A SHORT R TUTORIAL

Steven M. Holland

Department of Geology, University of Georgia, Athens, GA 30602-2501

24 August 2021

Installing R

R is open-source and freely available for macOS, Linux, and Windows. You can download compiled versions of R (called binaries, or precompiled binary distributions) by going to the home page for R (http://www.r-project.org), and following the link to CRAN (the Comprehensive R Archive Network). You will be asked to select a mirror; pick one that is geographically nearby. On the CRAN site, each operating system has a FAQ page, and there is also a more general FAQ. Both are worth reading.

To download R, macOS users should follow the macOS link from the CRAN page and select the file corresponding to the most recent version of R. This will download a disk image; double-click it to install R. R can be run from the R app, or from the command line in Terminal and XQuartz.

Linux users should follow the links for their distribution and version of Linux and download the most recent version of R. There is a read-me file that explains the download and installation process.

Windows users should follow the link for Windows and then the link for the base package. A read-me file contains the installation instructions for the R app.

Rstudio is alternative running R within a single window, which has a Matlab-like interface that many prefer. It is available for macOS, Linux, and Windows at http://rstudio.com.

For more details, follow the Manuals link on the left side of the R home page. *R Installation and Administration* gives detailed instructions for installation on all operating systems.

In addition to R, you should install a good text editor for writing and editing code; do not used a word processor (like Word) for this. For macOS, BBEdit is an excellent text editor and is available from Bare Bones Software; Atom is also good. For Windows, Notepad++ is highly recommended, and it is free. Sublime Text is another highly regarded text editor, and it is available for macOS, Linux, and Windows.

Learning R

There are an enormous number of books on R. Several I've read are listed below, starting with the more basic and passing to the more advanced. **The R Book** is my favorite. If you have a programming background or want to get serious about programming in R, I highly recommend **The Art of R Programming**.

Statistics : An Introduction using R, by Michael J. Crawley, 2014. John Wiley & Sons, 360 p. ISBN-13: 978-1118941096.

Using R for Introductory Statistics, by John Verzani, 2014. Chapman & Hall/CRC, 518 p. ISBN-13: 978-1466590731.

The R Book, by Michael J. Crawley, 2012. Wiley, 1076 p. ISBN-13: 978-0470973929.

An R and S-Plus® Companion to Multivariate Analysis, by Brian S. Everitt, 2007. Springer, 221 p. ISBN-13: 978-1852338824.

Data Analysis and Graphics Using R, by John Maindonald, 2010. Cambridge University Press, 549 p. ISBN-13: 978-0521762939.

Ecological Models and Data in R, by Benjamin M. Bolker , 2008. Princeton University Press, 408 p. ISBN-13: 978-0691125220.

The Art of R Programming: A Tour of Statistical Software Design, by Norman Matloff, 2011. No Starch Press, 400 p. ISBN-13: 978-1593273842.

The manuals link on the R home page points to three important guides. The *Introduction to R* is a basic source of information on R. *R Data Import/Export* is useful for understanding the many ways in which data may be imported into or exported from R. The *R Reference Index* is a gigantic pdf (3500 pages!) that comprehensively lists all help files in a standard R installation. These help files are also freely accessible in every installation of R.

Every experienced R user likely has their favorite web sites for R, and these three are mine:

R-bloggers (http://www.r-bloggers.com) is a good news and tutorial site that aggregates from over 750 contributors. Following its RSS feed is a good way to stay on top of what's new and to discover new tips and analyses.

Cookbook for R (http://www.cookbook-r.com) has recipes for working with data. This is a good source for how to do common operations.

Stack Overflow (http://stackoverflow.com/questions/tagged/r) is a question-and-answer site for programmers. Users post questions, other users post answers, and these get voted up or down by the community, so you can see what is regarded as the best answer as well as alternative approaches.

Remember when you run into a problem that **Google is your friend**. So is DuckDuckGo, if you're not a fan of all that tracking.

Objects and Arithmetic

When you launch R, you will be greeted with a prompt (**>**) and a blinking cursor:

```
>
```

For every command in this tutorial, I will show the prompt, but you should not type it.

R works with objects, and there are many types. Objects store data, and they have commands that can operate on them, which depend the type of data. A single number or string of text is the simplest object and is known as a scalar. [Note that a scalar in R is simply a vector of length one; there is no distinct object type for a scalar, although that is not critical for what follows.]

To give a value to an object, use one of the two assignment operators. Although the equals sign may be more familiar to you now, the arrow (less-than sign, followed by a dash: **<-**) is more common, and you should use it.

```
> x = 3
> x <- 3
```

Read both of those as "assign the value of 3 to the variable x" or more simply, "assign 3 to x". To display the value of any object, type its name at the prompt.

```
> x
```

Arithmetic operators follow standard symbols for addition, subtraction, multiplication, division, and exponents:

```
> x <- 3 + 4
> x <- 2 - 9
> x <- 12 * 8
> x <- 16 / 4
> x <- 2 ^ 3
```

Operators follow the standard order of operations: parentheses and brackets first, followed in order by exponents, multiplication and division, addition and subtraction. Within each of these, operations proceed from left to right. Although you can always force an order of operations, learn this order so that you use the fewest parentheses. Excess parentheses make code harder to read.

Comments are always preceded by a pound sign (**#**), and what follows the pound sign on that line will not be executed.

```
> # x <- 2 * 5; everything on this line is a comment
> x <- 2 * 5 # comments can also be placed after a statement
```

Spaces are generally ignored, but include them to make code easier to read. Both of these lines produce the same result.

```
> x<-3+7/14
> x <- 3 + 7 / 14
```

R is case-sensitive, so capitalization matters. This is especially important when calling functions, discussed below.

```
> x <- 3
> x      # correctly displays 3
> X      # produces an error, as X doesn't exist, but x does
```

Because capitalization matters, you should avoid giving objects names that differ only in their capitalization, and you should use capitalization consistently. One common pattern is camelcase, in which the first letter is lower case, but subsequent words are capitalized (for example, **`pacificOceanCarbon`**). Another common pattern is to separate words in an object's name with an underscore (**`pacific_ocean_carbon`**); this is called snake case. Pick one and be

consistent. Moreover, avoid overly short or cryptic names; it's better to have a longer name that is easily read because it will make your code easier to understand and debug.

You can use the up and down arrows to reuse old commands without retyping them. The up arrow lets you access progressively older previously typed commands. If you go too far, use the down arrow to return to more recent commands. These commands can be edited, so the up and down arrows are good time-savers, even if you don't need to use exactly the same command as before.

Functions

R has a rich set of functions, which are called with their *arguments* in parentheses and which generally *return* a value. Functions are also objects. The maximum of two values is calculated with the **`max()`** function:

```
> max(9, 5)
> max(x, 4) # objects can be arguments to functions
```

Some functions need no arguments, but the parentheses must still be included, otherwise R will interpret what you type as the name of a non-function object, like a vector or matrix. For example, the function **`objects()`**, which is used to display all R objects you have created, does not require any arguments and is called like this:

```
> objects()     # correct way to call a function
> objects       # error: this doesn't call the function
```

Neglecting the parentheses means that you are asking R to display the value of an object called **`objects`**. Since **`objects`** is a function, R will show you the source code for the function.

Functions commonly *return* a value. If the function is called without assigning the result to an object, then the returned value will be displayed on the screen. If an assignment is made, the value will be assigned to that object, but not displayed on the screen.

```
> max(9, 5)                 # displays the result
> myMax <- max(9, 5)        # result is stored, not displayed
```

Multiple functions can be called in a single line, and functions can be nested, so that the result of one function can be used as an argument for another function. Nesting functions too deeply can make the command long, confusing, and hard to debug.

```
> y <- (log(x) + exp(x * cos(x))) / sqrt(-cos(x) / exp(x))
```

To get help with any function, use the **`help()`** function or the **`?`** operator, along with the name of the function.

```
> help(max)
> ?max
> ? max
```

The help pages show useful options for a function, plus background on how the function works, references to the primary literature, and examples that illustrate how the function could be used. Use the help pages.

In time, you will write your own functions because they let you invoke multiple commands with a single command. To illustrate how a simple function is created and used, consider the **pow()** function shown below, which raises a base to an exponent. In parentheses after the word **function** is a list of *parameters* to the function, that is, values that must be input into the function. When you call a function, you supply *arguments* (values) for these parameters. The commands the function performs are enclosed in curly braces. Indenting these statements is not required, but it makes the function easier to read.

```
> pow <- function(base, exponent) {
      result <- base^exponent
      result
}
```

Arguments can be assigned to a function's parameters in two ways, by name and by position. When you assign arguments by name, they can be listed in any order and the function will give the same result:

```
> pow(base=3, exponent=2)  # returns 9
> pow(exponent=2, base=3)  # also returns 9
```

Assigning arguments by position involves less typing, but the arguments must in the correct order.

```
> pow(3, 2)    # returns 3^2, or 9
> pow(2, 3)    # returns 2^3, or 8
```

For **pow()**, the first position is assumed to hold the first parameter in the function definition (**base**) and the second position is assumed to hold the second parameter (**exponent**).

Calling arguments by position is faster, but calling arguments by name is less prone to errors, and it makes the code self-explanatory.

Some functions have default values for some parameters. For example, **pow()** could be written as

```
> pow <- function(base, exponent=2) {
      result <- base^exponent
      result
}
```

which makes the default exponent equal to 2. If you write **pow(5)**, you'll get 25 in return (5 to the default exponent of 2). If you want a different exponent, specify the **exponent** argument, such as **pow(2, 4)**, which would produce 16 (2 to the fourth power). Refer to a function's help page to see which function parameters have default assignments.

Several functions are useful for manipulating objects in R. To show all current objects, use **`objects()`** and **`ls()`**, which are identical.

```
> objects()
> ls()
```

To remove objects, use **`remove()`**. Removing an object is permanent; it cannot be undone.

```
> remove(someObject)
```

You can remove all objects, but since this cannot be undone, be sure that this is what you want to do. Even if you remove all objects, your command history is preserved, so you could reconstruct your objects, but this is likely laborious. To remove all objects, you must combine two commands, the **`ls()`** function and the **`remove()`** function.

```
> remove(list=ls())
```

Vectors

Data can be stored in several different types of objects, and a vector is the most common. A vector is a series of values, where all values have the same type, such as numbers, characters (strings), and logical (Boolean). Vectors are created most easily with the **`c()`** function (*c* as in *concatenate*). Short vectors are easily entered this way, but long ones are more easily imported (see Importing Larger Data Sets below).

```
> x <- c(3, 7, -8, 10, 15, -9, 8, 2, -5, 7, 8, 9, -2, -4, -1)
```

An element of a vector can be retrieved by using an index, which describes its position in the vector, starting with 1 for the first element. The element is placed in square brackets after the name of the vector. To see the fifth element of x, type:

```
> x[5]          # returns 15
```

Multiple consecutive indices can be specified with a colon. For example, to retrieve elements 3 through 10, type

```
> x[3:10]       # returns -8, 10, 15, -9, 8, 2, -5, 7
```

To retrieve non-consecutive elements, create a vector of those indices using the **`c()`** function. Failing to use **`c()`** will cause an error.

```
> x[c(1, 3, 5, 7)] # returns 3, -8, 15, 8
```

These two approaches can be combined in more complex ways. For example, if you wanted elements 3 through 7, followed by elements 5 through 9, you would type:

```
> x[c(3:7, 5:9)]
# returns a vector with -8, 10, 15, -9, 8, 15, -9, 8, 2, -5
```

You can use conditional logic to select elements meeting certain criteria. The logical statement inside the bracket produces a vector of boolean values (TRUE and FALSE), which tell whether a particular vector element should be returned.

```
> x[x>0]       # all positive values in x
> x[x<=2]      # values less than or equal to 2
> x[x==2]      # all values equal to 2
> x[x!=2]      # all values not equal to 2
> x[x>0 & x<3] # values greater than 0 and less than 3
> x[x>5 | x<1] # values greater than 1 or less than 1
```

Any Boolean vector can be negated with the **!** operator, but this is operator is easily overlooked when reading code. Except for **!=**, it is often better to use other operators that are more direct, for example, **x<2** rather than **!x>=2.**

Vectors can be sorted with the **sort()** function. Ascending order is the default.

```
> sort(x)
```

Sorting can be done in descending order by changing one of the default arguments to **decreasing = TRUE** or by reversing the sorted vector with **rev()**. There are often multiple ways to perform an operation in R, and you should choose the simplest one.

```
> sort(x, decreasing=TRUE)
> rev(sort(x))
```

R speeds calculations by using vectorized math. For example, suppose you wanted to multiply all values in a vector by a constant. If you have programmed before, you might think of doing this with a loop, in which you step through each value in a vector and multiply it by a constant. Avoid doing this in R, as loops are slow, so much so that they won't be presented until near the end of this tutorial. Instead, simply multiply the vector by the constant in one line. This example multiplies all of the values in a vector by 2.

```
> x <- c(1, 3, 5, 7, 9)
> 2 * x     # returns 2, 6, 10, 14, 18
```

Likewise, you can use vector arithmetic on two vectors. For example, given two vectors of the same length, it is easy to add the first element of the one vector to the second element of the second vector and so on, producing a third vector of the same length with all of the sums.

```
> x <- c(1, 3, 5, 7, 9)
> y <- c(2, 2, 2, 4, 4)
> x + y     # returns 3, 5, 7, 11, 13
```

Factors

Factors are similar to vectors, but they have a fixed number of levels and are ideal for expressing categorical variables. For example, if you have a series of sites, each representing a specific habitat type, store them as a factor rather than as a vector:

```
> habitats <- factor(c('marsh', 'grassland', 'forest', 'tun-
dra', 'grassland', 'tundra', 'forest', 'marsh', 'grassland'))
```

Inspecting the **habitats** object shows that it is stored differently than a vector.

```
> habitats
[1] marsh     grassland forest    tundra    grassland tundra
[7] forest    marsh     grassland
Levels: forest grassland marsh tundra
```

We can use the **str()** function to display the internal structure of any object:

```
> str(habitats)
 Factor w/ 4 levels "forest","grassland",..: 3 2 1 4 2 4 1 3 2
```

Although the names are displayed when we show the object, the data are actually saved as values 1–4, with each value corresponding to a named factor. For example, the first element is saved as a 3, that is, the third level of the factor (marsh). The second element is saved as a 2, the second level of the factor (grassland), and so on. This is not only a more compact way of saving the data, it also allows efficient ways to use these categories, such as using **tapply()** and **by()** to perform calculations on each category, **split()** to separate the data by category, and **table()** to create tables of the data by category.

If needed, a factor can always be converted to a vector:

```
> y <- as.vector(habitats)
```

Lists

Lists are another type of data structure, also closely related to vectors. Lists hold a series of values, like vectors, but the values in a list can be of different types, unlike a vector. For example, the first element of a list might be a string giving the name of a locality ('**Hueston Woods**'), the second element might be an integer expressing the number of beetle species there (**42**), and the third element might be a boolean value stating whether the locality is old-growth forest (**TRUE**). Lists are made with the **list()** function:

```
> x <- list('Hueston Woods', 42, TRUE)
```

Lists are commonly produced by statistical functions, such as regressions.

To access an element of a list, use double brackets instead of single brackets.

```
> x[[1]]       # returns the first item, 'Hueston Woods'
```

When the elements in a list are labelled, they can be accessed by their name. For example, when you perform a linear regression, a linear regression **lm** object is returned, and it is a list that contains twelve items, including the slope, the intercept, confidence limits, *p* values, residuals, etc. If you ran a regression and assigned the result to **forestRegression**, you could find the names of all the elements in the **forestRegression** list with the **names()** function.

```
> names(forestRegression)
```

One of the elements of the list is called **residuals**, and you can use *dollar-sign notation* to access it by name:

```
> forestRegression$residuals
```

Because residuals is the second element in the list, you could also access it by position with the double-brackets notation:

```
> forestRegression[[2]]
```

In the example above, the **residuals** object is a vector, so you can access individual elements of it by adding brackets. For example, you could get the residual for the third data point in two ways:

```
> forestRegression$residuals[3]         # method 1
> forestRegression[[2]][3]              # method 2
```

Note that the first command is more obvious. Although it requires more typing, accessing a value by name is more readable and less error-prone.

Matrices

Another type of R data structure is a matrix, which has multiple rows and columns of values, all of the same type, like a vector. You can think of a matrix as a collection of vectors, all of the same type and length.

Small matrices are entered easily with the **matrix()** function, in which you specify the data as a vector and the number of rows or columns.

```
> x <- matrix(c(3, 1, 4, 2, 5, 8), nrow=3)
```

This will generate a matrix of 3 rows and therefore 2 columns, since there are six elements in the matrix. By default, the matrix is filled by columns, such that column 1 is filled first from top to bottom, then column 2, etc. Displaying the **x** matrix gives

```
     [,1] [,2]
[1,]    3    2
[2,]    1    5
[3,]    4    8
```

The bracketed values along the right and the top give the row and column positions. For example, the value in the third row and second column could be retrieved with

```
> x[3, 2]     # returns 8
```

Memorize this order: row first, column second. This is standard across R.

A matrix can alternatively be filled by rows, with row 1 filled first, from left to right, then row 2, etc., by changing the default argument to be **byrow=TRUE**:

```
> x <- matrix(c(3, 1, 4, 2, 5, 8), nrow=3, byrow=TRUE)
```

This gives

```
     [,1] [,2]
[1,]    3    1
[2,]    4    2
[3,]    5    8
```

Large matrices can also be entered this way, but importing them is much easier (see Importing Larger Data Sets below).

Matrices also use vectorized math, greatly speeding computation.

The rows and columns can be swapped, such that the rows become the columns, and vice versa by transposing the matrix with the the **t()** function:

```
> y <- t(x)
```

Data Frames

A data frame is like a matrix, but the columns can be of different types. For example, one column might be locality names (strings), other columns might have counts of different species of beetles (numbers), and other columns might be Boolean values that describe properties of the localities, such as whether it is old-growth forest or on protected land. In this way, data frames are to matrices what lists are to vectors.

Nearly always, columns of a matrix or data frame should be the measured variables, and the rows should be cases (localities, samples, etc.).

One way to create a data frame is to combine several vectors, each of which is a different measure of a set of samples. For example, suppose we had vectors for the locality name, the abundance of four different beetle species, and whether the forest was old growth or not:

```
> locality <- c("Athens", "Comer", "Whitehall",
      "Watkinsville", "Commerce")
> beetle1 <- c(14, 23, 16, 19, 2)
> beetle2 <- c(5, 1, 0, 0, 18)
> beetle3 <- c(2, 3, 4, 2, 2)
> beetle4 <- c(0, 0, 2, 3, 20)
```

```
> oldGrowth <- c(TRUE, TRUE, TRUE, TRUE, FALSE)
```

These vectors can be combined into a data frame using the **data.frame()** command:

```
> forests <- data.frame(locality, beetle1, beetle2, beetle3,
beetle4, oldGrowth)
```

It is usually good practice to clean up and remove those vectors after creating the data frame:

```
> remove(locality, beetle1, beetle2, beetle3, beetle4,
oldGrowth)
```

When using a data frame, we often want to know the names of the columns (the variables), which is given by the **colnames()** function.

```
> colnames(forests)
```

If needed, column names can be created or changed by assigning a vector of names:

```
> colnames(forests) <- c('locality', 'S.andrewsi',
'P.pyralis', 'E.pennsylvanica', 'C.bidenticola', 'oldGrowth')
```

Retrieving elements of a data frame is the same as for a matrix, with the first index being the row and the second being the column.

```
> forests[3, 2]      # third row, second column
```

Individual columns can be accessed by name with the **$** operator. For example, this will return the column named **locality** from the **forests** data frame.

```
> forests$locality
```

Dollar-sign notation will not work for selecting multiple columns. To select multiple columns, you will need to pass a vector of column names or column numbers.

```
> forests[ , c('S.andrewsi', 'P.pyralis', 'E.pennsylvanica')]
> forests[ , 2:4]
```

To access individual elements (cases) of column in a data frame, use dollar-sign notation plus-brackets with indices, or specify the row and column numbers. Both of the following return the first four values (rows) of **locality**, which is in column 2 of the data frame.

```
> forests$locality[1:4]
> forests[1:4, 1]
```

If the name of the data frame is long, dollar-sign notation can become cumbersome. To avoid repeatedly typing the name of the data frame, you can use the **attach()** function, which creates copies of all the vectors (columns) in the data frame that can be directly accessed. It is good practice use the **detach()** function to delete these copies when you are done with those data.

```
> attach(forests)
> locality     # locality can be accessed without $ notation
```

```
> detach(forests)
> locality     # returns an error; detach() destroyed the copy
```

Because **attach()** creates copies, changes made to the attached data are not be saved in the original data frame.

The **attach()** and **detach()** commands also work on lists.

A danger in using **attach()** is that it may mask existing variables with the same name, making it unclear which version is being used. **attach()** is best used when you are working with a single data frame. If you must use it in more complicated situations, be alert for warnings about masked variables, and call **detach()** as soon as possible. There are alternatives to **attach()**, such as including the **data** parameter in some functions (such as regressions), which lets you access variables of data frame directly. You can also use the **with()** function, another safe alternative that can be used in many situations.

Sorting a matrix or data frame is more complicated than sorting a vector. It is often best to store the sorted matrix or data frame in another object in case something is typed wrong. The following sorts the data frame A by the values in its third column and stores the result in data frame B.

```
> B <- A[order(A[ ,3]), ]
```

To understand how this complex command or any complex command works, start at the innermost set of parentheses and brackets, and work outwards. At the innermost level, **A[,3]** returns all rows of column 3 from matrix A. Stepping out one level, **order(A[,3])** returns the order of the elements in that column as if they were sorted in ascending order. In other words, the smallest value in column 3 would have a value of 1, the next smallest would have a value of 2, and so on. Finally, these values specify the desired order of the rows in the A matrix, and **A[order(A[,3]),]** returns all the columns with the rows in that order.

This technique of taking apart a complex function call is known as *unpacking*. You should do this whenever a complex function call returns unexpected results or when you need to understand how a complex function call works.

In some cases, your data may be stored in a matrix, but a particular function may require it to be in a data frame for some function (or vice versa). You can try to coerce it into the correct type with the functions **as.data.frame()** and **as.matrix()**.

```
> yDataFrame <- as.data.frame(x)
> yMatrix <- as.matrix(yDataFrame)
```

If you need to find the type of an object, use the **class()** function.

```
> class(y)            # returns "matrix"
> class(yDataFrame)   # returns "data.frame"
> class(yMatrix)      # returns "matrix"
```

Importing Larger Data Sets

For larger data sets, it is easier to enter the data in a spreadsheet (like Excel), check it for errors, and export it into a format you can read into R. The most widely used file formats are comma-delimited and tab-delimited text. In Excel, choose Save As..., then select "Comma-separated values (.csv)" or "Tab-delimited text (.txt)". Both are text-only formats that can be read by almost any application.

To preserve the readability of your data for many years, always save a copy of your data in a text-only format. Binary file formats for programs like Excel change over time, and someday you may not be able to read your old files. This will not happen with plain text files.

To read or write files in R, you need to know the current working directory for R, that is, the directory in which R will look for files. To find the current working directory, use **`getwd()`**.

```
> getwd()
```

It is easiest to work with R by using a single working directory for any given session, and you can set this with the **`setwd()`** function. This directory will hold most or all of the files you will need to read, such as data and source code, and it will be where files are saved by default. R will stay in this working directory unless you specify a different path to a file.

Paths to files and directories are specified differently in UNIX (macOS, Linux) and Windows file systems. This is one of the few areas in which R works differently on the two platforms. If you share code across platforms, any command that accesses the file system will produce an error, so avoid including paths in your code.

Use the **`setwd()`** function to set the working directory of R, with the path as an argument. Because the path is a string, you must wrap it in quotes, and this is true for any string in R. Single-quotes and double-quotes both work; just be consistent. Inside the path, use single forward slashes for UNIX systems:

```
> setwd('/Users/myUserName/Documents/Rdata')
```

Use double back-slashes for Windows:

```
> setwd('C:\\Documents and Settings\\myUserName\\Desktop')
```

Remember to never include calls to **`setwd()`** in R code that you share with someone else. That path will most likely not exist on their computer, and including it will produce errors when they run your code. Likewise, avoid embedding file paths within other commands.

Importing a vector

Once you have set your working directory, you can read a vector into R with the **`scan()`** function. Remember to put your file name in quotes, and include the suffix.

```
> myData <- scan(file='myDataFile.txt')
```

This simple version of this command assumes that the cases are on different lines, but if cases are separated by commas, spaces, tabs, or other such characters, that can be set using the **sep** argument for **scan()**. Likewise, if the variable name is included in the file, or if other information is included at the beginning of the file, those lines can be skipped by setting the **skip** argument.

Importing a data frame

To read a table of values into R and save it as a data frame, use the **read.table()** function. This function assumes that your data are in the correct format, with columns corresponding to variables and rows corresponding to samples. All cells should contain a value; don't leave any empty. If a value is missing, enter **NA** in that cell. Variable names can be as long as you need them to be, and you will want to strike a balance between readability and the amount of typing needed. Variable names do not need to be single words, but any blank spaces will be replaced with a period when you import the data into R.

Of the many options for **read.table()**, you will routinely pay close attention to three, which you can determine by examining the data file in a text editor. First, check to see if there are column names for the variables, that is, whether a *header* is present. Second, check to see if there are *row names* (that is, unique identifiers for each case, such as a locality number) and what column they are in. If row names are present, they are most most commonly in the first column. Third, check to see what separates the columns: tabs or commas are the most common *separators*, also called *delimiters*.

For example, suppose that your data table has the names of its variables in its first row (that is, it has a header), that your samples names are in the first column of the table, and that your values are separated by commas. Given this, you would import your data as follows:

```
> myData <- read.table(file='myDataFile.txt', header=TRUE,
  row.names=1, sep=',')
```

If there is no header, leave out the argument for the **header** parameter, because the default is that there is no header. Likewise, if there are no row names, omit the argument for the **row.names** parameter. If the values are separated by tabs, use **sep='\t'**. There are other commands for opening files, like **read.csv()**, but you should become comfortable opening any table file with **read.table()**; it is the Swiss-army knife of file openers.

If additional explanatory lines about the data are included at the beginning of the file, they can be skipped with the **skip** argument of **read.table()**.

Always verify that an imported data file was read correctly. Although you could simply type the name of the object you assigned the data to, this is not recommended as it can produce voluminous output for large data sets. It is much better to use the **head()** and **tail()** functions to examine the first and last few lines of data.

```
> head(myData)
> tail(myData)
```

Generating Numbers

Writing long, regular sequences of numbers is tedious and error-prone. R can generate these quickly with the **seq()** function. If the increment (**by**) isn't set, it defaults to 1.

```
> seq(from=1, to=100)          # integers from 1 to 100
> seq(from=1, to=100, by=2)    # odd numbers from 1 to 100
> seq(from=0, to=100, by=5)    # counting by fives
```

R can generate random numbers from many statistical distributions. A few common examples include **runif()** for uniform or flat distributions, **rnorm()** for normal distributions, and **rexp()** for exponential distributions. For each, specify the number of random numbers (**n**) and one or more parameters that describe the distribution. For example, this generates ten random numbers from a uniform distribution that starts at 1 and ends at 6:

```
> runif(n=10, min=1, max=6)
```

This creates a hundred random numbers from a normal distribution with a mean of 5.2 and a standard deviation of 0.7:

```
> rnorm(n=100, mean=5.2, sd=0.7)
```

This produces fifty random numbers from an exponential distribution with a rate parameter of 7:

```
> rexp(n=50, rate=7)
```

Basic Statistics

R has functions for calculating simple descriptive statistics from a vector of data, such as:

```
> mean(x)   # arithmetic mean
> median(x) # middle value
> length(x) # number of elements in x (sample size, or n)
> range(x)  # largest and smallest value
> sd(x)     # standard deviation
> var(x)    # variance
```

Many statistical tests are built-in. Some common ones include:

```
> t.test(x, y)            # t-test for equality of means
> var.test(x, y)          # F-test for equality of variance
> cor(x, y)               # correlation coefficient
> lm(y~x)                 # least squares regression
> anova(lm(x~y))          # ANOVA (analysis of variance)
> wilcox.test (x, mu=183) # Mann-Whitney U test
> kruskal.test(x~y)       # Non-parametric ANOVA
> ks.test(x, y)           # Kolmogorov-Smirnov test
```

In addition to generating random numbers, the built-in statistical distributions of R can be used to find probabilities or *p* values, find critical values, and draw probability density functions. These commands are set up consistently, although their parameters may differ. For example, these can be performed with the normal distribution through the **pnorm()**, **qnorm()**, and **dnorm()** functions.

The **pnorm()** function will find the area under a standard normal curve, starting from the left tail, and in the example below, going to a value of 1.96. Note that 1 minus this value would correspond to the *p* value for a Z-test, where Z=1.96:

```
> pnorm(q=1.96)
```

The **qnorm()** function can find the critical value for a given probability. For example, in a two-tailed test at a significance level of 0.05, the following gives the critical value on the right tail.

```
> qnorm(p=0.975)
```

Last, the **dnorm()** function gives the density or height of the distribution at a particular value (at 1.5 below). By calculating this over all values, you can plot the shape of the distribution.

```
> dnorm(x=1.5)
```

Other distributions, such as the uniform, exponential, and binomial distributions follow a similar syntax, with **r** corresponding to a random number, **p** for probability, **q** for critical value, and **d** for density (the height of function at a particular value).

```
> rbinom(n=10, size=15, prob=0.1)   # binomial distribution
> pexp(q=7.2)                       # exponential distribution
> qunif(p=0.95)                     # uniform distribution
> dlnorm(10)                        # log-normal distribution
```

One of the great strengths of R is the incredibly large number of available statistical analyses. Many are built into the base package of R, and a mind-boggling number of them are freely available in user-contributed packages. There's a good chance that if you need to do an analysis, it is already available in R. Installing packages is described under Customizing R below.

Efficient computation

R's vectorized math eliminates the need for looping through vectors and matrices to perform many calculations. Such vectorized calculations are commonly orders of magnitude faster than using loops.

Another set of tools for efficient computation are the **apply()** family of functions. The most basic of these is **apply()**, which is used when you want to perform a function on every row or every column of a matrix or data frame and get the results as a vector. For example, the following calculates the median for every column of a matrix and returns a vector with those medians.

```
> medians <- apply(x, MARGIN=2, FUN=median)
```

Set **MARGIN** to 1 to calculate the median for every row instead. The easy way to remember whether to use 1 or 2 is to think of how rows and columns are specified for matrices and data frames: it is [row, column], so rows are 1, and columns are 2. The argument to **FUN** is the name of the function to be applied; this could be a function that you create.

The functions **sapply()** and **lapply()** are similar, but they are used when a function needs to be applied to all elements of a vector or a list. **lapply()** returns a list, and **sapply()** returns a vector or matrix. One possible use of both would be to have a series of similar data sets stored as elements of a list, then call **lapply()** or **sapply()** once to perform some function on each of the data sets. This would be simpler than looping through the list and performing the calculations separately on each data set. It is also better than storing the data sets as separate objects and performing the calculations separately on each object. By creating your own function as a wrapper around other functions, you could perform a complex series of analyses, including making and saving plots for each data set in one command.

Plots

A real strength of R is the quality of its plotting routines. Many types of plots are available, and complex plots can be produced. R's plots are far superior to those made in spreadsheets like Excel, which are commonly saddled by unnecessary shading, grid lines, and 3-D effects.

When calling an R plotting function, the plot will automatically be generated in the active plotting window, if one already exists. If no plot window exists, a new one will be created. Use **dev.new()** to create a new plot window, as this command will work on any platform. Avoid using the platform-specific ways of creating new plot windows, such as **quartz()** for macOS, **X11()** for Linux, and **windows()** for Windows, because these will generate errors if your code is run on a different platform.

The height and width (in inches) of a plot window can be specified, or they can be left off for the default window.

```
> dev.new(height=7, width=12)  # custom size, in inches
> dev.new()                    # default size
```

Histograms

Histograms, also called frequency distributions, are generated with the **hist()** function. Calling **hist()** with no arguments except the data generates a default histogram.

```
> hist(x)
```

Histograms can be customized by changing the number of bars or divisions with the **breaks** parameter. Oddly for R, this is only a suggestion, and it may not be honored.

```
> hist(x, breaks=20)
```

The divisions in the histogram can be forced by specifying the breaks as a vector.

```
> divisions <- c(-2.5, -2, -1, 0, 1, 1.5, 2, 2.5)
> hist(x, breaks=divisions)
```

SCATTERPLOTS

Scatterplots (x-y plots) are generated with the **plot()** function, with **x** being the variable on the horizontal axis and **y** being the variable on the vertical axis.

```
> plot(x=vegetationDensity, y=beetleAbundance)
```

It is common to call these arguments by position only, putting the x-axis variable first and the y-axis variable second.

```
> plot(vegetationDensity, beetleAbundance)
```

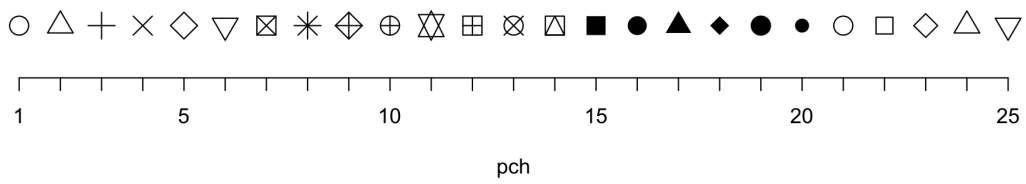
Labels for the x and y axes can be set with the **xlab** and **ylab** parameters, and the plot title can be set with the **main** parameter. Remember to put strings in quotes.

```
> plot(Fe, pH, main='Acid mine drainage', xlab='Fe (ppt)',
ylab='pH')
```

The **pch** parameter can be set in **plot()** to specify a particular type of symbol. The value of 16 is a filled circle, an easily visible symbol that you should use. For simplicity, I will just use **x** and **y** for the following examples, but you should always use informative names for your objects.

```
> plot(x, y, pch=16)
```

Twenty-five plotting symbols are available in **pch**:



By changing the argument to **type**, you can control whether lines or points are plotted.

```
> plot(x, y, type='p') # points only
> plot(x, y, type='l') # connecting lines only, no points
> plot(x, y, type='b') # both points and lines
> plot(x, y, type='n') # no points, only axes and labels
```

Symbol colors can be set with the **col** parameter. To see the full list of 657 named colors in R, use the **colors()** function. Colors can also be specified by RGB (red-green-blue) value, as used in creating colors for web pages.

```
> plot(x, y, pch=16, col='green') # small green filled circles
> colors()                        # lists all named colors
> plot(x, y, col='#00CCFF')       # 00CCFF is pale blue
```

Equations and special symbols can be used in figure labels and as text added to figures. See the help page for **plotmath()** for more details. Adding equations and special symbols commonly uses three functions. The function **text()** places additional text onto a figure, **expression()** creates an object of type "expression", and **paste()** is used to concatenate items. The help pages for these functions are useful for seeing the basic ways for combining these useful functions. The first example puts the square root of x as the y-axis label, and the second example shows the standard symbol for the oxygen isotope ratio ($\delta^{18}O$) on the y-axis.

```
> plot(x, y, ylab=expression(paste(sqrt(x))))
> plot(x, y, ylab=expression(paste(delta^18, 'O')))
```

Special symbols can also be added using Unicode symbols in **expression()** and **paste()**. For example, to create a label with a per-mil symbol, such as $\delta^{13}C$ (*‰* PDB), use the Unicode character 2030; all Unicode character numbers must be preceded with \U. Symbols for thousands of other characters can be found at https:\\home.unicode.org.

```
> expression(paste(delta^13, 'C (\U2030 PDB)'))
```

Adding to plots

Text annotations can be added with the **text()** function, where the **x** and **y** arguments describe the coordinates of the annotation, and **labels** specifies what text will be added. The coordinates are identical to those used for placing points on the plot.

```
> text(x=10, y=2, labels=expression(sqrt(x)))
```

Lines can be added to a plot with the **abline()** function. You can make horizontal and vertical lines, as well as lines with a specified slope and intercept.

```
> abline(h=5)                   # horizontal line at y=5
> abline(v=10)                  # vertical line at x=10
> abline(a=intercept, b=slope) # line with slope and intercept
> abline(lm(y~x))               # fitted regression line
```

Shapes can also be added to the plot, and these can be customized with particular line types (dashed, dotted, etc.), colors of lines, and colors of fills. See **lty** in **par()** for customizing line types. Become familiar with the help page for **par()**; it describes the many ways that plots can be customized.

The **rect()** function draws a box with the coordinates of the left, bottom, right, and top edges.

```
> rect(xleft=2, ybottom=5, xright=3, ytop=7)
```

The **arrows()** function draws an arrow, with the first two arguments specifying the x and y coordinate of the origin of the line, and the third and fourth arguments specifying the coordinates of the arrowhead tip.

```
> arrows(x0=1, y0=1, x1=3, y1=4)
```

The **segments()** function is identical to the **arrows()** function, except that no arrowhead is drawn.

```
> segments(x0=1, y0=1, x1=3, y1=4)
```

Similarly, lines connecting data points can be added with **lines()**, and polygons can be added with and **polygon()**. A box can be drawn around the entire plot with **box()**.

BUILDING A COMPLEX PLOT

Complex plots are typically built in stages, with groups of points added sequentially, and with custom axes, text, and graphic annotations added separately. To build a plot in stages, set **type='n'** to prevent data points from being plotted, and set **axes=FALSE** to prevent the axes from being drawn. Specify the values to be plotted along the x and y axes as the first two arguments.

```
> plot(x, y, type='n', axes=FALSE)
```

This will produce a plot window with only the x and y axis names: no points, ticks, boxes, axis labels, etc. Points can be added to the current plot with the **points()** function. Only those points that are visible within the current limits of the axes will be shown. The limits of the axes are based on the values used in the data supplied to the **plot()** function.

```
> points(x, y)
```

Axes can be added individually with the **axis()** function.

```
> axis(1, at=seq(-1.0, 2.0, by=0.5), labels=c('-1.0', '-0.5',
  '0.0', '0.5', '1.0', '1.5', '2.0'))
```

The initial argument specifies the axis to be drawn, with 1 indicating the x axis and 2 indicating the y-axis. The **at** parameter specifies where the tick marks should be drawn, and the **labels** parameter specifies the label to be displayed next to each tick mark. Values given to **labels** must be in quotes, as they are strings.

Here is an example of building a complex plot in parts. The commands that are shown were preceded by a good bit of experimentation (not shown) with different parts of the plot to determine the order in which these commands would be given. The data are read in, and **attach()** is used to make the variables accessible without dollar-sign notation.

```
> waves <- read.table(file='waves2.txt', header=TRUE, sep=',')
> attach(waves)
```

Plotting begins by making a window with an aspect ratio appropriate for the data being plotted. This is a time series, so we would like the plot to be relatively wide to emphasize the changes over time.

```
> dev.new(height=4, width=10)
```

Because this is a scatterplot, the basic plot is built with the **plot()** command, specifying the x and y data as the first two arguments, the titles, the axis labels, and the limits of the y data range (**ylim**). The type is set to **n**, because the data will be added subsequently.

```
> plot(juliandate, SwH, main='Station 46042 - Monterey - 27 NM
  West of Monterey Bay, California', xlab='',
  ylab='Significant Wave Height (m)', axes=FALSE,
  type='n', ylim=c(0, 6))
```

A light gray rectangle is added to emphasize a particular portion of the data. This is added first, so that the data will be added on top of the rectangle. If the data were drawn in the **plot()** command, this rectangle would cover the data. The **gray()** command is used to set the color to a shade of gray; with 0.0 being black and 1.0 being white.

```
> rect(12, 0, 18, 6, col=gray(0.9), border=gray(0.9))
```

Two lines are added to emphasize divisions of the data along the x-axis. Note that a vector of two values is passed as the **v** argument, which is simpler than drawing each line with a separate call to **abline()**.

```
> abline(v=c(1, 32), col=gray(0.5))
```

Because there are so many data points, the data are added as a series of lines (**type='l'**) that connect individual values, rather than showing a symbol for each data point. Date (**juliandate**) is plotted along the x-axis, and wave height (**SwH**) is plotted along the y-axis, matching the arguments in the **plot()** call.

```
> points(juliandate, SwH, type='l')
```

The x-axis is built in two stages. Closely spaced short ticks are added first, with **tcl** specifying the tick length.

```
> axis(1, at=seq(0, 46, by=1), labels=FALSE, tcl=-0.3)
```

Widely spaced longer ticks are added next, and these are labelled:

```
> axis(1, at=seq(0, 45, by=5), labels=c('30 Nov',
  '5 Dec', '10', '15', '20', '25', '30', '4 Jan', '9', '14'))
```

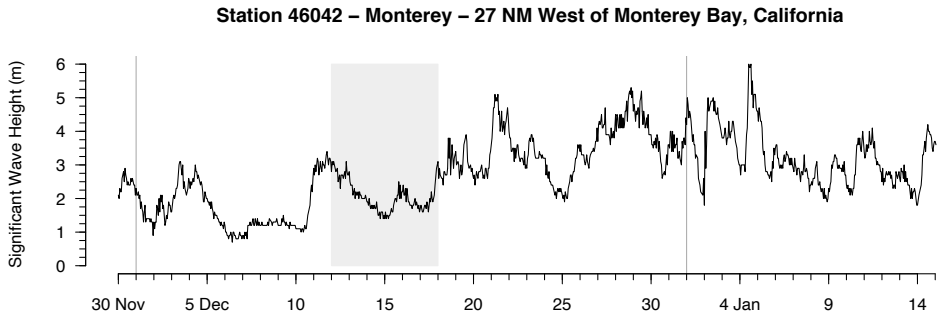
Likewise, the y-axis is built in two stages, beginning with closely spaced short ticks:

```
> axis(2, at=seq(0, 6, by=0.25), labels=FALSE, tcl=-0.3)
```

These are followed with widely spaced long ticks and their labels:

```
> axis(2, at=seq(0, 6, by=1), labels=
  c('0', '1', '2', '3', '4', '5', '6'), las=1)
```

This is the final plot. The advantage to making the plot this way, instead of making a simple plot and adding to it in a program like Illustrator, is that if you make a mistake, you can fix it, re-run all the commands and you'll have your final plot, saving you the laborious steps of making all the changes in Illustrator.



Multiple plots on a page

Multiple plots can be drawn on the same page by using the **`par()`** function and setting the argument to **`mfrow`**. Each successive call to **`plot()`** will fill the panel from left to right, or from the top to the bottom, starting in the upper left corner.

```
> par(mfrow=(c(3, 2)))
```

This example will produce a page with areas for six plots in 3 rows and 2 columns. Each plot is added subsequently. Note that once one plot is finished and another plot is started, the first plot can no longer be modified.

Saving and Exporting Results

Commands

From the R console window, you can copy and paste the entire R session into a text file. This will save a record of your commands and their displayed results. This is an easy and simple way to save your work. Making a habit of this allows you to replicate your work, to build on it subsequently, and to apply your analyses to other data sets.

You can edit this text file so that it consists of just the commands you need. First, delete all the commands and their output that produced results you don't want to keep. Next, delete all

lines of results, leaving only lines of commands. Last, delete the **>** prompts at the beginning of each line, and any **+** marks at the beginnings of lines. If you copy and paste this code into a new R session, it should run without errors or warnings. If you get errors or warnings, edit the code and repeat this process until it runs clean. Save this file for a permanent record of your analysis.

PLOTS

On macOS, graphics produced in R can be saved as pdf files by selecting the window containing the plot, and choosing Save or Save As from the File menu. In Windows, graphics can be saved this way to metafile, postscript, pdf, png, bmp, and jpeg formats. The pdf format is recommended because it can be edited in Adobe Illustrator. Raster (bitmap) formats such as png, bmp, and jpeg should be avoided because they cannot be edited and because they often suffer from compression artifacts that can make them look fuzzy.

Graphics can also be built and saved to a file from the command line. Use the **`pdf()`** function to open a pdf file, just as you would open a plotting window with **`dev.new()`**. Run all of your plotting commands, followed by **`dev.off()`** to close the pdf file. Note that your plot will not be displayed while you do this, although it is being written to a file.

```
> pdf(file='myPlot.pdf', height=7, width=7)
> plot(x, y)
>  # ... all other plotting commands
> dev.off()
```

A typical strategy is to first develop a plot onscreen to refine the order of the plot commands. Once these are established, then open a pdf file with **`pdf()`**, rerun all of the plotting commands, and close the pdf file with **`dev.off()`**.

OBJECTS

Any objects generated in R, such as functions, vectors, lists, matrices, data frames, etc. can be saved to a file by using the command **`save.image()`**. Although this will not save any graphics or the command history, it is useful for saving the objects produced by your analyses so they will not have to be regenerated in future R sessions. I recommend deleting unnecessary objects before saving the image, particularly temporary or test objects generated when you experiment with code. If your work involves long-running computations to build an object, use **`save.image()`** so that it you don't have to re-run those analyses the next time you work. The image should have the suffix .RData.

```
> save.image(file = 'myResults.RData')
```

When you resume your work, reload those objects with **`load()`**, allowing you to continue where you left off.

```
> load(file = 'myResults.RData')
```

Programming in R

If you have previous programming experience, get a copy of ***The Art of R Programming***; it will save you much time.

When coming from other programming languages, one of the largest conceptual hurdles is learning to avoid loops, which can dramatically slow your calculations, especially when loops or nested. Use vectorized operations and the **`apply()`** family of functions, as well as other similar types of functions to avoid loops.

Even so, some problems require loops.

R's **`for`** loop is a generalized loop like the *for* loop in C or the *do* loop in FORTRAN. If you have programmed before, the syntax is intuitive and this will likely be the most common loop that you use. Note the use of curly braces if there is more than one statement inside the loop. There are various styles as to where the curly braces may be placed to help readability of the code, but their placement does not affect the execution. Indenting the code inside the loop helps to make the contents of the loop more apparent. Although indenting isn't required, it is good form to do it.

Here is an example of calculating variance as you might in C or Fortran, that is, with loops

```
> x <- rnorm(10000000)   # simulating a very large data set

> sum <- 0
> for (i in 1:length(x)) sum <- sum + x[i]
> mean <- sum/length(x)             # mean() would be faster
> sumsquares <- 0
> for (i in 1:length(x)) {
      sumsquares <- sumsquares + (x[i]-mean)^2
  }
> variance <- sumsquares / (length(x)-1)
```

On an older test computer, this ran in 16.3 seconds. When vectorized math was used instead of loops,

```
> variance <- sum((x-mean)^2)/(length(x)-1)
```

it ran in a mere 0.23 seconds, a dramatic improvement (that is, 70 times faster). Using the built-in **`var()`** function, which has additional optimizations, it took only 0.082 seconds (almost 200 times faster). Note, however, that on a newer computer these three times were 1.30, 1.24, and 3.31 seconds: the **`var()`** function was the slowest! This raises an important issue: when optimizing your code, always measure the execution time (see Timing Your Code below) to be sure that you are actually making the code run faster.

R has other typical flow-control statements, including **if** and **if-else** branching, **while**, **repeat**, and **for** loops, **break** statements, and so on.

The **if** test will let you run one or more statements if some condition is true. If there are multiple statements, enclose them in curly braces.

```
> if (x < 3) y <- a + b
```

There is **if-else** test can let you act on a variety of conditions. Enclose the code for each condition in curly braces.

```
> if (x < 3) {
    y <- a + b
    z <- sqrt(x)
  } else {
    y <- a - b
    z <- log(a + b)
  }
```

The **while** loop will execute as long as the condition is true.

```
> while(x >= 0) {some expressions}
```

Note that if this condition never becomes false, this will generate an infinite loop.

TIMING YOUR CODE

Critical to optimizing your code is timing it. To do this, add one line before the code you wish to measure:

```
> startTime <- Sys.time()
```

and add these two lines after the code to be measured:

```
> endTime <- Sys.time()
> endTime - startTime
```

Copy all of these commands (the line above, your code, the two lines above) into R in one step. When your code executes, it will report how long it took.

Always measure your code before you optimize it to understand the effects of your optimizations. In some cases, code that seems like it ought to be faster might not be, and sometimes a loop is faster than the alternative, as shown with variance above. Be aware that some R users are dogmatic about avoiding loops. Instead of being dogmatic, measure your code and know which solution is fastest.

Customizing R

Packages

A great advantage to the open-source nature of R is that users have contributed an astonishing number of packages for solving a vast array of data analysis problems. For example, there are packages specifically directed to non-parametric statistics, signal processing, and ecological analyses. Using these packages requires two steps.

First, they must be *installed* on your system, and this is a one-time step. Most packages can be installed from the R site: http://CRAN.R-project.org/. On macOS, packages can be installed through the package installer under the Packages & Data menu. To check which packages have been installed and are available to be loaded, use the **`library()`** function.

```
> library()
```

Second, once a package is installed, it must be *loaded* by calling the **`library()`** function with the name of the library as an argument. The library is an object, not a string, so do not enclose its name in quotes. For example, this is how you would load the vegan package, a useful package for ecological analyses.

```
> library(vegan)
```

A library must be loaded before any R session in which you would like to use its functions.

Be aware that the correctness of code in these packages may not have been tested. Use trusted packages rather than obscure ones, and verify your results.

Packages are periodically updated, just as R is, so check to make sure that R and your packages are current. To check on the status of your packages, run **`packageStatus()`**, and to update your packages, run **`update.packages()`** and follow the prompts.

Function files

As you become familiar with R, you will write your own functions. To have easy access to these, save your functions to a text file. These can be loaded into an R session with the **`source()`** function. The r file suffix can be lower-case or upper-case.

```
> source('myfunctions.r')
```

I usually create one source file for each project, more for complex projects, and **`source()`** lets these functions be loaded in one step, rather than copying and pasting the code.

If you find yourself typing the same commands at the beginning or end of every R session, you can set up R to automatically execute those commands whenever it starts or quits by editing your **`.First()`** and **`.Last()`** functions. Common commands to place in your **`.First()`** function include loading any commonly used libraries or data sets, setting

the working directory, setting preferred graphics arguments, and loading any files of R functions. Here is an example of a **`.First()`** function:

```
.First <- function() {
   library(utils)
   library(grDevices)
   library(graphics)
   library(stats)
   library(vegan)
   library(MASS)
   library(cluster)
   setwd('/Users/myUserName/Documents/Rdata')
   source('myfunctions.r')
}
```

This function loads a series of libraries routinely needed for analyses, sets the working directory, and loads a set of functions written by the user.

On macOS and other UNIX/LINUX systems, the **`.First()`** and **`.Last()`** functions are stored in a file called .Rprofile, which is an invisible file located at the root level of a user's home directory (for example, /Users/username/). On Windows, these functions are stored in Rprofile.site, which is kept in the C:\Program Files\R\R-n.n.n\etc directory. If the .Rprofile or Rprofile.site file doesn't exist on your system, you will need to create one as a plain text file using a text editor. On macOS, Linux, and other UNIX-based systems, be sure to include the period at the beginning of the file name.