CHAPTER 7

User-Created Functions, Scripts, and S4 Methods

User-created functions and scripts often make the life of an R user easier. If a repetitive task involves several different lines of code, creating a function or script to do the task saves time. In S4, methods for generic functions are the functional side of S4 and require special treatment.

Designing plots is one example of when a user-created function or script makes sense. Plots often take several lines of code, and the design of a plot is usually an interactive process. From command line R, creating a function to do the plot and making changes to the function are often much easier than using the up arrow and changing lines.

Another example of when a user-created function or script is useful is when a user wants to try out a statistical technique that is not available in the R packages. Often, the user can create a function or script for the technique using functions that are available.

In R Studio, the Source window (the upper left window) provides a place to create and run code, which can then be saved as an R script, externally, or as a function, internally. The Source window is also a place into which to load R scripts or other text files.

M. Tollefson, *R Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-4405-0_7

Scripts

Scripts are code that is written in R and stored outside of the program. A file containing an R script is a text file and has the extension .R. R scripts can contain function definitions. From command line R, a script is run using the function source(). For example, let lm.example.R, a file in the working directory, contain

```
print( x )
print( y )
print( lm( y~x)  )
```

Then, running source() on the file gives

```
> source( "lm.example.R" )
[1]  1  2  3  4  5  6  7  8  9 10
[1] 21 22 23 24 25 26 27 28 29 30

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
         20            1
```

Note that only the results of the functions are printed.

In R Studio, things are simpler. There is no need to edit the script externally. If the script already exists outside of R, the script can be loaded into the Source window. Click on the icon of a yellow folder with a green arrow on the first menu just above the windows. Then, browse to the location of the script and click on the file. The file will open in the Source window. To run the file, click on the Run or Source icons to the right of the Source window menu bar. To run a portion of the file, highlight the portion and click on the Run icon.

To enter a new script, open the far left icon in the menu bar above the upper windows and choose the first option, "R Script." The Source window will open to a blank page. Just enter the lines of code. Run the code or sections of code to debug the script. R Studio helps with the debugging, flagging syntax errors. When done, you can save the script. Click on the floppy disk icon in the menu of the Source window and enter a name for the file. R Studio automatically gives the file an .R extension. To run the code when saving the code, check the "Source to Save" box.

The Structure of a Function

Functions that are not primitive functions all have the same structure. On the first line of the function is the word **function**, followed by open and close parentheses, which may or may not contain arguments. In most cases, an open bracket follows the parentheses. Usually, the body of the function is placed below the first line, and the last line is a blank line after the close bracket, which is usually on its own line. Normally, functions are assigned a name. For example:

```
> d.fun = function(){
+ print( 1:5 )
+ }

> d.fun
function(){
print( 1:5 )
}

> d.fun()
[1] 1 2 3 4 5
```

In this example, first, the function is assigned to **d.fun**; next, the content of `d.fun()` is listed; and, last, the function `d.fun()` is run.

The brackets are not necessary if the function consists of just one statement—which can be entered on the same line as the function statement or on the following line(s). For example:

```
> c.fun = function() print(1:5)

> c.fun
function() print(1:5)

> c.fun()
[1] 1 2 3 4 5
```

Again, the function is assigned a name, the function is listed, and the function is run.

Arguments are objects or values that are used by the function and that must be input to the function at the time the function is run, unless a default value exists for the argument. Arguments are placed within the parentheses when the function is created, separated by commas. A default value is supplied by setting the argument equal to the value. Arguments with default values do not have to be specified when the function is run. If the value is not specified, the function uses the default value.

An example follows of a function with two arguments, where **a** does not have a default value and must be specified, and **b** has the default value of 3:

```
> e.fun = function( a, b=3 ){
+ print( a:b )
+ }

> e.fun
function( a, b=3 ){
print( a:b )
}
```

```
> e.fun( 10 )
[1] 10  9  8  7  6  5  4  3

> e.fun()
Error in a:b : 'a' is missing
```

Again, the function is assigned a name, listed, and run. Note that since **a** is the first argument and **b** has a default value, **a** can be supplied without a name. In the second attempt to run `e.fun()`, no argument is supplied for **a**, so `e.fun()` returns an error.

Often, the user uses brackets within a function to enclose groups of statements, such as for **if**, **else**, **for**, **while**, and **repeat** groups. There must be the same number of opening brackets as closing brackets in a function; otherwise, the function will not save. Mismatched brackets are a common source of errors in R code and are flagged in R Studio.

Lines of code in R (both in a function and at the R prompt) can be broken and continued on the next line. R looks for things such as a closing parenthesis, bracket, or quotation mark to designate the end of a statement or a part of a statement.

Empty lines are legal in R functions. Also, any text can be commented out by placing a pound sign (#) in front of the text. On a line, anything entered after a pound sign is ignored. A piece of advice for writing functions is to write a little chunk at a time, debug at each step, and use plenty of comments.

How to Enter a Function into R

This section describes four ways to get a function into R using the command line and one way using R studio. The first involves using an editor. The second involves inline entry, as shown in the preceding section. The third involves creating a function outside of R and using `dget()` to get

the function into R. The fourth is a variation on the second and third and involves copying and pasting from a source that can be outside of R. The fifth involves using the R Studio Source window.

Using an Editor

For the Windows and OS X operating systems, there is a function, `edit()`, in the package utils that works well for creating new functions. The purpose of the function `edit()` is to call an editing function.

In Windows, the default editing function is the **internal** editor. The possible other choices for editor are `xedit()`, `emacs()`, `xemacs()`, `vi()`, and `pica()`, where the choice is available only if the editor is present on the system. The default editor is listed in `options()` and can be changed at any time (Chapter 15).

For OS X systems, the only editor available is the **`vi`** editor, which works well.

For Linux operating systems, calling `edit()` from the terminal window does not give a good result. A better editor is `emacs()`, which is available for Linux systems.

Most of the preceding information is from the help page for `edit()`. Enter **?edit** at the R prompt for more information about the editing functions.

To create an object that is a function by using an editor, the function is first assigned to a name. For example, let the name be **f.fun**. To create the function **f.fun()**, start by entering **f.fun = function(){}** at the R prompt. The object `f.fun` then contains a function with no arguments and no statements.

The next step is to edit the function. For simplicity, only the function `edit()` is shown in the example here. The other editors behave similarly. Enter **f.fun = edit(f.fun)** at the R prompt. An editing window opens up for editing (Figure 7-1).

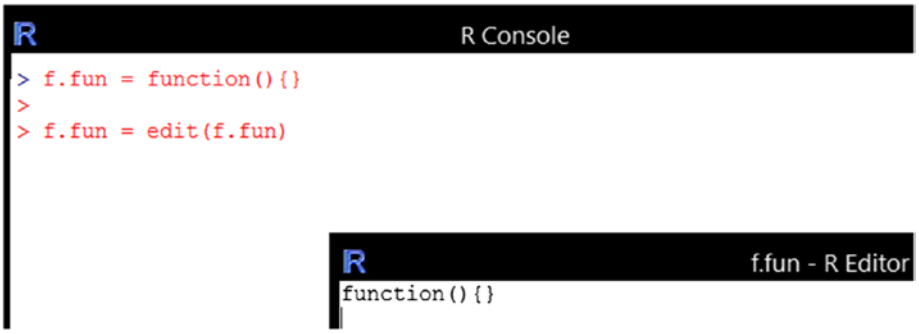

Figure 7-1. *Creating a function: the first and second steps*

For the third step, the arguments are entered within the parentheses, and the statements of the function are entered within the brackets (Figure 7-2).

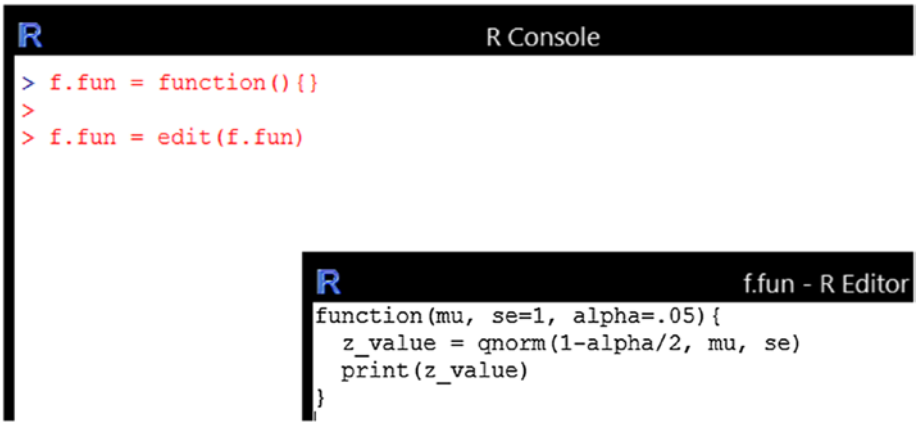

Figure 7-2. *Creating a function: the third step*

The fourth step is to exit the editor. To exit the editor, click the **x** at the top right-hand corner of the editing window. A window will appear with options to save the file, exit without saving, or to cancel the request and

go back to editing. (If no changes were made to the file, the options screen does not appear.) Click **Yes** to save the changes, **No** to revert to the earlier version, or **Cancel** to go back to editing.

If the function is syntactically correct, the function will save. Otherwise, `edit()` returns an error, such as the following:

```
Error in .External2(C_edit, name, file, title, editor) :
  unexpected '}' occurred on line 4
 use a command like
 x <- edit()
 to recover
```

To recover the work already done, enter **f.fun = edit()**. Using parentheses with no content is very important. If the name of the function is entered within the parentheses, the editing changes are lost, and the function reverts to the version before the edit. Note that the error message gives information about the problem with the R code.

The following shows the input and output at the R console when creating the function `f.fun()` with the editor, followed by the listing of the function, and the running of the function with the first argument set to zero.

```
> f.fun = function(){}

> f.fun = edit( f.fun )

> f.fun
function(mu, se=1, alpha=.05){
  z_value = qnorm(1-alpha/2, mu, se)
  print(z_value)
}

> f.fun( 0 )
[1] 1.959964.
```

Inline Entry

As shown in the first section of this chapter, a function can be entered inline. Let `b.fun` be the name of a new function created to list the digits three through six. Then, the steps to create the function, to list the code, and to run the function are as follows:

```
> b.fun = function(){
+ print( 3:6 )
+ }

> b.fun
function(){
print( 3:6 )
}

> b.fun()
[1] 3 4 5 6
```

If a syntactical error is made in the process of entering a function inline, R will give an error and return to the R prompt. For example:

```
> b.fun = function(){
+ print( 3:6
+ }
Error: unexpected '}' in:
"print( 3:6
}"
```

For longer functions, using the R editor or an external editor tends to be less frustrating.

An Outside Editor: dget() and Copying and Pasting

An outside editor can be used to create a function. Any editor that produces text files, such as **Notepad**, **TextEdit**, or **gedit**, can be used to create an R function. The rules for creating a function are the same as those described in the first section. Once the function is created, the function can be imported into the workspace by using the function `dget()` or by copying and pasting. (The function `dget()` and the corresponding function `dput()` are one way to import and export functions in R.)

Say that a function is in a file called **function.txt** in the same folder as the R workspace and that the function is syntactically correct. Then, the following line imports the function into the object `g.fun`:

```
g.fun = dget( "function.txt" )
```

(Note that R accepts more complex file paths for files, including absolute addresses on the hard drive and URLs.)

If the text file is not syntactically correct, R returns an error with information about the syntactical problem in the file.

If the file does not contain a function, or contains more than a function, R will attempt to run the code.

The file can also be copied and pasted from an outside source—or from elsewhere in the R session—into an object in R. Start by copying the function onto the clipboard of the computer. Next, enter the name that the object is to be called, followed by an equal sign, at the R prompt. The cursor should then be to the right of the equal sign. Next, paste.

If the function is syntactically correct, the cursor stops to the right of the close bracket. Press the **Return** key to complete the process. If the function is not syntactically correct, copying and pasting will give an error containing information about the problem with the syntax.

In R Studio

In R Studio, to create a new function, open a new R script (far left icon). Use an R script rather than a text file so that you can run the code from the Source window while debugging. Do not enter the function statement or the enclosing brackets. Enter variables to be entered as arguments first, assigning them values. When the lines of code run and run correctly, click on the wand icon above the Source window and choose "Extract Function." R Studio will cue you for a name and create the function in the Source window with the name assignment. The arguments should be in the correct place, but you may need to do some editing on the result. Running the resulting script assigns the function to the name within the workspace.

Clicking on an existing function under the Environment tab, in the upper right window in R Studio, opens the function in its own tab in the Source window. You cannot edit or run the code, but you can copy it, open a new R script, and paste the code into the R script for editing.

S4 Methods

S4 methods structure the functions of S4. An S4 method includes a name for the function being created, the class(s) of the data to be used by the function, and the function definition. The method can also specify where to store the method, if different from the workspace in which the method is created, as well as whether to seal the definition (not allow future changes.) S4 methods depend on the existence of a generic function that has the same name given in the method.

In S3, generic functions are functions for which the way the function behaves depends on the class of an argument. In S4, all functions are generic. In S4, a generic function is either created or exists and methods are created for the generic function. A method depends on the class(s) of the data objects used by the method.

The help page for `setMethod()` says that S4 usually does the creation automatically when `setMethod()` is run for functions that are S3 generics, but it must be done manually for new functions. Methods can be added to existing S4 generic functions.

The function `setGeneric()` creates a generic function. There are ten arguments to `setGeneric()`, two of which are usually assigned. The first is "name," which is a character string containing the name to be assigned to the function. The standard for S4 function names is **lowercasefirstUpperCaseAfter**. The second is **def**, which is a function definition and is optional in some cases. For new generic functions, the argument def is set equal to the function `standardGeneric()`, which normally has one argument, **f**, set equal to the name of the function in quotes. The rest of the arguments to `setGeneric()` are optional or have a default value that should be used. Enter **?setGeneric** and **?standardGeneric** at the R prompt or use the Help tab in R Studio for more information about the two functions.

The function **setMethod()**, is similar to **setGeneric()**, but also includes the class(s) associated with the method. The first argument is **f** and is a character string of the name to be assigned to the function. For a newly defined function, you must create the generic function with the desired name before creating the method. (See the succeeding example.) If the generic function exists, the names must match. The second argument is **signature** which is a character vector and gives the class(s) associated with the data objects used by the method.

The third argument, **definition**, defines the function of the method. The function definition is usually a mixture of S3 and S4; however, variables entered through the signature class(s) are subscripted using **@** or the function `slots()` rather than **$** or **[[**. But, if—say—there is a slot **mat** in class **mats**, where **mat** is a matrix, then mat could be subscripted with a combination of S3 and S4 methods; for example, `mats@mat[ 1:3, 4 ]`.

The last three arguments to `setMethod()` are usually left as their default values. The argument **where** tells R where to store the method, by default the namespace of the package for which the function is being defined. The argument **valueClass** is obsolete and by default is set to **NULL**. The argument **sealed** lets you freeze changes to the method and by default is set to **FALSE**.

This information is from the help page for `setMethod()`, which can be accessed by entering ?setMethod at the R prompt or by using the Help tab in R Studio.

A second example of creating and running a method (there is one in chapter 5) is as follows:

```
> setClass( "xyz", slots=c( x="numeric", y="numeric" ) )

> setMethod( "lmFunction", signature="xyz", function( x="xyz",
y="missing", ... )
{ print( lm( x@y ~ x@x ) ) } )
Error in setMethod("lmFunction", signature = "xyz", function
(x = "xyz",  :
  no existing definition for function ‘lmFunction’

> setGeneric( "lmFunction", function( x, y, ... ) {
standardGeneric( "lmFunction" ) } )
[1] "lmFunction"

> setMethod( "lmFunction", signature="xyz", function( x="xyz",
y="missing", ... )
{ print( lm( x@y ~ x@x ) ) } )

> xy1=new( "xyz", x=1:10, y=21:30 )

> lmFunction( xy1 )
```

```
Call:
lm(formula = x@y ~ x@x)

Coefficients:
(Intercept)          x@x
         20            1
```

You can see that the method cannot be defined until the generic function is defined. Note that **x** as defined in `setMethod()` has both the **x** and the **y** from the definition of **xy1** and that **y** is set to **missing**. In `setMethod()`, a variable set to **missing** is not used. Also note that the function is a new function, not in any of the loaded packages.

There are some testing functions to determine qualities of a function. The functions `isGeneric()`, `isS4()`, `isS3method()`, and `isS3stdGeneric()` can help determine if a S4 method can be defined with a function name. For example:

```
> isGeneric( "lm" )
[1] FALSE

> isS4( "lm" )
[1] FALSE

> isS3method( "lm" )
[1] FALSE

> isS3stdGeneric( "lm" )
[1] FALSE

> setClass( "lm", slots=c( fo="formula", df="data.frame" ) )
Error in setClass("lm", slots = c(fo = "formula", df = "data.
frame")) :
  "lm" has a sealed class definition and cannot be redefined
```

Here, lm() has a sealed class definition, so a new method cannot be defined for the function. For some S3 functions, methods can be defined. For example:

```
> isGeneric( "plot" )
[1] FALSE

> isS4( "plot" )
[1] FALSE

> isS3method( "plot" )
[1] FALSE

> isS3stdGeneric( "plot" )
plot
TRUE

> setClass( "plot", slots=c( x="numeric", y="numeric" ) )

> setMethod( "plot", signature="plot", definition=function
( x, y ,... ){ plot( x@x, x@y ) } )

> tester=new( "plot", x=1:10, y=21:30 )

> plot ( tester )

> isGeneric( "plot" )
[1] TRUE

> isS4(  "plot"  )
[1] FALSE

> isS3method( "plot" )
[1] FALSE

> isS3stdGeneric( "plot" )
[1] FALSE
```

Note that after creating the method for `plot()`, the function becomes an S4 generic function rather than an S3 standard generic function, at least in the workspace environment.

There are a number of other functions associated with S4 methods. The functions `selectMethod()`, `findMethod()`, `getMethod()`, `existsMethod()`, and `hasMethod()` are all grouped together in one help page. The functions `selectMethod()` and `getMethod()` return the function and the class of the method. The functions `existsMethod()` and `hasMethod()` return a logical value of **TRUE** or **FALSE** depending on if the method is found or not. The functions differ as to whether they allow inheritance. The functions `selectMethod()` and **existsMethod()** do. The function **findMethod()** returns the location of the method.

The first argument to all of the functions is **f**, the name of a generic function. The second argument is **signature**, a character vector of class name(s) consisting of class(s) for which method is defined. The functions `selectMethod()` and `getMethod()` behave similarly, except that selectMethod() has three arguments that `getMethod()` does not have; **useInherited**, **verbose**, and **doCache**, none of which are normally used.

All of the functions except `selectMethod()` have the argument **where**, which is an optional character variable giving the environment in which to look for the method. Both `selectMethod()` and `getMethod()` have the arguments **optional**, **mlist**, and **fdef**. The argument **optional**, if set to **TRUE**, tells R to return **NULL** rather than an error if `selectMethod()` does not find a method. The argument does not appear to affect `getMethod()`. The default value is **FALSE** for both functions. According to the help page for these functions, the other arguments are rarely used. More information can be found by entering **?getMethod** at the R prompt or by using the R Studio help tab.

The function `removeMethod()` removes a method. The function takes the **f**, **signature**, and **where** arguments, which are as described previously. Note that to make changes to a method, the method must be removed and assigned again after making the changes. For more information, enter **?removeMethod** at the R prompt or use the Help tab in R Studio.

Generic functions also must be removed at times. Generic functions are removed by using `removeGeneric()`. The function is one of a group of functions under the same help page: **Tools for Managing Generic Functions**. The arguments to `removeGeneric()` are **f** and **signature**, which are as described previously. Some of the other functions listed in the help page are `isGeneric()` already described; `findFunction()` which finds the locations of a function; `removeMethods()` which removes all methods associated with a function; and `getGenerics()` which lists all generics. For the last two functions, the location can be specified (use **.GobalEnv** for the workspace.) The function `standardGeneric()` is also at this help page. Enter **?removeGeneric** at the R prompt or use the Help tab in R Studio for more information.

The function `showMethods()` shows the methods for S4 generic functions. The function takes eight arguments. The first is **f**, the name(s) of the function(s). The argument is optional. If not used, the function returns all S4 generic functions. The second is **where**, the environment(s) in which to look for the function(s). By default, **where** is set to the parent environment of the workspace. To see the methods in the workspace, set **where** equal to **.GlobalEnv**.

The third argument is **classes** and is a list of classes used to restrict the search. The argument is optional. The fourth argument is **includeDefs**, a logical variable. If **TRUE**, the functions are printed out. The default value is **FALSE**. The fifth argument is **inherited**, a logical variable. If **TRUE**, the inherited methods that have been used during the session are included in the list of methods. The default value is the opposite value of **includeDefs**.

The sixth argument is **showEmpty**, a logical variable. See the help page for more information. The seventh argument is **printTo**, which tell R where to print the result of the call to the function. By default, the function prints to the standard output, usually the terminal. The last argument is **fdef** which allows you the option of choosing which generic function definition to use. The argument is optional. Enter **?showMethods** for more information or use the R Studio Help tab.

CRAN'S introduction to S4 methods can be found by entering **?methods::Introduction** at the R prompt or by using the Help tab in R Studio.