Introduction to R

Nathaniel E. Helwig

Associate Professor of Psychology and Statistics
University of Minnesota

August 27, 2020

Table of Contents

Table of Contents

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.

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.

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.

[1] `https://www.r-project.org/`

[2] `https://cran.r-project.org/`

What is RStudio?

RStudio is an "integrative development environment" (IDE) for R that is freely available for desktops and servers running R.[3]

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)
- view various other information (e.g., related to File paths, created Plots, installed Packages, and Help files)

[3] `https://rstudio.com/`

RStudio GUI — Looks Like MATLAB, Huh?

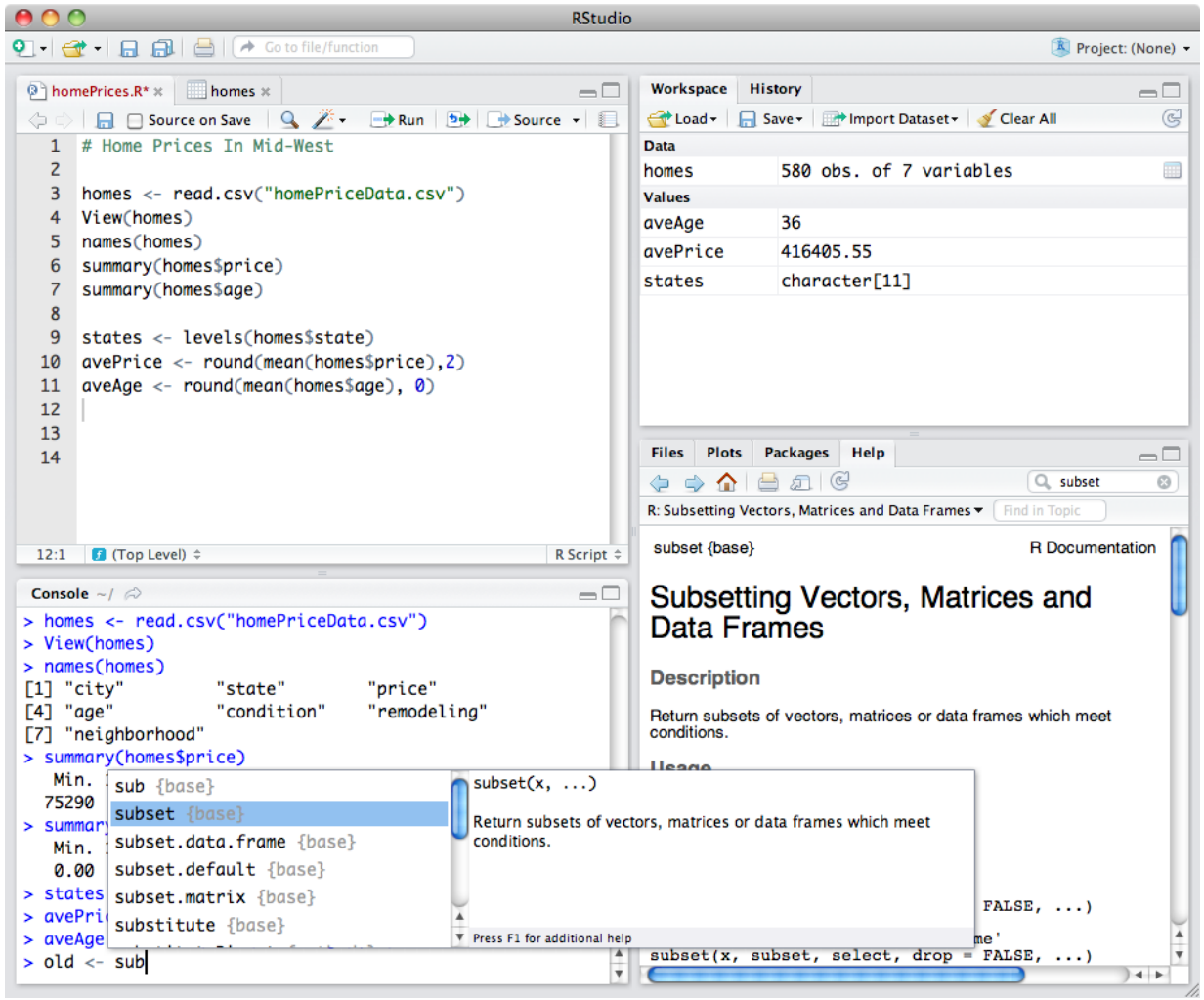

Figure 1: The RStudio GUI for a Mac. https://rstudio.com/products/rstudio/features/

Table of Contents

The R Console

Within the R Console, the symbol > comes before R code that is executed, and the symbol # denotes that what follows are comments.

You can use R for basic arithmetic calculations:

```
> 3 + 2      # addition
[1] 5
> 3 - 2      # subtraction
[1] 1
> 3 / 2      # multiplication
[1] 1.5
> 3 * 2      # division
[1] 6
> 3^2        # exponents
[1] 9
```

Understanding the R Console Output

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 16^{th} output letter).

Logical Operators in R

The R language has all of the standard logical operators

```
> 2 < 2       # less than
[1] FALSE
> 2 <= 2      # less than or equal to
[1] TRUE
> 2 > 2       # greater than
[1] FALSE
> 2 >= 2      # greater than or equal to
[1] TRUE
> 2 == 2      # exactly equal to
[1] TRUE
> 2 != 3      # not equal to
[1] TRUE
```

Combining Logical Operators: AND and OR

You can combine multiple logical operators using

- `&` denotes AND
- `|` denotes OR

```
> # x AND y
> (2 < 3) & (2 > 3)
[1] FALSE

> # x OR y
> (2 < 3) | (2 > 3)
[1] TRUE
```

Special Constants and Values in R

```
> pi                          # constant pi
[1] 3.141593

> exp(1)                      # base of natural logarithm
[1] 2.718282

> .Machine$double.eps         # machine epsilon
[1] 2.220446e-16

> Inf                         # infinity
[1] Inf

> NULL                        # empty data
NULL

> NA                          # missing data
[1] NA
```

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: `<-` is two symbols in a row: `<` followed by `-`

```
> x <- 3          # assign x value of 3
> y <- 2          # assign y value of 2
> x + y           # x plus y
[1] 5
> x * y           # x times y
[1] 6
> x^y             # x raised to the power of y
[1] 9
```

Using Functions in R

R is a function based language where a "function" takes in some input and produces some output.

- Vegas rules: what happens in a function, stays in a function
- Manipulations inside functions don't change values in Workspace

Syntax for using a function in R and assigning the output object:

```
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 (also known as arguments) for the function.

Examples of Built-In R Functions

Thankfully, many built-in R functions have intuitive names:

```
> c(1, 3, -2)              # combine values
[1]  1  3 -2
> sort(c(1, 3, -2))        # sort values
[1] -2  1  3
> min(c(1, 3, -2))         # minimum value
[1] -2
> max(c(1, 3, -2))         # maximum value
[1] 3
> sum(c(1, 3, -2))         # summation of values
[1] 2
> mean(c(1, 3, -2))        # mean of values
[1] 0.6666667
```

Viewing a Function's Help File

All built-in R functions have documentation (i.e., a help file) that...

- describes what the function does "under the hood"
- provides the syntax and argument options for usage
- explains what type of output values are produced
- illustrates how to use the function via examples

To see a help file, type a question mark before the function's name.

Example of how to see the help file for R's `sort` function:

```
> ?sort
```

Table of Contents

What is an Object 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.

The class of `X` informs R how to interact with the object when it is input into functions.

Classes for Numbers in R

R has two general classes for numbers:

- `numeric`: default class for real valued (double precision) numbers
- `integer`: special class for integers

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

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

Classes for Letters/Words in R

R has two general classes for letters/words:

- `character`: default class for letters and/or words
- `factor`: special class for categorical variables

A `factor` variable can have unordered levels (default) or ordered levels

```
> # 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"
```

Classes for Data in R

R has three general classes for data:

- `matrix`: vectors of the same length and same class
- `data.frame`: vectors of the same length and different classes
- `list`: collection of objects of different lengths or classes

Note that...

- 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

Matrix and Data Frame Classes 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 in R

```
> # 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"
```

Object Oriented Programming in R

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.

- Some functions are only applicable to objects of a particular class
- Some functions perform different operations depending on the class of the input object

As a user of R, it is your responsibility to ensure that you've correctly informed R how your data should be interpreted.

When you input your data into the R workspace, make sure...

- it is being stored as a data frame (or a list if needed)
- each column of the data frame has the class that you have intended

Example of Class Customized Functions in R

Functions 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.

```
> # 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
```

Table of Contents

If/Else Statements in R

IF/ELSE statements implement one collection of code if a particular condition is met, and implement a different collection of code otherwise.

Basic structure of IF/ELSE statements:

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

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

Example of If/Else Statement in R

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.

For Loops in R

For loops repeatedly implement the same R code with the loop index sequentially changing each time. For loops should *only* be used if you cannot vectorize code (i.e., apply the code to an entire vector of data).

Basic structure of a for loop:

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

`i` is the loop index and `1:n` is the index set (the values that `i` will take)

Example of For Loop in R

For loop version:

```
> 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
```

Vectorized version:

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

The vectorized version is preferred given that it is more efficient.

While Loops (or Statements) in R

While loops allow you to execute the same R code repeatedly until some condition is met. Similar to a for loop but there is a difference:

- For loops implement the same code a fixed number of times
- While loops implement the same code a variable number of times

Basic structure of a while loop:

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

The while loop repeats the code until the `condition` is no longer true.

Example of While Loop in R

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

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, which is 22 in this case. At the 22nd iteration $x = 100.8293 > 100$, so the while loop stops.

Table of Contents

Why You Should Write R Functions

In R, you can write your own function to accomplish whatever task(s) you desire—which makes it possible to streamline routine tasks.

You should write a function (instead of a script) whenever you need to implement the same multiline code more than once, e.g., for

- routine data preprocessing
- data analysis pipelines
- tabling or plotting results

Writing a function has the benefit of standardizing and streamlining your research, and it can make your code useable for other researchers.

Syntax for Creating Functions in R

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) for your function.

Create an R Function for Recoding Factors

```
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 satisfy 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 and fourth arguments control:

- should the new levels be treated as ordered (in the given order)?
- should the function return the key showing the old and new levels?

Example of Using the recode() 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

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

Table of Contents

Conducting Reproducible Research in R

There are two different ways to create reproducible R code:

- an R script file (.R file)
- an R Markdown document (.Rmd file)

Both an R script file and an R Markdown document contain a collection of R code that can be executed to reproduce 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.

Script or Markdown?

R script files are the "old school" way of doing things, and they are the tried-and-true way to reproduce your analysis results in R.

Dynamic reports created using R Markdown[4] are a more recent development that have some benefits over using an R script file:

- you can add detailed descriptions of your data or R code
- you can include all of the output that is produced by the code

However, R Markdown documents can be a bit finicky to compile, especially if you are running R in the Windows operating system.

[4] See `https://rmarkdown.rstudio.com/` for details on R Markdown.

Examples of R Markdown Documents

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, including downloading/importing data, preprocessing data, basic visualizations, simple statistical tests, fitting advanced statistical models, and outputting results via tables and figures.