

Introduction to R

Nathaniel E. Helwig

University of Minnesota

1 What is R?

R is a free and open source software environment and programming language for statistics.¹ R was created by Ross Ihaka and Robert Gentleman at the University of Auckland (in New Zealand), and is based on the S language that was created by John Chambers at Bell Laboratories. Unlike S (which is for profit), the R software is a part of the GNU General Public License, which means that it is available to anyone free of charge. The R statistical environment works for most standard operating systems (i.e., Mac, Windows, Linux), but certain features (particularly related to graphics and parallel computing) are only available on certain operating systems. R is currently maintained by the “R Core Team”, which is composed of some of the world’s best computational and applied statisticians.

When you download and install R, you get a collection of basic packages (or “libraries”) that can be used to implement several common data manipulations, graphical displays, and statistical models. However, the real power of R comes in the form of the Comprehensive R Archive Network (CRAN)², which is a repository where individuals can upload their own R packages for others to use. Most modern developments in statistics are first implemented and released in R, and it often takes other (for profit) softwares years—or maybe decades—to implement new statistical methods that you can download (for free) from CRAN. To see a full list of all of the R packages currently available on CRAN, click [here](https://cran.r-project.org/). However, the name of an R package may not be particularly descriptive of the packages’ capabilities. To figure out how to do something in R, you can just Google “How to I [fill in the blank] in R?”

¹<https://www.r-project.org/>

²<https://cran.r-project.org/>

2 What is RStudio?

RStudio is an “integrative development environment” (IDE) for R that is freely available.³ In its most basic sense, you can use RStudio as a graphical user interface (GUI) for developing and running R code. Note that R comes with a GUI for Mac and Windows, but the default R GUI is quite minimalistic—particularly on Windows machines. The RStudio IDE has the benefit of allowing you to develop R code (in the Editor), run R code (in the Console), see the objects in your R environment (in the Workspace), review past R code that you’ve run (in the History), and view various other information (e.g., related to File paths, created Plots, installed Packages, and Help files) all within the same user-friendly application.

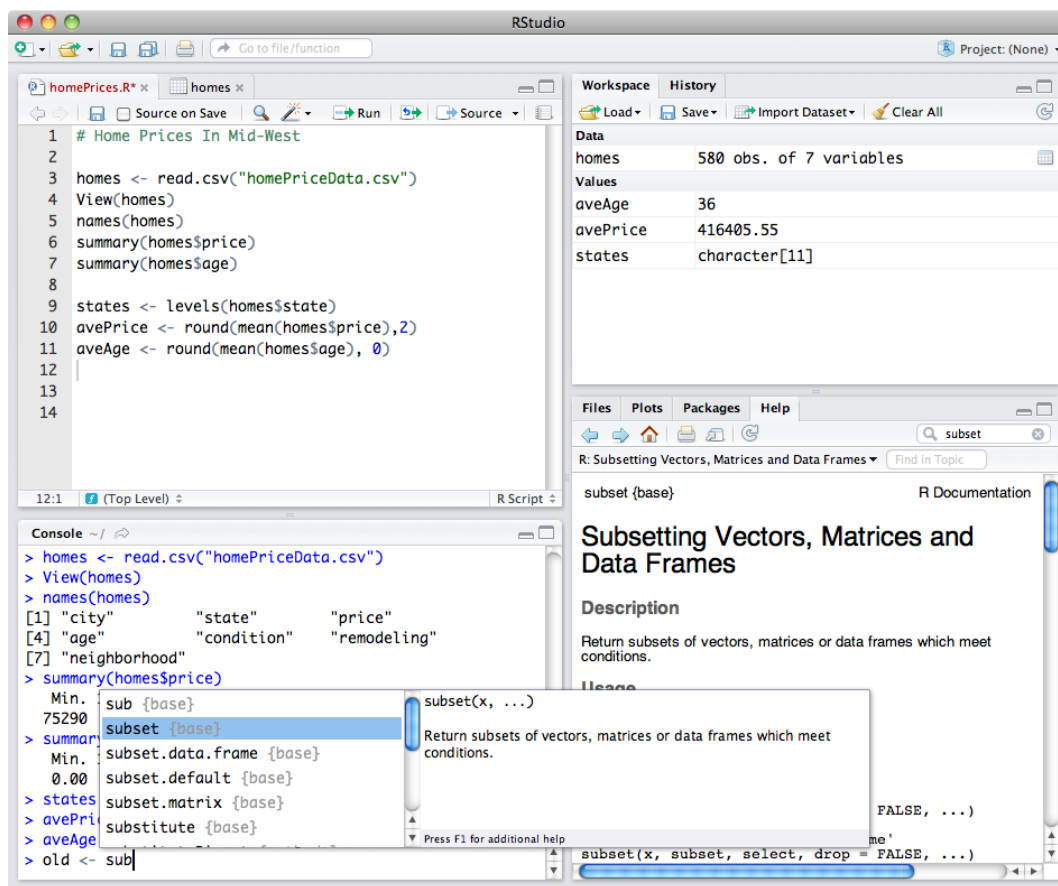


Figure 1: The RStudio GUI for a Mac. <https://rstudio.com/products/rstudio/features/>

³<https://rstudio.com/>

3 Using the R Console

Within the R Console, the symbol `>` comes before R code that is executed, and the symbol `#` denotes that the following words are comments. When R produces output in the Console, the output prints a number inside a box (e.g., `[1]`) that is used to index the output.

```
> # addition and subtraction
> 3 + 2
[1] 5
> 3 - 2
[1] 1

> # multiplication and division
> 3 / 2
[1] 1.5
> 3 * 2
[1] 6

> # exponents
> 3^2
[1] 9
> 2^3
[1] 8
```

Note that the `[1]` that proceeds each piece of output gives the index of the first output for each line of output. Since there is only one number, we only see a single index (i.e., `[1]`).

```
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
[16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Note that `[1]` corresponds to “a” (the first output on the first line) and `[16]` corresponds to “p” (the first output on the second line—which is the 16th output letter).

4 Logical Operators in R

The R language has all of the standard logical operators that you would find in any typical programming language.

```
> # less than / less than or equal to
```

```
> 2 < 2
```

```
[1] FALSE
```

```
> 2 <= 2
```

```
[1] TRUE
```

```
> # greater than / greater than or equal to
```

```
> 2 > 2
```

```
[1] FALSE
```

```
> 2 >= 2
```

```
[1] TRUE
```

```
> # exactly equal to
```

```
> 2 == 2
```

```
[1] TRUE
```

```
> # not equal to
```

```
> 2 != 3
```

```
[1] TRUE
```

```
> # x OR y
```

```
> (2 < 3) | (2 > 3)
```

```
[1] TRUE
```

```
> # x AND y
```

```
> (2 < 3) & (2 > 3)
```

```
[1] FALSE
```

5 Special Values in R

In R, there are a handful of symbols that relate to special values

```
> # mathematical constants
```

```
> pi
```

```
[1] 3.141593
```

```
> exp(1)
```

```
[1] 2.718282
```

```
> # infinity
```

```
> Inf
```

```
[1] Inf
```

```
> Inf + 1
```

```
[1] Inf
```

```
> # empty data
```

```
> NULL
```

```
NULL
```

```
> NULL + 1
```

```
numeric(0)
```

```
> # missing data
```

```
> NA
```

```
[1] NA
```

```
> NA + 1
```

```
[1] NA
```

```
> # machine epsilon
```

```
> .Machine$double.eps
```

```
[1] 2.220446e-16
```

6 Assigning Values in R

In R, you can assign values to objects using the syntax

```
object <- value
```

where `object` is the object's name and `value` is the value that you assign to the object. Note that the assignment symbol `<-` is really two symbols in a row (`<` followed by `-`).

```
> # assigning values to objects
> x <- 3
> y <- 2

> # calculations using the objects
> x + y
[1] 5
> x * y
[1] 6
> x^y
[1] 9
```

7 Using Functions in R

R is a function based language where a “function” takes in some input and produces some output. Functions in R follow Vegas rules: what happens in a function, stays in a function. In other words, the manipulations that you make to data inside of a function will not affect the data outside of the function (i.e., in your R Workspace).

Assuming that you want to assign the output to an R object (e.g., instead of simply printing the output in the R Console), you would use the syntax

```
output <- function(input, ...)
```

where `output` is the object being output by the function, `function` is the name of the function that you are calling, `input` is the object being input to the function, and `...` denotes additional inputs (or arguments) for the function.

R has many built-in functions for accomplishing all sorts of tasks. The real beauty of R is that you can create your own (user defined) functions, which allows you to automate any sort of data preprocessing, analysis, or visualization methods that you use for your research. We will cover how to write your own R function in a later section, but first we will review a few of the built-in functions to give you a sense of how functions in R work.

```
> # combine values
> c(1, 3, -2)
[1] 1 3 -2
> c("a", "a", "b", "b", "a")
[1] "a" "a" "b" "b" "a"

> # sort values
> sort(c(1, 3, -2))
[1] -2 1 3
> sort(c(1, 3, -2), decreasing = TRUE)
[1] 3 1 -2

> # minimum and maximum
> min(c(1, 3, -2))
[1] -2
> max(c(1, 3, -2))
[1] 3

> # sum and mean
> sum(c(1, 3, -2))
[1] 2
> mean(c(1, 3, -2))
[1] 0.6666667
```

Note that the second use of the `sort` function uses the argument `decreasing = TRUE` to return the values from largest to smallest (instead of from smallest to largest, which is the default). You can see the help file of any function by typing a question mark before the function's name (e.g., `?sort`), which will show you the available arguments.

On the previous page, I just printed the function's output to the R Console (instead of saving it as an object), which is not how you will use most functions in R. Typically, you are using a function because you want to produce some output that you will use again later (by inputting the output into another function). For example:

```
> # define x and y
> x <- c(0, 2, 4, 6, 8)
> y <- c(1, 3, 5, 7, 9)

> # combine as columns
> cbind(x, y)
      x y
[1,] 0 1
[2,] 2 3
[3,] 4 5
[4,] 6 7
[5,] 8 9

> # combine as rows
> rbind(x, y)
  [,1] [,2] [,3] [,4] [,5]
x    0    2    4    6    8
y    1    3    5    7    9

> # plot x versus y
> plot(x, y)
```

The above example uses the combine function `c()` to create the vectors `x` and `y`, which are then used as input to other functions, i.e., the `cbind`, `rbind`, and `plot` functions. Note that R can produce beautiful publication quality graphics, which are customizable in an infinite number of ways. I will not cover the data visualization capabilities of R in this document,⁴ but we will take a data visualization detour one day.

⁴See <http://stat.umn.edu/~helwig/publications.html> for some examples.

8 Object Classes in R

R is an object oriented programming language where an “object” is a generic term to describe something in R. For example, numbers, vectors, datasets, functions, etcetera are all considered objects in R. Each object **X** in the R language has an associated “class”, which indicates the type of object that **X** represents.

Classes for numbers in R:

```
> # numeric class
> x <- c(1, 3, -2)
> class(x)
[1] "numeric"
```

```
> # integer class
> x <- c(1L, 3L, -2L)
> class(x)
[1] "integer"
```

The “numeric” class is the default class for (double precision) numeric values in R. The “integer” class is a special class for numbers that are integers, which are denoted by placing a capital letter L after the number. Assuming that it is permissible, you can easily convert between the numeric and integer classes such as:

```
> # convert from numeric to integer (and back)
> x <- c(1, 3, -2)
> class(x)
[1] "numeric"
> x <- as.integer(x)
> class(x)
[1] "integer"
> x <- as.numeric(x)
> class(x)
[1] "numeric"
```

Classes for non-numerics in R:

```
> # character class
> x <- c("a", "a", "b")
> x
[1] "a" "a" "b"
> class(x)
[1] "character"
> levels(x)
NULL

> # factor class
> x <- factor(c("a", "a", "b"))
> x
[1] a a b
Levels: a b
> class(x)
[1] "factor"
> levels(x)
[1] "a" "b"

> # ordered factor class
> x <- factor(c("a", "a", "b"), ordered = TRUE)
> x
[1] a a b
Levels: a < b
> class(x)
[1] "ordered" "factor"
> levels(x)
[1] "a" "b"
```

The “character” class is the default class for non-numerics in R. The “factor” class informs R that the characters should be interpreted as a categorical variable (e.g., in a linear model) where the unique values are the “levels”, which can be unordered (default) or ordered.

Classes for data in R:

```
> # matrix class
> x <- c(1, 3, -2)
> y <- c(2, 0, 7)
> z <- cbind(x, y)
> z
      x y
[1,]  1 2
[2,]  3 0
[3,] -2 7
> class(z)
[1] "matrix" "array"
> class(z[,1])
[1] "numeric"
> class(z[,2])
[1] "numeric"

> # data.frame class
> x <- c(1, 3, -2)
> y <- c("a", "a", "b")
> z <- data.frame(x, y)
> z
      x y
1  1 a
2  3 a
3 -2 b
> class(z)
[1] "data.frame"
> class(z$x)
[1] "numeric"
> class(z$y)
[1] "character"
```

```

> # list class
> x <- c(1, 3, -2)
> y <- c("a", "a", "b", "b")
> z <- list(x = x, y = y)
> z
$x
[1] 1 3 -2

$y
[1] "a" "a" "b" "b"

> class(z)
[1] "list"
> class(z$x)
[1] "numeric"
> class(z$y)
[1] "character"

```

The “matrix” class is the default class for storing collections of vectors that are (i) of the same length and (ii) of the same class (e.g., all numeric or all character). The “data.frame” class is the default class for storing collections of vectors that are (i) of the same length and (ii) possibly of different classes (e.g., some numeric and some character). Finally, the “list” class is the generic class for storing collections of objects of different classes—the elements of a list could be anything (e.g., vectors, matrices, data frames, functions, or other lists).

- The matrix class is more specific than the data.frame class.
- The data.frame class is a special case of the list class.
- The list class is the most general class in the R language.

When you input your data into R, you will likely want to ensure that (i) it is being stored as a data frame, and (ii) each column of the data frame has the class that you have intended (e.g., numeric, integer, character, or factor). Otherwise bad things may happen...

9 Object Oriented Programming

When inputting data into R, it is very important to ensure that it is being stored as the correct class because how R interacts with the data will depend on the class of the object. More specifically, some R functions are only applicable to objects of a particular class, and some R functions perform different operations depending on the class of the input object. This is what I mean when I say that R is an “object oriented” programming language. Consequently, as a user of R, it is your responsibility to ensure that you’ve correctly informed R how your data should be interpreted.

For example, suppose that you (made the unfortunate⁵ choice to) code your data such that 1 = female, 2 = male, and 3 = other. If you input such data into R, you need to make sure that you inform R that the integer values are actually meant to represent the levels of a factor variable. Otherwise some strange things can/will happen...

```
> # print and summary (class customized)
> x <- rep(c(1, 2, 3), each = 3)
> y <- factor(x, levels = 1:3, labels = c("female", "male", "other"))
> print(x)
[1] 1 1 1 2 2 2 3 3 3
> print(y)
[1] female female female male   male   male   other  other  other
Levels: female male other
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     1       1       2       2       3       3
> summary(y)
female  male  other
     3     3     3
```

Note that the `print` and `summary` functions are class customized, and would produce nonsensical results (e.g., calculating the mean or median) if you don’t inform R that the integers 1, 2, and 3 should be interpreted as female, male, and other (respectively).

⁵This is an unfortunate thing to do because now you have to keep a data key to remember how your data are coded—which is an antiquated way of doing things. It would be better to use characters, which can convey meaning without needing a data key (e.g., use “female”, “male”, and “other” instead of 1, 2, and 3).

10 If/Else Statements in R

The “if/else” statement is one of the standard functionalities of any programming language. This statement allows the user to implement one collection of code IF a particular condition is met, and implement a different collection if the condition is not met. The basic structure of an IF/ELSE statement in R has the form

```
if(condition) {
  # some R code
} else {
  # more R code
}
```

You can combine multiple IF/ELSE statements into a longer version:

```
if(condition1) {
  # some R code
} else if(condition2) {
  # more R code
} else {
  # even more R code
}
```

Here is a simple example:

```
> x <- "cat"
> if(x == "dog"){
+   y <- "bone"
+ } else {
+   y <- "yarn"
+ }
> y
[1] "yarn"
```

Note that the + signs are NOT a part of the R code. These indicate that you are entering a multiline statement in the R Console.

11 For Loops in R

For loops (called “do loops” in some languages) are another one of the standard functionalities of any programming language. The for loop allows you to repeatedly implement the same R code with the loop index sequentially changing each time. Note that for loops should *only* be used if you cannot vectorize (i.e., apply the R code to an entire vector of data). The basic structure of a for loop in R has the form

```
for(i in 1:n){
  # some R code depending on i
}
```

Here is a simple example:

```
> x <- 11:15
> x
[1] 11 12 13 14 15
> for(i in 1:5){
+   x[i] <- x[i] + 1
+ }
> x
[1] 12 13 14 15 16
```

And here is the better (vectorized) way to do the same thing:

```
> x <- 11:15
> x
[1] 11 12 13 14 15
> x <- x + 1
> x
[1] 12 13 14 15 16
```

Vectorization is preferred, but it is not always possible to vectorize things. As a rule of thumb, always start with the obvious (for loop) way of doing things, and then think about how to make your R code more efficient by vectorizing.

12 While Loops in R

While loops (or while statements) are the last of the standard functionalities of any programming language. The while loop allows you to execute the same R code repeatedly until some condition is met. Note that this is similar to a for loop but there is a key difference: a for loop implements the same R code a specified number of times (which is controlled by the index set), whereas a while loop implements the same R code an unspecified number of times (which is controlled by the specified condition).

The basic structure of a while loop in R has the form

```
while(condition){  
  # some R code  
}
```

Here is a simple example:

```
> x <- 80  
> iter <- 0  
> while(x < 100){  
+   x <- x + sqrt(x) / 10  
+   iter <- iter + 1  
+ }  
> x  
[1] 100.8293  
> iter  
[1] 22
```

Note that the above while loop will keep adding $\sqrt{x}/10$ to the initial value of $x = 80$ until the condition $x < 100$ is no longer true. The while loop also keeps track of the number of iterations (i.e., times that the R code within the loop was run), which is 22 in this case. Note that at the 22nd iteration, we have that $x = 100.8293 > 100$, so the while loop stopped running. When writing while loops, you need to be careful to ensure that your logical condition will work properly; otherwise you could have an invalid while loop or an infinite while loop (see the next page).

Here is an example of an invalid while loop:

```
> x <- 80
> iter <- 0
> while(x < 100){
+   x <- x - sqrt(x) / 10
+   iter <- iter + 1
+ }
```

Error in while (x < 100) { : missing value where TRUE/FALSE needed
In addition: Warning message:
In sqrt(x) : NaNs produced

Note that the above while loop produces an error message because x becomes negative (and then the square root is no longer a real number).

Here is an example of an infinite while loop:

```
> x <- 80
> iter <- 0
> while(x < 100){
+   x <- x - x / 10
+   iter = iter + 1
+ }
```

Note that the above while loop will run forever (or until we manually stop it) because the logical statement $x < 100$ is always true.

While loops are frequently used when you are programming iterative routines where you are doing something until the parameter estimates converge. However, in terms of standard applications of analyzing data, you will likely not need to use while loops too often. In most typical data analysis applications, you would likely be using if/else statements and for loops to accomplish your data pre-processing and analysis goals. So if you find yourself writing a while loop for a typical data analysis task, you should probably think about why you're doing it and if the while loop can potentially be replaced by a for loop.

13 Writing R Functions

In R, you can write your own function to accomplish whatever task(s) you desire. Note that you should consider writing a function (instead of an R script) whenever you need to implement the same multiline code more than once. For example, if you have a standard way that you preprocess your data in all of your publications, you should consider writing a function to preprocess your data—instead of copy-pasting the same preprocessing code into multiple different R script files. Writing a function has the benefit of standardizing and streamlining your research, and it can also make your code easily useable by other researchers (assuming that you share/publish your R code, which you should ALWAYS do!).

To write a function in R, you use the `function()` function to specify the function's name, arguments, and code that it should execute. The basic syntax for defining an R function is

```
name <- function(...) {
  # some R code
}
```

where `name` is the name that you give your new function, `function()` is the R function that you are calling to create your own function, and `...` denote the arguments (or inputs) that you want your new function to have.

Here is an example of a function for recoding a factor variable:

```
recode <- function(x, levels, ordered = FALSE, key = FALSE){
  x <- as.factor(x)
  xlev <- levels(x)
  if(length(levels) != nlevels(x)){
    stop("Input 'x' needs to be a factor with nlevels(x) == length(levels)")
  }
  x <- factor(x, levels = xlev, labels = levels, ordered = ordered)
  if(key){
    return(list(x = x, key = data.frame(old = xlev, new = levels)))
  }
  return(x)
}
```

The function has two required inputs: `x` is the factor variable that will be recoded, and `levels` are the desired levels for the recoded version of the input factor variable. The third argument determines whether or not the (new) levels of the factor should be `ordered` (default is `unordered`). The fourth argument controls what type of output the function will produce. If `key = FALSE` (default), the function only returns the recoded factor. If `key = TRUE`, the function returns a list with two elements: `x` is the recoded factor, and `key` is a data frame that shows how the old factor levels map onto the new factor levels. Note that setting `key = TRUE` can be useful for confirming that you've correctly recoded the levels of the factor.

Here is an example of how to use the function:

```
> x <- rep(c(1, 2, 3), each = 3)
> x
[1] 1 1 1 2 2 2 3 3 3

> xr <- recode(x, c("female", "male", "other"))
> xr
[1] female female female male   male   male   other  other  other
Levels: female male other
```

And here is an example of how to use the `key` argument:

```
> xr <- recode(x, c("female", "male", "other"), key = TRUE)
> names(xr)
[1] "x"   "key"

> xr$x
[1] female female female male   male   male   other  other  other
Levels: female male other

> xr$key
  old   new
1   1 female
2   2  male
3   3  other
```

14 R Script Files and R Markdown Documents

There are two different ways to create reproducible R code for data analysis: using an R script file (.R) or an R Markdown document (.Rmd). Both an R script file and an R Markdown document contain a collection of R code that can be executed to generate/reproduce data analysis results. The primary difference between the two is that R script files only contain executable R code (and comments created using #), whereas R Markdown can be used to produce documents that include both R code and R output in a high quality report.

The R script file is the “old school” way of doing things, and it is the tried-and-true way to reproduce your analysis results in R. The dynamic reports that are created using R Markdown are a more recent development that have some clear benefits over using just an R script file. For example, using R Markdown, you can add detailed descriptions of your data or R code, and you can also include all of the output that is produced by the code, e.g., printed results, tables, plots, etc. Thus, R Markdown lets you create a stand alone document that contains all of the R code to reproduce your results, as well as the results of the analyses. See <https://rmarkdown.rstudio.com/> for details on R Markdown.

For a simple example of using R Markdown you can see:

- Document: <http://stat.umn.edu/~helwig/notes/Rmarkdown-ex.pdf>
- Source code: <http://stat.umn.edu/~helwig/notes/Rmarkdown-ex.Rmd>

For a more advanced example of using R Markdown you can see:

- Document: <http://stat.umn.edu/~helwig/talks/ReproducibleCode.html>
- Source code: <http://stat.umn.edu/~helwig/talks/ReproducibleCode.Rmd>

The advanced example shows you how to create a fully reproducible analysis of data. Specifically, it shows you how to implement all steps of a typical data analysis process, i.e., downloading/importing data, preprocessing data, basic visualizations, simple statistical tests, fitting advanced statistical models, and outputting resulting tables and figures. This example also shows you how to create an R script file from an R Markdown file, e.g., in case you decide that you would like a file that only includes the executable analysis code.