CHAPTER 8

How to Use a Script or Function

While scripts are just listings of code stored outside of R, functions are objects of mode function and are stored in the workspace. Most functions require specific kinds of arguments, which must be input into the function correctly. For example, if a function calls for a matrix and a data.frame is input, the function will return an error. Since external tables are often read into the R workspace as data.frames, using a data.frame for a matrix is quite a common error. This chapter covers calling a function, using arguments in a function, and accessing the output of a function, as well as an example of using a script to do a simple mining of Twitter.

Calling a Function

Calling a function is straightforward. The name of the function is entered at the R prompt followed by a set of parentheses which may or may not contain arguments, depending on the function. If the function does require arguments, the arguments are separated by commas within the parentheses.

Sometimes the argument name must be used, but not always. For values that are entered without names, R assigns the values to the arguments which are unnamed in the call, starting with the first unnamed

M. Tollefson, *R Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-4405-0_8

variable and continuing in order until the unnamed arguments are exhausted. The order of the arguments is the order of the arguments within the parentheses of the function definition.

To illustrate the use of arguments, an example follows using a function named **f.fun()**. The function **f.fun()** calculates a quantile of the normal distribution given the mean, the standard deviation, and alpha. The function returns the (1-alpha/2) x 100th percentile of the distribution. The arguments "se" and "alpha" are given default values and "mu" is not.

The example starts with a definition of the function, which is followed by five different calls to the function:

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

> f.fun( mu=0, se=1, alpha=0.05 )
[1] 1.959964
```

In the first call, each of the arguments is specified by name. In R, arguments can be in any order if specified by name.

```
> f.fun( 0, 1, 0.10 )
[1] 1.644854
```

In the second call, the values for the arguments are entered without names. Since the arguments are entered in order, the function knows which argument to assign to which value. The argument "mu" takes on the value of "0", "se" the value of "1," and "alpha" the value of "0.10," which is the order of the arguments within the parentheses in the function.

```
> f.fun( 0, alpha=0.20 )
[1] 1.281552
```

In the third call, the first argument is entered without a name, and the third argument is entered with a name. The second argument takes on the default value. The argument "mu" takes on the value of "0," "se" the value of "1," and "alpha" the value of "0.20."

```
> f.fun( 4, 4 )
[1] 11.83986
```

In the fourth call, values for the first two arguments are entered without names, and the third argument takes on the default value. The argument "mu" takes on the value of "4," as does "se." The argument "alpha" takes on the default value of "0.05."

```
> f.fun( se=1, 0, 0.2 )
[1] 1.281552
```

In the fifth call, the second argument is named, and the first and third are not, so "mu" takes on the value "0" and "alpha" takes on the value "0.2," while "se" takes on the value "1." Note that the named argument can be placed anywhere in the list.

Arguments

Given a function, a listing of the arguments to the function can be found at the help page for the function. Most help pages distinguish between the S3 and S4 versions of the functions. The S3 versions give the arguments for the S3 form of the function. The S4 versions give only those arguments that must be included, plus the "..." argument. In S4, each method for a generic function is different, so the arguments may vary by the method.

For some functions, the user must know something about the theory behind the function to understand the arguments, but for many functions the arguments are straightforward. As noted in the last section, arguments with default values do not have to be given a value when the function is called.

Arguments to a function must be of the correct mode and class. On the help page of a function, descriptions of the arguments are listed in the "Arguments" section, sometimes giving the mode and(or) class, but not always. Sometimes, the mode and(or) class is obvious. Sometimes, more information can be found in the "Details" section. Sometimes, looking in the "Examples" section is enough to clear up the form of an argument.

One argument which needs a little explaining is the "..." argument. The "..." argument tells the user that there are more arguments that may be entered. The arguments would be to a lower-level function called by the higher-level function. An example follows.

The example starts by listing two vectors, "x" and "y," and then continues with two calls to the function lm() with two different values for the argument "tol." (The function lm() fits a linear model.) On the help page for lm(), there is no argument "tol." However, there is the argument "...," indicating that lm() calls another function for which an argument can be entered.

The function lm.fit() is a lower level function which lm() calls and lm.fit() has the argument "tol." (The argument "tol" gives the tolerance for the QR decomposition as to whether a matrix is singular.) In the first call to lm(), the default value for "tol" is used, since "tol" is not specified. In the second call, lm() passes the value for "tol" to lm.fit().

```
> x
[1] 2.001 2.000 2.000

> y
[1] 4.03 4.00 4.01

> lm( y~x )

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

```
Coefficients:
(Intercept)            x
     -45.99        25.00

> lm( y~x, tol=.001 )

Call:
lm(formula = y ~ x, tol = 0.001)

Coefficients:
(Intercept)            x
      4.013           NA
```

In the first call, the default value for "tol" is 1.0e-7, so lm.fit() does not find a linear dependency in the matrix consisting of a column of ones and "x." As a result two coefficients are fit.

In the second call, "tol" is set to 1.0e-3, and lm() determines that there is a linear dependency in the matrix consisting of a column of ones and "x," so only one coefficient is fit.

The Output from a Function

The output from a function will vary with the function. Plotting functions mainly give plots. Summary functions give summarized results. Functions that test a hypothesis give the results from the test.

Most packaged functions print some results directly to the screen, but most packaged functions also have output which can be accessed through subscripting. For example, looking at the help page of the function lm(), under the "Value" Section, coefficients, residuals, fitted. values, rank, weights, df.residual, call, terms, contrasts, xlevels, offset, y, x, model, and na.action are all values which can be accessed from a call to the function.

The most common method used to access values is with the "$" operator, although index subscripting can be used, too. For most functions, the output is of mode list. The elements of the list can be of any mode.

For the first simple regression model fit in the last section, the accessible 15 values are as follows:

```
> a.lm = lm( y~x )

> a.lm$coef
(Intercept)           x
    -45.995      25.000

> a.lm$res
             1             2             3
-4.336809e-19 -5.000000e-03  5.000000e-03

> a.lm$fit
    1     2     3
4.030 4.005 4.005

> a.lm$rank
[1] 2

> a.lm$weights
NULL

> a.lm$df
[1] 1

> a.lm$call
lm(formula = y ~ x)

> a.lm$terms
y ~ x
attr(,"variables")
list(y, x)
```

```
attr(,"factors")
  x
y 0
x 1
attr(,"term.labels")
[1] "x"
attr(,"order")
[1] 1
attr(,"intercept")
[1] 1
attr(,"response")
[1] 1
attr(,".Environment")
<environment: R_GlobalEnv>
attr(,"predvars")
list(y, x)
attr(,"dataClasses")
        y         x
"numeric" "numeric"

> a.lm$contrasts
NULL

> a.lm$xlevels
named list()

> a.lm$offset
NULL

> a.lm$y
NULL

> a.lm$x
named list()
```

```
> a.lm$model
     y     x
1 4.03 2.001
2 4.00 2.000
3 4.01 2.000

> a.lm$na.action
NULL
```

In the example, the call to lm() was assigned a name, but lm() could have been subscripted directly. An example is lm(y~x)$coef. Values accessed from a call to a function are often used in another function.

Running an R function takes a little care, but with some experimentation and determination, the results can be very useful.

Example of a Script: Mining Twitter

This example demonstrates a way to mine Twitter and gives a result from a mining call. The example is stored on the hard drive as a script and is not a function. A mixture of S3 and S4 is used in the script. Most of the objects in "tm," the text mining package used here, are S4 objects and are methods.

In order to mine Twitter, you must create a developer account on Twitter and create an app. To create a developer account, you must have a Twitter account. If you have a Twitter account, create a developer account at https://developer.twitter.com. Otherwise, open a Twitter account, then create a developer account.

To open an account at the developer site, open the "Apply" button toward the right side of the top menu. Follow the instructions. When the account is approved, the name you have chosen for the developer account should then be on the top menu of the developer page, to the right. Choose the "Apps" item in the dropdown menu below the name.

In the window that opens, choose the "Create an App" button. Follow the instructions to create an app. On the page that opens after the app is created, click on "Details" and look under the "Keys and tokens" tab. There, you will find the consumer API key and API secret key:

Consumer API keys

```
############################################## (API key)
############################################## (API secret key)
```

(The pound signs in the above will be letters and numbers in the actual result.)

Below the customer API keys is a button to regenerate the keys. You can regenerate the keys at any time. Below the regenerate button is a button to generate tokens.

Select the button to generate tokens. The result will be:

Access token & access token secret

```
########################################## (Access token)
########################################## (Access token secret)
```

(The pound signs will be mostly letters and numbers in the actual result.) Below the tokens is the access level. The default access is read and write:

Read and write (Access level)

Below the access level are buttons to revoke the tokens or to regenerate them. The tokens can be revoked or regenerated at any time. The tokens and keys are used by the "twitteR" package in R to connect to the Twitter API.

The Twitter developer app must be open when R or R Studio is run or the program will crash when R attempts to connect with Twitter and all of your work will be lost.

The libraries “twitteR” and “tm” (for text mining) are loaded first in the script. The script is below:

```
library( twitteR )
library( tm )

# Connect to Twitter; the consumer_key, consumer_secret,
# access_token, access_secret are the ones generated by Twitter.

setup_twitter_oauth(
 consumer_key    = "##########################",
 consumer_secret = "########################################",
 access_token    = "########################################",
 access_secret   = "########################################"
)

# Fetch at most 100 tweets about "Clinton" and within 70 miles of
# 42 N and 95.5 W. The tweets are fetched backwards in time.

tweetsClinton = searchTwitter( "Clinton", n = 100,
geocode = "42,-95.5,70mi" )

# The types and classes of tweetsClinton and of the elements of
# tweetsClinton (tweetsClinton is a list).

print( typeof( tweetsClinton ) )
print( class(  tweetsClinton ) )

print( typeof( tweetsClinton[[1]] ) )
print( class(  tweetsClinton[[1]] ) )

# Manipulate the S4 objects in tweetsClinton into an S3 matrix of
# the number of a given word in each tweet. The words are assigned
# to the row names.
```

```
tweetsClintonDF = twListToDF( tweetsClinton )

ClintonCorpus = Corpus( VectorSource( tweetsClintonDF$text ) )

ClintonTDM = TermDocumentMatrix( ClintonCorpus )

ClintonMatrix = as.matrix( ClintonTDM )

# Create a data frame with words in the first column and
# the frequencies of the words in the second. Only keep words
# with a frequency greater than 12. Print out the result.

ClintonFrqMat = data.frame( Word = rownames( ClintonMatrix ),
                            Frequency = rowSums( ClintonMatrix ) )

ClintonFrqMatReduced = ClintonFrqMat[ ClintonFrqMat[ , 2 ] > 12, ]

print( ClintonFrqMatReduced )
```

Sourcing the script in R gives:

```
> source('~/Documents/RQSRexample.R')
[1] "Using direct authentication"
[1] "list"
[1] "list"
[1] "S4"
[1] "status"
attr(,"package")
[1] "twitteR"
                     Word Frequency
and                   and        19
for                   for        13
hillary           hillary        23
https               https        77
clinton           clinton        43
that                 that        15
```

```
the                    the        58
was                    was        16
trump                trump        18
you                    you        14
georgepapa19 georgepapa19        15
```

The tweets have not been cleaned in any way. Usually tweets are reduced to nontrivial words. Note that "clinton" only shows up in 43 times in the tweets. The tweets include tweets related to Clinton as well as tweets including the word "Clinton."

The sources for the above information include the R help pages and stack overflow.

More information about the above functions can be found by entering ??tm::tm or ??twitteR::twitteR at the R prompt or by using the "Help" tab in R Studio.