Engineering Data Analysis in R

John Volckens and Kathleen E. Wendt

2020-07-19

Contents

```
## -- Attaching packages ----------------------------------- tidyverse 1.2.1 --

## v ggplot2 3.3.0     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts -------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()

##
## Attaching package: 'kableExtra'

## The following object is masked from 'package:dplyr':
##
##     group_rows
```

Chapter 1

Introduction

Preface

I developed this coursebook to help engineers begin to *think* about data. Most of my job at the university is to conduct research, yet most of the students who show up to my lab don't know where to begin when presented with data. The irony is that, while engineering students are continuously drilled on how to solve problems, they are rarely taught how to seek them out. My undergraduate advisor, Dr. David Hemenway, once told me: "The data are always trying to tell you something." This book is an introduction to *data listening* because listening comes before understanding.

In broad terms, scientific and engineering research is about discovery: finding out something new. In practice, however, research is really about failure. Over the short term, researchers fail in the act of discovery much more often than they succeed. Such failure is to be expected because cutting-edge research ain't easy. Once you come to terms with accepting failure as a regular occurrence, you put yourself in a position to learn from it. To learn from failure requires that you observe and diagnose "what went wrong" and to do that, you need to listen to what the data are telling you. Let's begin.

1.1 How to use this book

The coursebook is intended to be a sort of *self-help* guide for students who want to learn R programming and the art of engineering data science. The book is designed to get you started in the art, not master it. I'm not qualified to teach mastery in the art of R or engineering data science, so look elsewhere for that level of tutelage.

If you are new to these topics, you probably want to start at the beginning and proceed through each chapter sequentially. Some sections or material might seem boring or too easy. In that case, just skip to the end of the section and see if you can complete the exercises and answer the questions.

Nearly all of the graphics and data presented in this book were created or manipulated in R. In many places, however, I have hidden the code in order to streamline the message. If you ever wonder "How did he do that?" you can download any of the R Markdowns on the GitHub repository, where this coursebook is hosted.

1.2 Prerequisites

This course is intended for upper-level undergraduates who have completed MECH 231 (Experimental Design).

1.3 Grading

1.3.1 Grading for MECH 481A6

Course grades will be determined by the following five components:

Assessment component	Percent of grade
Exams	60
Quizzes	25
Homework	15

1.4 Course set-up

Please download and install the latest version of R and RStudio.

- R: https://cran.r-project.org
- RStudio: https://rstudio.com/products/rstudio/download/#download

Students will also need to download and install git software and create a GitHub account.

- Install git: https://git-scm.com/downloads
- Create a GitHub account: https://github.com

Happy Git with R by Dr. Jenny Bryan is a helpful resource for installation and set-up. Also, a team associated with the Colorado State University Statistics Department is actively developing a series of one-credit modules for undergraduate students new to R. Their videos on software installation and RStudio orientation may be particularly helpful at this stage.

If you want to generate PDF output from R Markdown documents, you will also need to install LaTex. I suggest taking the following approach, if you have never installed LaTex on your personal computer. More installation guidance can be found here.

```r
# install R package
install.packages("tinytex")
# install LaTex "ingredients"
tinytex::install_tinytex()
```

1.5 Coursebook

This coursebook will serve as the only required textbook for this course. I regularly edit and add to this book, so content may change somewhat over the semester. We typically cover about a chapter of the book every 1-2 weeks of the course.

This coursebook includes:

- Links to the slides presented in class for each topic
- In-course exercises, typically including links to the data used in the exercise
- Homework assignments
- Appendix of reference distributions
- A list of vocabulary and concepts that should be mastered for each quiz

If you find any typos or bugs, or if you have any suggestions for how the book can be improved, feel free to post it on the book's GitHub Issues page.

This book was developed using Yihui Xie's bookdown framework. The book is built using code that combines R code, data, and text to create a book for which R code and examples can be re-executed every time the book is re-built, which helps identify bugs and broken code examples quickly. The online book is hosted using GitHub's free GitHub Pages. All material for this book is available and can be explored at the book's GitHub repository.

1.5.1 Other helpful books (not required)

The best book to supplement the coursebook and lectures for this course is R for Data Science by Garrett Grolemund and Hadley Wickham. The entire book is freely available online through the same format of the coursebook. You can also purchase a paper version of the book published by O'Reilly for around $40. This book is an excellent and up-to-date reference by some of the best R programmers in the world.

There are a number of other useful books available on general R programming, including:

- R for Dummies
- R Cookbook
- R Graphics Cookbook
- Roger Peng's Leanpub books

- Various books on bookdown.org

The R programming language is used extensively within certain fields, including statistics and bioinformatics. If you are using R for a specific type of analysis, you will be able to find many books with advice on using R for both general and specific statistical analysis, including many available in print or online through the CSU library.

1.6 Acknowledgements

Most of the introductory material for this book were adapted from Professor Brooke Anderson's R Programming Coursebook, to whom I owe thanks not only for the materials but for the many helpful discussions. I would also like to acknowledge John Tukey, one of the pioneers of exploratory data analysis, and the creators of the NIST Engineering Statistics Handbook, from which I have drawn many techniques.

Chapter 2

The R Programming Environment

2.1 Objectives

This chapter is designed around the following learning objectives. Upon completing this chapter, you should be able to:

- Define free and open source software and list some of its advantages over proprietary software
- Recognize the difference between R and RStudio
- Describe the differences between base R code that you initially download and "package" code that you use to expand base R
- Use RStudio to download and install a package from the Comprehensive R Archive Network (CRAN) to your computer
- Use RStudio to load a package that you have installed within an R session
- Demonstrate how to access help documentation including vignettes and helpfiles for a package and its functions
- Demonstrate how to submit R expressions at the console
- Define the general syntax for calling a function and for specifying both required and optional arguments for that function
- Describe what an R object is and how to assign an R object a name to reference it in later code
- Describe how to create vector objects of numeric and character classes
- Describe how to explore and extract elements from vector objects
- Describe how to create dataframe objects
- Describe how to explore and extract elements from dataframe objects
- Compare the key differences between running R code from the console versus writing and running R code in an R script

2.2 R and R Studio

2.2.1 What is R?

R in an open-source programming language that evolved from the S language. The S language was developed at Bell Labs in the 1970s, which is the same place (and about the same time) that the C programming language was developed.

R itself was developed in the 1990s-2000s at the University of Auckland. It is open-source software, freely and openly distributed under the GNU General Public License (GPL). The base version of R that you download when you install R on your computer includes the critical code for running R, but you can also install and run "packages" that people all over the world have developed to extend R.

With new developments, R is becoming more and more useful for a variety of programming tasks. It really shines in working with data and doing statistical analysis. R is currently popular in a number of fields, including statistics, machine learning, and data analysis.

R is an **interpreted language**. That means that you can communicate with it interactively from a command line. Other common interpreted languages include Python and Perl.

Compared to Python, R has some of the same strengths (e.g., quick and easy to code, interfaces well with other languages, easy to work interactively) and weaknesses (e.g., slower than compiled languages). For data-related tasks, R and Python are fairly neck-and-neck, with Julia an up-and-coming option. Nonetheless, R is still the first choice of statisticians in most fields, so I would argue that R has a an advantage, if you want to have access to cutting-edge statistical methods.

> "The best thing about R is that it was developed by statisticians. The worst thing about R is that…it was developed by statisticians." – Bo Cowgill, Google, at the Bay Area R Users Group

2.2.2 Free and open-source software

> "Life is too short to run proprietary software." – Bdale Garbee

R is **free and open-source software**. Conversely, many other popular statistical programming languages such as SAS and SPSS are proprietary. It's useful to know what it means for software to be "open-source", both conceptually and in terms of how you will be able to use and add to R in your own work.

R is free, and it's tempting to think of open-source software just as "free software". It is a little more subtle than that. It helps to consider some different meanings of the word "free". "Free" can mean:

- *Gratis*: Free as in free beer

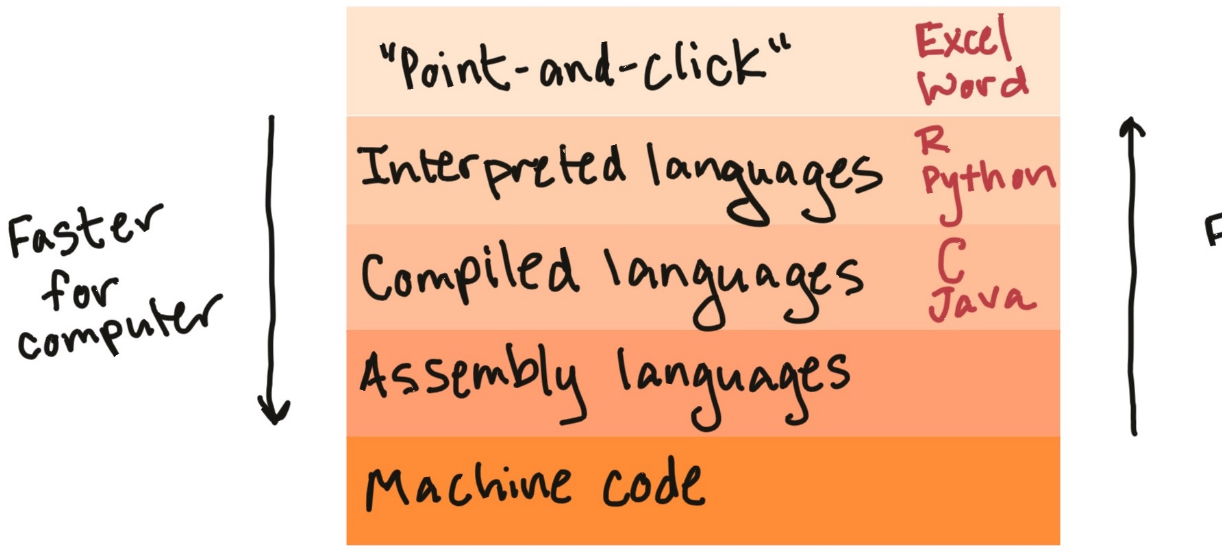


Figure 2.1: Broad types of software programs. R is an interpreted language. 'Point-and-click' programs, like Excel and Word, are often easiest for a new user to get started with, but are slower for the computer and are restricted in the functionality they offer. By contrast, compiled languages (like C and Java), assembly languages, and machine code are faster for the computer and allow you to create a wider range of things, but can take longer to code and take longer for a new user to learn to work with.

- *Libre*: Free as in free speech

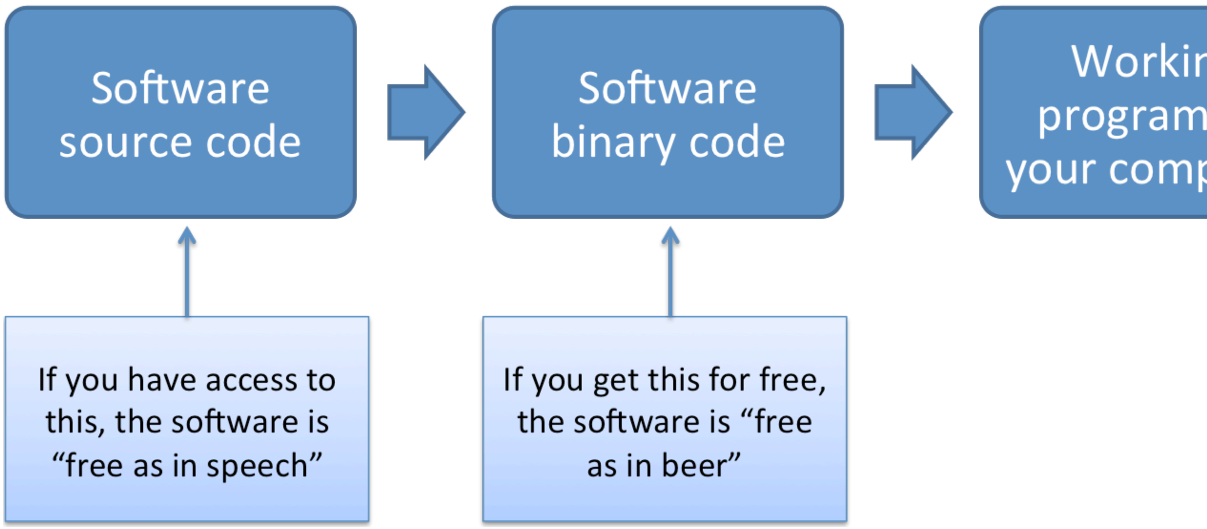

Software source code
Software binary code
Workin program your comp
If you have access to this, the software is "free as in speech"
If you get this for free, the software is "free as in beer"

Figure 2.2: An overview of how software can be each type of free (beer and speech). For software programs developed using a compiled programming language, the final product that you open on your computer is run by machine-readable binary code. A developer can give you this code for free (as in beer) without sharing any of the original source code with you. This means you can't dig in to figure out how the software works and how you can extend it. By contrast, open-source software (free as in speech) is software for which you have access to the human-readable code that was used as in input in creating the software binaries. With open-source code, you can figure out exactly how the program is coded.

Open-source software is the *libre* type of free (Figure **??**). This means that, with software that is open-source, you can:

- Access all of the code that makes up the software
- Change the code as you'd like for your own applications
- Build on the code with your own extensions
- Share the software and its code, as well as your extensions, with others

Often, open-source software is also free, making it "free and open-source software", or "FOSS".

Popular open source licenses for R and R packages include the GPL and MIT licenses.

> "Making Linux GPL'd was definitely the best thing I ever did." – Linus Torvalds

In practice, this means that, once you are familiar with the software, you can dig deeply into the code to figure out exactly how it's performing certain tasks. This can be useful for finding and eliminating bugs and can help researchers figure out if there are any limitations in how the code works for their specific research.

It also means that you can build your own software on top of existing R software and its extensions. I explain a bit more about R packages a bit later, but this open-source nature of R has created a large community of people worldwide who develop and share extensions to R. As a result, you can pull in packages that let you do all kinds of things in R, like visualizing Tweets, cleaning up accelerometer data, analyzing complex surveys, fitting maching learning models, and a wealth of other cool things.

> "Despite its name, open-source software is less vulnerable to hacking than the secret, black box systems like those being used in polling places now. That's because anyone can see how open-source systems operate. Bugs can be spotted and remedied, deterring those who would attempt attacks. This makes them much more secure than closed-source models like Microsoft's, which only Microsoft employees can get into to fix." – Woolsey and Fox. *To Protect Voting, Use Open-Source Software.* New York Times. August 3, 2017.

You can download the latest version of R from CRAN. Be sure to select the distribution for your type of computer system. R is updated occasionally; you should plan to reinstall R at least once a year to make sure you're working with one of the newer versions. Check your current R version (e.g., by running `sessionInfo()` at the R console) to make sure you're not using an outdated version of R.

> "The R engine …is pretty well uniformly excellent code but you have to take my word for that. Actually, you don't. The whole engine is open source so, if you wish, you can check every line of it. If people were out to push dodgy software, this is not the way they'd go about it." – Bill Venables, R-help (January 2004)
>
> "Talk is cheap. Show me the code." – Linus Torvalds

2.2.3 What is RStudio?

To get the R software, you'll download R from the R Project for Statistical Computing. This is enough for you to use R on your own computer. But, for a more user-friendly experience, you should also download RStudio, an integrated development environment (IDE) for R. It provides you an interface for using R, with a lot of nice extras like R Projects that will make your life easier. All of the code chunks shown in this book were produced using RStudio.

As Chapter 1 outlined, you should download R first, then the RStudio IDE.

RStudio, PBC is a leader in the R community. Currently, the company:

- Develops and freely provides the RStudio IDE
- Provides excellent resources for learning and using R (e.g., cheat sheets, free online books)

- Is producing some of the popular R packages
- Employs some of the top people in R development
- Is a key member of The R Consortium in addition to others such as Microsoft, IBM, and Google

R has been advancing by leaps and bounds in terms of what it can do and the elegance with which it does it, in large part because of the enormous contributions of people involved with RStudio.

2.3 Communicating with R

Because R is an interpreted language, you can communicate with it interactively. You do this using the following general steps:

1. Open an **R session**
2. At the **prompt** in the **console**, enter an **R expression**
3. Read R's "response" (i.e., **output**)
4. Repeat 2 and 3
5. Close the R session

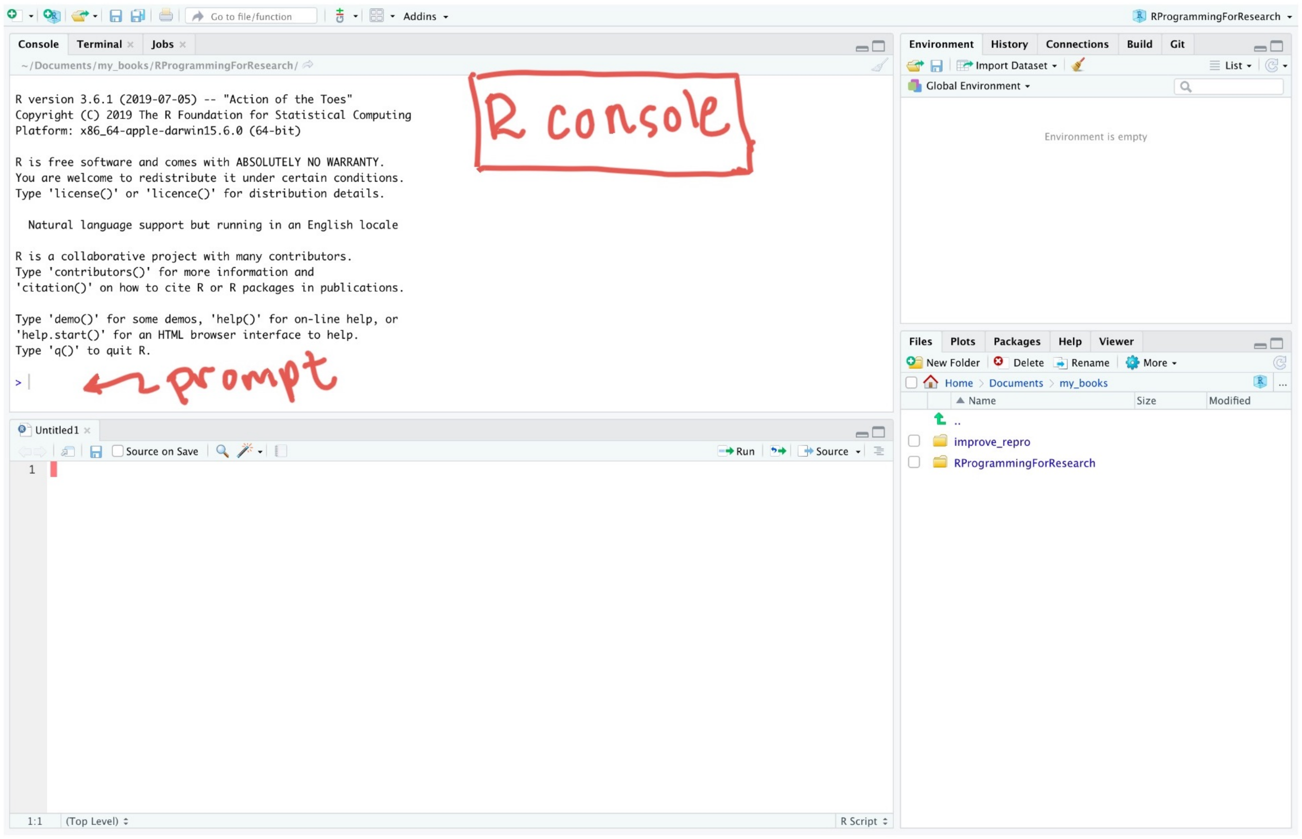
2.3.1 R sessions, console, and command prompt

An **R session** is an "instance" of you using R. To open an R session, double-click on the icon for the RStudio IDE on you computer. When RStudio opens, you will be in a "fresh" R session, unless you restore a saved session, which is not best practice. To avoid saving work sessions, you should change the defaults in RStudio's Preferences menu, such that RStudio never saves the workspace to .RData on exit. A "fresh" R session means that, once you open RStudio, you will need to "set up" your session, including loading packages and importing data (discussed later).

In RStudio, the screen is divided into several "panes". We'll start with the pane called "Console". The **console** lets you "talk" to R. This is where you can "talk" to R by typing an **expression** at the **prompt** (the caret symbol, ">"). You press the "Return" key to send this message to R.

Once you press "Return", R will respond in one of three ways:

1. R does whatever you asked it to do with the expression and prints the output, if any, of doing that, as well as a new prompt so you can ask it something new.
2. R doesn't think you've finished asking for something, and instead of giving you a new prompt (">") it gives you a "+". This means that R is still listening, waiting for you to finish asking it something.
3. R tries to do what you asked it to, but it can't. It gives you an **error message**, as well as a new prompt so you can try again or ask it something new.



```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Figure 2.3: Finding the 'Console' pane and the command prompt in RStudio.

2.3.2 R expressions, function calls, and objects

To "talk" with R, you need to know how to give it a complete **expression**. Most expressions you'll want to give R will be some combination of two elements:

1. **Function calls**
2. **Object assignments**

We'll go through both these pieces and also look at how you can combine them together for some expressions.

According to John Chambers, one of the creators of the S language (precursor to R):

1. Everything that exists in R is an **object**
2. Everything that happens in R is a **call to a function**

In general, function calls in R take the following structure:

```
# generic code (this won't run)
function_name(formal_argument_1 = named_argument_1,
              formal_argument_2 = named_argument_2,
              [etc.])
```

Sometimes, we'll show "generic" code in a code block, that doesn't actually work if you put it in R, but instead shows the generic structure of an R call. We'll try to always include a comment with any generic code, so you'll know not to try to run it in R.

A function call forms a complete R expression, and the output will be the result of running `print()` or `show()` on the object that is output by the function call. Here is an example of this structure:

```
print(x = "Hello, world!")
```

```
## [1] "Hello, world!"
```

Figure **??** shows an example of the typical elements of a function call. In this example, we're **calling** a function with the **name** `print`. It has one **argument**, with a **formal argument** of `x`, which in this call we've provided the **named argument**: "Hello, world!".

The **arguments** are how you customize the call to an R function. For example, you can use change the named argument value to print different messages with the `print()` function. Note that the formal argument never changes.

```
print(x = "Hello, world!")
```

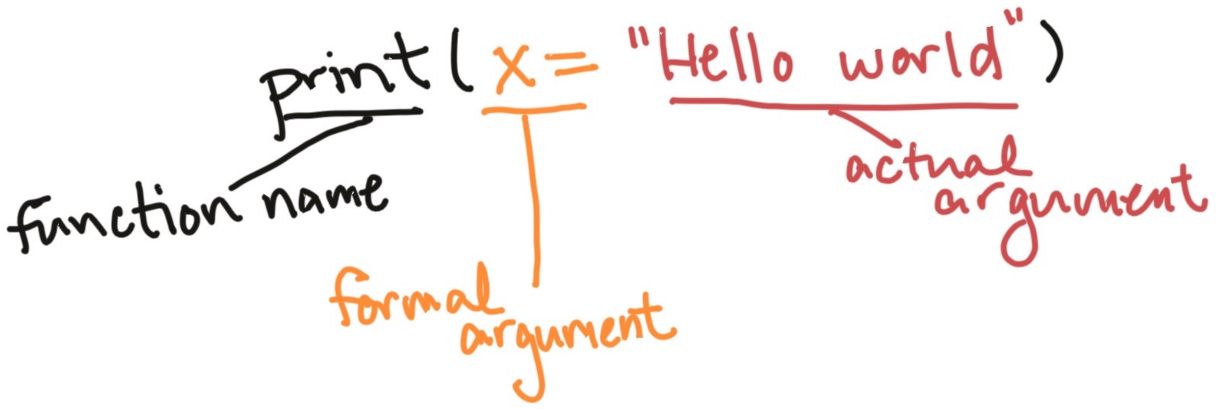


Figure 2.4: Main parts of a function call. This example is calling a function with the name 'print'. The function call has one argument, with a formal argument of 'x', which in this call is provided the named argument 'Hello world'.

```
## [1] "Hello, world!"
```

```r
print(x = "Hi, Fort Collins!")
```

```
## [1] "Hi, Fort Collins!"
```

Some functions do not require any arguments. For example, the `getRversion()` function will print out the version of R you are using.

```r
getRversion()
```

```
## [1] '3.6.2'
```

Some functions will accept multiple arguments. For example, the `print()` function allows you to specify whether the output should include quotation marks, using the `quote` formal argument:

```r
print(x = "Hello world", quote = TRUE)
```

```
## [1] "Hello world"
```

```r
print(x = "Hello world", quote = FALSE)
```

```
## [1] Hello world
```

Arguments can be **required** or **optional**.

For a required argument, if you don't provide a value for the argument when you call the function, R will respond with an error. For example, `x` is a **required argument** for the `print()` function, so if you try to call the function without it, you'll get an error:

```
print()
```

```
Error in print.default() : argument "x" is
  missing, with no default
```

For an **optional argument** on the other hand, R knows a **default value** for that argument, so if you don't give it a value for that argument, it will just use the default value provided by the R developer who wrote the function.

For example, for the `print()` function, the `quote` argument has the default value `TRUE`. So if you don't specify a value for that argument, R will assume it should use `quote = TRUE`. That's why the following two calls give the same result:

```
print(x = "Hello, world!", quote = TRUE)
```

```
## [1] "Hello, world!"
```

```
print(x = "Hello, world!")
```

```
## [1] "Hello, world!"
```

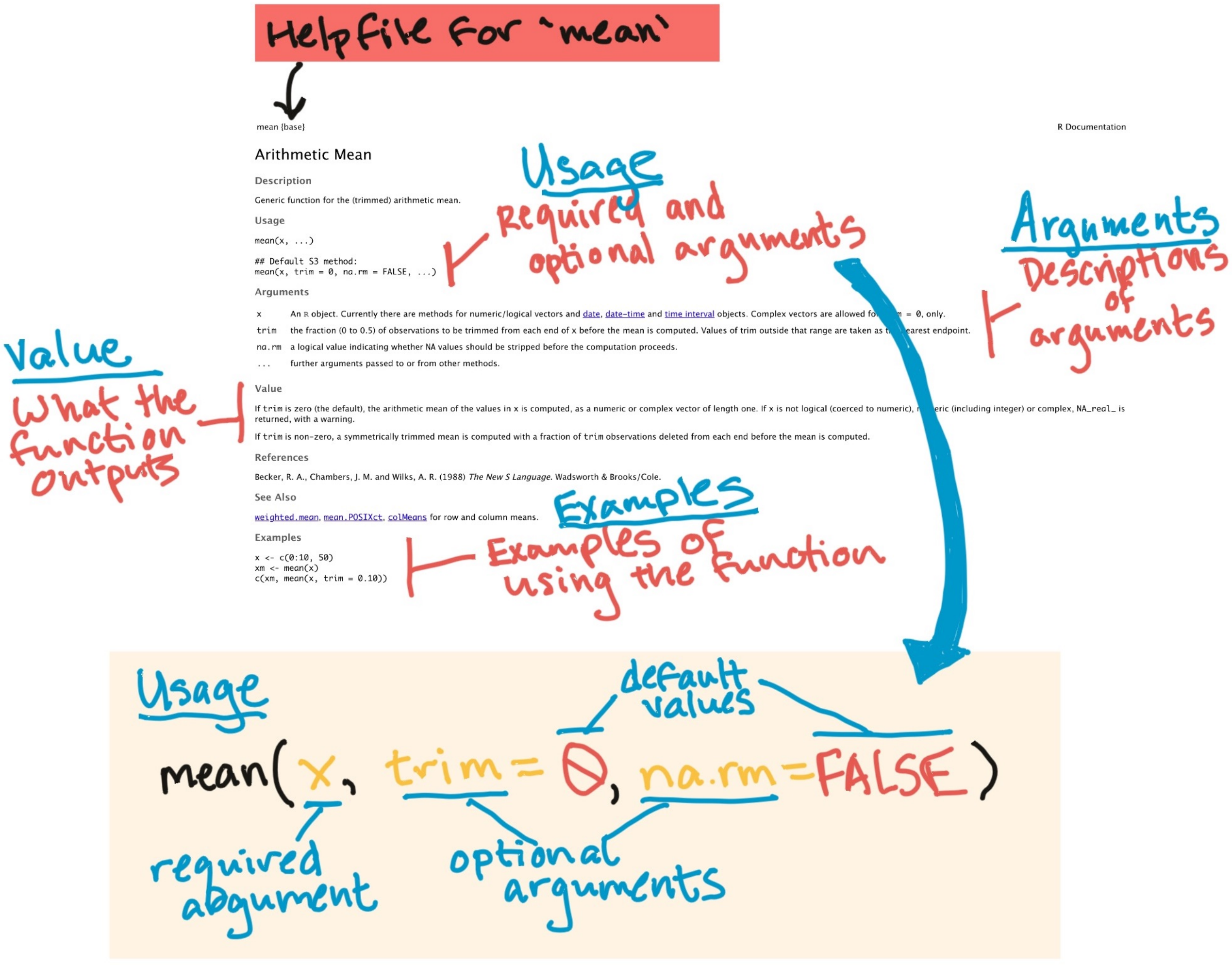
Often, you'll want to find out more about a function, including:

- Examples of how to use the function
- Which arguments you can include for the function
- Which arguments are required versus optional
- What the default values are for optional arguments

You can find out all this information in the function's **helpfile**, which you can access using the function `?`. For example, the `mean()` function will let you calculate the mean (average) of a group of numbers. To find out more about this function, at the console type:

```
?mean
```

This will open a helpfile in the "Help" pane in RStudio. Figure **??** shows some of the key elements of an example helpfile, the helpfile for the `mean()` function. In particular, the "Usage" section helps you figure out which arguments are **required** and which are **optional** in the Usage section of the helpfile.


Helpfile for `mean`
Usage
Required and optional arguments
Arguments
Descriptions of arguments
Value
What the function outputs
Examples
Examples of using the function
Usage
default values
mean(x, trim = 0, na.rm = FALSE)
required abgument
optional arguments

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed fo m = 0, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as t earest endpoint.
`na.rm`	a logical value indicating whether NA values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), n eric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`weighted.mean`, `mean.POSIXct`, `colMeans` for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

Figure 2.5: Navigating a helpfile. This example shows some key parts of the helpfile for the 'mean' function.

There's one class of functions that looks a bit different from others. These are the infix **operator** functions. Instead using parentheses after the function name, they usually go *between* two arguments. One common example is the + operator:

```
2 + 3
```

```
## [1] 5
```

There are operators for several mathematical functions: +, -, *, /. There are also other operators, including **logical operators** and **assignment operators**, which we'll cover later.

In R, a variety of different types and structures of data can be saved in **objects**. For right now, you can just think of an R object as a discrete container of data in R.

Function calls will produce an object. If you just call a function, as we've been doing, then R will respond by printing out that object. But, we often want to use that object more. For example, we might want to use it as an argument later in our "conversation" with R, when we call another function later. If you want to re-use the results of a function call later, you can **assign** that **object** to an **object name**. This kind of expression is called an **assignment expression**.

Once you do this, you can use that *object name* to refer to the object. This means that you don't need to re-create the object each time you need it—instead, you can create it once, and then just reference it by name each time you need it after that. For example, you can read in data from an external file as a dataframe object and assign it an object name. Then, when you need that data later, you won't need to read it in again from the external file.

The **"gets arrow"** (<-) is R's assignment operator. It takes whatever you've created on the right hand side of the <- and saves it as an object with the name you put on the left hand side of the <-:

```
# generic code-- this will not work
[object name] <- [object]
```

For example, if I just type `"Hello, world!"`, R will print it back to me, but it won't save it anywhere for me to use later:

```
"Hello, world!"
```

```
## [1] "Hello, world!"
```

If I assign it to an object, I can "refer" to that object in a later expression. For example, the code below assigns the **object** `"Hello, world!"` the **object name** `message`. Later, I can just refer to this object using the name `message`, for example in a function call to the `print()` function:

```r
message <- "Hello, world!"
print(x = message)
```

```
## [1] "Hello, world!"
```

When you enter an **assignment expression** like this at the R console, if everything goes right, then R will "respond" by giving you a new prompt, without any kind of message. There are three ways you can check to make sure that the object was successfully assigned to the object name:

1. Enter the object's name at the prompt and press return. The default if you do this is for R to "respond" by calling the `print()` function with that object as the `x` argument.
2. Call the `ls()` function, which doesn't require any arguments. This will list all the object names that have been assigned in the current R session.
3. Look in the "Environment" pane in RStudio. This also lists all the object names that have been assigned in the current R session.

Here are examples of these strategies:

1. Enter the object's name at the prompt and press return:

```r
message
```

```
## [1] "Hello, world!"
```

2. Call the `ls()` function:

```r
ls()
```

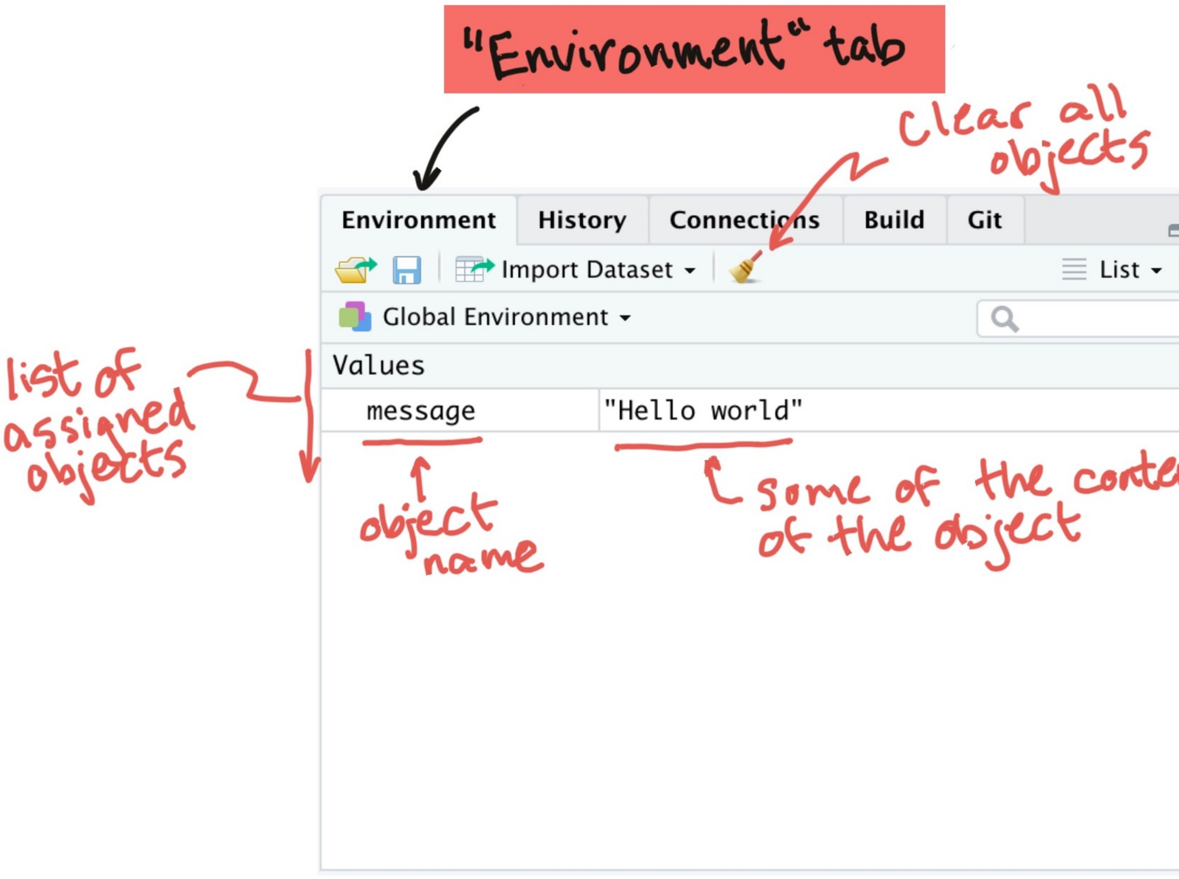
```
## [1] "message"
```

3. Look in the "Environment" pane in RStudio (see Figure **??**).

You can make assignments in R using either the "gets arrow" (<-) or =. When you read other people's code, you'll see both. R gurus advise using <- rather than = when coding in R, because as you move to doing more complex things, some subtle problems might crop up if you use =. You can tell the age of a programmer by whether he or she uses the "gets arrow" or =, with = more common among the young and hip. For this course, however, I am asking you to code according to Hadley Wickham's R style guide, which specifies using the "gets arrow" for object assignment.

While the "gets arrow" takes two key strokes, you can somewhat get around this limitation by using RStudio's keyboard shortcut for the "gets arrow". This shortcut is Alt + - on Windows and Option + - on Macs. To see a full list of RStudio keyboard shortcuts, go to the "Help" tab in RStudio and select "Keyboard Shortcuts".

There are some absolute **rules** for the names you can use for an object name:



```
message    "Hello world"
```

Figure 2.6: 'Environment' pane in RStudio. This shows the names and first few values of all objects that have been assigned to object names in the global environment.

- Use only letters, numbers, and underscores
- Don't start with anything but a letter

If you try to assign an object to a name that doesn't follow the "hard" rules, you'll get an error. For example, all of these expressions will give you an error:

```
1message <- "Hello world"
_message <- "Hello world"
message! <- "Hello world"
```

In addition to these fixed rules, there are also some guidelines for naming objects that you should adopt now, since they will make your life easier as you advance to writing more complex code in R. The following three guidelines for naming objects are from Hadley Wickham's R style guide:

- Use lower case for variable names (`message`, not `Message`)
- Use an underscore as a separator (`message_one`, not `messageOne`)
- Avoid using names that are already defined in R (e.g., don't name an object `mean`, because a `mean()` function exists)

> "Don't call your matrix 'matrix'. Would you call your dog 'dog'? Anyway, it might clash with the function 'matrix'." – Barry Rowlingson, R-help (October 2004)

Another good practice is to name objects after nouns (e.g., `message`) and later, when you start writing functions, name those after verbs (e.g., `print_message`). You'll want your object names to be short enough that they don't take forever to type as you're coding, but not so short that you can't remember to what they refer.

Sometimes, you'll want to create an object that you won't want to keep for very long. For example, you might want to create a small object to test some code, but you plan to not need the object again once you've done that. You may want to come up with some short, generic object names that you use for these kinds of objects, so that you'll know that you can delete them without problems when you want to clean up your R session.

There are all kinds of traditions for these placeholder variable names in computer science. `foo` and `bar` are two popular choices, as are, evidently, `xyzzy`, `spam`, `ham`, and `norf`. There are different placeholder names in different languages: for example, `toto`, `truc`, and `azerty` (French); and `pippo`, `pluto`, `paperino` (Disney character names in Italian). See the Wikipedia page on metasyntactic variables to find out more.

What if you want to "compose" a call from more than one function call? One way to do it is to assign the output from the first function call to a name and then use that name for the next call. For example:

```
message <- paste("Hello", "world")
print(x = message)
```

```
## [1] "Hello world"
```

If you give two objects the same name, the most recent definition will be used; objects can be overwritten by assigning new content to the same object name. For example:

```
a <- 1:10
b <- LETTERS [1:3]
a
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
b
```

```
## [1] "A" "B" "C"
```

```
a <- b
a
```

```
## [1] "A" "B" "C"
```

To create an R expression you can "nest" one function call inside another function call. For example:

```
print(x = paste("Hello", "world"))
```

```
## [1] "Hello world"
```

Just like with math, the order that the functions are evaluated moves from the inner set of parentheses to the outer one (Figure **??**). There's one more way we'll look at later called "piping".

2.4 R scripts

This is a good point in learning R for you to start putting your code in R scripts, rather than entering commands at the console.

An R script is a plain text file where you can save a series of R commands. You can save the script and open it up later to see or re-do what you did earlier, just like you could with something like a Word document when you're writing a paper.

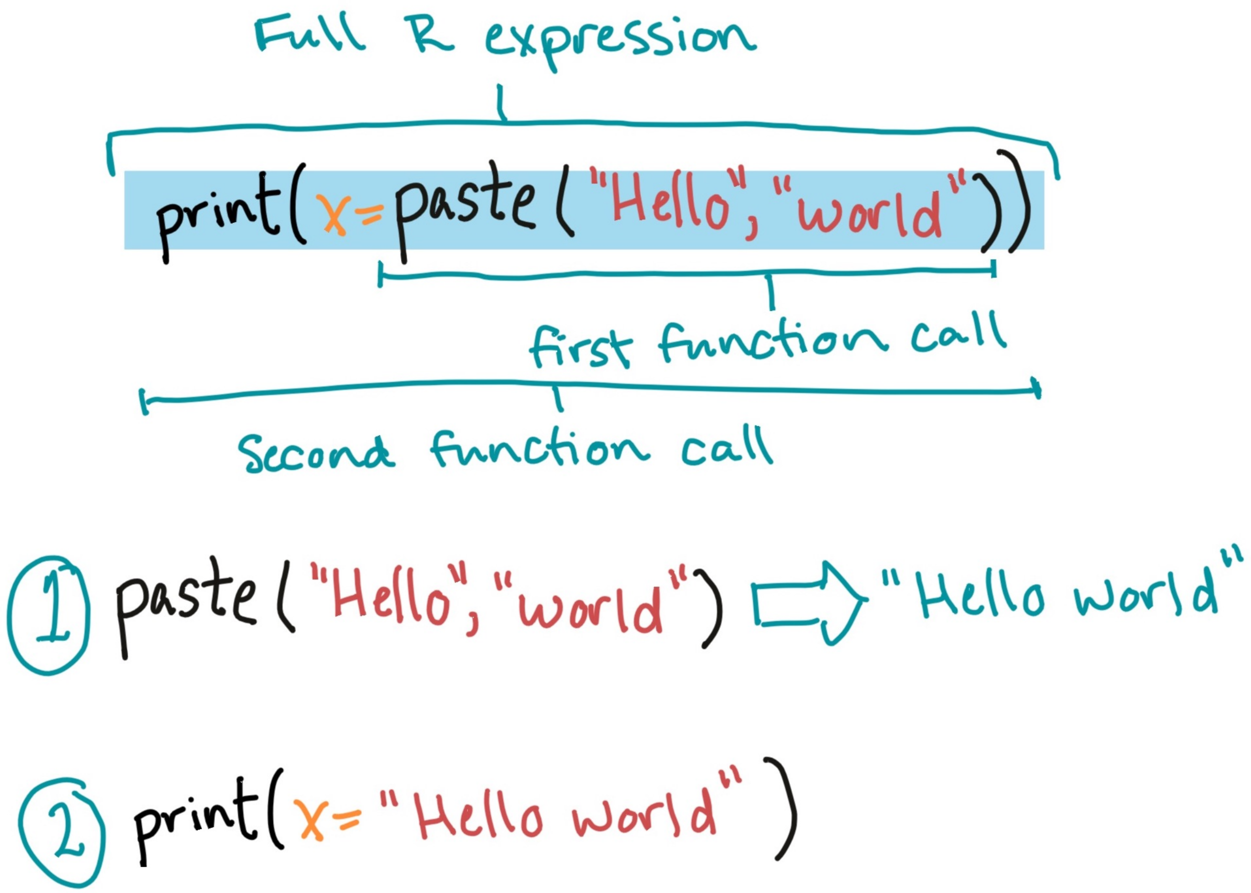


Figure 2.7: 'Environment' pane in RStudio. This shows the names and first few values of all objects that have been assigned to object names in the global environment.

To open a new R script in RStudio, go to the menu bar and select "File" -> "New File" -> "R Script". Alternatively, you can use the keyboard shortcut Command-Shift-N. Figure **??** gives an example of an R script file opened in RStudio and points out some interesting elements.

To save a script you're working on, you can click on the "Save" button, which looks like a floppy disk, at the top of your R script window in RStudio or use the keyboard shortcut Command-S. You should save R scripts using a ".R" file extension.

Within the R script, you'll usually want to type your code so there's one command per line. If your command runs long, you can write a single call over multiple lines. It's unusual to put more than one command on a single line of a script file, but you can if you separate the commands with semicolons (`;`). These rules all correspond to how you can enter commands at the console.

Running R code from a script file is very easy in RStudio. You can use either the "Run" button or Command-Return, and any code that is selected (i.e., that you've highlighted with your cursor) will run at the console. You can use this functionality to run a single line of code, multiple lines of code, or even just part of a specific line of code. If no code is highlighted, then R will instead run all the code on the line with the cursor and then move the cursor down to the next line in the script.

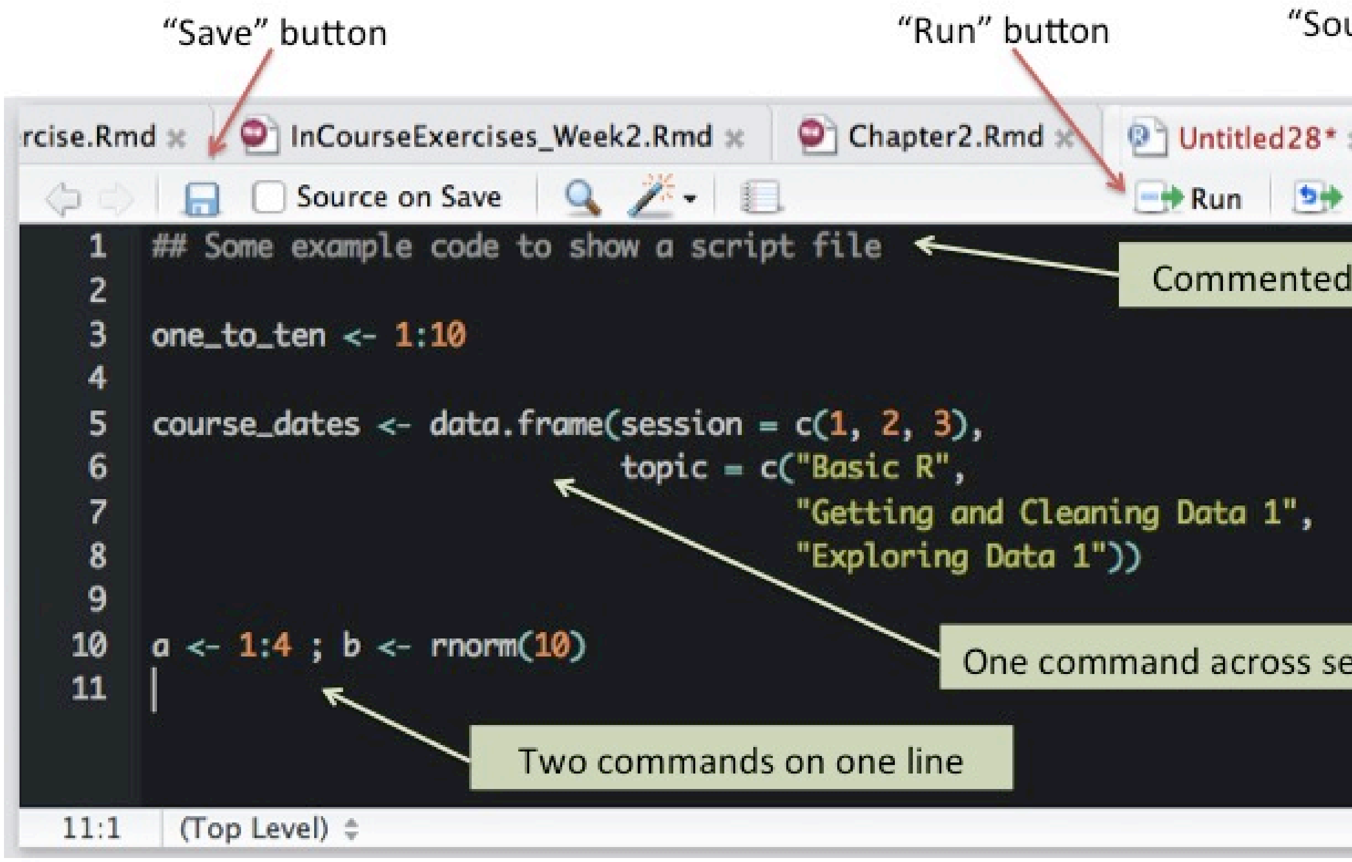
You can also run all of the code in a script. To do this, use the "Source" button at the top of the script window. You can also run the entire script either from the console or from within another script by using the `source()` function, with the filename of the script you want to run as the argument. For example, to run all of the code in a file named "MyFile.R" that is saved in your current working directory, run:

```
source("MyFile.R")
```

While it's generally best to write your R code in a script and run it from there rather than entering it interactively at the R console, there are some exceptions. A main example is when you're initially checking out a dataset to make sure you've imported it correctly. It often makes more sense to run commands for this task, like `str()`, `head()`, `tail()`, and `summary()`, at the console. These are all examples of commands where you're trying to look at something about your data **right now**, rather than code that builds toward your analysis, or helps you import or wrangle your data.

2.4.1 Commenting code

Sometimes, you'll want to include notes in your code. You can do this in all programming languages by using a *comment character* to start the line with your comment. In R, the comment character is the hash symbol, #. You can add comments into an R script to let others know (and remind yourself) what you're doing and why. Any line on a script line that starts with # will not be read by R. You can also take advantage of commenting to comment out certain parts of code that you don't want to run at the



```
## Some example code to show a script file

one_to_ten <- 1:10

course_dates <- data.frame(session = c(1, 2, 3),
                           topic = c("Basic R",
                                     "Getting and Cleaning Data 1",
                                     "Exploring Data 1"))


a <- 1:4 ; b <- rnorm(10)
```

Figure 2.8: Example of an R script in RStudio.