Coding in R

An introduction to practical programming in R

Gaston Sanchez

2022-02-05

Contents

Welcome **5**
Yet Another R Book 5

I Getting Started with R and RStudio **7**

1 Installing R and RStudio **9**
1.1 Interacting with R 9
1.2 Installing R 11
1.3 Installing RStudio 14

2 Breaking the Ice with R **17**
2.1 First Contact with R (via RStudio) 17
2.2 Getting Help 22
2.3 Installing Packages 23
2.4 Exercises 25

3 A Quick Tour Around RStudio **27**
3.1 First Contact with RStudio 27
3.2 RStudio Panes in a Nutshell 30
3.3 Exercises 32

4 Session Management **33**
4.1 Starting a Session 33
4.2 Working During a Session 35
4.3 Closing a Session 37

II Data Objects in R **39**

5 Vectors **41**
5.1 Motivation: Compound Interest 41
5.2 About R Vectors 44
5.3 Vectors are Atomic Objects 45

5.4 Creating Vectors 48
5.5 Coercion 51

6 More About Vectors **53**
6.1 Motivation: Future Value 53
6.2 Vectorization 55
6.3 Recycling 56
6.4 Manipulating Vectors: Subsetting 59

7 Matrices and Arrays **63**
7.1 Motivation 63
7.2 Tables with Matrices 64
7.3 Creating matrices with `matrix()` 66

Welcome

This book is a work in progress for a text on programming in R.

We discuss the basics of R, covering how to manipulate data objects such as vectors, factors, matrices, data frames, and lists. We also cover data manipulation, reshaping, tidying, etc. Likewise, we describe how to make basic (and not so) graphics.

Yet Another R Book

Why writing another book about programming in R? Quick answer: "Why not?" The truth is that for many years I refrained myself from writing a book like the one you are reading now.

However, after using R—almost on a daily basis—for more than 16 years, and having taught STAT 133 "Concepts in Computing with Data", for the past 6 years, I've accumulated a large body of notes, slides, scripts, exercises and that sort of things, that deserve a place such as this a textbook. In addition, it fits my teaching needs for courses such as STAT 133, STAT 33A, STAT 33B, STAT 243, and similar courses on programming, data analysis, data science, with R.

As you will see, the book has an overall theme based on financial math. The main reason for this decision is to provide a unifying theme under which most topics can be tidely presented, instead of offering a number of examples in a vacuum.

Part I

Getting Started with R and RStudio

Chapter 1

Installing R and RStudio

To learn how to program in R, you obviously need to have access to it. This chapter guides you through the installation process so that you can have both R and RStudio up and running in your computer.

If you already have R and RStudio installed in your machine, you can safely skip this chapter.

1.1 Interacting with R

Before I show you how to install R and the integrated development environment RStudio, let me tell you first about the different ways in which users can work with R.

Overall, you can work with R in two major ways:

1) Interactive Mode, and

2) Non-interactive Mode

Interactive Mode. Most R users work with R in an interactive way, and this is certainly the way I personally interact with R in my daily coding activities. Interactive means that you launch R (i.e. you open a session), having direct access to its console. This is where you type in commands, and then R does its magic reading the commands, parsing them, evaluating them, and—typically—printing an output back into the console, waiting for you to type in the next command(s).

Non-interactive Mode. In contrast to interactive mode, working with R in non-interactive way involves writing all the commands in a text file (for example in an `R` script file), and then asking your computer—via the command line interface—to pass this file to R so that it runs the commands without you launching R or having direct access to its console.

Think of interactive way as having a direct conversation with R, establishing a dialogue in which you type in a command, R interprets it, gives you an answer, and then you type more commands, continuing the dialogue. In constrast, non-interactive is like writing a letter (or an email) to R. Here you don't have that synchronous conversation, instead R will see the script file, try to execute all the commands "behind the scenes", and it will disappear when it finishes the computations. You may get some output back, but R is gone in the sense that there is no open session waiting for you to execute the next instructions.

Most of what I discuss in this book is applicable to writing code in R regardless of how you decide to interact with R. However, to learn R and to follow the examples of the book it is definitely much better to do it in an interactive way using: a) R's built-in graphical user interface (R's GUI), b) an integrated development environment (IDE) such as RStudio, or c) R from a command line interface (CLI) commonly referred to as a *terminal*.

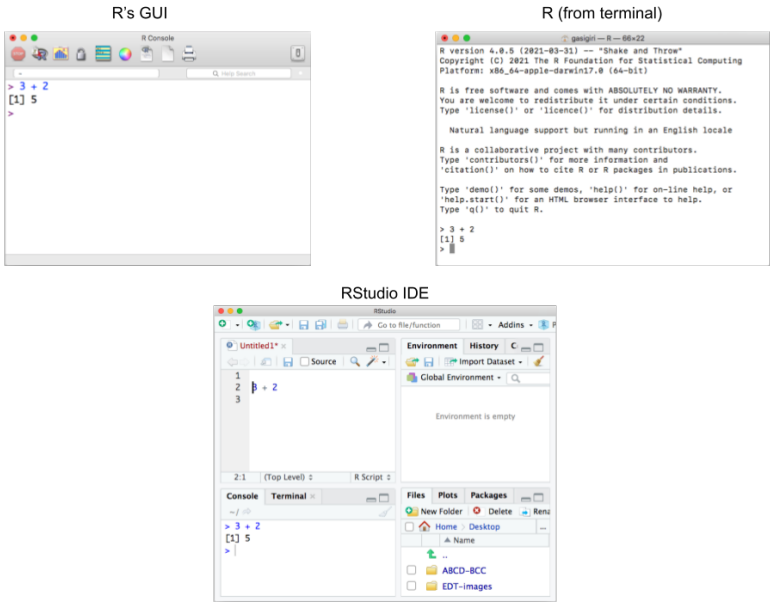


Figure 1.1: Interacting with R in various ways: R's GUI, RStudio, and R from a command-line terminal

While you can interact with R using its built-in graphical user interface (GUI) or launching R from the terminal (command line interface), nowadays I highly recommend that you interact with it using an *Integrated Development Environment* (IDE) such as **RStudio**. Simply put, programs like RStudio provide a nice working space that make your life easier while writing code, creating all sorts of reports, documents, and slides, running analysis, making graphs, generating outputs, creating web apps, etc.

Figure 1.2: Main computational tools: R and RStudio

Keep in mind that R and RStudio are not the same thing. R is like the main "engine" or computational core. RStudio is just a convenient layer that talks directly to R, and gives us a convenient working space to organize our files, to type in code, to run commands, visualize plots, interact with our filesystem, etc. Having said that, everything that happens in RStudio, can be done in R alone. Yes, you may need to write more code and work in a more rudimentary way, but nothing should stop your work in R if one day RStudio disappears from the face of the earth.

By the way, both R and RStudio are free, and available for Mac (OS X), Windows, and Linux (e.g. Ubuntu, Fedora, Debian). More about this in the following sections.

1.2 Installing R

To download and install R in your computer, follow the steps listed below.

Step 1) Go to the **R project** website: https://r-project.org

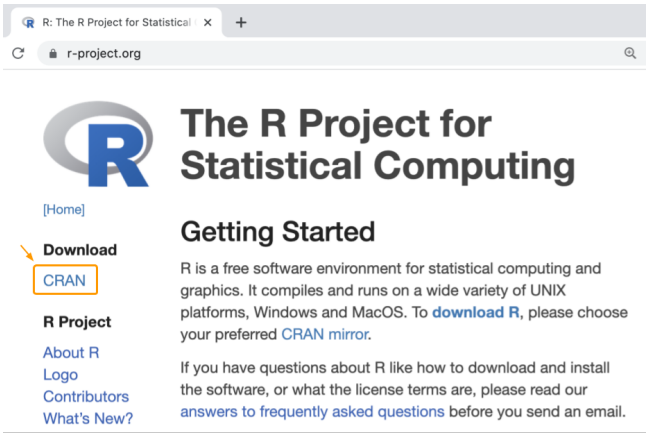


Figure 1.3: R project's home webpage

Step 2) Click on the CRAN link, located in the navigation bar (on the left side). This will take you to the *Comprehensive R Archive Network* page (see screenshot below).

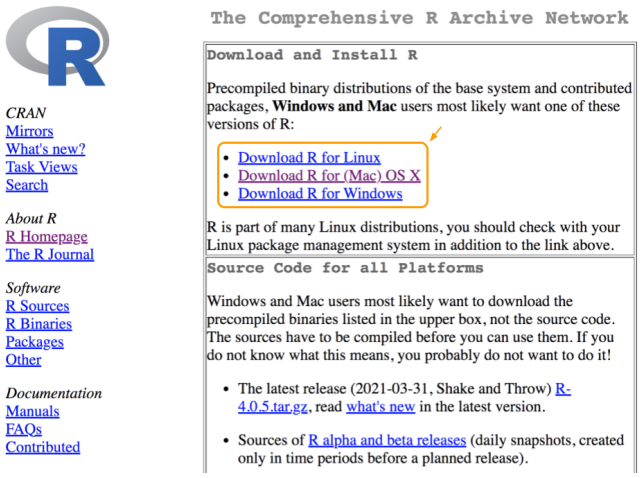


Figure 1.4: R is available for MacOS, Windows, and Linux

Step 3) Click on the download option that corresponds to your operating system (e.g. Linux, Mac, or Windows). In my case, I have a Mac computer, which explains why the Mac OS-X link is highlighted in the above screeenshot.

For most users, you will want to install the **Latest release**, which in the screenshot above happens to be `R 4.0.5 "Shake and Throw"`. Keep in mind that by the time you read this book, R will very likely have a more recent version.

Step 4) Click on the package link, which in the screenshot corresponds to `R-4.0.5.pkg`. This is the link of a compressed file that contains the binary code. Before installing a given version of R, read the description of the release to make sure the operating system in your computer is compatible with a specific version of R.

After clicking on the `R-4.0.5.pkg` link, the compressed file will be downloaded to your computer.

Step 5. Click on the downloaded file. An installation wizard will open automatically, ready to guide you through the installation process, step by step (see image below).

In most cases, you will want to use the default settings. Personally, I've been using the default settings for several years without having the need to customize anything.

At the end of the installation, if everything went well, you should be able to see a successful message (see figure below):

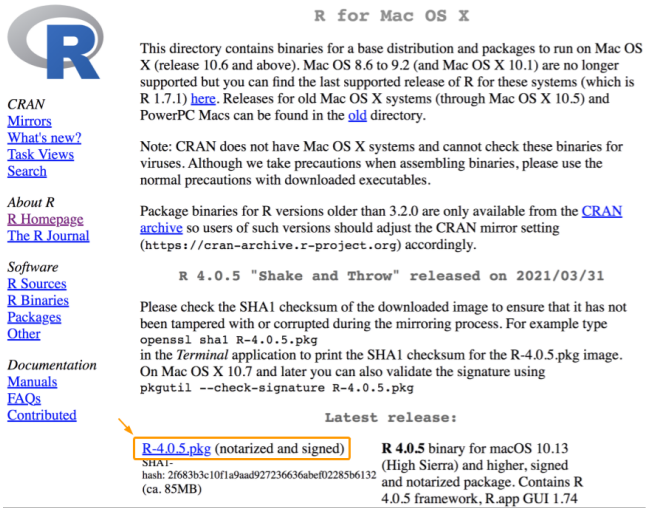


Figure 1.5: CRAN download for Mac

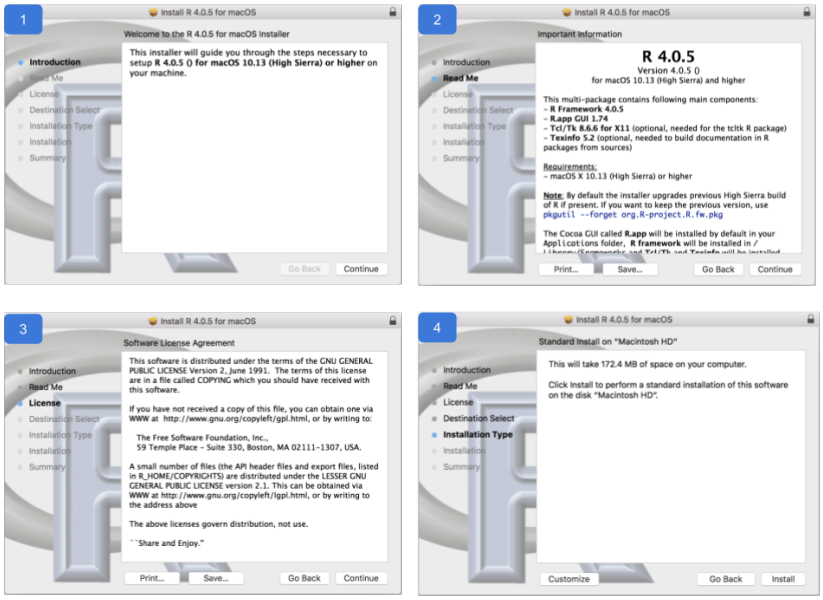


Figure 1.6: R Installation wizard for Mac

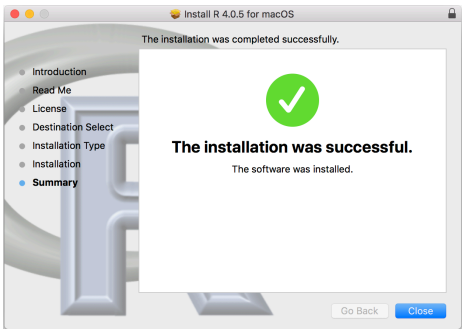


Figure 1.7: R Installation wizard for Mac

1.3 Installing RStudio

In addition to R, the other program you will need to have installed in your machine is RStudio. To download and install it, follow the steps listed below.

Step 1) Go to **RStudio's** download webpage:

https://www.rstudio.com/products/rstudio/download/

Figure 1.8: RStudio download options

At the time of this writing, there are four options of RStudio.

Step 2) Choose the **free** version of RStudio Desktop (see image below), and click on the "DOWNLOAD" button.

Step 3) Select the version that matches your operating system (e.g. Windows, macOS, linux). Double check that the operating system in your computer is compatible with a specific version of RStudio.

Once the installation of RStudio is completed, you should be able to open a new

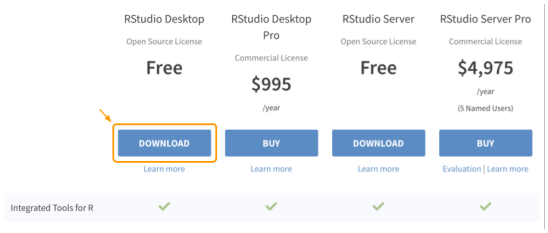


Figure 1.9: Choose RStudio free desktop

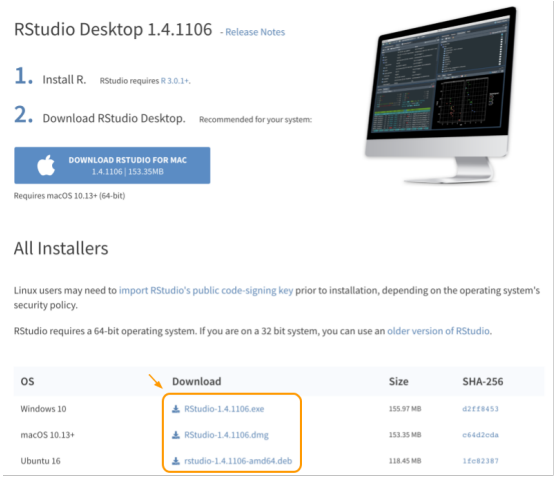


Figure 1.10: RStudio Desktop versions

session in RStudio, and also to interact with R.

Chapter 2

Breaking the Ice with R

If you are new to R and don't have any programming experience, then you should read this chapter in its entirety. If you already have some previous experience working with R and/or have some programming background, then you may want to skim over most of the introductory chapters of part I.

This chapter, and the rest of the book, assumes that you have installed both R and RStudio in your computer. If this is not the case, then go to chapter Installing R and RStudio and follow the steps to download and install these programs.

R comes with a simple built-in graphical user interface (GUI), and you can certainly start working with it right out of the box. That is actually the way I got my first contact with R back in 2001 during my senior year in college. Nowadays, instead of using R's GUI, it is more convenient to interact with R using a third party software such as RStudio.

I describe more introductory details about RStudio in the next chapter A Quick Tour Around RStudio. For now, go ahead and launch RStudio in your computer.

2.1 First Contact with R (via RStudio)

When you open RStudio, you should be able to see its layout organized into quadrants offically called *panes*. The very first time you launch RStudio you will only see three panes, like in the screenshot below.

To help you break the ice with R, it's better if we start working directly on the **Console**.

As you can tell from the following screenshot, the console is located in the left-hand side quadrant of RStudio. Keep in mind that your RStudio's console pane may be located in a different quadrant.

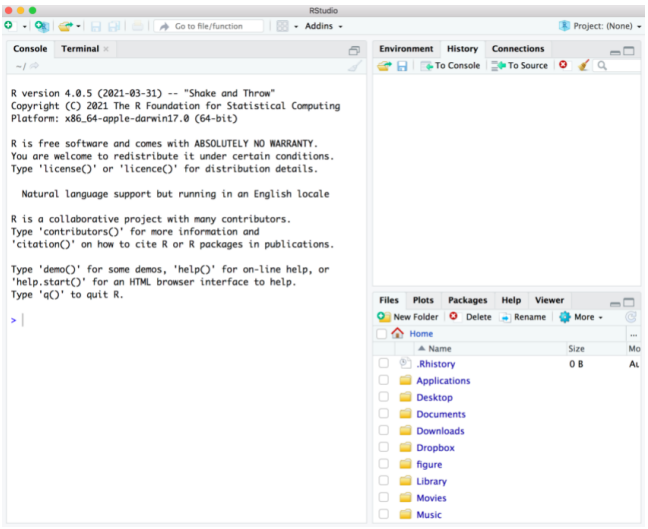


Figure 2.1: Screenshot of RStudio when launched for the first time.

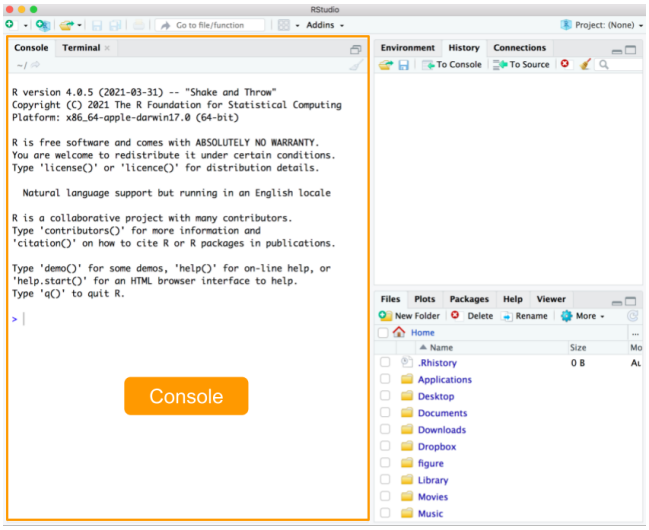


Figure 2.2: Console quadrant in RStudio.

Technically speaking, the console is a terminal where a user inputs commands and views output. Simply put, this is where you can directly interact with R by typing commands, and getting the output from the execution of the commands.

2.1.1 R as a scientific calculator

This first activity is dedicated for readers with little or no programming experience, especially those of you who have never used software in which you have to type commands. The idea is to start typing simple things in the **console**, basically using R as a scientific calculator.

Here's a toy example. Consider the monthly bills of an undergraduate student:

- cell phone $80
- transportation $20
- groceries $527
- gym $10
- rent $1500
- other $83

You can use R to find the student's total expenses by typing these commands in the console:

```
80 + 20 + 527 + 10 + 1500 + 83
```

There is nothing surprising or fancy about this piece of code. In fact, it has all the numbers and all the `+` symbols that you would use if you had to obtain the total expenses by using the calculator in your cellphone.

2.1.2 Assigning values to objects

Often, it will be more convenient to create **objects**, sometimes also called **variables**, that store one or more values. To do this, type the name of the object, followed by the assignment or "arrow" operator `<-`, followed by the assigned value. By the way, the arrow operator consists of a left-angle bracket `<` (or "less than" symbol) and a dash or hyphen symbol `-`.

For example, you can create an object `phone` to store the value of the monthly cell phone bill, and then inspect the object by typing its name:

```
phone <- 80
phone
#> [1] 80
```

All R statements where you create objects are known as **assignments**, and they have this form:

```
object <- value
```

this means you assign a `value` to a given `object`; one easy way to read the previous assignment is "phone gets 80".

Alternatively, you can also use the equals sign `=` for assignments:

```
transportation = 20
transportation
#> [1] 20
```

As you will see in the rest of the book, I've written most assignments with the arrow operator `<-`. But you can perfectly replace them with the equals sign `=`. The opposite is not necessarily true. There are some especial cases in which an equals sign cannot be replaced with the arrow, but we'll talk about this later.

Pro tip. RStudio has a keyboard shortcut for the arrow operator`<-`:

- Windows & Linux users: `Alt` + `-`
- Mac users: `Option` + `-`

In fact, there is a large set of keyboard shortcuts. In the menu bar, go to the *Help* tab, and then click on the option *Keyboard Shorcuts Help* to find information about all the available shortcuts.

2.1.3 Object Names

There are certain rules you have to follow when creating objects and variables. Object names cannot start with a digit and cannot contain certain other characters such as a comma or a space.

The following are invalid names (and invalid assignments)

```
# cannot start with a number
5variable <- 5

# cannot start with an underscore
_invalid <- 10

# cannot contain comma
my,variable <- 3

# cannot contain spaces
my variable <- 1
```

People use different naming styles, and at some point you should also adopt a convention for naming things. Some of the common styles are:

```
snake_case

camelCase

period.case
```

Pretty much all the objects and variables that I created in this book follow the "snake_case" style. It is certainly possible that you may endup working with a team that has a styleguide with a specific naming convention. Feel free to try various style, and once you feel comfortable with one of them, then stick to it.

2.1.4 Case Sensitive

R is case sensitive. This means that `phone` is not the same as `Phone` or `PHONE`

```
# case sensitive
phone <- 80
Phone <- -80
PHONE <- 8000

phone + Phone
#> [1] 0

PHONE - phone
#> [1] 7920
```

Again, this is one more reason why adopting a naming convention early on in a data analysis or programming project is very important. Being consistent with your notation may save you from some headaches down the road.

2.1.5 Calling Functions

Like any other programming language, R has many functions. To use a function type its name followed by parenthesis. Inside the parenthesis you typically pass one or more inputs. Most functions will produce some type of output:

```
# absolute value
abs(10)
abs(-4)

# square root
sqrt(9)

# natural logarithm
log(2)
```

In the above examples, the functions are taking a single input. But often you will be working with functions that accept several inputs. The `log()` function is one them. By default, `log()` computes the natural logarithm. But it also has the `base` argument that allows you to specify the base of the logarithm, say to `base = 10`

```
log(10, base = 10)
#> [1] 1
```

2.1.6 Comments in R

All programming languages use a set of characters to indicate that a specifc part or lines of code are **comments**, that is, things that are not to be executed. R uses the hash or pound symbol `#` to specify comments. Any code to the right of `#` will not be executed by R.

```
# this is a comment
# this is another comment
2 * 9

4 + 5  # you can place comments like this
```

You will notice that I have included comments in almost all of the code snippets shown in the book. To be honest, some examples may have too many comments, but I've done that to be very explicit, and so that those of you who lack coding experience understand what's going on. In real life, programmers use comments, but not so much as I do in the book. The main purpose of writing comments is to describe—conceptually—what is hapenning with certain lines of code. Some would even argue that comments should only be used to express not the what but the **why** a developer is doing something.

2.2 Getting Help

Because we work with functions all the time, it's important to know certain details about how to use them, what input(s) is required, and what is the returned output.

So how do you find all this information technically known as **documentation**? There are several ways to access this type of information.

If you know the name of a function you are interested in knowing more about, you can use the function `help()` and pass it the name of the function you are looking for:

```
# documentation about the 'abs' function
help(abs)

# documentation about the 'mean' function
help(mean)
```

Alternatively, you can use a shortcut using the question mark `?` followed by the name of the function:

```
# documentation about the 'abs' function
?abs

# documentation about the 'mean' function
?mean
```

`help()` only works if you know the name of the function your are looking for. Sometimes, however, you don't know the name of the function but you may know some keyword(s). To look for related functions associated to a keyword, use `help.search()` or simply type double question marks `??`

```
# search for 'absolute'
help.search("absolute")

# alternatively you can also search like this:
??absolute
```

Notice the use of quotes surrounding the input name inside `help.search()`

Often overlooked by beginners but extremely helpful is to understand the anatomy of the information displayed in the technical documentation. The content is typical organized into seven sections listed below (although sometimes you have less or more sections)

- Title
- Description
- Usage of function
- Arguments
- Details
- See Also
- Examples

The three screenshots below show the "Help" or technical documentation of the `log()` function. This information is in RStudio's `Help` tab, located in the pane that contains other tabs such as `Files`, `Plots`, `Packages`.

2.3 Installing Packages

R comes with a large set of functions and packages. A package is a collection of functions that have been designed for a specific purpose. One of the great advantages of R is that many analysts, scientists, programmers, and users can create their own pacakages and make them available for everybody to use them. R packages can be shared in different ways. The most common way to share a package is to submit it to what is known as **CRAN**, the *Comprehensive R Archive Network*.

You can install a package using the `install.packages()` function. To do this, I recommend that you run this command directly on the console. In other words, do **not** include this command in a source file (e.g. `R` script file, `Rmd` file). The reason for running this command directly on the console is to avoid getting an error message when running code from a source file.

To use `install.packages()` just give it the name of a package, surrounded by quotes, and R will look for it in CRAN, and if it finds it, R will download it to

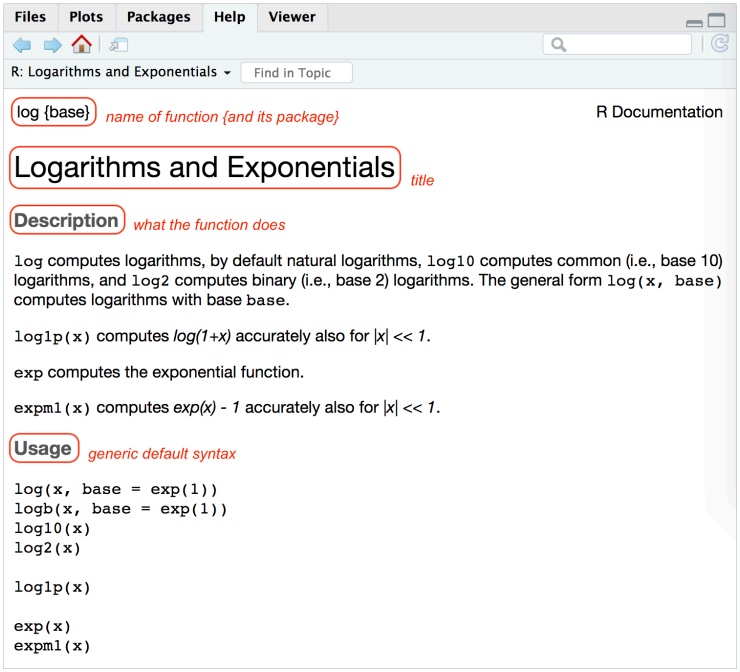


Figure 2.3: Help documentation for the log function (part 1)

your computer.

```
# installing (run this on the console!)
install.packages("knitr")
```

You can also install a bunch of packages at once by placing their names, each name separated by a comma, inside the `c()` function:

```
# run this command on the console!
install.packages(c("readr", "ggplot2"))
```

Once you installed a package, you can start using its functions by *loading* the package with the function `library()`. For better of worse, `library()` allows you to specify the name of the package with or without quotes. Unlike `install.packages()` you can only specify the name of one package in `library()`

```
# (this command can be included in an Rmd file)
library(knitr)      # without quotes
library("ggplot2")  # with quotes
```

By the way, you only need to install a package once. After a package has been installed in your computer, the only command that you need to invoke to use

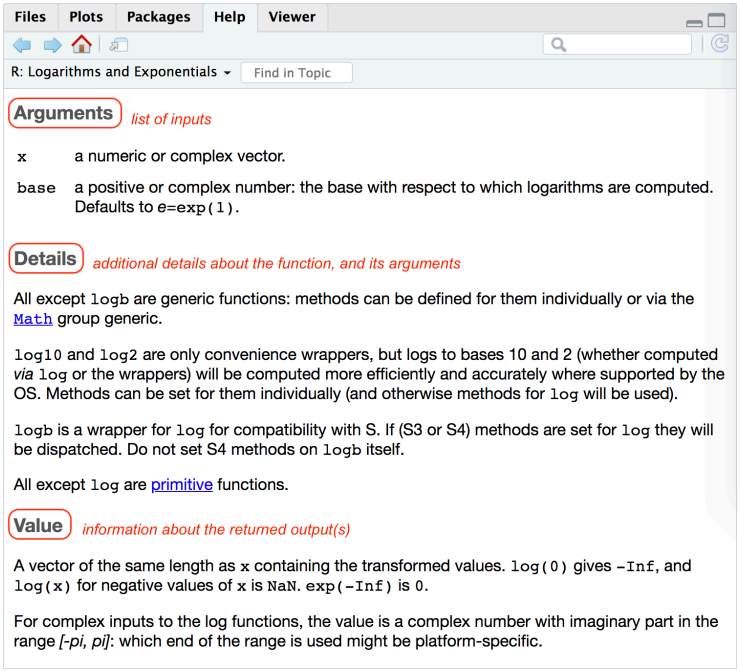


Figure 2.4: Help documentation for the log function (part 2)

its functions is the `library()` function.

2.4 Exercises

1) Here's the list of monthly expenses for a hypothetical undergraduate student

- cell phone $80
- transportation $20
- groceries $527
- gym $10
- rent $1500
- other $83

a) Using the `console` pane of RStudio, create objects (i.e. variables) for each of these expenses and create an object `total` with the sum of the expenses.

b) Assuming that the student has the same expenses every month, how much would she spend during a school "semester"? (assume the semester involves five months). Write code in R to find this value.

c) Maintaining the same assumption about the monthly expenses, how much would she spend during a school "year"? (assume the academic year is 10

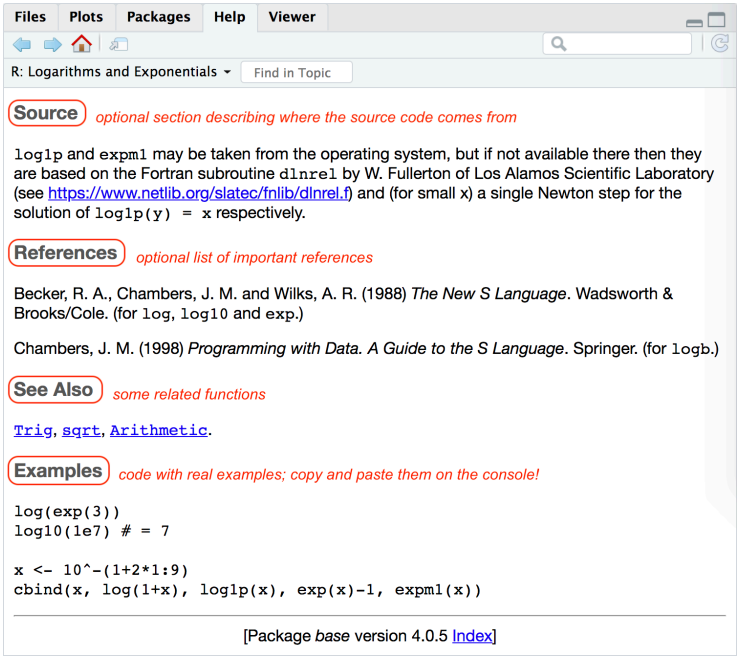


Figure 2.5: Help documentation for the log function (part 3)

months). Write code in R to find this value.

2) Use the function `install.packages()` to install packages `"stringr"`, `"RColorBrewer"`, and `"bookdown"`

3) Use the console to write code for calculating: $3x^2 + 4x + 8$ when $x = 2$

4) Calculate: $3x^2 + 4x + 8$ but now with a numeric sequence for x using `x <- -3:3`

5) Find out how to look for information about math binary operators like `+` or `^` (without using `?Arithmetic`). *Tip*: quotes are your friend.

Chapter 3

A Quick Tour Around RStudio

As I mentioned in the previous chapter, R comes with a simple built-in graphical user interface, or *GUI* for short. While you can use this interface to work with R, it is more convenient if you interact with R using a third party software such as RStudio.

Technically speaking, RStudio is an IDE which is the acronym for *Integrated Development Environment*. This is just the fancy name for any software application that provides comprehensive facilities to programmers for making their lives easier when writing code and developing programs.

Simply put, you can think of RStudio as a "workbench" that gives you an organized working space for interacting with R, while taking care of many of the little tasks than can be a hassle.

3.1 First Contact with RStudio

When you open RStudio, you should be able to see its layout organized into quadrants offically called *panes* (or panels).

The very first time you launch RStudio you will only see three panes, like in the screenshot below.

As you can tell from the previous screenshot, the left-hand side shows the Console pane which is what we used in the previous chapter to write a handful of simple commands, execute them, and inspect the output provided by R.

If RStudio only displays three panes, why do I call them "quadrants"? Where is the fourth pane? Well, to see the extra pane you need to open a file. One way to do this is by clicking the icon of a blank file with a green plus sign. This

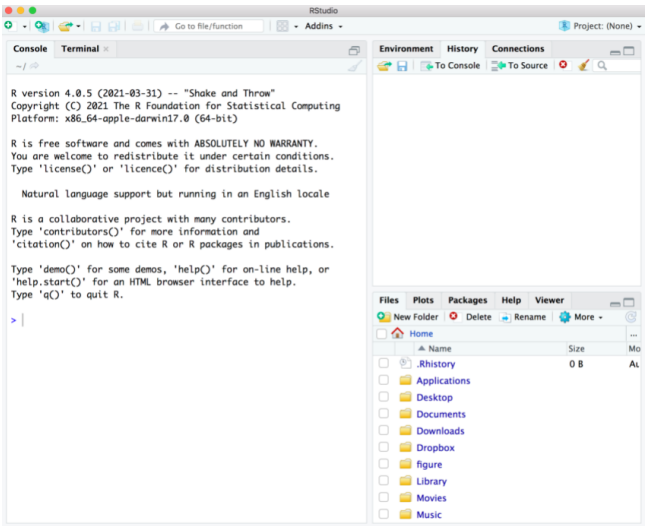


Figure 3.1: Screenshot of RStudio when launched for the first time.

button is located in the top-left corner of the icons menu bar of RStudio. A drop-down menu with a long list of available file formats will be displayed, the first option being an "R Script" file (see image below).

Once you open a (text) file, the layout of RStudio will show the Editor quadrant, officially called the *Source* pane, like in the following screenshot

The appearance of RStudio's quadrants can be a bit intimidating for beginners. But fear not. In the above screenshot, the panes are:

- `Source` or editor pane (top left quadrant)
- `Console` pane (bottom left quadrant)
- `Environment/History/Connections` pane (top right quadrant)
- `Files/Plots/Packages/Help` pane (bottom right quadrant)

Pro tip: you can change the default location of the panes, among many other things. If you are interested in knowing what customizing options are available, visit this link for Customizing RStudio

https://support.rstudio.com/hc/en-us/articles/200549016-Customizing-RStudio

If you have no previous programming experience, you don't have to customize anything right now. It's better if you wait some days until you get a better feeling of the working environment. You will probably be experimenting (trial and error) some time with the customizing options until you find what works for you.

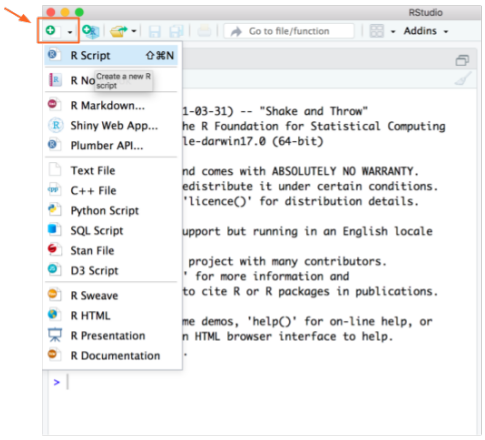


Figure 3.2: Opening a new (text) file in RStudio.

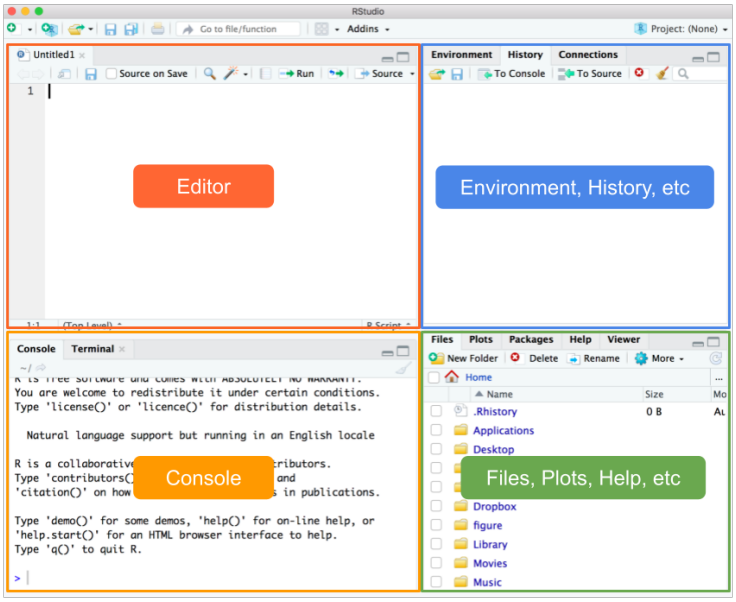


Figure 3.3: RStudio layout organized into quadrants.

3.2 RStudio Panes in a Nutshell

Sooner or later you will be using all four panes in RStudio. Most programming activities will require working with both the `Source` and `Console` panes. Certain operations will involve using the `Files` tab. Occasionally you will also use one of the tabs in the `Environment/History/etc` pane. The set of specifc tabs that you have to use really depends on the type of work you plan to carry out. You will have time to learn the basics—and not so basics—of every pane throughout the book. The more time you spend in RStudio, and the more you use it, the more features you will discover about it.

3.2.1 Console

The `Console` is supposed to be the terminal—or the place—where you type in commands, which R then executes, and where the output of those commands is typically displayed. The truth is that most programmers don't write commands *directly* in the console. Instead, what we use is the `Source` pane to write commands in a text file (e.g. an R-Script file, an R-Markdown file), and then execute the commands from that pane.

The reason for writing commands in a text file and not directly in the console, is because of convenience and organization. *Convenience* because, as you will see, many commands involve writing several lines of code which can be tricky to do it correctly just by typing on the console. *Organization* in the sense that having all your commands in a text file makes it easy to store your code, keep track of all the work you do, build upon it, and share it with others.

So, knowing that programmers rarely make direct use of the `Console`, when do you actually use this pane? I don't know about the rest of programmers but I can tell you how I personally use it.

One common use is when I want to calculate basic things like the monthly balance in my credit cards, or the overall score for one of my students, or some other quick computation. These are types of calculations that I could perfectly perform with any scientific calculator like the one in my smartphone. But many times I prefer to do them in R, typically when that's the tool I have at hand (which happens almost every day).

The other typical situation in which I use the console is when I'm trying out some idea or testing if a certain command could work. I like to explore the feasibility of my code with a small example in the console, and then refine it or generalize it by writing code in a text file—using the `Source` pane.

3.2.2 Files, Plots, Packages, Help

The `Files` quadrant contains multiple tabs

- `Files`: this tab lets you navigate your file system without the need of leaving RStudio. You can move to any directory or folder in your home

directory, inspect the contents of a given folder, create a new folder, and perform standard operations on files such as opening, renaming, copying, and deleting. In addition, you can also see the working directory, or change it if you want to.

- `Plots`: this tab is used by R Graphics Devices to display any graphic or image produce by an R plotting function.
- `Packages`: this tab allows you to install and update R packages. Often, you will want to use functions from external R packages, and to do this you must first install those packages in your system. While it is possible to write commands for doing this, the `Packages` tab gives you a richer interface to see what packages are already available in your computer, what their versions are, update them if necessary, or delete them in case you no longer need them.
- `Help`: this is the tab that gives you access to the "help" or manual documentation of functions, objects, tutorials, and demos of a given R package. In the previous chapter we provided an example of the manual documentation for the `log()` function, showing the main anatomy of the so-called *R Documentation* files.

3.2.3 Source or Editor

The `Source` pane is basically the text editor of RStudio. This is the quadrant you use to edit any text file, again, without the need to leave RStudio. The reason why is called "source" is because the text files edited in this pane are, for the most part, files that contain the commands that R will run. In other words, these files are the **source** of the commands to be executed.

3.2.4 Environment, History, Connections

The last quadrant is the pane that contains, at least, the following three tabs: `Environment`, `History`, and `Connections`.

I provide a deeper explanation of the `Environment` and `History` tabs in the following chapter Session Management. In the meantime, what you need to know about `Environment` is that this tab is used to list the objects that have being created, or that are available, in a given R session.

In turn, the `History` tab is a very useful resource that lists **all** the R commands that you've executed so far. In theory, R will track all the invoked commands since the first time you used it, unless you've removed the auxiliary `.Rhistory` file linked to your working directory, or unless you've modified the history mechanism used by your R console.

As for the `Connections` tab, this plays a more advanced (and somewhat obscure) role that I briefly discuss in part IV of the book.

3.3 Exercises

1) In RStudio, one of the panes has tabs `Files, Plots, Packages, Help, Viewer`.

a) In the tab **Files**, what happens when you click the button with a House icon?
b) Go to the **Help** tab and search for the documentation of the function `mean`.
c) In the tab **Help**, what happens when you click the button with a House icon?

2) In RStudio, one of the panes has the tabs `Environment, History, Connections`.

a) If you click on tab **History**, what do see?
b) Find what the buttons of the menu bar in tab **History** are for.
c) Likewise, what can you say about the tab **Environment**?

3) When you start a new R session in Rstudio, a message with similar content to the text below appears on the console (the exact content will depend on your R version):

```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 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.
```

a) What happens when you type: `license()`?

b) What happens when you type: `contributors()`?

c) What happens when you type: `citation()`?

d) What happens when you type: `demo()`?

Chapter 4

Session Management

In this chapter I review some important aspects about managing your interactive session with R.

There are several things going on behind the scenes:

- when starting a session
- during the session
- when closing a session

4.1 Starting a Session

Starting a session can be done in two primary ways:

- launching the R application program, which will give you access to its graphical usier interface; or via RStudio or any other IDE that has the ability to open an R session.
- by clicking on a file that your computer associates with R (or RStudio). For example, R-script files (with file extension `.R` or `r`), R-Markdown files (`.Rmd` extension), R-Noweb files (`.Rnw` extension), RStudio project files (`.Rproj` extension), etc.

4.1.1 What happens when you open an R session via RStudio?

This is an important question that many users never stop to think about. However, it's worth reviewing what happens when a session is started. So let's talk about this.

- Every time you open RStudio, the console pane will display R's welcome message.
- The console is always linked to a working directory.
- Typically, the working directory of the console will be your home directory, unless you specified a different location when you installed R in your computer.
- You can change the working directory to a different directory if you want. This change can be permanent (for future sessions), or temporary (for current session).
- To permanently modify working directory (when a session is opened), go to the menu bar, select "RStudio" tab, click on Preferences, and modify the "R General" options.
- To temporary change the working directory, go to the menu bar, select the "Session" tab, and click on "Set Working Directory", or simply specify a working directory with the `setwd()` function executed from R's console.

4.1.2 Opening a session for the first time

If you are opening a session for the very fist time:

- the "Editor" pane will be collapsed, and
- all the tabs in the "Environment, History, Connections" pane will be empty

In general, when you open a session (**not** for the very fist time):

- the "Environment" tab may display some objects, which means you have some existing objects in your global environment

```
# what objects are in my global environment?
ls()
```

- the "History" tab may contain lines of previously used commands; if this is the case it means that there is a text file called `.Rhistory` in your working directory.

```
# is my history of commands being tracked?
history()
```

4.1.3 Working Directory

If you open R via launching an application program (e.g. RStudio) that lets you interact with R's console, your session will have an associated working directory, sometimes also referred to as the *current* working directory.

When you install R in your computer, during the installation process a default working directory is assigned with R. Be default, this directory is your home

directory. You can check whether this is the case if you run the *get working directory* function `getwd()` in your console.

```
# run this command in the console to find out
# the working directory of your session
getwd()
```

In RStudio, you can also look at the console pane, and inspect the text right below the `Console` tab that always displays the working directory of your session. If you see the symbols `~/` it means that the home directory (represented by the tilde) is your working directory.

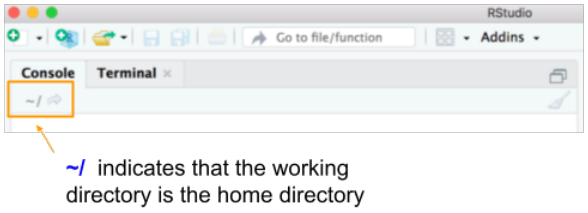


Figure 4.1: The working directory is indicated right below the Console tab.

4.2 Working During a Session

Working with R involves a series of common actions:

- writing commands (on a source document or on the console)
- executing commands (from a source doc or from the console)
- looking or examining outputs
- reading or importing files (e.g. data files, script files)
- saving or exporting output to files (e.g. data files, images, results)

From the logistical point of vew, it all boils down to executing commands, taking into account the following:

- where a command is being executed from
- if the command requires an input, where does that input comes from?
- if the command produces an output, where does that output goes to?

This is why we need to describe:

1. Working Directory
2. Workspace and Global Environment
3. History of commands

4.2.1 Working Directory

You will be writing commands either directly in the console or in a source document (e.g. `R` script file, `Rmd` file, etc.)

The console is always associated to a working directory (usually your home directory).

A source document, once it's saved, will live in some directory. **Ideally**, a source doc's directory whould be used as its working directory, using relative filepaths to handle all input and output resources required in the code of the source doc. Unfortunately, this ideal is far from what happens in practice.

- By default, when you execute code chunks in a **saved** `Rmd` file, the working directory is the directory where the `Rmd` file resides.
- By default, when you execute code from an `R` file, the working directory is that of the console.

4.2.2 Workspace and Global Environment

Regardless of where commands are being executed from, R will carry out all the necessary computations, and objects will be created along the way.

The collection of objects that are being created (*and kept alive*) during a session are part of what is considered to be your **workspace**.

At a more technical level, all the objects in your workspace are part of an R **environment**. To be more precise, the workspace is the **Global Environment**.

On the console, if you type `ls()`, R will display all the available objects in your workspace.

```
# available objects in your workspace
ls()
```

You can also go to the pane "Environment, History, …" and click on the `Environment` tab to see the objects in your workspace which are displayed by default under the option "Global Environment".

4.2.3 Commands History

When you start a session, R will track all the commands that you execute during that session. As you execute commands, they will become part of what is called the **Commands History**.

You can find the list of all used commands in the `History` tab, located in the *Environment, History, Connections* pane of RStudio. You can also access the commands history with the function `history()`.

By default, the history of commands are stored in a text file called `.Rhistory` that is saved in your session's working directory.

Figure 4.2: The comands history is avaialble in the History tab.

4.3 Closing a Session

When closing a session, what should you do?

This is a somewhat "very personal" type of question, because it is up to you to decide what should happen to all the work that you've done in R.

Having said that, you can always decide whether or not to:

- save changes in your source document(s)
- save the commands history in a text file
- save the objects in your workspace (i.e. objects in Global Environment) in a binary file

It turns out that RStudio comes with default settings that take place when you close a session:

- it will ask you if you want to save changes in your source documents
- it will ask you if you want to save the workspace in an `.RData` file (this is a file that uses R's native binary format; this is saved in your session's working directory)
- it will automatically save your commands history in a text file called `.Rhistory` (saved in your session's working directory)

Also, because of RStudio's default settings, the next time you open a new session it will:

- restore the previously open source document
- restore objects in the `.RData` file into your workspace
- give you access to all the commands stored in the `.Rhistory` file

Part II

Data Objects in R

Chapter 5

Vectors

In order to enjoy and exploit R as a computational tool, one of the first things you need to learn about is the objects R provides to handle data. The formal name for these programming elements is **data objects** also known as **data structures**. They form the ecosystem of data containers that we can use to handle various types of data sets, and be able to operate with them in different forms.

I'm going to use financial math examples as an excuse to introduce and explain the material. I've found that having a common theme helps avoiding falling into the "teaching trap" of presenting isolated examples in a vacuum.

5.1 Motivation: Compound Interest

I would like to ask you if you have any of the following accounts:

- Savings account?
- Retirement account?
- Brokerage account?

Don't worry if you don't have any of these accounts. I certainly didn't have any of those accounts until I started my first job right after I finished college.

Anyway, let's consider a hypothetical scenario in which you have $1000, and you decide to deposit them in a savings account that pays you an annual interest rate of 2%. Assuming that you leave that money in the savings account, an important question could be:

> How much money will you have in your savings account **one** year from now?

The answer to this question is given by the **compound interest** formula:

$$\text{deposit} + \text{paid interest} = \text{amount in one year}$$

In this example, you deposit \$1000, and the bank pays you 2% of \$1000 = \$20. In mathematical terms, we can write the following equation to calculate the amount that you should expect to have in your savings account within a year:

$$1000 + 1000(0.02) = 1000 \times (1 + 0.02) = 1020$$

You can confirm this by running the following R command:

```
# in one year
1000 * (1.02)
#> [1] 1020
```

Now, if you leave the \$1020 in the savings account for one more year, assuming that the bank keeps paying you a 2% annual return, how much money will you have at the end of the second year?

Well, all you have to do is repeat the same computation, this time by letting the \$1020—accumulated during the first year—compound for one more year:

$$1020 + 1020(0.02) = 1020 \times (1 + 0.02) = 1040.40$$

which in R can be computed as:

```
# in two years
1020 * (1.02)
#> [1] 1040
```

How much money will you have at the end of three years? Again, take the amount saved at the end of year 2, and compund it for one more year:

$$1040.4 + 1040.4(0.02) = 1040 \times (1 + 0.02) = 1061.208$$

We can confirm this in R by running the following command:

```
# in three years
1040.40 * (1.02)
#> [1] 1061
```

5.1.1 Creating Objects

Often, it will be more convenient to create **objects** , also referred to as **variables**, that store both input and output values. To do this, type the name of the object, followed by the equals sign `=`, followed by the assigned value. For

example, you can create an object `d` for the initial deposit of $1000, and then inspect the object by typing its name:

```
# deposit 1000
d = 1000
d
#> [1] 1000
```

Alternatively, you can also use the *arrow operator* `<-`, technically known as the **assignment operator** in R. This operator consists of the left-angle bracket (i.e. the less-than symbol) and the dash (i.e. hyphen character).

```
# interest rate of 2%
r <- 0.02
r
#> [1] 0.02
```

Assignment Statements

All R statements where you create objects are known as "assignments", and they have this form:

```
object <- value
```

this means you assign a `value` to a given `object`; you can read the previous assignment as "r gets 0.02".

RStudio has a keyboard shortcut for the arrow operator `<-`: `Alt` + `-` (the minus sign).

Here are more assignments for each of the savings amounts at the end of years 1, 2, and 3:

```
# amounts at the end of years 1, 2, and 3
a1 = d * (1 + r)
a2 = a1 * (1 + r)
a3 = a2 * (1 + r)
a3
#> [1] 1061.21
```

Use Descriptive Names

While the names of these objects—`d`, `r`, `a1`, etc—are good for a computer, they can be a bit cryptic for a human being. To be more transparent, we can use more descriptive names, for example:

```
# inputs
deposit = 1000
rate = 0.02
```

```
# amounts at the end of years 1, 2, and 3
amount1 = deposit * (1 + rate)
amount2 = amount1 * (1 + rate)
amount3 = amount2 * (1 + rate)
amount3
#> [1] 1061.21
```

The names of the above objects are good for a computer and also for a human being (that reads English).

Whenever possible, make an effort to use descriptive names. While they don't matter that much for the computer, they definitely can have a big impact on any person that takes a look at the code. As it turns out, we tend to spend more time reviewing and reading code than writing it. So do yourself (and others) a favor by using descriptive names for your objects.

Combining various objects into a single one

We can store various computed values in a single object using the combine or catenate function `c()`. Simply list two or more objects inside this function, separating them by a comma `,`. Here's an example for how to use `c()` to define an object `amounts` containing the amounts at the end of years 1, 2, and 3.

```
# inputs
deposit = 1000
rate = 0.02

# amounts at the end of years 1, 2, and 3
amount1 = deposit * (1 + rate)
amount2 = amount1 * (1 + rate)
amount3 = amount2 * (1 + rate)

# combine (catenate) in a single object
amounts = c(amount1, amount2, amount3)
amounts
#> [1] 1020.00 1040.40 1061.21
```

So far we have created a bunch of objects. You can use the list function `ls()` to display the names of the available objects. But what kind of objects are we dealing with?

It turns out that all the objects we have so far are **vectors**.

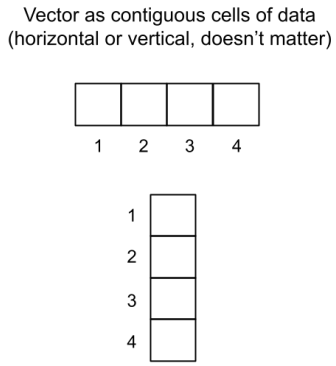
5.2 About R Vectors

Vectors are the **most basic** kind of data objects in R. Pretty much all other R data objects are derived (or are built) from vectors. This is the reason why I

personally like to say that, to a large extent, R is a *vector-based programming language*.

Based on my own experience, becoming proficient in R requires a solid understanding of the properties and behavior of R vectors.

To give you a mental picture of what a vector could like, you can think of a vector as set of contiguous "cells" of data, like in the diagram below:



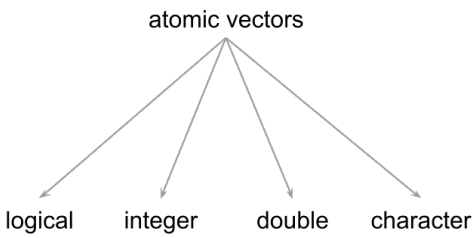
They can be of any length (including 0), and the starting position or index is always number 1.

5.3 Vectors are Atomic Objects

The first thing you should learn about R vectors is that they are consired to be **atomic structures**, which is just the fancy name to indicate that all the elements in a vector are of the **same type**.

R has four main basic types of atomic vectors:

- `logical`
- `integer`
- `double` or `real`
- `character`



There are also two additional types that are less commonly used: `complex` (for complex numbers), and `raw` which is a binary format used by R.

Here are simple examples of the four common types of vectors:

```
# logical
a = TRUE

# integer
x = 1L

# double (real)
y = 5

# character
b = "yosemite"
```

Logical values, known as boolean values in other languages, are `TRUE` and `FALSE`. These values can be abbreviated by using their first letters `T` and `F`, although I discourage you from doing this because it can make code review a bit harder.

Integer values have an awkward syntax. Notice the appended `L` when assigning number 1 to object `x`. This is not a typo. Rather, this is the syntax used in R to indicate that a number (with no decimals) is an integer.

If you just simply type a number like `1` or `5`, even though cosmetically they correspond to the mathematical notion of integer numbers, R stores those numbers as `double` type. So if you want to declare those numbers as type `integer`, you should append an upper case letter `L` to encode them as `1L` and `5L`.

Character types, referred to as strings in other languages, are specified by surrounding characters within quotes: either double quotes `"a"` or single quotes `'b'`. The important thing is to have an opening and a closing quote of the same kind.

5.3.1 Types and Modes

How do you know that a given vector is of a certain data type? For better or worse, there is a couple of functions that allow you to answer this question:

- `typeof()`
- `mode()`

Although not commonly used within the R community, my recommended function to determine the data type of a vector is `typeof()`. The reason for this recommendation is because `typeof()` returns the data types previously listed which are what most other programming languages use:

```
typeof(deposit)
typeof(rate)
typeof(amount1)
```

You should know that among the R community, many useRs don't really talk about *types*. Instead, because of historical reasons related to the S language—on which R is based—you will often hear useRs talking about *modes* as given by the `mode()` function:

```
mode(deposit)
mode(rate)
mode(amount1)
```

`mode()` gives the storage mode of an object, and it actually relies on the output of `typeof()`.

When applied to vectors, the main difference between `mode()` and `typeof()` is that `mode()` groups together types `"double"` and `"integer"` into a single mode called `"numeric"`.

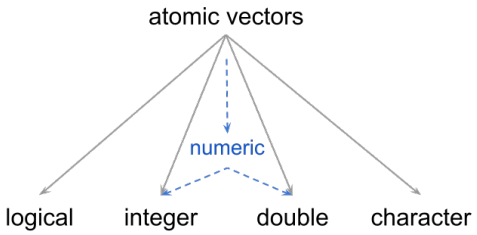


Figure 5.1: Data types "integer" and "double" correspond to "numeric" mode

5.3.2 Special Values

In addition to the four common data types, R also comes with a series of special values

- `NULL` is the null object (it has length zero)
- `NA`, which stands for *Not Available*, indicates a missing value. By default, typing `NA` is stored as a logical value. But there are also special types of missing values.
  - `NA_integer_`
  - `NA_real_`
  - `NA_character_`
- `NaN` indicates *Not a Number*. An example of this value is the output returned by computing the square root of a negative number: `sqrt(-5)`

- `Inf` indicates positive infinite, e.g. `100/0`
- `-Inf` indicates negative infinite, e.g. `-100/0`

5.3.3 Length of Vectors

The simplest kind of vectors are single values—i.e. vectors with just one element. For example, objects such as `deposit` and `rate` are one-element vectors

```
length(deposit)
#> [1] 1
length(rate)
#> [1] 1
```

In most other languages, a number like `5` or a logical `TRUE` are usually considered to be "scalars". R, however, does not have the concept of "scalar", instead the simplest data structure is that of a one-element vector.

5.4 Creating Vectors

We've already seen how to create simple vectors, that is, vectors containing just one element (i.e. length-1 vectors)

```
a = TRUE  # logical
x = 1L    # integer
y = 5     # double (real)
b = "abc" # character
```

5.4.1 Creating vectors with `c()`

Among the main functions to work with vectors we have the **combine** function `c()`. This is the workhorse function to create vectors of length greater than one. Here's how to create a vector `flavors` with some ice-cream flavors:

```
flavors <- c('lemon', 'vanilla', 'chocolate')

flavors
#> [1] "lemon"     "vanilla"   "chocolate"
```

Basically, you call `c()` and you type in the values, separating them by commas.

5.4.2 Numeric Sequences

A common situation when creating vectors involves creating numeric sequences. If the numeric sequence is short and simple, it could be created with the combine function `c()`, for example:

```
s1 = c(1, 2, 3, 4)
s1
#> [1] 1 2 3 4
```

Often, you will have to create less simpler and/or longer sequences. For these purposes there are two useful functions:

- the colon operator `":"`
- the sequence function `seq()` and its siblings `seq.int()`, `seq_along()` and `seq.len()`

Sequences with :

The colon operator `:` lets you create numeric sequences by indicating the starting and ending values. For instance, if you want to generate an integer sequence starting at 1 and ending at 10, you use this command:

```
ints = 1:10
ints
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Notice that the the colon operator, when used with whole numbers, will produce an integer sequence

```
typeof(ints)
#> [1] "integer"
```

However, when the starting value is not a whole number, then the generated sequence will be of type `double`, with unit-steps. For example:

```
1.5:5.5
#> [1] 1.5 2.5 3.5 4.5 5.5
```

Run the following commands to see the how R generates different sequences:

```
1.5:5
1.5:5.1
1.5:5.5
1.5:5.9
```

You can also create a decreasing sequence by starting with a value on the left-hand side of `:` that is greater than the value on the right-hand side:

```
# decreasing (reversed) sequence
10:1
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

Sequences with `seq()`

In addition to the colon operator, R also provides the more generic `seq()` function for creating numeric sequences. This function comes with a couple of parameters that let you generate sequences in various forms.

The simplest usage of `seq()` involves passing values for the arguments `from` (the starting value) and `to` (the ending value):

```
# equivalent to 1:10
seq(from = 1, to = 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

As you can tell, the sequence is created with one unit-steps. But this can be changed with the `by` argument. Say you want steps of two-units:

```
seq(from = 1, to = 10, by = 2)
#> [1] 1 3 5 7 9
```

Now, what if you want a decreasing sequence, for example 10, 9, ..., 1? You can also use `seq()` to achieve this goal. The starting value `from` is 10, the ending value `to` is 1, and the step size `by` has to be `-1`

```
seq(from = 10, to = 1, by = -1)
#>  [1] 10  9  8  7  6  5  4  3  2  1
```

Sometimes you may be interested in creating a sequence of a specific length. When this is the case, you need to use the `length.out` argument. For example, say we want to start with 2, getting the sequence of the first six even numbers. One way to obtain this sequence is with `from = 2`, steps of size `by = 2`, and a length of `length.out = 6`

```
seq(from = 2, length.out = 6, by = 2)
#> [1]  2  4  6  8 10 12
```

5.4.3 Replicated Vectors

Another interesting function for creating repeated sequences is `rep()`. This function takes a vector as the main input, and then it optionally takes various arguments: `times`, `length.out`, and `each`.

```
rep(1, times = 5)        # repeat 1 five times
#> [1] 1 1 1 1 1
rep(c(1, 2), times = 3)  # repeat 1 2 three times
#> [1] 1 2 1 2 1 2
rep(c(1, 2), each = 2)
#> [1] 1 1 2 2
rep(c(1, 2), length.out = 5)
#> [1] 1 2 1 2 1
```

Here are some more complex examples:

```
rep(c(3, 2, 1), times = 3, each = 2)
#>  [1] 3 3 2 2 1 1 3 3 2 2 1 1 3 3 2 2 1 1
```

5.5 Coercion

Another fundamental concept that you should learn about vectors is that of **coercion**. This has to do with the mechanism that R uses to make sure that all the elements in a vector are of the same data type.

There are two coercion mechanisms or approaches:

- implicit coercion rules
- explicit coercion functions

5.5.1 Implicit Coercion Rules

Implicit coercion is what R does when we try to combine values of different types into a single vector. Here's an example:

```
mixed <- c(TRUE, 1L, 2.0, "three")
mixed
#> [1] "TRUE"  "1"     "2"     "three"
```

In this command we are mixing different data types: a logical `TRUE`, an integer `1L`, a double `2.0`, and a character `"three"`. Now, even though the input values are of different data flavors, R has decided to convert everything into type `"character"`. Technically speaking, R has **implicitly coerced** the values as characters, without asking for our permission and without even letting us know that it did so.

If you are not familiar with implicit coercion rules, you may get an initial impression that R is acting weirdly, in a nonsensical form. The more you get familiar with R, you will notice some interesting coercion patterns. But you don't need to struggle figuring out what R will do. You just have to remember the following hierarchy:

$$\text{character} > \text{double} > \text{integer} > \text{logical}$$

Here's how R works in terms of coercion:

- characters have priority over other data types: as long as one element is a character, all other elements are coerced into characters
- if a vector has numbers (double and integer) and logicals, double will dominate

- finally, when mixing integers and logicals, integers will dominate

5.5.2 Explicit Coercion Functions

The other type of coercion mechanism, known as **explicit coercion**, is done when you explicitly tell R to convert a certain type of vector into a different data type by using explicit coercion functions:

- `as.integer()`
- `as.double()`
- `as.character()`
- `as.logical()`

Depending on the type of input vector, and the coercion function, you may achieve what you want, or R may fail to convert things accordingly.

We can take `deposit`, which is of type double, and convert it into an integer with no issues:

```
int_deposit = as.integer(deposit)
int_deposit
#> [1] 1000
```

Interestingly, the way an `integer` number is displayed is exactly the same as its `double` version. To confirm that `int_deposit` is indeed of type `integer` you can use the `is.integer()` function

```
is.integer(deposit)
#> [1] FALSE
is.integer(int_deposit)
#> [1] TRUE
```

What about trying to convert a character string such as `"string"` into an integer? You can try to apply `as.integer()` but in this case the attempt is fruitless:

```
as.integer("string")
#> Warning: NAs introduced by coercion
#> [1] NA
```

Chapter 6

More About Vectors

In the previous chapter we started the topic of data objects by introducing R vectors and some of their basics properties. In this chapter we continue the discussion of vectors, specifically the notions of vectorization, and recycling.

6.1 Motivation: Future Value

Let's bring back the savings example from the previous chapter: you have $1000 and you decide to deposit this money in a savings account that pays you an annual interest rate of 2%. We've already seen how to calculate the amount of money that you would have at the end of the first, second and third years. Let's now calculate the saved amount for a 10-year period.

> How much money will you have at the end of each year during a 10-year period?

To answer this question, we could compute individual amount objects (e.g. `amount1`, `amount2`, `amount3`, etc) to get the saved amount at the end of each year. For example:

```
# inputs
deposit <- 1000
rate <- 0.02

# amounts at the end of years 1, 2, 3, ..., 10
amount1 = deposit * (1 + rate)
amount2 = amount1 * (1 + rate)
amount3 = amount2 * (1 + rate)
amount4 = amount3 * (1 + rate)
amount5 = amount4 * (1 + rate)
amount6 = amount5 * (1 + rate)
```

```
amount7 = amount6 * (1 + rate)
amount8 = amount7 * (1 + rate)
amount9 = amount8 * (1 + rate)
amount10 = amount8 * (1 + rate)
```

The problem with this piece of code is that it is too repetitive, time consuming, boring, and error prone (can you spot the error?). Even worse, imagine if you were interested in computing the amount of your investment for a 20-year or a 30-year or a longer year period?

The good news is that we don't have to be so repetitive. Before describing what the alternative—and more efficient—approach is, we need to do a bit of algebra.

6.1.1 Future Value Formula

In one year you'll have:

$$1000 \times (1.02) = 1020$$

In two years you'll have:

$$1000 \times (1.02) \times (1.02) = 1000 \times (1.02)^2 = 1040.4$$

In three years you'll have:

$$1000 \times (1.02) \times (1.02) \times (1.02) = 1000 \times (1.02)^3 = 1061.208$$

Do you see a pattern?

If you deposit \$1000 at a rate of return r, how much will you have at the end of year t? The answer is given by the Future Value (FV) formula. In its simplest version, the formula is:

$$\text{FV} = \text{PV} \times (1 + r)^t$$

- FV = future value (how much you'll have)
- PV = present value (the initial deposit)
- r = rate of return (e.g. annual rate of return)
- t = number of periods (e.g. number of years)

Keep in mind that there are more sophisticated versions of the FV formula. For now, let's keep things simple and use the above equation.

If you deposit \$1000 at a rate of 2%, how much will you have at the end of year 10?

```
deposit <- 1000
rate <- 0.02
year <- 10

amount10 <- deposit * (1 + rate)^year
amount10
#> [1] 1218.994
```

Using the formula of the Future Value you can directly compute the amount that you would have at the end of the tenth year. But what about calculating the amounts at the end of each year during that time period? Enter vectorization!

6.2 Vectorization

In order to explain what vectorization is, let me first show you the following R code. Compared to the code snippet above, note that the code below uses a vector `years` containing a numeric sequence from 1 to 10, thanks to the `:` ("colon") operator. This vector `years` is then used to play the role of the exponent in the Future Value formula:

```
deposit <- 1000
rate <- 0.02
years <- 1:10  # vector of years

# example of vectorization (or vectorized code)
amounts <- deposit * (1 + rate)^years
amounts
#>  [1] 1020.000 1040.400 1061.208 1082.432 1104.081 1126.162 1148.686 1171.659
#>  [9] 1195.093 1218.994
```

The computed object `amounts` is exactly what we are looking for. This vector contains the saved amounts at the end of each year, from the first year till the tenth year.

The code used to obtain `amounts` is an example of one of the most fundamental and powerful kinds of operations (computations) in R, and it has its special name: **vectorization**, also referred to as **vectorized** code.

When you write code like this:

```
amounts = deposit * (1 + rate)^years
```

we say that your code is **vectorized**. Technically speaking, this code uses not just vectorization but it also uses something else called *recycling*, which we will explain in the next section. But let's describe vectorization first.

6.2.1 Definition of Vectorization

Simply put, vectorization means that a given function or operation will be applied to all the elements of one or more vectors, element by element.

Say you want to create a vector `log_amounts` by taking the logarithm of `amounts`. All you have to do is apply the `log()` function to `amounts`:

```
log_amounts <- log(amounts)
```

When you create the vector `log_amounts`, what you're doing is applying a function to a vector, which in turn acts on all the elements of the vector. Hence the reason why we call it *vectorization* in R parlance.

Most functions that operate with vectors in R are vectorized functions. This means that an action is applied to all elements of the vector without the need to explicitly type commands to traverse all of its values, element by element.

In many other programming languages, you would have to use a set of commands to loop over each element of a vector (or list of numbers) to transform them. But not in R.

Another simple example of vectorization would be the calculation of the square root of all the amounts:

```
sqrt(amounts)
```

Why should you care about vectorization?

If you are new to programming, learning about R's vectorization will be very natural and you won't stop to think about it too much. If you have some previous programming experience in other languages (e.g. C, python, perl), you know that vectorization does not tend to be a native thing.

Vectorization is essential in R. It saves you from typing many lines of code, and you will exploit vectorization with other useful functions known as the *apply* family functions (we'll talk about them later in the book).

6.3 Recycling

Closely related with the concept of *vectorization* we have the notion of **Recycling**. To explain recycling let's see an example.

The values in the vector `amounts` are given in dollars, but what if you need to convert them into values expressed in thousands of dollars?. To convert from dollars to thousands-of-dollars you just need to divide by 1000; for example

- 1,000 dollars becomes 1 thousands-dollars
- 10,000 dollars becomes 10 thousands-dollars
- 1 dollar becomes 0.001 thousands-dollars

Here is how to create a new vector `thousands`:

```
thousands <- amounts / 1000
thousands
#>  [1] 1.020000 1.040400 1.061208 1.082432 1.104081 1.126162 1.148686 1.171659
#>  [9] 1.195093 1.218994
```

What you just did (assuming that you did things correctly) is called **Recycling**, which is what R does when you operate with two (or more) vectors of **different length**.

To understand this concept, you need to remember that R does not have a data structure for scalars (single numbers). Scalars are in reality vectors of length 1.

The conversion from dollars to thousands-of-dollars requires this operation: `amounts / 1000`. Although it may not be obvious, we are operating with two vectors of different length: `amounts` has 10 elements, whereas `1000` is a one-element vector. So how does R know what to do in this case?

Well, R uses the **recycling rule**, which takes the shorter vector (in this case `1000`) and recycles its content to form a temporary vector that matches the length of the longer vector (i.e. `amounts`).

Another recycling example

Here's another example of recycling. Saved amounts of elements in an odd number position will be divided by two; values of elements in an even number position will be divided by 10:

```
units <- c(1/2, 1/10)
new_amounts <- amounts * units
new_amounts
#>  [1] 510.0000 104.0400 530.6040 108.2432 552.0404 112.6162 574.3428 117.1659
#>  [9] 597.5463 121.8994
```

In this piece of code, the elements of `units` are recycled (i.e. repeated) as many times as the number of elements in `amounts`.

To achieve the same result without using recycling you would have to create a vector `new_units` (i.e. the values to divide by) of the same length as `amounts`. For example, you could create a vector `new_units` with the replicate function `rep()` having ten elements in which those values in odd positions are `1/2` and those values in even positions are `1/10`:

```
new_units <- rep(c(1/2, 1/10), length.out = length(amounts))
amounts * new_units
#>  [1] 510.0000 104.0400 530.6040 108.2432 552.0404 112.6162 574.3428 117.1659
#>  [9] 597.5463 121.8994
```

6.3.1 Vectorization and Recycling

Let's bring back the code that uses the Future Value to obtain the vector `amounts`:

```
amounts = deposit * (1 + rate)^years
```

Recall that `deposit` and `rate` have length 1. So does the number `1`, it is a vector containing just one element. In contrast, `years` has 10 elements. This means that R is dealing with four vectors some of which have different lengths.

In pictures, we have the following diagram:

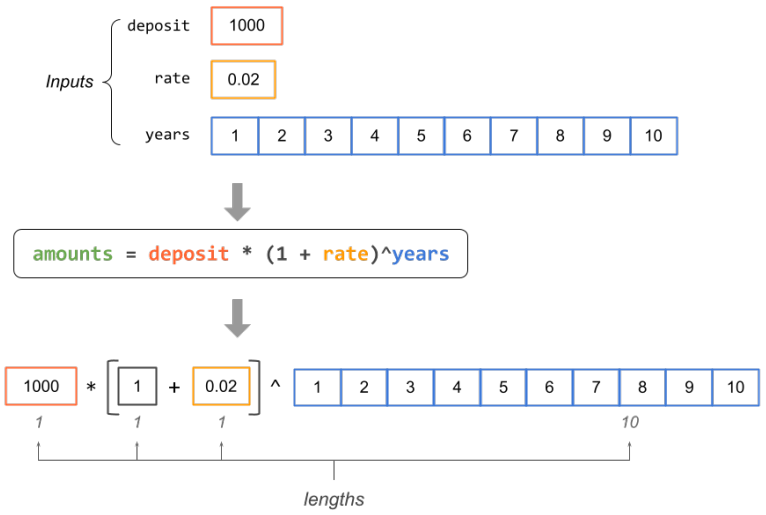
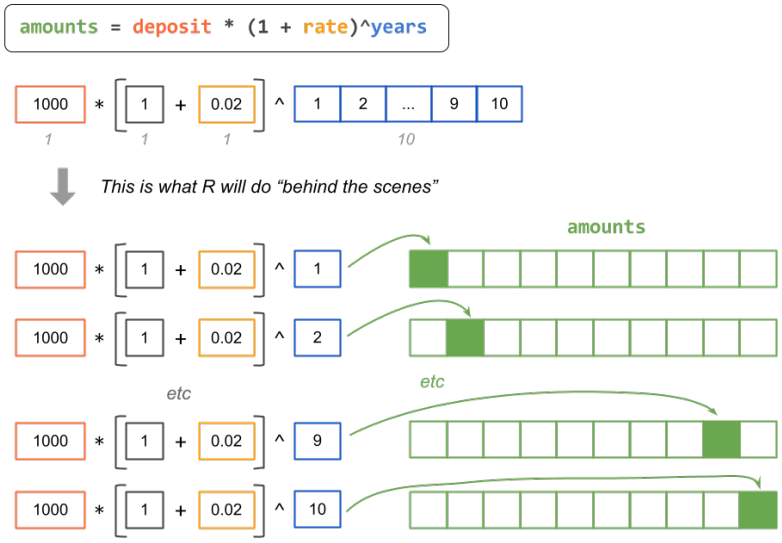


Figure 6.1: Diagram depicting vectors of different lengths.

How does R take care of this?

The following diagram depicts what R does behind the scenes: R recycles the shorter vectors to match the length of the longest vector. In this example, vectors `deposit`, `rate`, and `1` are the shorter vectors, which are then recycle to match the length of the longest vector `years`. The computation process is completed with vectorization.



As you can tell, this is an example of vectorization & recycling rules in R.

6.4 Manipulating Vectors: Subsetting

In addition to creating vectors, you should also learn how to do some basic manipulation of vectors. The most common type of manipulation is called *subsetting*, also known as *indexing* or *subscripting*, which refers to extracting elements of a vector (or another R object). To do so, you use what I like to call **bracket notation**. This implies using (square) brackets `[ ]` to get access to the elements of a vector.

To subset a vector, you type the name of the vector, followed by an opening and a closing bracket. Inside the brackets you specify

6.4.1 Numeric Subsetting

This type of subsetting, as the name indicates, is when the indexing vector consists of a numeric vector with one or more values that correspond to the position(s) of the vector element(s).

The simplest type of numeric subsetting is when we use single number (which is a vector of length one).

```
# amount at end of year 1
amounts[1]
#> [1] 1020
```

The indexing numeric vector can have more than one element. For example, if we want to extract the elements in positions 1, 2 and 3, we could provide a

numeric sequence `1:3`:

```
# amounts at end of years 1, 2, and 3
amounts[1:3]
#> [1] 1020.000 1040.400 1061.208
```

The numeric positions don't have to be consecutive numbers. You can also use a vector of non-consecutive numbers:

```
# amounts at end of years 2 and 4
amounts[c(2, 4)]
#> [1] 1040.400 1082.432
```

Likewise, we can also use a vector with repeated numbers:

```
# repeated amounts
amounts[c(2, 2, 2)]
#> [1] 1040.4 1040.4 1040.4
```

In addition to the previous subscripting options, we can specify negative numbers to indicate that we want to exclude an element in the associated position:

```
# exclude 2nd year
amounts[-2]
#> [1] 1020.000 1061.208 1082.432 1104.081 1126.162 1148.686 1171.659 1195.093
#> [9] 1218.994

# exclude 2nd and 4th years
amounts[-c(2, 4)]
#> [1] 1020.000 1061.208 1104.081 1126.162 1148.686 1171.659 1195.093 1218.994
```

6.4.2 Character Subsetting

Sometimes, you may have a vector with named elements. When this is the case, you can use a character vector—containing one or more of the element names—as the indexing vector.

None of the vectors that we have created so far have named elements. So let's see how to do this. One way to give names to the elements of an existing vector is with the function `names()`

```
amounts3 = amounts[1:3]
names(amounts3) = c("y1", "y2", "y3")
amounts3
#>       y1       y2       y3
#> 1020.000 1040.400 1061.208
```

When a vector, like `amounts3`, has named elements, we can use those names for subsetting purposes. Instead of using a numeric vector we use a character vector. Hence the term *character subsetting*.

For example, to extract the element in `amounts3` that has name `"y1"` we pass this string inside the brackets:

```
amounts3["y1"]
#>   y1
#> 1020
```

To get the elements in `amounts3` that have names `"y1"` and `"y3"`, we can write

```
amounts3[c("y1", "y3")]
#>       y1       y3
#> 1020.000 1061.208
```

And like in the numeric subsetting case, we can also write a command such as:

```
amounts3[c("y2", "y2", "y2", "y1")]
#>     y2     y2     y2     y1
#> 1040.4 1040.4 1040.4 1020.0
```

6.4.3 Logical Subsetting

Another type of subsetting is when we use a logical vector as the indexing vector.

Let me show you an example of logical subsetting. In this case, we will use a logical vector with three elements `c(TRUE, FALSE, FALSE)` and pass this inside the brackets:

```
amounts3[c(TRUE, FALSE, FALSE)]
#>   y1
#> 1020
```

As you can tell, the retrieved element in `amounts3` is the one associated to the `TRUE` position, whereas those elements associated to the `FALSE` values are excluded. This is how the logical values (in the indexing vector) are used:

- `TRUE` means inclusion
- `FALSE` means exclusion

So, if we want to extract only the element in the second position, we could write something like this:

```
amounts3[c(FALSE, TRUE, FALSE)]
#>     y2
#> 1040.4
```

Now, I have to say that doing logical subsetting in this way is not really how we tend to use it in practice. In other words, we won't be providing an explicit logical vector, typing a bunch of `TRUE`'s and `FALSE`'s values. Instead, what we typically do is to provide a command that, when executed by R, will return a logical vector.

Consider the following example. We create a vector `x`, and then we use the greater than symbol `>` to compute a mathematical comparison which in turn will return a logical vector.

```
x = c(2, 4, 6, 8)
x > 5
#> [1] FALSE FALSE  TRUE  TRUE
```

Knowing that `x > 5` produces a logical vector in which `FALSE` indicates that the number is less than 5, and `TRUE` indicates that the number is greater than five, we can write the following command to subset those elements in `x` that are greater than five:

```
x[x > 5]
#> [1] 6 8
```

This is a simple example of logical subsetting because the indexing vector is the logical vector that comes from executing the comparison `x > 5`.

Here is a less simple example of logical subsetting to extract the elements in `x` that are greater than 3 and less than or equal to 6. This requires two comparison expressions, `x > 3` and `x <= 6`, and the use of the logical *AND* operator `&` to form a compound expression:

```
x[x > 3 & x <= 6]
#> [1] 4 6
```

6.4.4 Summary of Subsetting

In summary, the things that you can specify inside the brackets are three kind of vectors:

- numeric vectors
- logical vectors (the length of the logical vector must match the length of the vector to be subset)
- character vectors (if the elements have names)

In addition to the brackets `[]`, some common functions that you can use on vectors are:

- `length()` gives the number of values
- `sort()` sorts the values in increasing or decreasing ways
- `rev()` reverses the values
- `unique()` extracts unique elements

```
length(amounts3)
amounts3[length(amounts3)]
sort(amounts3, decreasing = TRUE)
rev(amounts3)
```

Chapter 7

Matrices and Arrays

In this chapter we introduce R `arrays`, which are multidimensional data objects including 2-dimensional arrays better known as matrices, and N-dimensional arrays (generic arrays).

7.1 Motivation

Let us continue discussing the savings-investing scenario in which you deposit $1000 into a savings account that pays you an annual interest rate of return of 2%.

Assuming that you leave that money in the bank for several years, without changing the rate of return r, you can use the Future Value (FV) formula to calculate how much money you'll have at the end of year t:

$$\text{FV} = \text{PV}(1 + r)^t$$

where:

- FV = future value
- PV = present value
- r = annual interest rate
- t = number of years

Here's some R code to obtain a vector `amounts` containing the amount of money that you would have at the end of every year during a 5 year period:

```
# inputs
deposit = 1000
rate = 0.02
years = 1:5
```

```
# future values
amounts = deposit * (1 + rate)^years
amounts
#> [1] 1020.000 1040.400 1061.208 1082.432 1104.081
```

Recall that this code is an example of vectorized (and recycling) code because the FV formula is applied to all the elements of the involved vectors, some of which have different lengths.

So far, so good.

Now, consider a seemingly simple modification. What if you want to organize the amount values in a table? Something like this:

year	amount
0	1000.000
1	1020.000
2	1040.400
3	1061.208
4	1082.432
5	1104.081

In other words, what if you are interested not in getting the set of future values in a vector, but instead you want them to be arranged in some sort of tabular object? How could you create a table in which the first column corresponds to the years and the second column corresponds to the future amounts? Well, let's find out.

7.2 Tables with Matrices

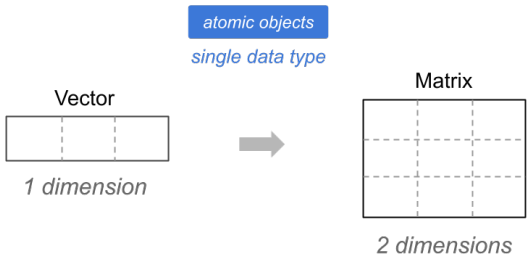
R provides two main ways to organize data in a tabular (i.e. rectangular) object

- using an R `matrix`
- using an R `data.frame`

Let's talk about the first one.

7.2.1 R Matrices

- R matrices are rectangular arrays
- While vectors are one-dimensional objects, matrices are **two-dimensional** objects
- Matrices are also **atomic** objects (all their elements must be of the same type)



Column binding vectors

You can build a matrix by **column binding** vectors using the function `cbind()`:

```
# inputs
deposit = 1000
rate = 0.02
years = 0:5

# future values
amounts = deposit * (1 + rate)^years

# output as a matrix via cbind()
savings = cbind(years, amounts)
savings
#>      years  amounts
#> [1,]     0 1000.000
#> [2,]     1 1020.000
#> [3,]     2 1040.400
#> [4,]     3 1061.208
#> [5,]     4 1082.432
#> [6,]     5 1104.081
```

As you can tell, the use of `cbind()` is straightforward. All you have to do is indicate the name of the vectors, separating them with a comma. Each vector will become a column of the returned matrix.

Row binding vectors

You can also build a matrix by **row binding** vectors.

```
savings = rbind(years, amounts)
savings
#>         [,1] [,2]   [,3]     [,4]     [,5]
#> years      0    1    2.0    3.000    4.000
#> amounts 1000 1020 1040.4 1061.208 1082.432
#>             [,6]
```

```
#> years      5.000
#> amounts 1104.081
```

The difference between `cbind()` and `rbind()` is that the latter will "stack" the given vectors on top of each other. That is, each vector will become a row of the returned matrix.

7.3 Creating matrices with `matrix()`

More commonly, you use the function `matrix()` to create a matrix by providing an input vector, and defining the number of rows and columns (i.e. the *matrix dimensions*).

```
savings = matrix(c(years, amounts), nrow = 6, ncol = 2)
savings
#>      [,1]     [,2]
#> [1,]    0 1000.000
#> [2,]    1 1020.000
#> [3,]    2 1040.400
#> [4,]    3 1061.208
#> [5,]    4 1082.432
#> [6,]    5 1104.081
```

The input vector `c(years, amounts)` is arranged into 6 rows and 2 columns.

Giving names to rows and columns

Often, you may need to provide names for either rows and columns

```
savings = matrix(c(years, amounts), nrow = 6, ncol = 2)
rownames(savings) = 1:6
colnames(savings) = c("year", "amount")
savings
#>   year   amount
#> 1    0 1000.000
#> 2    1 1020.000
#> 3    2 1040.400
#> 4    3 1061.208
#> 5    4 1082.432
#> 6    5 1104.081
```

7.3.1 More Matrices

Let's make things a bit more complex. Say you have the following investments:

- $1000 in a **savings account** that pays 2% annual return, during 4 years

- $2000 in a **money market** account that pays 2.5% annual return, during 2 years
- $5000 in a **certificate of deposit** that pays 3% annual return, during 3 years

In R, we can calculate the future values of each type of investment product:

```
# savings account
savings = 1000 * (1 + 0.02)^(0:4)
savings
#> [1] 1000.000 1020.000 1040.400 1061.208 1082.432
```

```
# money market
moneymkt = 2000 * (1 + 0.025)^(0:2)
moneymkt
#> [1] 2000.00 2050.00 2101.25
```

```
# certificate of deposit
certificate = 5000 * (1 + 0.03)^(0:3)
certificate
#> [1] 5000.000 5150.000 5304.500 5463.635
```

Separated matrices

We can create individual matrices:

```
sav_mat = cbind(0:4, savings)

mm_mat = cbind(0:2, moneymkt)

cd_mat = cbind(0:3, certificate)
```

Or we can stack everything into a single matrix:

```
cbind(c(0:4, 0:2, 0:3), c(savings, moneymkt, certificate))
#>       [,1]     [,2]
#>  [1,]    0 1000.000
#>  [2,]    1 1020.000
#>  [3,]    2 1040.400
#>  [4,]    3 1061.208
#>  [5,]    4 1082.432
#>  [6,]    0 2000.000
#>  [7,]    1 2050.000
#>  [8,]    2 2101.250
#>  [9,]    0 5000.000
#> [10,]    1 5150.000
#> [11,]    2 5304.500
#> [12,]    3 5463.635
```

What if you want some table like this:

account	year	amount
savings	0	1000.000
savings	1	1020.000
savings	2	1040.400
savings	3	1061.208
savings	4	1082.432
moneymkt	0	2000.000
moneymkt	1	2050.000
moneymkt	2	2101.250
certif	0	5000.000
certif	1	5150.250
certif	2	5304.500
certif	3	5463.635

We could use the `cbind()` function in an attempt to obtain a matrix having a similar rectangular structure as in the above table:

```
investments = cbind(
  rep(c("savings", "moneymkt", "certif"), times = c(5, 3, 4)),
  c(0:4, 0:2, 0:3),
  c(savings, moneymkt, certificate))

investments
#>       [,1]       [,2] [,3]
#>  [1,] "savings"  "0"  "1000"
#>  [2,] "savings"  "1"  "1020"
#>  [3,] "savings"  "2"  "1040.4"
#>  [4,] "savings"  "3"  "1061.208"
#>  [5,] "savings"  "4"  "1082.43216"
#>  [6,] "moneymkt" "0"  "2000"
#>  [7,] "moneymkt" "1"  "2050"
#>  [8,] "moneymkt" "2"  "2101.25"
#>  [9,] "certif"   "0"  "5000"
#> [10,] "certif"   "1"  "5150"
#> [11,] "certif"   "2"  "5304.5"
#> [12,] "certif"   "3"  "5463.635"
```

Do you notice something funny with the matrix `investments`?

As you can tell, all the values in `investments` are displayed being surrounded with double quotes. This indicates that all the values are of type `character`. Why?

Recall that matrices are **atomic** objects. Therefore, all the values in a matrix

must be of the same data type. In this example, because the first input vector to `cbind()` is a character vector, this will dictate the type flavor of the produced matrix `investments`.