Getting Started

Chapter 1

Stats 20 Lec 2

Fall 2017

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Install R and RStudio.
- Generate R scripts and R Markdown files.
- Get and set working directories.
- Apply good programming etiquette to write readable code.
- Compute basic calculator computations in R.
- Understand the order of operations in R.
- Store objects with the assignment operator (`<-`).
- Write and call basic functions.
- Understand how to view and remove objects from the workspace.

1 Introduction

1.1 What is Statistical Programming?

Computer programming involves writing instructions (**code**) to control a computer: what to calculate, what to display, etc.

Statistical computing involves creating and/or using software to do computations to aid in statistical analysis:

(a) Data is summarized and displayed.
(b) Statistical models are fit to the data.
(c) Results are displayed.

Statistical programming involves writing code to aid in statistical analysis (i.e., the "creating" part of statistical computing).

1.2 Why R?

1.2.1 What is R?

There are many applications that can be used for statistical computing: Excel, Fathom, Python, R, SAS, SPSS, Stata, etc.

We are interested in providing a foundation for understanding how statistical applications work. To do this, we will use R.

R is a statistical software package and programming language.

- R is widely used by statisticians all over the world.
- It is open source and free to download.
- Users can see how it is written and improve it.
- Users can contribute their own code to expand its functionality.

1.2.2 Interfaces

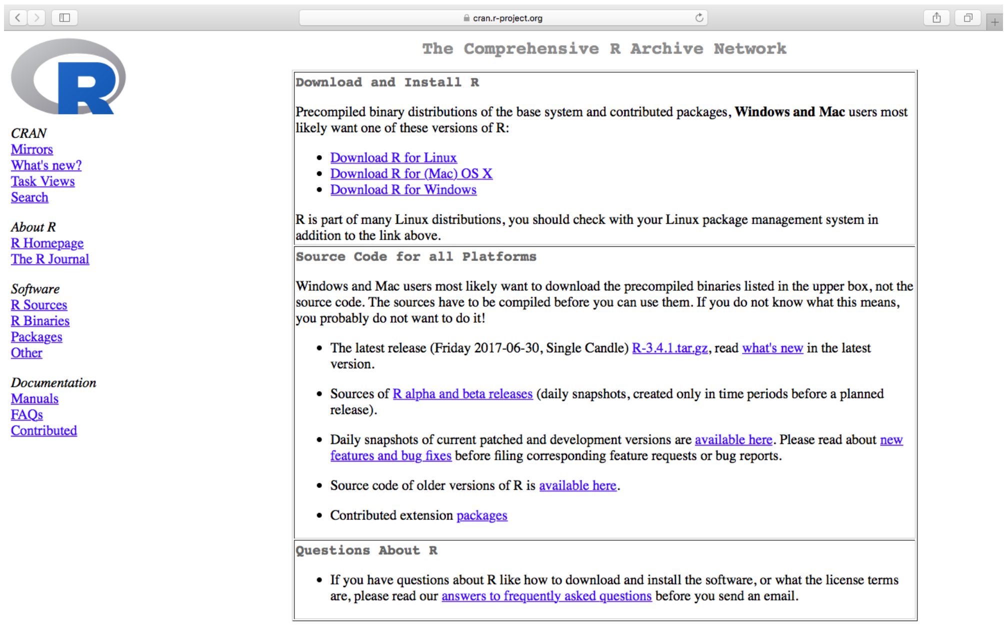
Some applications (e.g., Fathom and SPSS) have a **menu-based interface**. Menu-based applications are easy to learn and use. However, they are also inflexible, as there is a limited set of commands from which to choose.

Some applications (including R) have a **command-line interface** that requires the user to type commands for the computer to run. These are open ended applications in which you can write your own customized programs. These are harder to learn, since they require more programming knowledge. But by taking the time to learn, command-line applications allow for a more fundamental understanding of programming, and the skills you learn will be transferrable to other programming languages you may learn in the future.

2 Preliminaries

2.1 Installing R

To install R, go to https://cran.r-project.org/. Download and install the binaries for the base distribution for your operating system.



The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (Friday 2017-06-30, Single Candle) R-3.4.1.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

Figure 1: CRAN R Download page

Note: All computers in the computer lab and CLICC laptops have R preinstalled.

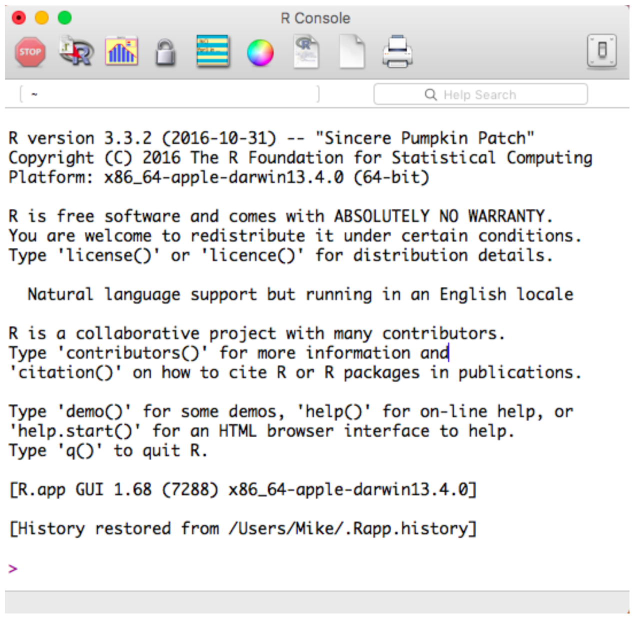
When you first open R, the **R console** will appear. The `>` symbol in the corner indicates the **command prompt**, where R commands are entered and processed by the computer.

To make sure R has been properly installed, enter the basic command "`2 + 3`" (without quotation marks) into the command prompt. Press `Enter` to submit (**run**) the command.

```r
2 + 3
```

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

The `[1]` in front of the output `5` is an **index** to tell us that the first entry (or element) of the output is the number 5. This will be important later, when we have output vectors with more than one entry.



```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (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.

[R.app GUI 1.68 (7288) x86_64-apple-darwin13.4.0]

[History restored from /Users/Mike/.Rapp.history]

>
```

Figure 2: The R Console

2.2 Incomplete Commands

After our previous command, notice that the R console now shows a new command prompt below the output. This means R is ready for another command.

What happens if you enter the incomplete command "`2 + `" into the command prompt?

The bottom left corner will show a `+` symbol, indicating that the command from the previous line is not complete. For R to be able to run, you need to complete the command.

Note: If you make a mistake and do not want to complete the command, press the Escape (`Esc`) key to skip the incomplete line and start a fresh command prompt.

2.3 Commenting

The # sign creates a comment so that the text is not read as an R command. It can be added at the beginning of the line or at the end of a command.

```
# This is a comment. This line is not evaluated by R.

2 + 3 # This line will be evaluated by R.
```

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

Note: It is *absolutely necessary* to add comments to your code. Not only do comments help other readers to understand your code, but it also makes sure that you can remember what your code is doing. Well documented programs will often have more comments than actual lines of code.

2.4 R Scripts

In most cases, typing directly into the R console is not ideal. For long and/or complicated programs, it is much more convenient to type the code in a separate text editor. This allows you to save the code you write and run multiple lines of code at once.

An **R script** (or **script file**) is a text file containing a set of R commands (i.e., a program). Within R, there is a basic built-in text editor that reads and writes R scripts. Any line of text written in the script file can be read into the console by pressing `Ctrl + R` on Windows or `Command + Enter` on Mac. Multiple lines can be run by first highlighting the lines you want to run.

You can save these text/script files as .R files. Using this filename extension, opening the script file will open in R.

2.5 Installing RStudio

To make coding in R a little easier, we will also use **RStudio**, which is an **integrated development environment (IDE)** designed to help users program in R. It allows you to more easily edit your programs, keep track of the objects in your workspace, search for help, and fix errors.

To install RStudio, go to https://www.rstudio.com/. Download and install the "Open Source License" version of "RStudio Desktop" for your operating system.

Note: All computers in the computer lab and CLICC laptops have RStudio preinstalled.

A sample RStudio display is shown in Figure 3. Notice that there are four panes.

- Bottom left: The R console
- Top left: The R script file
- Top right: Environment/History (more on this later), currently showing the `magnitudes` object in the global environment.
- Bottom right: Files/Plots/Packages/Help/Viewer, currently showing the histogram output from the last R command inputted into the console.

2.6 R Markdown

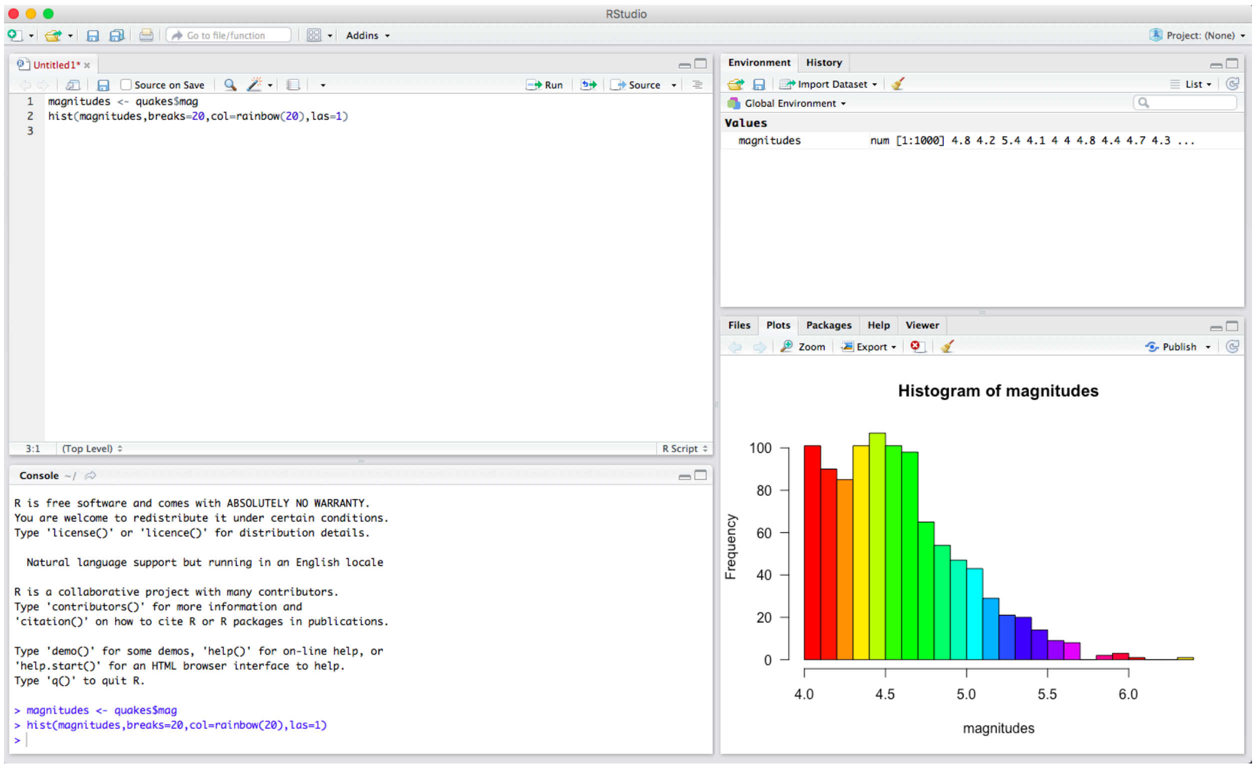
R Markdown is a useful and fairly simple way to generate documents that include R code and output. R Markdown files are text files that have a .Rmd extension. With a little bit of formatting syntax, R Markdown combines plain text and sections (or **chunks**) of R code that RStudio is able to compile (or *knit*) into a well formatted document for assignments, reports, and presentations. This set of notes was made using R Markdown.

An R Markdown document is dynamic and reproducible. The .Rmd file can be recompiled if the embedded R code or data changes, and the R output in the document will be updated accordingly.

We can create new R Markdown files in RStudio.

- Using the menu bar, choose "File – New File – R Markdown".
- A popup should appear. Input the title and author of the document.
- Choose what output format the document will be in (HTML, PDF, or Word). The default is HTML.
- After clicking "OK", an R Markdown template file will be created.
- To knit the file into an HTML document, click on the "Knit" button and choose "Knit to HTML".

Note: To knit to a PDF file, LaTeX needs to be installed on your computer.



```
magnitudes <- quakes$mag
hist(magnitudes,breaks=20,col=rainbow(20),las=1)
```

```
Values
magnitudes        num [1:1000] 4.8 4.2 5.4 4.1 4 4 4.8 4.4 4.7 4.3 ...
```

```
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.

> magnitudes <- quakes$mag
> hist(magnitudes,breaks=20,col=rainbow(20),las=1)
>
```

Figure 3: A sample RStudio display

2.7 Working Directories

A working directory is the default folder (or directory) from which R reads and writes data.

```
getwd() # Returns the current working directory
```

```
## [1] "/Users/Mike/Dropbox/School/Teaching/Fall 2017/Stats 20/Lecture Notes/Chapter 1"
```

```
setwd("~/Desktop") # Changes working directory to desktop
```

You can also get/set the working directory from the menu bar in R or RStudio.

Note: Since different data sets may use the same variable names, it is helpful to use a different working directory for each project or assignment.

2.8 Quitting R

To quit an R session, run the quit function **q()**:

```
q()
```

Upon quitting, a popup will ask if you want to save an image of the current workspace. The workspace image will save a record of the computations and the saved results from the session in the working directory (the record of the computations are saved in a file called .Rhistory and the results are saved in a file called .RData).

Rather than saving the workspace, it is usually preferrable to save the R commands (in an R script) and reproduce the results later. For extensive programs that are difficult to reproduce quickly, manually saving objects in the workspace before quitting R gives more control over what you are saving (more on this later).

3 Calculator Computations

3.1 Basic Arithmetic

By typing arithmetic expressions into the command prompt, R is able to do basic calculator computations. Some examples:

```
2 + 1 # This is addition
```

```
## [1] 3
```

```
7 * 3 # This is multiplication
```

```
## [1] 21
```

```
2 - 1 # This is subtraction
```

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

```
117 / 3 # This is division
```

```
## [1] 39
```

```
1 + pi # pi is a stored constant
```

```
## [1] 4.141593
```

Note: The spaces between the numbers and arithmetic operators (`+`, `-`, `*`, `/`) are not strictly necessary, but they are helpful for readability.

R computes powers (exponentiation) with the ^ operator.

```
3^4
```

```
## [1] 81
```

```
5^3
```

```
## [1] 125
```

3.2 Modular Arithmetic

R is also able to do modular arithmetic using the `%%` and `%/%` operators. The `%%` operator computes the remainder after division. For example, the remainder after division of 46 by 7, i.e., 46 (mod 7), is written as:

```
46 %% 7
```

```
## [1] 4
```

The integer part of a fraction uses the `%/%` operator. If we want to know how many times 46 goes into 7, we can write:

```
46 %/% 7
```

```
## [1] 6
```

Question: What is the output of the following line?

```
67 %% 8
```

Question: What is the output of the following line?

```
67 %/% 8
```

3.3 Order of Operations

Question: What is the output of the following line?

```
2 + 3 * 3
```

The way the line is written, it is not clear which operation R will compute first: `2 + 3` or `3 * 3`.

Computing `2 + 3` would be true if R reads commands sequentially from left to right, the way that most simple calculators do. Fortunately, R is not a simple calculator. R follows the standard **order of operations** from basic arithmetic: PEMDAS (or BEDMAS).

- Parentheses (or Brackets)
- Exponents
- Multiplication and Division
- Addition and Subtraction

Question: What is the output of the following line?

```
4 * 6 + 2 * 3
```

Note: Whenever an expression could be ambiguous, it is *highly recommended* to use parentheses, even if they are not strictly necessary to yield the correct computation. This adds to the readability and clarity of your code.

```
(4 * 6) + (2 * 3) # Yields the same results as above but is more readable
```

Round parentheses `()` and curly braces `{}` can be used here, but square brackets `[]` will give you an error. Square brackets have a specific use in R syntax (as we will see), so they cannot be used here.

4 Objects and Functions

John Chambers (co-inventor of S, the precursor to R):

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.

R is what is known as an **object-oriented language**: Everything in R is stored as an object, and programs are written around manipulating objects. Formally, an **object** is a storage space (or data structure) with an associated name.

All objects created and stored in an R session are collectively called the **workspace**, also known the **global environment**.

4.1 Object Assignment

To save objects/functions/anything into R's current workspace, use the **assignment** operator `<-`. Names of objects/functions/anything can contain letters, numbers, periods, or underscores and must start with a letter.

Caution: R is case-sensitive. `object`, `OBJECT`, and `ObJeCt` are all treated as different names.

The name of the object you want to make goes on the left of the `<-`. The `<-` is meant to resemble an arrow pointing to the left, to signify that you are assigning the output of the code on the right to the object name on the left.

Note: The single equal sign `=` can also be used for assignment, but it is recommended to use the arrow `<-`, since the direction of the assignment is clearer.

4.1.1 Example

Andy Dwyer is interested in opening a savings account at Pawnee National Bank. Suppose the bank has a savings account that offers a 0.25% interest rate per year, compounded annually. What is the overall interest rate after 20 years? If Andy Dwyer put in an initial balance of $5000, what will the final balance be?

The interest rate over 20 years is $(1 + 0.0025)^{20} = 1.0025^{20}$. We can save this value in our workspace to use later.

```
interest.20 <- 1.0025^20 # Save this value in the interest.20 object
```

Notice that R will not show any output after running this command. R has done what we asked (assigned the value to an object), so it is ready for the next command.

Now that we have stored the value in `interest.20`, we can recall the value by typing the name of the object.

```
interest.20
```

```
## [1] 1.051206
```

We can now use this object in our computations of the final balance.

```
initial.balance <- 5000
final.balance <- initial.balance * interest.20
final.balance
```

```
## [1] 5256.028
```

For an initial balance of $5000, the final balance after 20 years at 0.25% interest, compounded annually, is $5256.03.

4.1.2 Common Errors

Even the best programmers sometimes make mistakes. When you give R a command with a syntax mistake, R will return (or **throw**) an error message to tell you something is wrong. Here are some common errors associated with object assignment and recall:

```
20.interest <- 1.0025^20 # This gives an error. Why?
```

```
## Error: unexpected symbol in "20.interest"
```

```
INTEREST.20 # This also gives an error. Why?
```

```
## Error in eval(expr, envir, enclos): object 'INTEREST.20' not found
```

Knowing what error messages mean and why you get them will help you debug your code and reinforce your comfort with R syntax.

4.1.3 Masking Built-In Objects

R has many built-in objects predefined in R, including a few built-in constants. For example, the mathematical constant π can be accessed by typing **`pi`**:

```
pi
```

```
## [1] 3.141593
```

Caution: If we assign the name of a built-in constant to a new value, R will *not* give an error. The user assigned object will **mask** the built-in constant, so the object name will reference the user assigned object rather than the built-in one.

```
pi <- 3 # Declare pi to be exactly 3
pi
```

```
## [1] 3
```

By redefining the `pi` object, any calculations involving π will be wrong. For example, suppose we are interested in the area of a circle of radius 3:

```
pi * (3^2) # Compute (incorrectly) the area of a circle of radius 3
```

```
## [1] 27
```

To properly reference the built-in constant again, we first need to remove the user defined object from the workspace. This can be done with the following commands:

```
rm(pi) # Remove the pi object
pi # pi now references the built-in constant again
```

```
## [1] 3.141593
```

```
pi * (3^2) # Compute (correctly) the area of a circle of radius 3
```

```
## [1] 28.27433
```

Always use caution when naming objects to make sure you are not unknowingly masking built-in objects.

4.2 Functions

Functions are a special type of object in R. They have a similar use and purpose as functions in mathematics. In mathematics, a function is a relation between a set of inputs and a set of outputs such that each input is related to a single output. A function in R takes in certain input objects called **arguments** and returns an output by executing a set of commands.

4.2.1 Function Syntax

The basic syntax to use (or **call**) a function is:

```
functionname(argument(s))
```

A function has a name, parentheses, and arguments. Some arguments are required, some are optional, and some have built-in or **default** values. Multiple arguments are separated by commas.

There are many built-in functions already in R. For example, consider the quit function `q()`:

```
q
```

```
## function (save = "default", status = 0, runLast = TRUE)
## .Internal(quit(save, status, runLast))
## <bytecode: 0x7f9206aaf8d0>
## <environment: namespace:base>
```

By typing the name of the quit function without parentheses, R shows us how the function is defined. There are three arguments: `save`, `status`, and `runLast`. Each argument has a default value specified by a single equal sign `=`: `"default"`, `0`, and `FALSE`, respectively. We typically call `q()` with empty parentheses so that the arguments are left with their default values.

R function arguments can be matched positionally or by name. If we know the order of the arguments, we do not need to use the name in order to specify the arguments. However, it is typically better to use named arguments for clarity.

```r
q("no") # Sets the save argument to "no"
q(save = "no") # Same thing (but clearer)

q(,,FALSE) # Sets the runLast argument to FALSE
q(runLast = FALSE) # Same thing (but clearer)
```

Notice that blank arguments do not erase the default values.

```r
q(FALSE) # This gives an error. Why?
```

```
## Error in quit(save, status, runLast): one of "yes", "no", "ask" or "default" expected.
```

The first argument of `q()` is the `save` argument, which expects an input of either `"yes"`, `"no"`, or `"default"`. If a different input is given (such as the value `FALSE`), R will throw an error.

Some other basic functions:

```r
sqrt(16) # The square root function
```

```
## [1] 4
```

```r
exp(1) # The exponential function
```

```
## [1] 2.718282
```

```r
log(exp(2)) # The logarithm function (notice the default base value)
```

```
## [1] 2
```

```r
log(10,base=10) # Changes the optional base argument to 10
```

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

The code `log(10,10)` is technically acceptable, but it requires knowing that the optional second argument of `log()` is the `base` argument that controls the base of the logarithm.

Side Note: *Everything* that R does is done through calls to functions, though the calls are sometimes hidden (like when we click on menus) or very basic (like calculator computations). For example:

```r
`*`(4,3) # The functional way to write multiplication
```

```
## [1] 12
```

```r
`-`(7,2) # The functional way to write subtraction
```

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

Notice that the name of the functions use backticks, since the arithmetic symbols alone are inadmissible object names. It is important to see this functional representation at least once, but we will not use this notation for the rest of the course.

4.2.2 Creating Functions

Any user can create new functions that build on top of the base R objects and functions. By creating our own functions, we can extend the functionality of R in practically unlimited ways.

Since functions are considered objects in R, we can still assign a name to the function and save it to the workspace using the assignment `<-` operator.

The definition of a function typically has the following components:

- The word `function`.
- A pair of round parentheses `()` containing the argument list (it may be empty).
- A pair of curly braces `{}` containing a statement or sequence of statements (called the **body** of the function).

The basic syntax to create your own function is

```
function.name <- function(arguments){
  # The body of the function goes here.
}
```

A few notes:

- Indenting the body of the function is highly recommended, as it helps make the code more readable.
- The output of the last command in the body will be the output of the function.
- The body of the function creates a **local environment**, so objects created inside the function are accessible only within the function. The objects created locally will not appear in the global environment (your workspace).
- Any objects in the global environment are still accessible within the function's local environment. The global environment is thus also called the **parent environment** in relation to the local environment.

Caution: Objects in the local environment will take precedence over objects in the global environment. In particular, if you use the same name for something within the body of a function as an object in the global environment, the global object will be **masked** by the local object: The object name within the function will reference the local object instead of the global one.

We will go into more detail about functions in a later chapter.

4.2.3 Example

Now that Andy Dwyer has a savings account with Pawnee National Bank, he tells Tom Haverford to open one too. Tom's account has the same annual interest rate of 0.25%, compounded annually, but his initial balance is $3500. We want to write a function to compute the final balance after 20 years.

Here is one possible way to write this function:

```
final.balance <- function(initial){
  # This function inputs the argument initial = initial balance
  # and outputs the final balance after 20 years
  # at an annual interest rate of 0.25% compounded annually.

  # Specify interest rate as a decimal and save it to the interest.rate object.
  interest.rate <- 0.25/100

  # Specify the number of years to accrue interest.
  years <- 20

  # Compute the final balance and save it to the final object.
  final <- initial * (1 + interest.rate)^years
  final # Output the final balance
}
```

```r
final.balance(3500) # Compute final balance for an initial balance of 3500
```

```
## [1] 3679.219
```

```r
final.balance(initial=3500) # Same thing
```

```
## [1] 3679.219
```

Question: Typing `final.balance()` will give an error. Why? How can we fix it?

Question: How would we rewrite the function if we also wanted to vary the interest rate and the number of years?

4.3 Managing the Workspace

You can see what is saved in your workspace with the **`objects()`** function. The **`ls()`** function does the exact same thing.

```r
objects()
```

```
## [1] "final.balance"   "initial.balance" "interest.20"
```

```r
ls() # Same as objects()
```

```
## [1] "final.balance"   "initial.balance" "interest.20"
```

The **`rm()`** function allows you to remove objects from the workspace. For example, to remove the `initial.balance` object from the workspace we can input the object name between the parentheses, we can write:

```r
rm(initial.balance)

ls()
```

```
## [1] "final.balance" "interest.20"
```

To remove *everything* from the workspace (what I call *the nuclear option*), we can write:

```r
rm(list=ls()) # Kaboom!

ls() # Verify that the workspace is empty
```

```
## character(0)
```

The output `character(0)` means that the list of objects in the workspace is a character vector of length 0. We will discuss vectors in the next chapter. In this case, the vector of length 0 just means that there are no objects in the workspace.

Caution:

- *Only* use `rm()` (and especially `rm(list=ls())`) if you are certain that you do not need those objects anymore. *Removing objects is not reversible.*
- If you quit the R session without saving the workspace image, then all of the objects in the workspace will disappear.