or programs. This is documented in the *Stored results* section of the particular command in the reference manuals. Commands store results in one of three places:

1. r-class commands, such as `summarize`, store their results in `r()`; most commands are r-class.
2. e-class commands, such as `regress`, store their results in `e()`; e-class commands are Stata's model estimation commands.
3. s-class commands (there are no good examples) store their results in `s()`; this is a rarely used class that programmers sometimes find useful to help parse input.

Commands that do not store results are called n-class commands. More correctly, these commands require that you state where the result is to be stored, as in `generate` *newvar* `=` ....

▷ Example 1

You wish to write a program to calculate the standard error of the mean, which is given by the formula $\sqrt{s^2/n}$, where s^2 is the calculated variance. (You could obtain this statistic by using the `ci` command, but we will pretend that is not true.) You look at [R] **summarize** and learn that the mean is stored in `r(mean)`, the variance in `r(Var)`, and the number of observations in `r(N)`. With that knowledge, you write the following program:

```
program meanse
        quietly summarize `1'
        display "      mean = " r(mean)
        display "SE of mean = " sqrt(r(Var)/r(N))
end
```

The result of executing this program is

```
. meanse mpg
      mean = 21.297297
SE of mean = .67255109
```

◁

If you run an r-class command and type `return list` or run an e-class command and type `ereturn list`, Stata will summarize what was stored:

```
. use https://www.stata-press.com/data/r18/auto
(1978 automobile data)

. regress mpg weight displ
  (output omitted)

. ereturn list

scalars:
                  e(N) =  74
               e(df_m) =  2
               e(df_r) =  71
                  e(F) =  66.78504752026517
                 e(r2) =  .6529306984682528
               e(rmse) =  3.45606176570828
                e(mss) =  1595.409691543724
                e(rss) =  848.0497679157351
               e(r2_a) =  .643154098425105
                 e(ll) =  -195.2397979466294
               e(ll_0) =  -234.3943376482347
               e(rank) =  3

macros:
            e(cmdline) : "regress mpg weight displ"
              e(title) : "Linear regression"
          e(marginsok) : "XB default"
                e(vce) : "ols"
              e(depvar) : "mpg"
                e(cmd) : "regress"
         e(properties) : "b V"
            e(predict) : "regres_p"
              e(model) : "ols"
           e(estat_cmd) : "regress_estat"

matrices:
                  e(b) :  1 x 3
                  e(V) :  3 x 3
               e(beta) :  1 x 2

functions:
             e(sample)

. summarize mpg if foreign
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         22    24.77273    6.611187         14         41

. return list

scalars:
                  r(N) =  22
              r(sum_w) =  22
               r(mean) =  24.77272727272727
                r(Var) =  43.70779220779221
                 r(sd) =  6.611186898567625
               r(min) =  14
               r(max) =  41
               r(sum) =  545
```

In the example above, we ran `regress` followed by `summarize`. As a result, `e(N)` records the number of observations used by `regress` (equal to 74), and `r(N)` records the number of observations used by `summarize` (equal to 22). `r(N)` and `e(N)` are not the same.

If we now ran another r-class command—say, `tabulate`—the contents of `r()` would change, but those in `e()` would remain unchanged. You might, therefore, think that if we then ran another e-class command, say, `probit`, the contents of `e()` would change, but `r()` would remain unchanged. Although it is true that `e()` results remain in place until the next e-class command is executed, do

not depend on r() remaining unchanged. If an e-class or n-class command were to use an r-class command as a subroutine, that would cause r() to change. Anyway, most commands are r-class, so the contents of r() change often.

❑ Technical note

It is, therefore, of great importance that you access results stored in r() immediately after the command that sets them. If you need the mean and variance of the variable ‘1’ for subsequent calculation, do *not* code

```
summarize ‘1’
...
... r(mean) ... r(Var) ...
```

Instead, code

```
summarize ‘1’
local mean = r(mean)
local var = r(Var)
...
... ‘mean’ ... ‘var’ ...
```

or

```
tempname mean var
summarize ‘1’
scalar ‘mean’ = r(mean)
scalar ‘var’ = r(Var)
...
... ‘mean’ ... ‘var’ ...
```

❑

Stored results, whether in r() or e(), come in three types: scalars, macros, and matrices. If you look back at the ereturn list and return list output, you will see that regress stores examples of all three, whereas summarize stores just scalars. (regress also stores the “function” e(sample), as do all the other e-class commands; see [U] **20.7 Specifying the estimation subsample**.)

Regardless of the type of e(*name*) or r(*name*), you can just refer to e(*name*) or r(*name*). That was the rule we gave in [U] **13.6 Accessing results from Stata commands**, and that rule is sufficient for most uses. There is, however, another way to refer to stored results. Rather than referring to r(*name*) and e(*name*), you can embed the reference in macro-substitution characters ‘’ to produce ‘r(*name*)’ and ‘e(*name*)’. The result is the same as macro substitution; the stored result is evaluated, and then the evaluation is substituted:

```
. display "You can refer to " e(cmd) " or to ‘e(cmd)’"
You can refer to regress or to regress
```

This means, for instance, that typing ‘e(cmd)’ is the same as typing regress because e(cmd) contains “regress”:

```
. ‘e(cmd)’
      Source |       SS           df       MS      Number of obs   =        74
-------------+----------------------------------   F(2, 71)        =     66.79
       Model |  1595.40969         2  797.704846   Prob > F        =    0.0000
  (remaining output omitted)
```

In the ereturn list, e(cmd) was listed as a macro, and when you place a macro’s name in single quotes, the macro’s contents are substituted, so this is hardly a surprise.

What is surprising is that you can do this with scalar and even matrix stored results. `e(N)` is a scalar equal to 74 and may be used as such in any expression such as "`display e(mss)/e(N)`" or "`local meanss = e(mss)/e(N)`". `` `e(N)' `` substitutes to the string "74" and may be used in any context whatsoever, such as "`local val`e(N)' = e(N)`" (which would create a macro named `val74`). The rules for referring to stored results are

1. You may refer to `r(`*name*`)` or `e(`*name*`)` without single quotes in any expression and only in an expression. (Referring to s-class `s(`*name*`)` without single quotes is not allowed.)
   1.1 If *name* does not exist, missing value (`.`) is returned; it is not an error to refer to a nonexistent stored result.
   1.2 If *name* is a scalar, the full double-precision value of *name* is returned.
   1.3 If *name* is a macro, it is examined to determine whether its contents can be interpreted as a number. If so, the number is returned; otherwise, the string contents of *name* are returned.
   1.4 If *name* is a matrix, the full *matrix* is returned.
2. You may refer to `` `r(`*name*`)' ``, `` `e(`*name*`)' ``, or `` `s(`*name*`)' ``—note the presence of quotes indicating macro substitution—in any context whatsoever.
   2.1 If *name* does not exist, nothing is substituted; it is not an error to refer to a nonexistent stored result. The resulting line is the same as if you had never typed `` `r(`*name*`)' ``, `` `e(`*name*`)' ``, or `` `s(`*name*`)' ``.
   2.2 If *name* is a scalar, a string representation of the number accurate to no less than 12 digits of precision is substituted.
   2.3 If *name* is a macro, the full contents are substituted.
   2.4 If *name* is a matrix, the word `matrix` is substituted.

In general, you should refer to scalar and matrix stored results without quotes—`r(`*name*`)` and `e(`*name*`)`—and to macro stored results with quotes—`` `r(`*name*`)' ``, `` `e(`*name*`)' ``, and `` `s(`*name*`)' ``—but it is sometimes convenient to switch. Say that stored result `r(example)` contains the number of periods patients are observed, and assume that `r(example)` was stored as a macro and not as a scalar. You could still refer to `r(example)` without the quotes in an expression context and obtain the expected result. It would have made more sense for you to have stored `r(example)` as a scalar, but really it would not matter, and the user would not even have to know how the stored result was stored.

Switching the other way is sometimes useful, too. Say that stored result `r(N)` is a scalar that contains the number of observations used. You now want to use some other command that has an option `n(#)` that specifies the number of observations used. You could not type `n(r(N))` because the syntax diagram says that the `n()` option expects its argument to be a literal number. Instead, you could type `n(`` `r(N)' ``)`.

18.9 Accessing results calculated by estimation commands

Estimation results are stored in `e()`, and you access them in the same way you access any stored result; see [U] 18.8 Accessing results calculated by other programs above. In summary,

1. Estimation commands—`regress`, `logistic`, etc.—store results in `e()`.
2. Estimation commands store their name in `e(cmd)`. For instance, `regress` stores "`regress`" and `poisson` stores "`poisson`" in `e(cmd)`.

3. Estimation commands store the command they executed in `e(cmdline)`. For instance, if you typed `reg mpg displ`, stored in `e(cmdline)` would be "`reg mpg displ`".
4. Estimation commands store the number of observations used in `e(N)`, and they identify the estimation subsample by setting `e(sample)`. You could type, for instance, `summarize if e(sample)` to obtain summary statistics on the observations used by the estimator.
5. Estimation commands store the entire coefficient vector and variance–covariance matrix of the estimators in `e(b)` and `e(V)`. These are matrices, and they may be manipulated like any other matrix:

```
. matrix list e(b)
e(b)[1,3]
        weight       displ       _cons
y1  -.00656711   .00528078   40.084522

. matrix y = e(b)*e(V)*e(b)'
. matrix list y
symmetric y[1,1]
            y1
y1  6556.982
```

6. Estimation commands set `_b[`*name*`]` and `_se[`*name*`]` as convenient ways to use coefficients and their standard errors in expressions; see **[U] 13.5 Accessing coefficients and standard errors**.
7. Estimation commands may set other `e()` scalars, macros, or matrices containing more information. This is documented in the *Stored results* section of the particular command in the command reference.

Estimation commands also store results in `r()`. The `r(table)` matrix stores the results that you see in the coefficient table in the output. This includes the coefficients, standard errors, test statistics, p-values, and confidence intervals.

▷ Example 2

If you are writing a command for use after `regress`, early in your code you should include the following:

```
if "`e(cmd)'" != "regress" {
        error 301
}
```

This is how you verify that the estimation results that are stored have been set by `regress` and not by some other estimation command. Error 301 is Stata's "last estimates not found" error.

◁

18.10 Storing results

If your program calculates something, it should store the results of the calculation so that other programs can access them. In this way, your program not only can be used interactively but also can be used as a subroutine for other commands.

Storing results is easy:

1. On the `program` line, specify the `rclass`, `eclass`, or `sclass` option according to whether you intend to return results in `r()`, `e()`, or `s()`.

2. Code

`return scalar` *name* `=` *exp*	(same syntax as `scalar` without the `return`)
`return local` *name* ...	(same syntax as `local` without the `return`)
`return matrix` *name matname*	(moves *matname* to `r(`*name*`)`)

to store results in `r()`.

3. Code

`ereturn` *name* `=` *exp*	(same syntax as `scalar` without the `ereturn`)
`ereturn local` *name* ...	(same syntax as `local` without the `ereturn`)
`ereturn matrix` *name matname*	(moves *matname* to `e(`*name*`)`)

to store results in `e()`. You do not store the coefficient vector and variance matrix `e(b)` and `e(V)` in this way; instead you use `ereturn post`.

4. Code

`sreturn local` *name* ... (same syntax as `local` without the `sreturn`)

to store results in `s()`. (The s-class has only macros.)

A program must be exclusively r-class, e-class, or s-class.

18.10.1 Storing results in r()

In [U] **18.8 Accessing results calculated by other programs**, we showed an example that reported the mean and standard error of the mean. A better version would store in `r()` the results of its calculations and would read

```
program meanse, rclass
        quietly summarize ‘1’
        local mean = r(mean)
        local sem  = sqrt(r(Var)/r(N))
        display "      mean = " ‘mean’
        display "SE of mean = " ‘sem’
        return scalar mean = ‘mean’
        return scalar se = ‘sem’
end
```

Running `meanse` now sets `r(mean)` and `r(se)`:

```
. meanse mpg
      mean = 21.297297
SE of mean = .67255109
. return list
scalars:
       r(se)       =  .6725510870764975
       r(mean)     =  21.2972972972973
```

In this modification, we added the `rclass` option to the `program` statement, and we added two `return` commands to the end of the program.

Although we placed the `return` statements at the end of the program, they may be placed at the point of calculation if that is more convenient. A more concise version of this program would read

```
program meanse, rclass
        quietly summarize ‘1’
        return scalar mean = r(mean)
        return scalar se = sqrt(r(Var)/r(N))
        display "      mean = " return(mean)
        display "SE of mean = " return(se)
end
```

The return() function is just like the r() function, except that return() refers to the results that this program *will* return rather than to the stored results that currently *are* returned (which here are due to summarize). That is, when you code the return command, the result is not immediately posted to r(). Rather, Stata holds onto the result in return() until your program concludes, and then it copies the contents of return() to r(). While your program is active, you may use the return() function to access results you have already "returned". (return() works just like r() works after your program returns, meaning that you may code 'return()' to perform macro substitution.)

18.10.2 Storing results in e()

Storing in e() is in most ways similar to saving in r(): you add the eclass option to the program statement, and then you use ereturn ... just as you used return ... to store results. There are, however, some significant differences:

1. Unlike r(), estimation results are stored in e() the instant you issue an ereturn scalar, ereturn local, or ereturn matrix command. Estimation results can consume considerable memory, and Stata does not want to have multiple copies of the results floating around. That means you must be more organized and post your results at the end of your program.
2. In your code when you have your estimates and are ready to begin posting, you will first clear the previous estimates, set the coefficient vector e(b) and corresponding variance matrix e(V), and set the estimation-sample function e(sample). How you do this depends on how you obtained your estimates:
   2.1 If you obtained your estimates by using Stata's likelihood maximizer ml, this is automatically handled for you; skip to step 3.
   2.2 If you obtained estimates by "stealing" an existing estimator, e(b), e(V), and e(sample) already exist, and you will not want to clear them; skip to step 3.
   2.3 If you write your own code from start to finish, you use the ereturn post command; see [P] **ereturn**. You will code something like "ereturn post 'b' 'V', esample('touse')", where 'b' is the name of the coefficient vector, 'V' is the name of the corresponding variance matrix, and 'touse' is the name of a variable containing 1 if the observation was used and 0 if it was ignored. ereturn post clears the previous estimates and moves the coefficient vector, variance matrix, and variable into e(b), e(V), and e(sample).
   2.4 A variation on (2.3) is when you use an existing estimator to produce the estimates but do not want all the other e() results stored by the estimator. Then you code

```
tempvar touse
tempname b V
matrix 'b' = e(b)
matrix 'V' = e(V)
quietly generate byte 'touse' = e(sample)
ereturn post 'b' 'V', esample('touse')
```

3. You now store anything else in e() that you wish by using the ereturn scalar, ereturn local, or ereturn matrix command.
4. Save e(cmdline) by coding

```
ereturn local cmdline '"cmdname '0'"'
```

   This is not required, but it is considered good style.

5. You code `ereturn local cmd "`*cmdname*`"`. Stata does not consider estimation results complete until this command is posted, and Stata considers the results to be complete when this is posted, so you must remember to do this and to do this last. If you set `e(cmd)` too early and the user pressed *Break*, Stata would consider your estimates complete when they are not.

Say that you wish to write the estimation command with syntax

`myest` *depvar* *var*$_1$ *var*$_2$ [`if` *exp*] [`in` *range*]`,` *optset1* *optset2*

where *optset1* affects how results are displayed and *optset2* affects the estimation results themselves. One important characteristic of estimation commands is that, when typed without arguments, they redisplay the previous estimation results. The outline is

```
program myest, eclass
        local options "optset1"
        if replay() {
                if "`e(cmd)'"!="myest" {
                        error 301             /* last estimates not found       */
                }
                syntax [, `options']
        }
        else {
                syntax varlist [if] [in] [, `options' optset2]
                marksample touse
Code contains either this,
                tempnames b V
                commands for performing estimation
                assume produces `b' and `V'
                ereturn post `b' `V', esample(`touse')
                ereturn local depvar "`depv'"
or this,
                ml model ... if `touse' ...
and regardless, concludes,
                perhaps other ereturn commands appear here
                ereturn local cmdline `"myest `0'"'
                ereturn local cmd "myest"
        }
                                              /* (re)display results ...        */
code typically reads
        code to output header above coefficient table
        ereturn display                      /* displays coefficient table      */
or
        ml display                           /* displays header and coef. table */
end
```

Here is a list of the commonly stored `e()` results. Of course, you may create any `e()` results that you wish.

`e(N)` (scalar)
Number of observations.

`e(df_m)` (scalar)
Model degrees of freedom.

`e(df_r)` (scalar)
"Denominator" degrees of freedom if estimates are nonasymptotic.

`e(r2_p)` (scalar)
Value of the pseudo-R^2 if it is calculated. (If a "real" R^2 is calculated as it would be in linear regression, it is stored in (scalar) `e(r2)`.)

`e(F)` (scalar)
Test of the model against the constant-only model, if relevant, and if results are nonasymptotic.

`e(ll)` (scalar)
Log-likelihood value, if relevant.

`e(ll_0)` (scalar)
Log-likelihood value for constant-only model, if relevant.

`e(N_clust)` (scalar)
Number of clusters, if any.

`e(chi2)` (scalar)
Test of the model against the constant-only model, if relevant, and if results are asymptotic.

`e(rank)` (scalar)
Rank of `e(V)`.

`e(cmd)` (macro)
Name of the estimation command.

`e(cmdline)` (macro)
Command as typed.

`e(depvar)` (macro)
Names of the dependent variables.

`e(wtype)` and `e(wexp)` (macros)
If weighted estimation was performed, `e(wtype)` contains the weight type (`fweight`, `pweight`, etc.) and `e(wexp)` contains the weighting expression.

`e(title)` (macro)
Title in estimation output.

`e(clustvar)` (macro)
Name of the cluster variable, if any.

`e(vcetype)` (macro)
Text to appear above standard errors in estimation output; typically `Robust`, `Bootstrap`, `Jackknife`, or `""`.

`e(vce)` (macro)
vcetype specified in `vce()`.

`e(chi2type)` (macro)
`LR` or `Wald` or other depending on how `e(chi2)` was performed.

`e(properties)` (macro)
Typically contains `b V`.

`e(predict)` (macro)
Name of the command that `predict` is to use; if this is blank, `predict` uses the default `_predict`.

`e(b)` and `e(V)` (matrices)
The coefficient vector and corresponding variance matrix. Stored when you coded `ereturn post`.

`e(sample)` (function)
This function was defined by `ereturn post`'s `esample()` option if you specified it. You specified a variable containing 1 if you used an observation and 0 otherwise. `ereturn post` stole the variable and created `e(sample)` from it.

18.10.3 Storing results in s()

`s()` is a strange class because, whereas the other classes allow scalars, macros, and matrices, `s()` allows only macros.

`s()` is seldom used and is for subroutines that you might write to assist in parsing the user's input prior to evaluating any user-supplied expressions.

Here is the problem that `s()` solves: say that you create a nonstandard syntax for some command so that you have to parse through it yourself. The syntax is so complicated that you want to create subroutines to take pieces of it and then return information to your main routine. Assume that your syntax contains expressions that the user might type. Now say that one of the expressions the user types is, for example, `r(mean)/sqrt(r(Var))`—perhaps the user is using results left behind by `summarize`.

If, in your parsing step, you call subroutines that return results in `r()`, you will wipe out `r(mean)` and `r(Var)` before you ever get around to seeing them, much less evaluating them. So, you must be careful to leave `r()` intact until your parsing is complete; you must use no r-class commands, and any subroutines you write must not touch `r()`. You must use s-class subroutines because s-class routines return results in `s()` rather than `r()`. S-class provides macros only because that is all you need to solve parsing problems.

To create an s-class routine, specify the `sclass` option on the `program` line and then use `sreturn local` to return results.

S-class results are posted to `s()` at the instant you issue the `sreturn()` command, so you must organize your results. Also, `s()` is never automatically cleared, so occasionally coding `sreturn clear` at appropriate points in your code is a good idea. Few programs need s-class subroutines.

The `collect` suite of commands is one of the few examples in which results are posted to `s()`. This collection system gathers results from `r()` and `e()`, so posting its results to `s()` allows it to leave the `r()` and `e()` results intact.

18.11 Ado-files

Ado-files were introduced in [U] **17 Ado-files**.

When a user types '*gobbledygook*', Stata first asks itself if *gobbledygook* is one of its built-in commands. If so, the command is executed. Otherwise, it asks itself if *gobbledygook* is a defined program. If so, the program is executed. Otherwise, Stata looks in various directories for *gobbledygook*`.ado`. If there is no such file, the process ends with the "unrecognized command" error.

If Stata finds the file, it quietly issues to itself the command '`run` *gobbledygook*`.ado`' (specifying the path explicitly). If that runs without error, Stata asks itself again if *gobbledygook* is a defined program. If not, Stata issues the "unrecognized command" error. (Here somebody wrote a bad ado-file.) If the program is defined, as it should be, Stata executes it.

Thus you can arrange for programs you write to be loaded automatically. For instance, if you were to create `hello.ado` containing

begin hello.ado

```
program hello
        display "hi there"
end
```

end hello.ado

and store the file in your current directory or your personal directory (see [U] **17.5.2 Where is my personal ado-directory?**), you could type `hello` and be greeted by a reassuring

```
. hello
hi there
```

You could, at that point, think of `hello` as just another part of Stata.

There are two places to put your personal ado-files. One is the current directory, and that is a good choice when the ado-file is unique to a project. You will want to use it only when you are in that directory. The other place is your *personal ado-directory*, which is probably something like `C:\ado\personal` if you use Windows, `~/ado/personal` if you use Unix, and `~/ado/personal` if you use a Mac. We are guessing.

To find your personal ado-directory, enter Stata and type

```
. personal
```

❑ Technical note

Stata looks in various directories for ado-files, defined by the c-class value `c(adopath)`, which contains

```
BASE;SITE;.;PERSONAL;PLUS;OLDPLACE
```

The words in capital letters are codenames for directories, and the mapping from codenames to directories can be obtained by typing the `sysdir` command. Here is what `sysdir` shows on one particular Windows computer:

```
. sysdir
   STATA:  C:\Program Files\Stata18\
    BASE:  C:\Program Files\Stata18\ado\base\
    SITE:  C:\Program Files\Stata18\ado\site\
    PLUS:  C:\ado\plus\
PERSONAL:  C:\ado\personal\
OLDPLACE:  C:\ado\
```

Even if you use Windows, your mapping might be different because it all depends on where you installed Stata. That is the point of the codenames. They make it possible to refer to directories according to their logical purposes rather than their physical location.

The c-class value `c(adopath)` is the search path, so in looking for an ado-file, Stata first looks in `BASE` then in `SITE`, and so on, until it finds the file. Actually, Stata not only looks in `BASE` but also takes the first letter of the ado-file it is looking for and looks in the lettered subdirectory. For files with the extension `.style`, Stata will look in a subdirectory named `style` rather than a lettered subdirectory. Say that Stata was looking for `gobbledygook.ado`. Stata would look up `BASE` (`C:\Program Files\Stata18\ado\base` in our example) and, if the file were not found there, it would look in the `g` subdirectory of `BASE` (`C:\Program Files\Stata18\ado\base\g`) before looking in `SITE`, whereupon it would follow the same rules. If Stata were looking for `gobbledygook.style`, Stata would look up `BASE` (`C:\Program Files\Stata18\ado\base` in our example) and, if the file were not found there, it would look in the `style` subdirectory of `BASE` (`C:\Program Files\Stata18\ado\base\style`) before looking in `SITE`, whereupon it would follow the same rules.

Why the extra complication? We distribute hundreds of ado-files, help files, and other file types with Stata, and some operating systems have difficulty dealing with so many files in the same directory. All operating systems experience at least a performance degradation. To prevent this, the ado-directory we ship is split 31 ways (letters *a*–*z*, underscore, `jar`, `py`, `resource`, and `style`). Thus the Stata command `ci`, which is implemented as an ado-file, can be found in the subdirectory `c` of `BASE`.

If you write ado-files, you can structure your personal ado-directory this way, too, but there is no reason to do so until you have more than, say, 250 files in one directory.

❑

❑ Technical note

After finding and running *gobbledygook*`.ado`, Stata calculates the total size of all programs that it has automatically loaded. If this exceeds `adosize` (see [P] **sysdir**), Stata begins discarding the oldest automatically loaded programs until the total is less than `adosize`. Oldest here is measured by the time last used, not the time loaded. This discarding saves memory and does not affect you, because any program that was automatically loaded could be automatically loaded again if needed.

It does, however, affect performance. Loading the program takes time, and you will again have to wait if you use one of the previously loaded-and-discarded programs. Increasing `adosize` reduces this possibility, but at the cost of memory. The `set adosize` command allows you to change this parameter; see [P] **sysdir**. The default value of `adosize` is 1,000. A value of 1,000 for `adosize` means that up to 1,000 K can be allocated to autoloaded programs. Experimentation has shown that this is a good number—increasing it does not improve performance much.

❑

18.11.1 Version

We recommend that the first line following `program` in your ado-file declare the Stata release under which you wrote the program; `hello.ado` would read better as

begin hello.ado

```
program hello
        version 18.0                        // (or version 18.5 for StataNow)
        display "hi there"
end
```

end hello.ado

We introduced the concept of version in [U] **16.1.1 Version**. In regular do-files, we recommend that the `version` line appear as the first line of the do-file. For ado-files, the line appears after the `program` because loading the ado-file is one step and executing the program is another. It is when Stata executes the program defined in the ado-file that we want to stipulate the interpretation of the commands.

The inclusion of the `version` line is of more importance in ado-files than in do-files because ado-files have longer lives than do-files, so it is more likely that you will use an ado-file with a later release and ado-files tend to use more of Stata's features, increasing the probability that any change to Stata will affect them.

18.11.2 Comments and long lines in ado-files

Comments in ado-files are handled the same way as in do-files: you enclose the text in `/*` comment `*/` brackets, or you begin the line with an asterisk (`*`), or you interrupt the line with `//`; see [U] **16.1.2 Comments and blank lines in do-files**.

Logical lines longer than physical lines are also handled as they are in do-files: either you change the delimiter to a semicolon (`;`) or you comment out the new line by using `///` at the end of the previous physical line.

18.11.3 Debugging ado-files

Debugging ado-files is a little tricky because it is Stata and not you that controls when the ado-file is loaded.

Assume that you wanted to change `hello` to say "Hi, Mary". You open `hello.ado` in the Do-file Editor and change it to read

begin hello.ado

```
program hello
        version 18.0                        // (or version 18.5 for StataNow)
        display "hi, Mary"
end
```

end hello.ado

After saving it, you try it:

```
. hello
hi there
```

Stata ran the old copy of `hello`—the copy it still has in its memory. Stata wants to be fast about executing ado-files, so when it loads one, it keeps it around a while—waiting for memory to get short—before clearing it from its memory. Naturally, Stata can drop `hello` anytime because it can always reload it from disk.

You changed the copy on disk, but Stata still has the old copy loaded into memory. You type `discard` to tell Stata to forget these automatically loaded things and to force itself to get new copies of the ado-files from disk:

```
. discard
. hello
hi, Mary
```

You had to type `discard` only because you changed the ado-file while Stata was running. Had you exited Stata and returned later to use `hello`, the `discard` would not have been necessary because Stata forgets things between sessions anyway.

18.11.4 Local subroutines

An ado-file can contain more than one `program`, and if it does, the other programs defined in the ado-file are assumed to be subroutines of the main program. For example,

begin decoy.ado

```
program decoy
        ...
        duck ...
        ...
end
program  duck
        ...
end
```

end decoy.ado

`duck` is considered a local subroutine of `decoy`. Even after `decoy.ado` was loaded, if you typed `duck`, you would be told "unrecognized command". To emphasize what *local* means, assume that you have also written an ado-file named `duck.ado`:

```
begin duck.ado
program duck
        ...
end
end duck.ado
```

Even so, when `decoy` called `duck`, it would be the program `duck` defined in `decoy.ado` that was called. To further emphasize what *local* means, assume that `decoy.ado` contains

```
begin decoy.ado
program decoy
        ...
        manic ...
        ...
        duck ...
        ...
end
program duck
        ...
end
end decoy.ado
```

and that `manic.ado` contained

```
begin manic.ado
program manic
        ...
        duck ...
        ...
end
end manic.ado
```

Here is what would happen when you executed `decoy`:

1. `decoy` in `decoy.ado` would begin execution. `decoy` calls `manic`.
2. `manic` in `manic.ado` would begin execution. `manic` calls `duck`.
3. `duck` in `duck.ado` (yes) would begin execution. `duck` would do whatever and return.
4. `manic` regains control and eventually returns.
5. `decoy` is back in control. `decoy` calls `duck`.
6. `duck` in `decoy.ado` would execute, complete, and return.
7. `decoy` would regain control and return.

When `manic` called `duck`, it was the global ado-file `duck.ado` that was executed, yet when `decoy` called `duck`, it was the local program `duck` that was executed.

Stata does not find this confusing and neither should you.

18.11.5 Development of a sample ado-command

Below we demonstrate how to create a new Stata command. We will program an influence measure for use with linear regression. It is an interesting statistic in its own right, but even if you are not interested in linear regression and influence measures, the focus here is on programming, not on the particular statistic chosen.

Belsley, Kuh, and Welsch (1980, 24) present a measure of influence in linear regression defined as

$$\frac{\text{Var}\left(\widehat{y}_i^{(i)}\right)}{\text{Var}(\widehat{y}_i)}$$

which is the ratio of the variance of the ith fitted value based on regression estimates obtained by omitting the ith observation to the variance of the ith fitted value estimated from the full dataset. This ratio is estimated using

$$\text{FVARATIO}_i \equiv \frac{n-k}{n-(k+1)}\left\{1-\frac{d_i^2}{1-h_{ii}}\right\}(1-h_{ii})^{-1}$$

where n is the sample size; k is the number of estimated coefficients; $d_i^2 = e_i^2/\mathbf{e}'\mathbf{e}$ and e_i is the ith residual; and h_{ii} is the ith diagonal element of the hat matrix. The ingredients of this formula are all available through Stata, so, after estimating the regression parameters, we can easily calculate FVARATIO_i. For instance, we might type

```
. regress mpg weight displ
. predict hii if e(sample), hat
. predict ei if e(sample), resid
. quietly count if e(sample)
. scalar nreg = r(N)
. generate eTe = sum(ei*ei)
. generate di2 = (ei*ei)/eTe[_N]
. generate FVi = (nreg - 3) / (nreg - 4) * (1 - di2/(1-hii)) / (1-hii)
```

The number 3 in the formula for `FVi` represents k, the number of estimated parameters (which is an intercept plus coefficients on `weight` and `displ`), and the number 4 represents $k+1$.

❑ Technical note

Do you understand why this works? `predict` can create h_{ii} and e_i, but the trick is in getting $\mathbf{e}'\mathbf{e}$—the sum of the squared e_is. Stata's `sum()` function creates a running sum. The first observation of `eTe` thus contains e_1^2; the second, $e_1^2+e_2^2$; the third, $e_1^2+e_2^2+e_3^2$; and so on. The last observation, then, contains $\sum_{i=1}^{N} e_i^2$, which is $\mathbf{e}'\mathbf{e}$. (We specified `if e(sample)` on our `predict` commands to restrict calculations to the estimation subsample, so `hii` and `eii` might have missing values, but that does not matter because `sum()` treats missing values as contributing zero to the sum.) We use Stata's explicit subscripting feature and then refer to `eTe[_N]`, the last observation. (See **[U] 13.3 Functions** and **[U] 13.7 Explicit subscripting**.) After that, we plug into the formula to obtain the result.

❑

Assuming that we often wanted this influence measure, it would be easier and less prone to error if we canned this calculation in a program. Our first draft of the program reflects exactly what we would have typed interactively:

begin fvaratio.ado, version 1

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        predict hii if e(sample), hat
        predict ei if e(sample), resid
        quietly count if e(sample)
        scalar nreg = r(N)
        generate eTe = sum(ei*ei)
        generate di2 = (ei*ei)/eTe[_N]
        generate FVi = (nreg - 3) / (nreg - 4) * (1 - di2/(1-hii)) / (1-hii)
        drop hii ei eTe di2
end
```

end fvaratio.ado, version 1

All we have done is to enter what we would have typed into a file, bracketing it with `program fvaratio` and `end`. Because our command is to be called `fvaratio`, the file must be named `fvaratio.ado` and must be stored in either the current directory or our personal ado-directory (see **[U] 17.5.2 Where is my personal ado-directory?**).

Now when we type `fvaratio`, Stata will be able to find it, load it, and execute it. In addition to copying the interactive lines into a program, we added the line '`drop hii ...`' to eliminate the working variables we had to create along the way.

So, now we can interactively type

```
. regress mpg weight displ
. fvaratio
```

and add the new variable `FVi` to our data.

Our program is not general. It is suitable for use after fitting a regression model on two, and only two, independent variables because we coded a `3` in the formula for k. Stata statistical commands such as `regress` store information about the problem and answer in `e()`. Looking in *Stored results* in [R] **regress**, we find that `e(df_m)` contains the model degrees of freedom, which is $k-1$, assuming that the model has an intercept. Also, the sample size of the dataset used in the regression is stored in `e(N)`, eliminating our need to count the observations and define a scalar containing this count. Thus the second draft of our program reads

begin fvaratio.ado, version 2

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        predict hii if e(sample), hat
        predict ei if e(sample), resid
        gen eTe = sum(ei*ei)
        gen di2 = (ei*ei)/eTe[_N]
        gen FVi = (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *     /// changed this
                  (1 - di2/(1-hii)) / (1-hii)                    // version
        drop hii ei eTe di2
end
```

end fvaratio.ado, version 2

In the formula for `FVi`, we substituted `(e(df_m)+1)` for the literal number `3`, `(e(df_m)+2)` for the literal number `4`, and `e(N)` for the sample size.

Back to the substance of our problem, `regress` also stores the residual sum of squares in `e(rss)`, so calculating `eTe` is not really necessary:

begin fvaratio.ado, version 3

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        predict hii if e(sample), hat
        predict ei if e(sample), resid
        gen di2 = (ei*ei)/e(rss)                          // changed this version
        gen FVi = (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *      ///
                  (1 - di2/(1-hii)) / (1-hii)
        drop hii ei di2
end
```

end fvaratio.ado, version 3

Our program is now shorter and faster, and it is completely general. This program is probably good enough for most users; if you were implementing this solely for your own occasional use, you could stop right here. The program does, however, have the following deficiencies:

1. When we use it with data with missing values, the answer is correct, but we see messages about the number of missing values generated. (These messages appear when the program is generating the working variables.)
2. We cannot control the name of the variable being produced—it is always called `FVi`. Moreover, when `FVi` already exists (say, from a previous regression), we get an error message that `FVi` already exists. We then have to `drop` the old `FVi` and type `fvaratio` again.
3. If we have created any variables named `hii`, `ei`, or `di2`, we also get an error that the variable already exists, and the program refuses to run.

Fixing these problems is not difficult. The fix for problem 1 is easy; we embed the entire program in a `quietly` block:

begin fvaratio.ado, version 4

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        quietly {                                           // new this version
                predict hii if e(sample), hat
                predict ei if e(sample), resid
                gen di2 = (ei*ei)/e(rss)
                gen FVi = (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *      ///
                          (1 - di2/(1-hii)) / (1-hii)
                drop hii ei di2
        }                                                   // new this version
end
```

end fvaratio.ado, version 4

The output for the commands between the `quietly {` and `}` is now suppressed—the result is the same as if we had put `quietly` in front of each command.

Solving problem 2—that the resulting variable is always called `FVi`—requires use of the `syntax` command. Let's put that off and deal with problem 3—that the working variables have nice names like `hii`, `ei`, and `di2`, and so prevent users from using those names in their data.

One solution would be to change the nice names to unlikely names. We could change `hii` to `MyHiiVaR`, which would not guarantee the prevention of a conflict but would certainly make it unlikely. It would also make our program difficult to read, an important consideration should we want to change it in the future. There is a better solution. Stata's `tempvar` command (see **[U] 18.7.1 Temporary variables**) places names into local macros that are guaranteed to be unique:

begin fvaratio.ado, version 5

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        tempvar hii ei di2                                    // new this version
        quietly {
                predict `hii' if e(sample), hat   // changed, as are other lines
                predict `ei' if e(sample), resid
                gen `di2' = (`ei'*`ei')/e(rss)
                gen FVi = (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *      ///
                          (1 - `di2'/(1-`hii')) / (1-`hii')
        }
end
```

end fvaratio.ado, version 5

At the beginning of our program, we declare the temporary variables. (We can do it outside or inside the `quietly`—it makes no difference—and we do not have to do it at the beginning or even all at once; we could declare them as we need them, but at the beginning is prettiest.) When we refer to a temporary variable, we do not refer directly to it (such as by typing `hii`); we refer to it indirectly by typing open and close single quotes around the name (`` `hii' ``). And at the end of our program, we

no longer bother to drop the temporary variables—temporary variables are dropped automatically by Stata when a program concludes.

❑ Technical note

Why do we type single quotes around the names? tempvar creates local macros containing the real temporary variable names. hii in our program is now a local macro, and ‘hii’ refers to the contents of the local macro, which is the variable's actual name.

❑

We now have an excellent program—its only fault is that we cannot specify the name of the new variable to be created. Here is the solution to that problem:

begin fvaratio.ado, version 6

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        syntax newvarname                                    // new this version
        tempvar hii ei di2
        quietly {
                predict ‘hii’ if e(sample), hat
                predict ‘ei’ if e(sample), resid
                gen ‘di2’ = (‘ei’*‘ei’)/e(rss)
                gen ‘typlist’ ‘varlist’ = ///               changed this version
                        (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *      ///
                        (1 - ‘di2’/(1-‘hii’)) / (1-‘hii’)
        }
end
```

end fvaratio.ado, version 6

It took a change to one line and the addition of another to obtain the solution. This magic all happens because of syntax (see [U] 18.4.4 Parsing standard Stata syntax above).

‘syntax newvarname’ specifies that one new variable name must be specified (had we typed ‘syntax [newvarname]’, the new varname would have been optional; had we typed ‘syntax newvarlist’, the user would have been required to specify at least one new variable and allowed to specify more). In any case, syntax compares what the user types to what is allowed. If what the user types does not match what we have declared, syntax will issue the appropriate error message and stop our program. If it does match, our program will continue, and what the user typed will be broken out and stored in local macros for us. For a newvarname, the new name typed by the user is placed in the local macro varlist, and the type of the variable (float, double, . . .) is placed in typlist (even if the user did not specify a storage type, in which case the type is the current default storage type).

This is now an excellent program. There are, however, two more improvements we could make. First, we have demonstrated that, by the use of ‘syntax newvarname’, we can allow the user to define not only the name of the created variable but also the storage type. However, when it comes to the creation of intermediate variables, such as ‘hii’ and ‘di2’, it is good programming practice to keep as much precision as possible. We want our final answer to be precise as possible, regardless of how we ultimately decide to store it. Any calculation that uses a previously generated variable would benefit if the previously generated variable were stored in double precision. Below we modify our program appropriately:

begin fvaratio.ado, version 7

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        syntax newvarname
        tempvar hii ei di2
        quietly {
                predict double `hii' if e(sample), hat        // changed, as are
                predict double `ei' if e(sample), resid       // other lines
                gen double `di2' = (`ei'*`ei')/e(rss)
                gen `typlist' `varlist' =  ///
                        (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *      ///
                        (1 - `di2'/(1-`hii')) / (1-`hii')
        }
end
```

end fvaratio.ado, version 7

As for the second improvement we could make, `fvaratio` is intended to be used sometime after `regress`. How do we know the user is not misusing our program and executing it after, say, `logistic`? `e(cmd)` will tell us the name of the last estimation command; see [U] **18.9 Accessing results calculated by estimation commands** and [U] **18.10.2 Storing results in e()** above. We should change our program to read

begin fvaratio.ado, version 8

```
program fvaratio
        version 18.0                        // (or version 18.5 for StataNow)
        if "`e(cmd)'"!="regress" {                      // new this version
                error 301
        }
        syntax newvarname
        tempvar hii ei di2
        quietly {
                predict double `hii' if e(sample), hat
                predict double `ei' if e(sample), resid
                gen double `di2' = (`ei'*`ei')/e(rss)
                gen `typlist' `varlist' = ///
                        (e(N)-(e(df_m)+1)) / (e(N)-(e(df_m)+2)) *      ///
                        (1 - `di2'/(1-`hii')) / (1-`hii')
        }
end
```

end fvaratio.ado, version 8

The `error` command issues one of Stata's prerecorded error messages and stops our program. Error 301 is "last estimates not found"; see [P] **error**. (Try typing `error 301` at the command line.)

In any case, this is a perfect program.

❑ Technical note

You do not have to go to all the trouble we did to program the FVARATIO measure of influence or any other statistic that appeals to you. Whereas version 1 was not really an acceptable solution—it was too specialized—version 2 was acceptable. Version 3 was better, and version 4 better yet, but the improvements were of less and less importance.

Putting aside the details of Stata's language, you should understand that final versions of programs do not just happen—they are the results of drafts that have been refined. How much refinement depends on how often and who will be using the program. In this sense, the "official" ado-files that come with Stata are poor examples. They have been subject to substantial refinement because they will be used by strangers with no knowledge of how the code works. When writing programs for yourself, you may want to stop refining at an earlier draft.

❑

18.11.6 Writing help files

When you write an ado-file, you should also write a help file to go with it. This file is a standard text file, named *command*`.sthlp`, that you place in the same directory as your ado-file *command*`.ado`. This way, when users type `help` followed by the name of your new command (or pull down **Help**), they will see something better than "help for ... not found".

You can obtain examples of help files by examining the `.sthlp` files in the official ado-directory; type "`sysdir`" and look in the lettered subdirectories of the directory defined as `BASE`:

```
. sysdir
   STATA:  C:\Program Files\Stata18\
    BASE:  C:\Program Files\Stata18\ado\base\
    SITE:  C:\Program Files\Stata18\ado\site\
    PLUS:  C:\ado\plus\
PERSONAL:  C:\ado\personal\
OLDPLACE:  C:\ado\
```

Here you would find examples of `.sthlp` files in the `a`, `b`, ... subdirectories of `C:\Program Files\Stata18\ado\base`.

Help files are physically written on the disk in text format, but their contents are Stata Markup and Control Language (SMCL). For the most part, you can ignore that. If the file contains a line that reads

```
Also see help for the finishup command
```

it will display in just that way. However, SMCL contains many special directives, so that if the line in the file were to read

```
Also see {hi:help} for the {help finishup} command
```

what would be displayed would be

```
Also see help for the finishup command
```

and moreover, `finishup` would appear as a hypertext link, meaning that if users clicked on it, they would see help on `finishup`.

You can read about the details of SMCL in [P] **smcl**.

If you would like to see an example of SMCL code for a Stata help file, type `viewsource examplehelpfile.sthlp`. You can view the equivalent help file by selecting **Help > Stata command**, typing `examplehelpfile`, and clicking on **OK**, or you can type `help examplehelpfile`.

Users will find it easier to understand your programs if you document them the same way that we document ours. We offer the following guidelines:

1. The first line must be

   ```
   {smcl}
   ```

   This notifies Stata that the help file is in SMCL format.

2. The second line should be

   ```
   {* *! version #.#.#  date}{...}
   ```

   The * indicates a comment and the {...} will suppress the blank line. Whenever you edit the help file, update the version number and the date found in the comment line.

3. The next several lines denote what will be displayed in the quick access toolbar with the three pulldown menus: Dialog, Also See, and Jump To.

   ```
   {vieweralsosee "[R] help" "help help "}{...}
   {viewerjumpto "Syntax" "examplehelpfile##syntax"}{...}
   {viewerjumpto "Description" "examplehelpfile##description"}{...}
   {viewerjumpto "Options" "examplehelpfile##options"}{...}
   {viewerjumpto "Remarks" "examplehelpfile##remarks"}{...}
   {viewerjumpto "Examples" "examplehelpfile##examples"}{...}
   ```

4. Then place the title.

   ```
   {title:Title}

   {phang}
   {bf:yourcmd} {hline 2} Your title
   ```

5. Include two blank lines, and place the Syntax title, syntax diagram, and options table:

   ```
   {title:Syntax}

   {p 8 17 2}
   syntax line

   {p 8 17 2}
   second syntax line, if necessary

   {synoptset 20 tabbed}{...}
   {synopthdr}
   {synoptline}
   {syntab:tab}
   {synopt:{option}}brief description of option{p_end}
   {synoptline}
   {p2colreset}{...}

   {p 4 6 2}
   clarifying text, if required
   ```

6. Include two blank lines, and place the Description title and text:

   ```
   {title:Description}

   {pstd}
   description text
   ```

   Briefly describe what the command does. Do not burden the user with details yet. Assume that the user is at the point of asking whether this is what is wanted.

7. If your command allows options, include two blank lines, and place the Options title and descriptions:

   {title:Options}

   {phang}
   {opt *optionname*} *option description*

   {pmore}
   continued option description, if necessary

   {phang}
   {opt *optionname*} *second option description*

   Options should be included in the order in which they appear in the option table. Option paragraphs are reverse indented, with the option name on the far left, where it is easily spotted. If an option requires more than one paragraph, subsequent paragraphs are set using {pmore}. One blank line separates one option from another.

8. Optionally include two blank lines, and place the Remarks title and text:

   {title:Remarks}

   {pstd}
   text

   Include whatever long discussion you feel necessary. Stata's official system help files often omit this because the discussions appear in the manual. Stata's official help files for features added between releases (obtained from the *Stata Journal*, the Stata website, etc.), however, include this section because the appropriate *Stata Journal* may not be as accessible as the manuals.

9. Optionally include two blank lines, and place the Examples title and text:

   {title:Examples}

   {phang}
   {cmd:. *first example*}

   {phang}
   {cmd:. *second example*}

   Nothing communicates better than providing something beyond theoretical discussion. Examples rarely need much explanation.

10. Optionally include two blank lines, and place the Author title and text:

    {title:Author}

    {pstd}
    Name, affiliation, etc.

    Exercise caution. If you include a telephone number, expect your phone to ring. An email address may be more appropriate.

11. Optionally include two blank lines, and place the References title and text:

    {title:References}

    {pstd}
    Author. year.
    Title. Location: Publisher.

We also warn that it is easy to use too much {hi:highlighting}. Use it sparingly. In text, use {cmd:...} to show what would be shown in typewriter typeface it the documentation were printed in this manual.

❑ Technical note

Sometimes it is more convenient to describe two or more related commands in the same .sthlp file. Thus xyz.sthlp might document both the xyz and abc commands. To arrange that typing help abc displays xyz.sthlp, create the file abc.sthlp, containing

```
begin abc.sthlp
.h xyz
end abc.sthlp
```

When a .sthlp file contains one line of the form '.h *refname*', Stata interprets that as an instruction to display help for *refname*.

❑

❑ Technical note

If you write a collection of programs, you need to somehow index the programs so that users (and you) can find the command they want. We do that with our contents.sthlp entry. You should create a similar kind of entry. We suggest that you call your private entry user.sthlp in your personal ado-directory; see **[U] 17.5.2 Where is my personal ado-directory?**. This way, to review what you have added, you can type help user.

We suggest that Unix users at large sites also add site.sthlp to the SITE directory (typically /usr/local/ado, but type sysdir to be sure). Then you can type help site for a list of the commands available sitewide.

❑

18.11.7 Programming dialog boxes

You not only can write new Stata commands and help files but also can create your own interface, or dialog box, for a command you have written. Stata provides a dialog box programming language to allow you to create your own dialog boxes. In fact, most of the dialog boxes you see in Stata's interface have been created using this language.

This is not for the faint of heart, but if you want to create your own dialog box for a command, see **[P] Dialog programming**. The manual entry contains all the details on creating and programming dialog boxes.

18.12 Tools for interacting with programs outside Stata and with other languages

Advanced programmers may wish to interact Stata with other programs or to call programs or libraries written in other languages from Stata. Stata supports the following:

Shell out synchronously or asynchronously to another program	See [D] **shell**
Call code in libraries written in C, C++, FORTRAN, etc.	See [P] **plugin**
Call code written in Java	See [P] **Java intro**
Interact Stata and Python code	See [P] **PyStata intro**
Interact with an H2O cluster	See [P] **H2O intro**
Control Stata—send commands to it and retrieve results from it—from an external program via OLE Automation	See [P] **Automation**

18.13 A compendium of useful commands for programmers

You can use any Stata command in your programs and ado-files. Also, some commands are intended solely for use by Stata programmers. You should see the section under the *Programming* heading in the subject table of contents at the beginning of the *Stata Index*.

Also see the *Mata Reference Manual* for all the details on the Mata language within Stata.

18.14 References

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

Belsley, D. A., E. Kuh, and R. E. Welsch. 1980. *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. New York: Wiley.

Drukker, D. M. 2015. Programming an estimation command in Stata: Global macros versus local macros. *The Stata Blog: Not Elsewhere Classified*. http://blog.stata.com/2015/11/03/programming-an-estimation-command-in-stata-global-macros-versus-local-macros/.

Gould, W. W. 2001. Statistical software certification. *Stata Journal* 1: 29–50.

Haghish, E. F. 2019. Seamless interactive language interfacing between R and Stata. *Stata Journal* 19: 61–82.

Herrin, J. 2009. Stata tip 77: (Re)using macros in multiple do-files. *Stata Journal* 9: 497–498.