Coding in R

An introduction to practical programming in R

Gaston Sanchez

2022-03-16

Contents

Welcome

This book is a work in progress for an introductory textbook on programming in R.

In a nutshell, we discuss the basics of R, covering properties of data objects, such as vectors, factors, matrices, data frames, and lists. We also cover main principles for data manipulation, reshaping, tidying, etc. Likewise, we discuss fundamental notions of programming (e.g. functions, conditionals, iterations). And last but not least we describe how to make basic (and not so basic) 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. I used to tell myself something along the lines of: "The world does not need another introductory book about R." And I still somewhat agree with this. Okay, maybe the world does not need another book like this … but I do.

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, and more important, it fits my needs when teaching courses such as STAT 133, STAT 33A, STAT 33B, STAT 243, and similar courses on programming, data analysis, data science, and related activities … with R.

As you will see, the book has an overarching theme based on financial math formulas and personal finance applications. 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.

I hope you enjoy reading the book, learn something (hopefully a lot) about coding in R, and also a bit of financial math. Enough chit-chat. Let's get started.

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*.

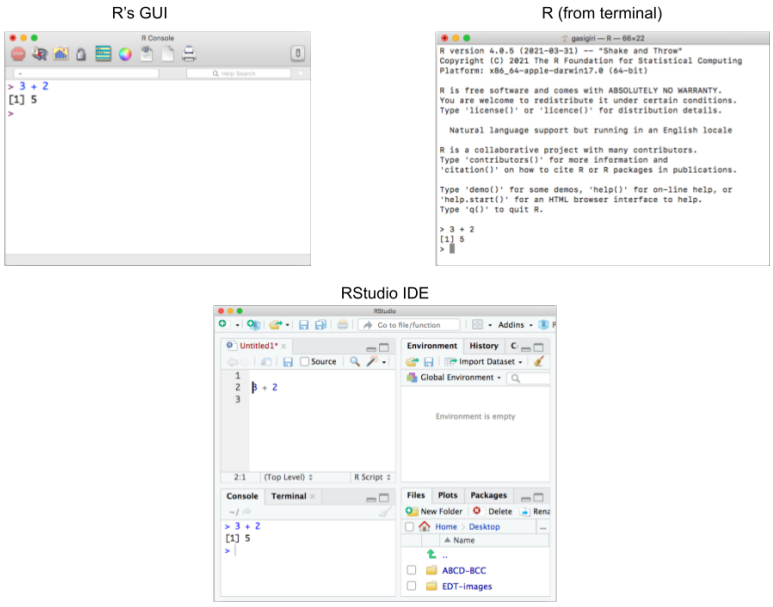


Figure 1.1: Interacting with R in various ways: R's GUI, RStudio, and R from a command-line terminal

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

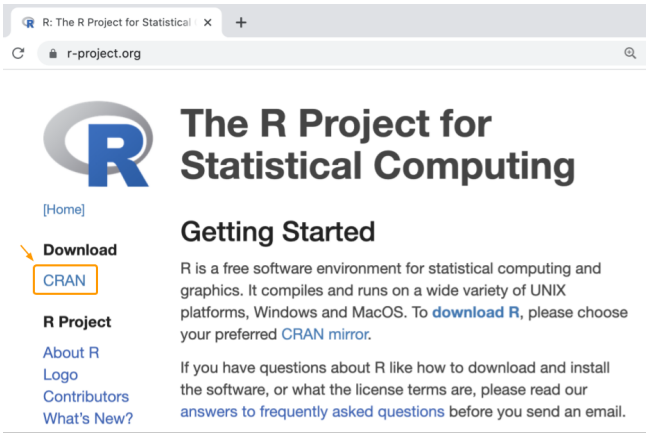


Figure 1.3: R project's home webpage

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).

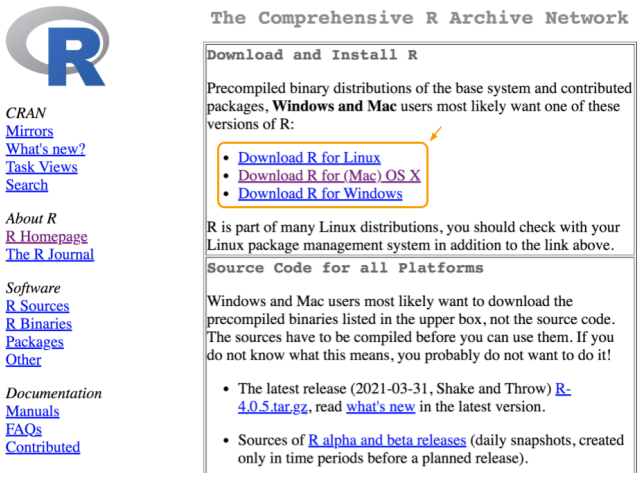


Figure 1.4: R is available for MacOS, Windows, and Linux

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):

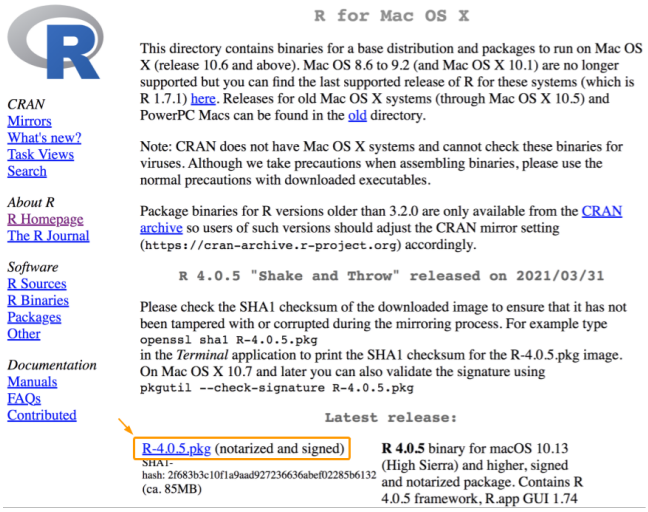


Figure 1.5: CRAN download for Mac

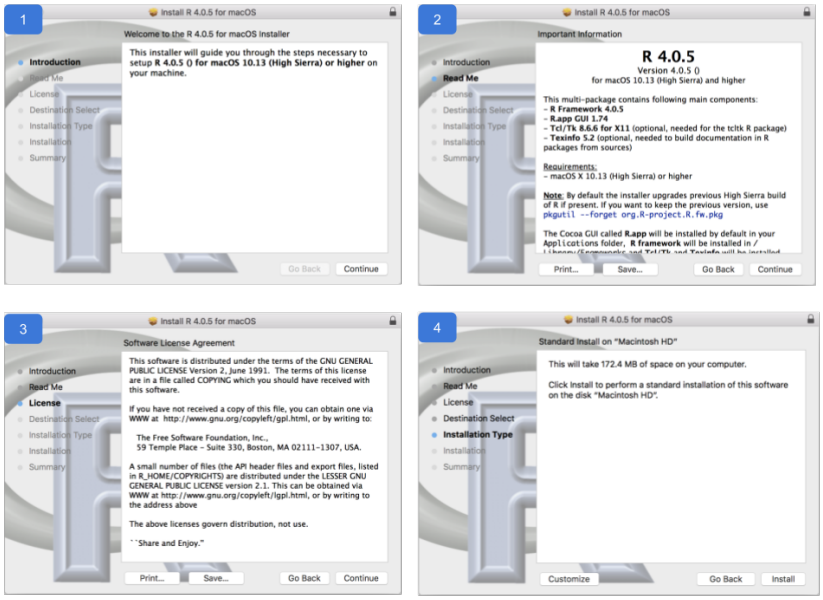


Figure 1.6: R Installation wizard for Mac

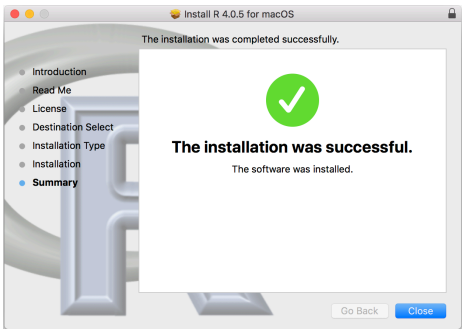


Figure 1.7: R Installation wizard for Mac

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

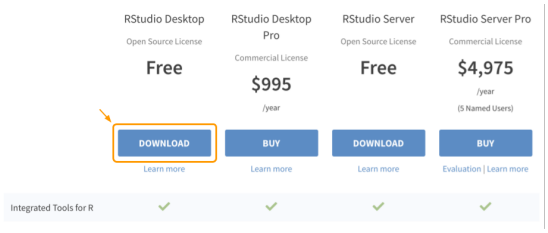


Figure 1.9: Choose RStudio free desktop

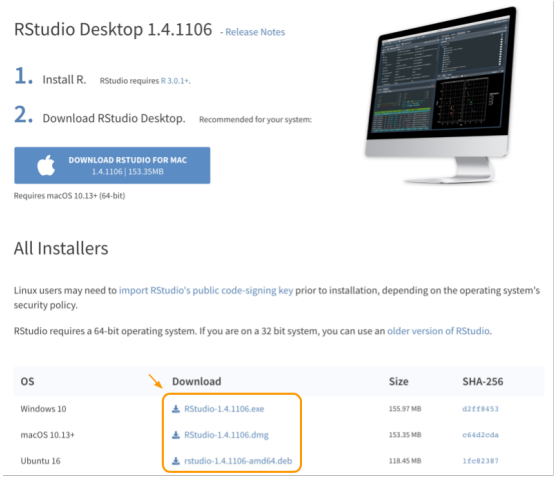


Figure 1.10: RStudio Desktop versions

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.

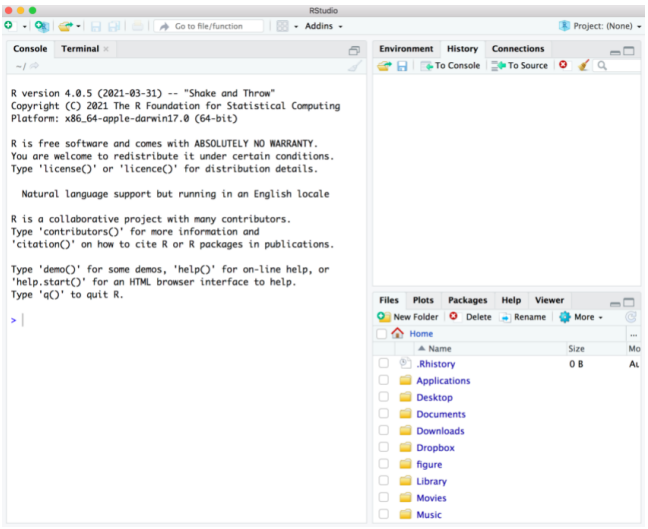


Figure 2.1: Screenshot of RStudio when launched for the first time.

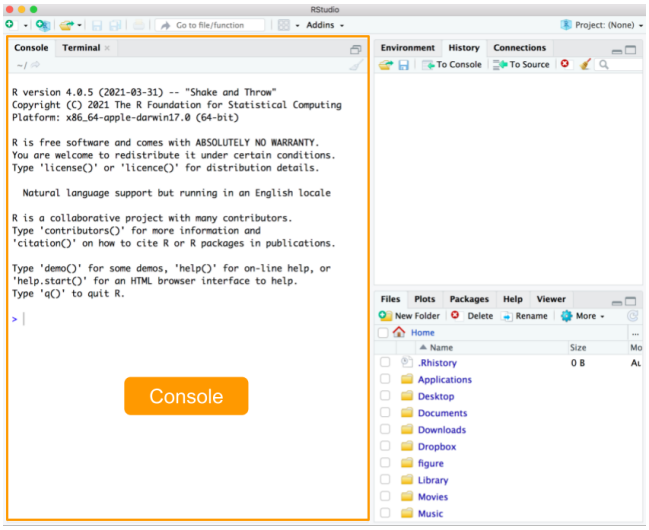


Figure 2.2: Console quadrant in RStudio.

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

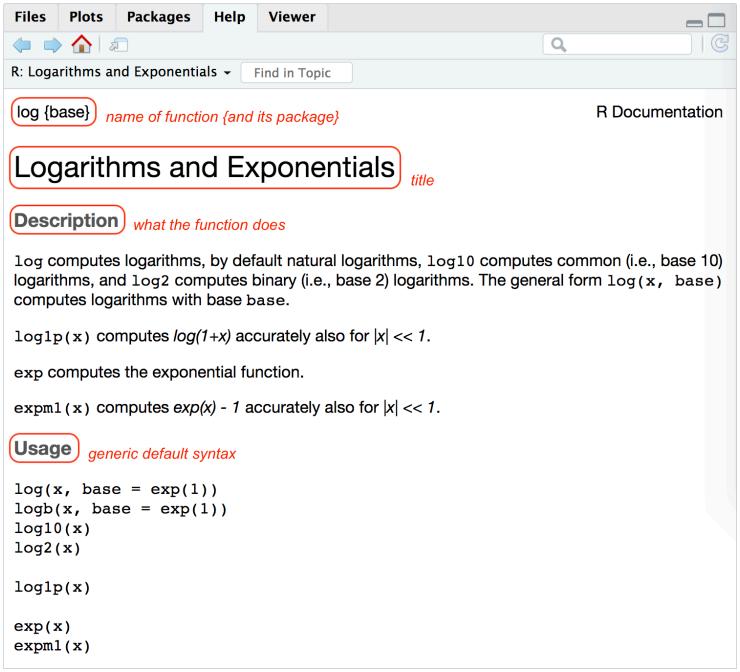


Figure 2.3: Help documentation for the log function (part 1)

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

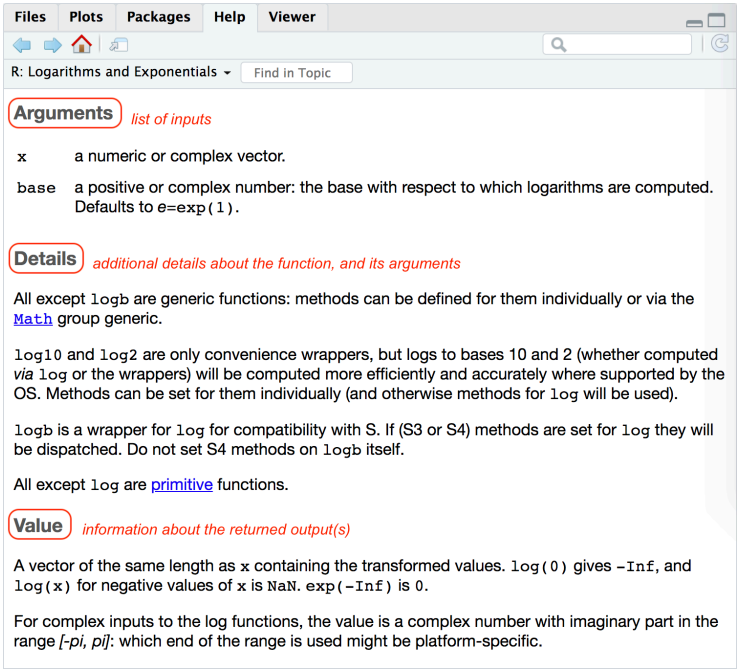


Figure 2.4: Help documentation for the log function (part 2)

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

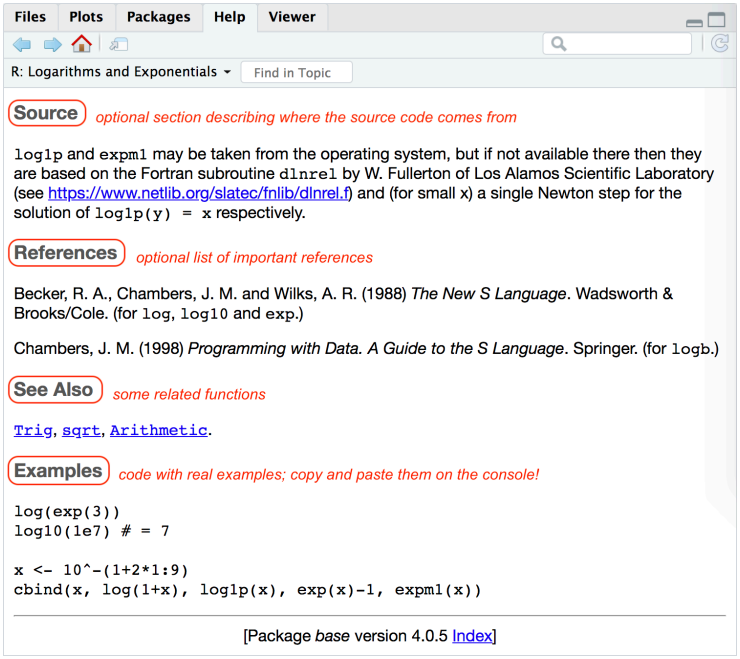


Figure 2.5: Help documentation for the log function (part 3)

would she spend during a school "year"? (assume the academic year is 10 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 officially 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

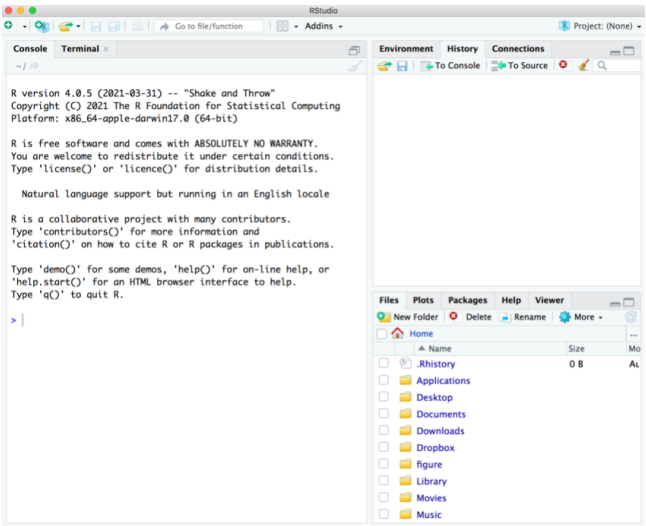


Figure 3.1: Screenshot of RStudio when launched for the first time.

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.

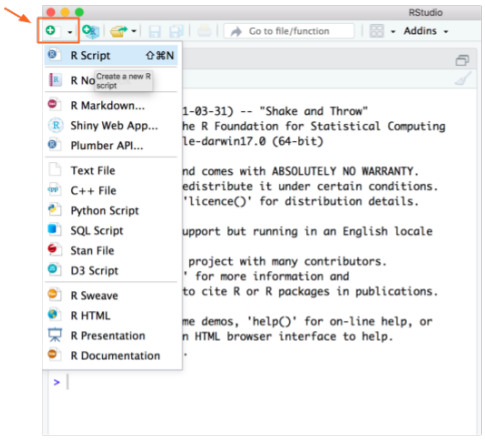


Figure 3.2: Opening a new (text) file in RStudio.

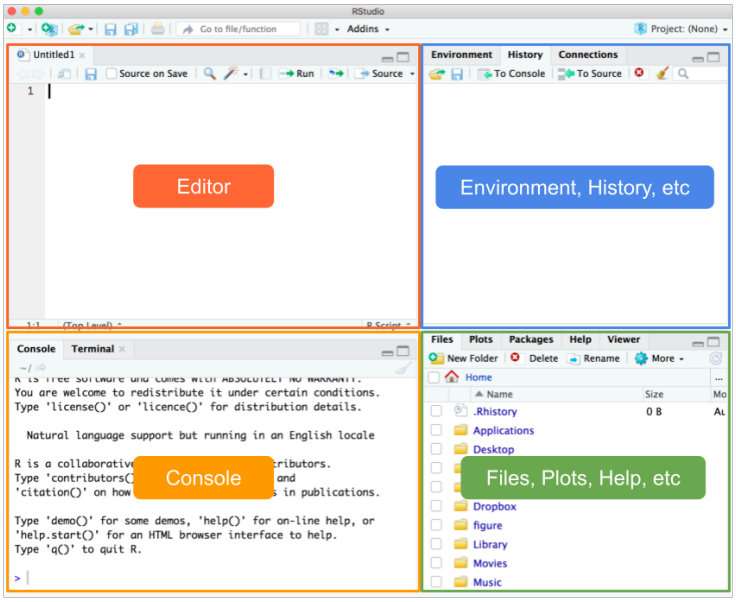


Figure 3.3: RStudio layout organized into quadrants.

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 in 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 of the `Console` is when I want to calculate basic things like the monthly balance in my credit cards, or the overall score for one of the students in my classes, 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 more often than not 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 have 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 using RStudio.

From the point of view of a session, all the work, activities, and actions you do with R can be classified into three categories:

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

Because there are several things going on behind the scenes in each of the categories listed above, it is important that we talk about them—at least briefly.

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 in RStudio 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. You can also invoke the list function `ls()` to list any available objects in your current session:

```
# 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 an associated a text file called `.Rhistory` in your working directory. You can also use the `history()` function to display the *Commands History*, that is, all the commands invoked in your interactive sessions:

```
# 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 to R. By 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.

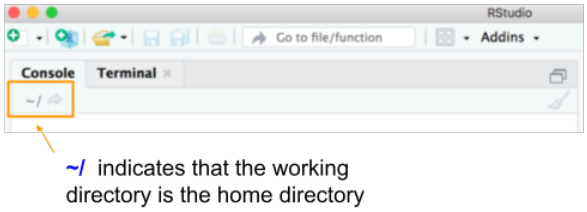


Figure 4.1: The working directory is indicated right below the Console tab.

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 come from?
- if the command produces an output, where does that output go to?

This is why we need to describe the following:

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 has been saved, will live in some directory. **Ideally**, a source document's directory would be used as its working directory, using relative filepaths to handle all input and output resources required in thecode of the source document. 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 in.
- 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()`.

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

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

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 actions 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] 1040.4
```

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] 1061.208
```

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] 1061.208
```

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] 1061.208
```

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] 1020.000 1040.400 1061.208
```

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**.

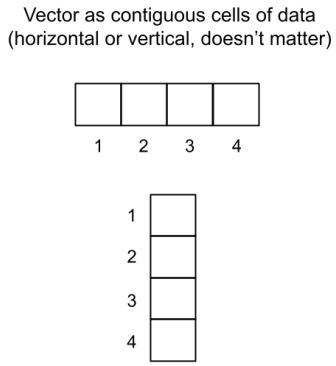
5.2 About R Vectors

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:



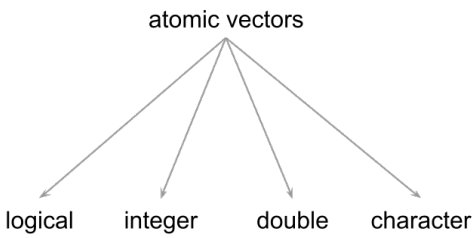
They can be of any length (including 0), and the starting position or index is always number 1.

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"`.

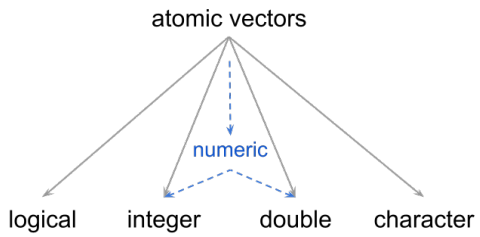


Figure 5.1: Data types "integer" and "double" correspond to "numeric" mode

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
```

5.6 Exercises

1) Consider the data—about so-called **Terrestrial** planets—provided in the table below. These planets include Mercury, Venus, Earth, and Mars. They are called terrestrial because they are "Earth-like" planets in contrast to the **Jovian** planets that involve planets similar to Jupiter (i.e. Jupiter, Saturn, Uranus and Neptune). The main characteristics of terrestrial planets is that

they are relatively small in size and in mass, with a solid rocky surface, and metals deep in its interior.

planet	gravity	daylength	temp	moons	haswater
Mercury	3.7	4222.6	167	0	FALSE
Venus	8.9	2802	464	0	FALSE
Earth	9.8	24	15	1	TRUE
Mars	3.7	24.7	-65	2	FALSE

Create vectors for each of the columns in the data table displayed above, according to the following data-type specifications:

- `planet`: character vector
- `gravity`: real (i.e. double) vector (m/s^2)
- `daylength`: real (i.e. double) vector (hours)
- `temp`: integer vector (mean temperature in Celsius)
- `moons`: integer vector (number of moons)
- `haswater`: logical vector indicating whether a planet has known bodies of liquid water on its surface

2) Refer to the vectors created in the previous question. Without running any R commands, try to guess the data type—as returned by `typeof()`—if you had to create a new vector by combining, using the function `c()`, the following:

a) combine `planets` with `gravity`

b) combine `planets` with `temp`

c) combine `planets` with `haswater`

d) `gravity` with `daylength`

e) combine `gravity` with `temp`

f) combine `temp` with `moons`

g) combine `temp` with `haswater`

3) What is the data type—as returned by `typeof()`—of each of the following vectors. Try guessing the data type without running any commands.

- `x`: where `x <- c(TRUE, FALSE)`
- `y`: where `y <- c(x, 10)`
- `z`: where `z <- c(y, 10, "a")`

4) What is the data type—as returned by `typeof()`—of each of the following vectors. Try guessing the data type without running any commands.

- `x`: where `x <- c('1', '2', '3', '4')`

- `y`: where `y <- (x == 1)`
- `z`: where `z <- y + 0`
- `w`: where `w <- c(x, "5.5")`
- `yz1`: where `yz1 <- c(y, z, pi)`

5) How do you use the function `seq()` to create the following vector?

```
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

6) Write a command using the function `rep()` and the input vector `1:3` to create the following vector:

```
[1] 1 1 2 2 3 3
```

7) Write a command using the function `rep()` and the input vector `1:3` to create the following vector:

```
[1] 1 2 3 1 2 3
```

8) Suppose `y <- c(1, 4, 9, 16, 25)`. Write down the R command to return a vector `z`, in which each element of `z` is the square root of each element of the vector `y`.

9) A student is trying to implement the following formula in R:

$$e^{\frac{-(X-\mu)^2}{2\sigma^2}}$$

However, the student gets unexpected results when using the code:

```
exp(-(x - mu)^2 / 2 * sigma^2)
```

Explain what the problem is, and correct the code.

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` are vectors of length 1. And so it is 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:

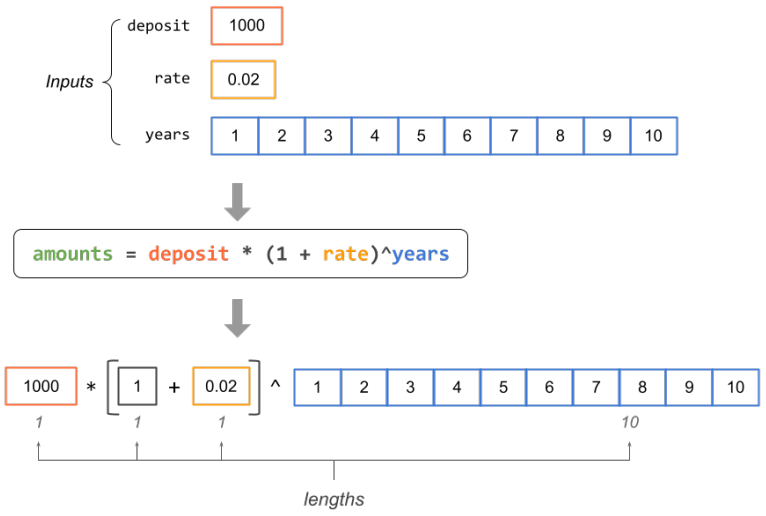
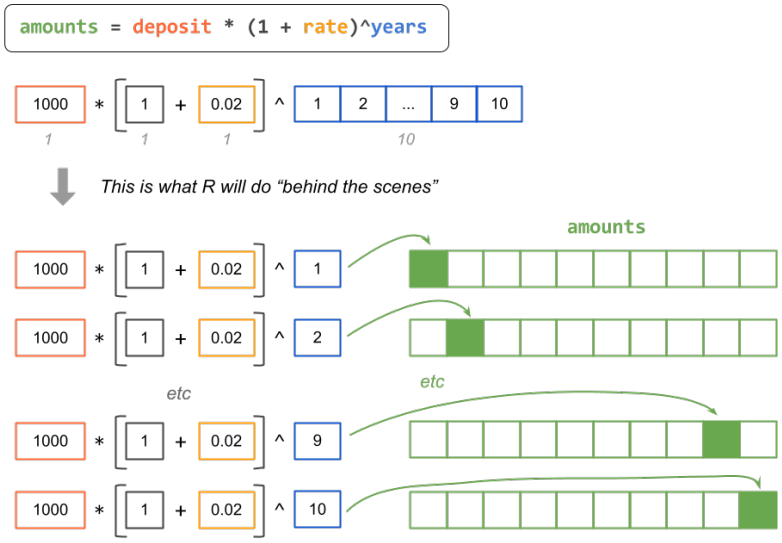


Figure 6.1: Diagram depicting vectors of different lengths.

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)
```

6.5 Exercises

1) Explain, with your own words, the concept of *vectorization* a.k.a. vectorized operations.

2) Write 2 different R commands to return the first five elements of a vector `x` (assume `x` has more than 5 elements).

3) Which command will fail to return the first five elements of a vector `x`? (assume `x` has more than 5 elements).

a) `x[1:5]`

b) `x[c(1,2,3,4,5)]`

c) `head(x, n = 5)`

d) `x[seq(1, 5)]`

e) `x(1:5)`

4) Consider the following code to obtain vectors `name` and `mpg` from the data set `mtcars` that contains data about 32 automobiles (1973–74 models).

```
# vectors name and mpg
name = rownames(mtcars)
mpg = mtcars$mpg
```

Use `seq()`, and bracket notation, to subset (extract):

a) all the even elements in `name` (i.e. extract positions 2, 4, 6, etc)

b) all the odd elements in `mpg` (i.e. extract positions 1, 3, 5, etc)

c) all the elements in positions multiples of 5 (e.g. extract positions 5, 10, 15, etc) of `mpg`

d) all the even elements in `name` but this time in reverse order; *hint* the `rev()` function is your friend.

5) Consider the following code to obtain vectors `name`, `mpg` and `cyl` from the data set `mtcars` that contains data about 32 automobiles (1973–74 models).

```
# vectors name, mpg and cyl
name = rownames(mtcars)
mpg = mtcars$mpg # miles-per-gallon
cyl = mtcars$cyl  # number of cylinders
```

Write commands, using bracket notation, to answer the parts listed below. You may need to use `is.na()`, `sum()`, `min()`, `max()`, `which()`, `which.min()`, `which.max()`:

a) name of cars that have a fuel consumption of less than 15 mpg

b) name of cars that have 6 cylinders

c) largest mpg value of all cars with 8 cylinders

d) name of car(s) with mpg equal to the median mpg

e) name of car(s) with mpg of at most 22, and at least 6 cylinders

Chapter 7

Factors

I'm one of those with the humble opinion that great software for data science and analytics should have a data structure dedicated to handle categorical data. Lucky for us, one of the nicest features about R is that it provides a data object exclusively designed to handle categorical data: **factors**.

The term "factor" as used in R for handling categorical variables, comes from the terminology used in *Analysis of Variance*, commonly referred to as ANOVA. In this statistical method, a categorical variable is commonly referred to as, surprise-surprise, *factor* and its categories are known as *levels*. Perhaps this is not the best terminology but it is the one R uses, which reflects its distinctive statistical origins. Especially for those users without a brackground in statistics, this is one of R's idiosyncracies that seems disconcerning at the beginning. But as long as you keep in mind that a factor is just the object that allows you to handle a qualitative variable you'll be fine. In case you need it, here's a short mantra to remember:

> factors have levels

7.1 Creating Factors

To create a factor in R you use the homonym function `factor()`, which takes a vector as input. The vector can be either numeric, character or logical. Let's see our first example:

```
# numeric vector
num_vector <- c(1, 2, 3, 1, 2, 3, 2)

# creating a factor from num_vector
first_factor <- factor(num_vector)
```

```
first_factor
#> [1] 1 2 3 1 2 3 2
#> Levels: 1 2 3
```

As you can tell from the previous code snippet, `factor()` converts the numeric vector `num_vector` into a factor (i.e. a categorical variable) with 3 categories—the so called `levels`.

You can also obtain a factor from a string vector:

```
# string vector
str_vector <- c('a', 'b', 'c', 'b', 'c', 'a', 'c', 'b')

str_vector
#> [1] "a" "b" "c" "b" "c" "a" "c" "b"

# creating a factor from str_vector
second_factor <- factor(str_vector)

second_factor
#> [1] a b c b c a c b
#> Levels: a b c
```

Notice how `str_vector` and `second_factor` are displayed. Even though the elements are the same in both the vector and the factor, they are printed in different formats. The letters in the string vector are displayed with quotes, while the letters in the factor are printed without quotes.

And of course, you can use a logical vector to generate a factor as well:

```
# logical vector
log_vector <- c(TRUE, FALSE, TRUE, TRUE, FALSE)

# creating a factor from log_vector
third_factor <- factor(log_vector)

third_factor
#> [1] TRUE  FALSE TRUE  TRUE  FALSE
#> Levels: FALSE TRUE
```

7.2 How R treats factors

Technically speaking, R factors are referred to as *compound objects*. According to the "R Language Definition" manual:

> "Factors are currently implemented using an integer array to specify the actual levels and a second array of names that are mapped to the integers."

What does this mean?

Essentially, a factor is internally stored using two ingredients: one is an integer vector containing the values of categories, the other is a vector with the "levels" which has the names of categories which are mapped to the integers.

Under the hood, the way R stores factors is as vectors of integer values. One way to confirm this is using the function `storage.mode()`

```
# storage of factor
storage.mode(first_factor)
#> [1] "integer"
```

This means that we can manipulate factors just like we manipulate vectors. In addition, many functions for vectors can be applied to factors. For instance, we can use the function `length()` to get the number of elements in a factor:

```
# factors have length
length(first_factor)
#> [1] 7
```

We can also use the square brackets `[ ]` to extract or select elements of a factor. Inside the brackets we specify vectors of indices such as numeric vectors, logical vectors, and sometimes even character vectors.

```
# first element
first_factor[1]

# third element
first_factor[3]

# second to fourth elements
first_factor[2:4]

# last element
first_factor[length(first_factor)]

# logical subsetting
first_factor[rep(c(TRUE, FALSE), length.out = 7)]
```

If you have a factor with named elements, you can also specify the names of the elements within the brackets:

```
names(first_factor) <- letters[1:length(first_factor)]
first_factor
#> a b c d e f g
#> 1 2 3 1 2 3 2
#> Levels: 1 2 3
```

```
first_factor[c('b', 'd', 'f')]
#> b d f
#> 2 1 3
#> Levels: 1 2 3
```

However, you should know that factors are NOT really vectors. To see this you can check the behavior of the functions `is.factor()` and `is.vector()` on a factor:

```
# factors are not vectors
is.vector(first_factor)
#> [1] FALSE

# factors are factors
is.factor(first_factor)
#> [1] TRUE
```

Even a single element of a factor is also a factor:

```
class(first_factor[1])
#> [1] "factor"
```

So what makes a factor different from a vector?

Well, it turns out that factors have an additional attribute that vectors don't: `levels`. And as you can expect, the class of a factor is indeed `"factor"` (not `"vector"`).

```
# attributes of a factor
attributes(first_factor)
#> $levels
#> [1] "1" "2" "3"
#>
#> $class
#> [1] "factor"
#>
#> $names
#> [1] "a" "b" "c" "d" "e" "f" "g"
```

Another feature that makes factors so special is that their values (the levels) are mapped to a set of character values for displaying purposes. This might seem like a minor feature but it has two important consequences. On the one hand, this implies that factors provide a way to store character values very efficiently. Why? Because each unique character value is stored only once, and the data itself is stored as a vector of integers.

Notice how the numeric value `1` was mapped into the character value `"1"`. And the same happens for the other values `2` and `3` that are mapped into the char-

acters "2" and "3".

What is the advantage of R factors?

Every time I teach about factors, there is inevitably one student who asks a very pertinent question: Why do we want to use factors? Isn't it redundant to have a factor object when there are already character or integer vectors?

I have two answers to this question.

The first has to do with the storage of factors. Storing a factor as integers will usually be more efficient than storing a character vector. As we've seen, this is an important issue especially when the data—to be encoded into a factor—is of considerable size.

The second reason has to do with categorical variables of *ordinal* nature. Qualitative data can be classified into nominal and ordinal variables. Nominal variables could be easily handled with character vectors. In fact, *nominal* means name (values are just names or labels), and there's no natural order among the categories.

A different story is when we have ordinal variables, like sizes "small", "medium", "large" or college years "freshman", "sophomore", "junior", "senior". In these cases we are still using names of categories, but they can be arranged in increasing or decreasing order. In other words, we can rank the categories since they have a natural order: small is less than medium which is less than large. Likewise, freshman comes first, then sophomore, followed by junior, and finally senior.

So here's an important question: How do we keep the order of categories in an ordinal variable? We can use a character vector to store the values. But a character vector does not allow us to store the ranking of categories. The solution in R comes via factors. We can use factors to define ordinal variables, like the following example:

```
sizes <- factor(
  x = c('sm', 'md', 'lg', 'sm', 'md'),
  levels = c('sm', 'md', 'lg'),
  ordered = TRUE)

sizes
#> [1] sm md lg sm md
#> Levels: sm < md < lg
```

As you can tell, sizes has ordered levels, clearly identifying the first category "sm", the second one "md", and the third one "lg".

7.3 A closer look at `factor()`

Since working with categorical data in R typically involves working with factors, you should become familiar with the variety of functions related with them. In the following sections we'll cover a bunch of details about factors so you can be better prepared to deal with any type of categorical data.

7.3.1 Function `factor()`

Given the fundamental role played by the function `factor()` we need to pay a closer look at its arguments. If you check the documentation—see `help(factor)`—you'll see that the usage of the function `factor()` is:

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax = NA)
```

with the following arguments:

- `x` a vector of data
- `levels` an optional vector for the categories
- `labels` an optional character vector of labels for the levels
- `exclude` a vector of values to be excluded when forming the set of levels
- `ordered` logical value to indicate if the levels should be regarded as ordered
- `nmax` an upper bound on the number of levels

The main argument of `factor()` is the input vector `x`. The next argument is `levels`, followed by `labels`, both of which are optional arguments. Although you won't always be providing values for `levels` and `labels`, it is important to understand how R handles these arguments by default.

Argument `levels`

If `levels` is not provided (which is what happens in most cases), then R assigns the unique values in `x` as the category levels.

For example, consider our numeric vector from the first example: `num_vector` contains unique values 1, 2, and 3.

```
# numeric vector
num_vector <- c(1, 2, 3, 1, 2, 3, 2)

# creating a factor from num_vector
first_factor <- factor(num_vector)

first_factor
#> [1] 1 2 3 1 2 3 2
#> Levels: 1 2 3
```

Now imagine we want to have `levels` 1, 2, 3, 4, and 5. This is how you can define the factor with an extended set of levels:

```
# numeric vector
num_vector
#> [1] 1 2 3 1 2 3 2

# defining levels
one_factor <- factor(num_vector, levels = 1:5)
one_factor
#> [1] 1 2 3 1 2 3 2
#> Levels: 1 2 3 4 5
```

Although the created factor only has values between 1 and 3, the `levels` range from 1 to 5. This can be useful if we plan to add elements whose values are not in the input vector `num_vector`. For instance, you can append two more elements to `one_factor` with values `4` and `5` like this:

```
# adding values 4 and 5
one_factor[c(8, 9)] <- c(4, 5)
one_factor
#> [1] 1 2 3 1 2 3 2 4 5
#> Levels: 1 2 3 4 5
```

If you attempt to insert an element having a value that is not in the predefined set of levels, R will insert a missing value (`<NA>`) instead, and you'll get a warning message like the one below:

```
# attempting to add value 6 (not in levels)
one_factor[1] <- 6
#> Warning in `[<-.factor`(`*tmp*`, 1, value = 6): invalid factor level, NA
#> generated
one_factor
#> [1] <NA> 2    3    1    2    3    2    4    5
#> Levels: 1 2 3 4 5
```

Argument `labels`

Another very useful argument is `labels`, which allows you to provide a string vector for naming the `levels` in a different way from the values in `x`. Let's take the numeric vector `num_vector` again, and say we want to use words as labels instead of numeric values. Here's how you can create a factor with predefined `labels`:

```
# defining labels
num_word_vector <- factor(num_vector, labels = c("one", "two", "three"))

num_word_vector
#> [1] one   two   three one   two   three two
#> Levels: one two three
```

Argument `exclude`

If you want to ignore some values of the input vector `x`, you can use the `exclude` argument. You just need to provide those values which will be removed from the set of `levels`.

```
# excluding level 3
factor(num_vector, exclude = 3)
#> [1] 1    2    <NA> 1    2    <NA> 2
#> Levels: 1 2

# excluding levels 1 and 3
factor(num_vector, exclude = c(1,3))
#> [1] <NA> 2    <NA> <NA> 2    <NA> 2
#> Levels: 2
```

The side effect of `exclude` is that it returns a missing value (`<NA>`) for each element that was excluded, which is not always what we want. Here's one way to remove the missing values when excluding 3:

```
# excluding level 3
num_fac12 <- factor(num_vector, exclude = 3)

# oops, we have some missing values
num_fac12
#> [1] 1    2    <NA> 1    2    <NA> 2
#> Levels: 1 2
# removing missing values
num_fac12[!is.na(num_fac12)]
#> [1] 1 2 1 2 2
#> Levels: 1 2
```

7.3.2 Unclassing factors

We've mentioned that factors are stored as vectors of integers (for efficiency reasons). But we also said that factors are more than vectors. Even though a factor is displayed with string labels, the way it is stored internally is as integers. Why is this important to know? Because there will be occasions in which you'll need to know exactly what numbers are associated to each level values.

Imagine you have a factor with `levels` 11, 22, 33, 44.

```
# factor
xfactor <- factor(c(22, 11, 44, 33, 11, 22, 44))
xfactor
#> [1] 22 11 44 33 11 22 44
#> Levels: 11 22 33 44
```

To obtain the integer vector associated to `xfactor` you can use the function

`unclass()`:

```
# unclassing a factor
unclass(xfactor)
#> [1] 2 1 4 3 1 2 4
#> attr(,"levels")
#> [1] "11" "22" "33" "44"
```

As you can see, the levels `"11"`, `"22"`, `"33"`, `"44"` were mapped to the vector of integers `(1 2 3 4)`.

An alternative option is to simply apply `as.numeric()` or `as.integer()` instead of using `unclass()`:

```
# equivalent to unclass
as.integer(xfactor)
#> [1] 2 1 4 3 1 2 4

# equivalent to unclass
as.numeric(xfactor)
#> [1] 2 1 4 3 1 2 4
```

Although rarely used, there can be some cases in which what you need to do is revert the integer values in order to get the original factor levels. This is only possible when the levels of the factor are themselves numeric. To accomplish this use the following command:

```
# recovering numeric levels
as.numeric(levels(xfactor))[xfactor]
#> [1] 22 11 44 33 11 22 44
```

7.4 Ordinal Factors

By default, `factor()` creates a *nominal* categorical variable, not an ordinal. One way to check that you have a nominal factor is to use the function `is.ordered()`, which returns `TRUE` if its argument is an ordinal factor.

```
# ordinal factor?
is.ordered(num_vector)
#> [1] FALSE
```

If you want to specify an ordinal factor you must use the `ordered` argument of `factor()`. This is how you can generate an ordinal value from `num_vector`:

```
# ordinal factor from numeric vector
ordinal_num <- factor(num_vector, ordered = TRUE)
ordinal_num
#> [1] 1 2 3 1 2 3 2
#> Levels: 1 < 2 < 3
```

As you can tell from the snippet above, the levels of `ordinal_factor` are displayed with less-than symbols "<'}, which means that the levels have an increasing order. We can also get an ordinal factor from our string vector:

```
# ordinal factor from character vector
ordinal_str <- factor(str_vector, ordered = TRUE)
ordinal_str
#> [1] a b c b c a c b
#> Levels: a < b < c
```

In fact, when you set `ordered = TRUE`, R sorts the provided values in alphanumeric order. If you have the following alphanumeric vector `("a1", "1a", "1b", "b1")`, what do you think will be the generated ordered factor? Let's check the answer:

```
# alphanumeric vector
alphanum <- c("a1", "1a", "1b", "b1")

# ordinal factor from character vector
ordinal_alphanum <- factor(alphanum, ordered = TRUE)
ordinal_alphanum
#> [1] a1 1a 1b b1
#> Levels: 1a < 1b < a1 < b1
```

An alternative way to specify an ordinal variable is by using the function `ordered()`, which is just a convenient wrapper for `factor(x, ..., ordered = TRUE)`:

```
# ordinal factor with ordered()
ordered(num_vector)
#> [1] 1 2 3 1 2 3 2
#> Levels: 1 < 2 < 3

# same as using 'ordered' argument
factor(num_vector, ordered = TRUE)
#> [1] 1 2 3 1 2 3 2
#> Levels: 1 < 2 < 3
```

A word of caution. Don't confuse the function `ordered()` with `order()`. They are not equivalent. `order()` arranges a vector into ascending or descending order, and returns the sorted vector. `ordered()`, as we've seen, is used to get ordinal factors.

Of course, you won't always be using the default order provided by the functions `factor(..., ordered = TRUE)` or `ordered()`. Sometimes you want to determine categories according to a different order.

For example, let's take the values of `str_vector` and let's assume that we want them in descending order, that is, `c < b < a`. How can you do that? Easy, you

just need to specify the `levels` in the order you want them and set `ordered = TRUE` (or use `ordered()`):

```
# setting levels with specified order
factor(str_vector, levels = c("c", "b", "a"), ordered = TRUE)
#> [1] a b c b c a c b
#> Levels: c < b < a

# equivalently
ordered(str_vector, levels = c("c", "b", "a"))
#> [1] a b c b c a c b
#> Levels: c < b < a
```

Here's another example. Consider a set of size values `"xs"` extra-small, `"sm"` small, `"md"` medium, `"lg"` large, and `"xl"` extra-large. If you have a vector with size values you can create an ordinal variable as follows:

```
# vector of sizes
sizes <- c("sm", "xs", "xl", "lg", "xs", "lg")

# setting levels with specified order
ordered(sizes, levels = c("xs", "sm", "md", "lg", "xl"))
#> [1] sm xs xl lg xs lg
#> Levels: xs < sm < md < lg < xl
```

Notice that when you create an ordinal factor, the given `levels` will always be considered in an increasing order. This means that the first value of `levels` will be the smallest one, then the second one, and so on. The last category, in turn, is taken as the one at the top of the scale.

Now that we have several nominal and ordinal factors, we can compare the behavior of `is.ordered()` on two factors:

```
# is.ordered() on an ordinal factor
ordinal_str
#> [1] a b c b c a c b
#> Levels: a < b < c
is.ordered(ordinal_str)
#> [1] TRUE

# is.ordered() on a nominal factor
second_factor
#> [1] a b c b c a c b
#> Levels: a b c
is.ordered(second_factor)
#> [1] FALSE
```

Chapter 8

Matrices and Arrays

In the previous three chapters, we discussed a number of ideas and concepts that basically have to do with vectors and their cousins factors. You can think of vectors and factors as one-dimensional objects. While many data sets can be handled through vectors and factors, there are occasions in which one dimension is not enough. The classic example for when one-dimensional objects are not enough involves working with data that fits better into a tabular structure consisting of a series of rows (one dimension) and columns (another dimension).

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

8.1 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 2%.

Assuming that you leave that money in the bank for several years, with a constant 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 from the beginning of time, and at the end of every year during a 5 year period:

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

# future values
amounts = deposit * (1 + rate)^years
amounts
#> [1] 1000.000 1020.000 1040.400 1061.208 1082.432
#> [6] 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 can you create a table in which the first column `year` corresponds to the years, and the second column `amount` corresponds to the future amounts? Well, let's find out.

8.2 Matrices

R provides two main ways to organize data in a tabular (i.e. rectangular) object. One of them is a `matrix`—the topic of this chapter—and the other one is a `data.frame`—to be discussed in a subsequent chapter.

Creating a matrix by column-binding vectors

You can build a matrix by **column binding** vectors using the function `cbind()`. In the code below we pass `years` and `amount` to the `cbind()` function, which returns a matrix having the tabular structure that we are looking for: `years` in the first column, and `amounts` in the second column.

```
# 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 pass the vectors, separating them with a comma. Each vector will become a column of the returned matrix.

Creating a matrix by row-binding vectors

You can also build a matrix by **row binding** vectors. For instance, pretend for a minute that we are interested in obtaining a tabular object in which the first row corresponds to `years`, and the second row to `amounts`. To obtain this object we use `rbind()` as follows:

```
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.

8.2.1 What kind of object is a matrix?

In turns out that an R `matrix` is a special type of multi-dimensional atomic object called `array`. Both classes of objects, together with vectors and factors, form the triad of atomic objects. This is illustrated in the following diagram in terms of their number of dimensions.

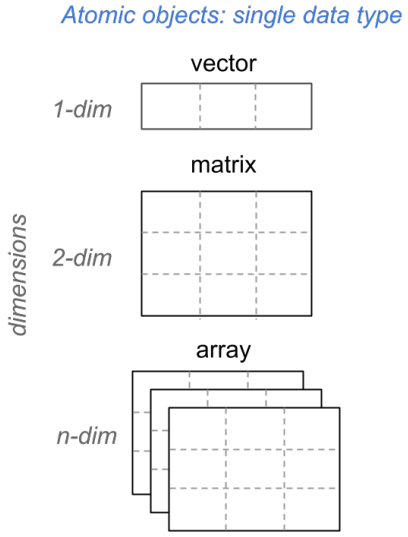


Figure 8.1: Triad of atomic data objects in R.

Personally, I prefer to reserve the term `array` for three or more dimensional arrays. As you can tell from the above diagram, this is how I'm using this term in the book. However, you should always keep in mind that a `matrix` is an `array`. The other way around is not necessarily true: not all arrays are matrices.

8.3 Creating matrices with `matrix()`

The `cbind()` and `rbind()` functions provide a convenient way to create matrices from different input vectors. But the kind of matrices that you can create with them is limited if all you have is just one input vector.

So, in addition to `cbind()` and `rbind()`, R comes with the function `matrix()` which is the workhorse function for creating matrices. Usually, you provide an input vector, and also the number of rows and columns (i.e. the *matrix dimensions*) that the returned matrix should have.

Here is how to use `matrix()` to create the `savings` matrix that we are interested in obtaining:

```
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
```

This is an interesting piece of code. Notice that `years` and `amounts` are combined into a single vector, which is the main input of `matrix()`. The two other arguments correspond to the matrix dimensions: `nrow = 6` tells R that we want to produce a matrix with 6 rows; `ncol = 2` indicates that we want the matrix to have 2 columns.

8.3.1 Column-Major Matrices

When creating a matrix via the function `matrix()`, R takes into consideration three important aspects:

1) the length of the input vector
2) the "size" of the matrix given by its number of rows and columns
3) whether the length of the input vector is a multiple or sub-multiple of the size of the matrix

In the current example, the input vector `c(years, amounts)` has 12 elements. In turn, the size of the desired matrix is given by the multiplication of the number of rows (2) times the number of columns (6), that is:

$$\text{size of matrix} = 6 \times 2 = 12$$

R then compares the length of the input against the size of the matrix. If these numbers are the same, like in this example, R proceeds to split the elements of the input vector into 2 sections or sub-vectors, each one containing 6 elements. Each of these sections will become a column of the output matrix.

In other words, the vector `c(years, amounts)` is split into 2 sub-vectors:

```
# the first sub-vector is:
0  1  2  3  4  5

# the second sub-vector is:
1000.000  1020.000  1040.400  1061.208  1082.432  1104.081
```

The first sub-vector, which corresponds to `years`, becomes the first column. The second sub-vector, which corresponds to `amounts`, becomes the second column. In technical terms we say that R matrices are stored **column-major** because of the mechanism used by R to arrange the elements of an input vector in order to create a matrix.

Mismatch between length of vector and size of matrix

What about those cases in which the length of the input vector does not match the size of the desired matrix? For example, consider the following commands illustrating this type of situation:

```
# examples in which length of input of vector
# doe snot match size of matrix
m1 = matrix(1:3, nrow = 3, ncol = 2)

m2 = matrix(1:3, nrow = 2, ncol = 3)

m3 = matrix(1:12, nrow = 3, ncol = 2)

m4 = matrix(1:4, nrow = 3, ncol = 2)
#> Warning in matrix(1:4, nrow = 3, ncol = 2): data
#> length [4] is not a sub-multiple or multiple of
#> the number of rows [3]

m5 = matrix(1:8, nrow = 2, ncol = 3)
#> Warning in matrix(1:8, nrow = 2, ncol = 3): data
#> length [8] is not a sub-multiple or multiple of
#> the number of columns [3]
```

In matrices `m1` and `m2` the input vector `1:3` is a sub-multiple of the size of the matrix 6.

In matrix `m3` the input vector `1:12` is longer than the size of matrix: 6. Although the entire length of the vector, 12, is a multiple of the size 6.

In matrices `m4` and `m5`, all the input vectors have lengths that are neither a multiple or sub-multiple of the size of the returned matrix.

When the length of the input vector does not match the size of the desired matrix, R applies its recycling rules. Let's pay attention to `m1`:

```
m1
#>      [,1] [,2]
#> [1,]    1    1
#> [2,]    2    2
#> [3,]    3    3
```

Note how the values of the input vector `1:3` are recycled to form the columns of

`m1`. The values appear in the first column, but they also appear in the second column after being recycled.

In contrast, matrix `m3` does not use all the elements in the input vector `1:12`. Instead, only the first six values are retained.

As for the matrices `m4` and `m5`, they all have an input vector whose length is neither a multiple or sub-multiple of the size of the matrix. R will also apply its recycling rules, but it will also display a warning message letting us know that the length of the input vector is not a multiple or sub-multiple of either the number of rows or the number of columns.

8.3.2 Giving names to rows and columns

Often, you may need to provide names for either the rows and/or the columns. R comes with the functions `rownames()` and `colnames()` that can be used to assign names for the rows and columns, for example:

```
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
```

8.3.3 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

account	year	amount
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`.

Recall that matrices are **atomic** objects. Usually, you provide an input vector containing the elements to be arranged into a rectangular array with a certain number of rows and columns. Because vectors are atomic, this property is "inherited" to the returned matrix.

It turns out that you can use other classes of data objects, not necessarily atomic, for creating a matrix. If the input object is non-atomic, R will coerce it into a vector, making the input an atomic object.

So either way, whether you provide an atomic input or a non-atomic input, to any of the matrix-creation functions, R will always produce an atomic output.

This is the reason why the below command produces a `character` matrix:

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

The three input vectors are coerced into a single vector of `character` data type, causing the `investments` matrix to be of type `character`.

Chapter 9

Lists

In this chapter, you will learn about R lists, the most generic type of data container in R. Here's a summary of the main features of R lists:

- Lists are the most general class of data container
- Like vectors, lists group data into a one-dimensional set
- Unlike vectors, lists can store all kinds of objects
- Lists can be of any length
- Elements of a list can be named, or not

9.1 Creating Lists

The typical way to create a list is with the function `list()`. This function creates a list the same way `c()` creates a vector. Let's start with a simple example creating three numeric vectors of same length, that we then use to store them in a list:

```
vec1 <- 1:3
vec2 <- 4:6
vec3 <- 7:9

# list with unnamed elements
lis <- list(vec1, vec2, vec3)
lis
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] 4 5 6
#>
```

```
#> [[3]]
#> [1] 7 8 9
```

Note how the contents of a list with unnamed elements are displayed: there is a set of double brackets with an index indicating the position of each element, and below each double bracket the corresponding vector is printed.

For illustration purposes, we could visualize the three input vectors and the list with the following conceptual diagram.

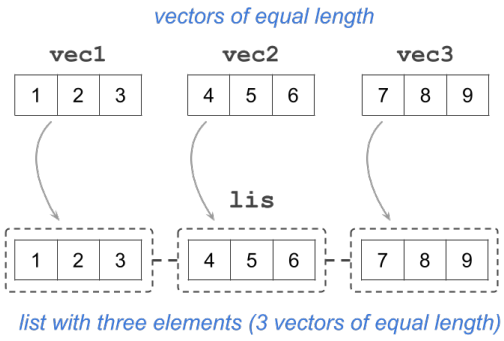


Figure 9.1: A list containing three unnamed elements (numeric vectors of length 3)

Our intention with the depicted list as a set of discontinuous cells is to convey the idea that a list is also a one-dimensional vector, albeit a very special type of vector: a **non-atomic vector**. This means that each element of a list can be any kind of object.

In the same way you can give names to elements of a vector, you can also give names to elements of a list:

```
# list with named elements
lis <- list("vec1" = vec1, "vec2" = vec2, "vec3" = vec3)
lis
#> $vec1
#> [1] 1 2 3
#>
#> $vec2
#> [1] 4 5 6
#>
#> $vec3
#> [1] 7 8 9
```

When you create a list in this form, you can actually omit the quotes of the given names. While this option of naming elements may create a bit of confusion

for beginners and inexperienced users in R, we believe it's not a big deal (based on our experience):

```
# another option for giving names to elements in a list
lis <- list(vec1 = vec1, vec2 = vec2, vec3 = vec3)
lis
#> $vec1
#> [1] 1 2 3
#>
#> $vec2
#> [1] 4 5 6
#>
#> $vec3
#> [1] 7 8 9
```

Observe how the contents of a list with named elements are displayed: this time, instead of the set of double brackets, there is a dollar sign followed by the name of the element, e.g. `$vec1`. Below each name, the corresponding vector is printed.

The conceptual diagram in this case could look like this:

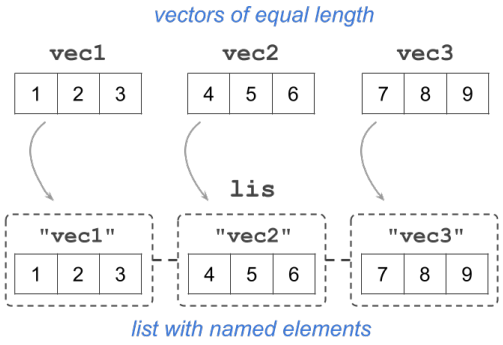


Figure 9.2: A list with named elements (numeric vectors of length 3)

As we just said, the elements of a list can be any kind of R object. For example, here's a list called `lst` that contains a character vector, a numeric matrix, a factor, and another list:

```
lst <- list(
  c("Harry", "Ron", "Hermione"),
  matrix(1:6, nrow = 2, ncol = 3),
  factor(c("yes", "no", "no", "no", "yes")),
  list("Harry", "Ron", "Hermione")
)
```

```
lst
#> [[1]]
#> [1] "Harry"    "Ron"      "Hermione"
#>
#> [[2]]
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> [[3]]
#> [1] yes no  no  no  yes
#> Levels: no yes
#>
#> [[4]]
#> [[4]][[1]]
#> [1] "Harry"
#>
#> [[4]][[2]]
#> [1] "Ron"
#>
#> [[4]][[3]]
#> [1] "Hermione"
```

Whenever possible, we strongly recommend giving names to the elements of a list. Not only this makes it easy to identify one element from the others, but it also gives you more flexibility to rearrange the contents of the list without having to worry about the exact order or position they occupy.

```
# whenever possible, give names to elements in a list
lst <- list(
  first = c("Harry", "Ron", "Hermione"),
  second = matrix(1:6, nrow = 2, ncol = 3),
  third = factor(c("yes", "no", "no", "no", "yes")),
  fourth = list("Harry", "Ron", "Hermione")
)
```

9.2 Manipulating Lists

To manipulate the elements of a list you can use bracket notation. Because a list is a vector, you can use single brackets (e.g. `lis[1]`) as well as double brackets (e.g. `lis[[1]]`).

9.2.1 Single brackets

Just like any other vector, and any other data object in R, you can use single brackets on a list. For example, consider the unnamed version of a list, and the

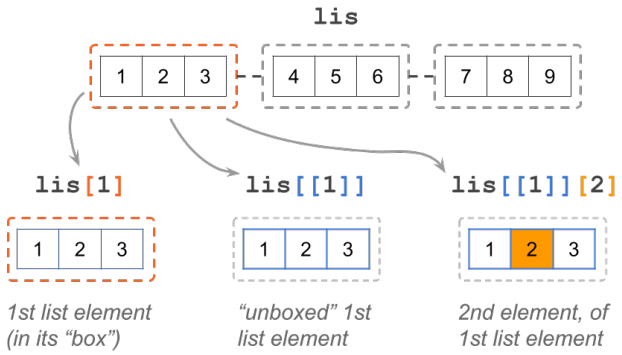


Figure 9.3: Bracket notation with lists

use of single brackets with index `1` inside them:

```
# list with unnamed elements
lis <- list(vec1, vec2, vec3)

lis[1]
#> [[1]]
#> [1] 1 2 3
```

What a single bracket does, is give you access to the "container" of the specified element but without "unboxing" its contents. This is reflected by the way in which the output is displayed: note the double bracket `[[1]]` in the first line, and then `[1] 1 2 3` in the second line.

In other words, `lis[1]` gives you the first element of the list, which contains a vector, but it does not give you direct access to the vector itself. Put another way, `lis[1]` lets you see that the first element of the list is a vector, but this vector is still *inside* its "box".

9.2.2 Double Brackets

In addition to single brackets, lists also accept double brackets: e.g. `lis[[1]]`

```
lis[[1]]
#> [1] 1 2 3
```

Double brackets are used when you want to get access to the content of the list's elements. Notice the output of the previous command: now there are no double brackets, just the output of the vector in the first position. Think of this command as "unboxing" the object of the first element in `lis`.

What if you want to manipulate the elements of vector `vec1` or `vec2`? Use

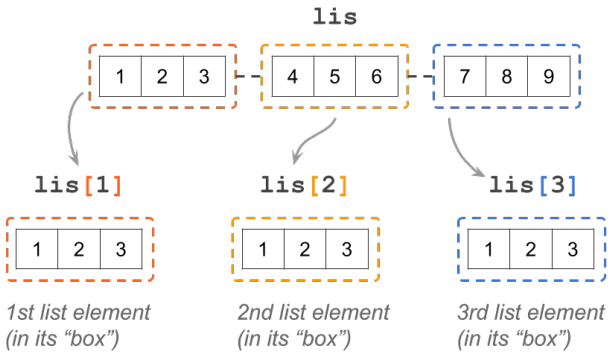


Figure 9.4: Use single brackets to select an element

double brackets followed by single brackets

```
# second index of first list's element
lis[[1]][2]
#> [1] 2

# first index of second list's element
lis[[2]][1]
#> [1] 4
```

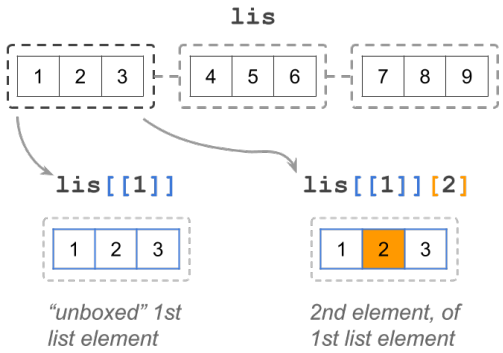


Figure 9.5: Use double brackets to extract an element

9.2.3 Dollar signs

R lists—and data frames—follow an optional second system of notation for extracting **named elements** using the dollar sign **$**

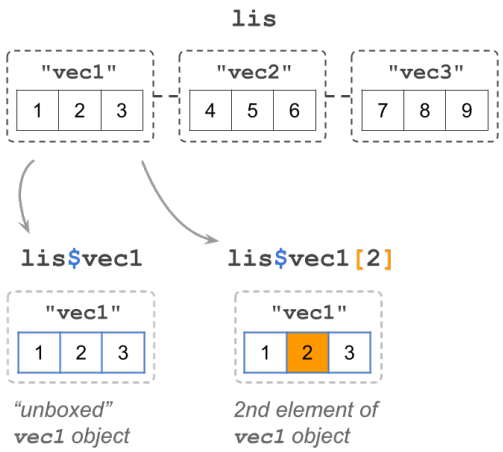


Figure 9.6: Dollar notation with lists

Let's use the named version of `lis`:

```
# list with named elements
lis <- list("vec1" = vec1, "vec2" = vec2, "vec3" = vec3)

lis$vec1
#> [1] 1 2 3
```

The dollar sign `$` notation works for selecting **named elements** in a list. Notice the output of the above command: `lis$vec1` gives you vector `1 2 3`. In other words, dollar notation "unboxes" the object that is associated to the specified name.

9.2.4 Adding new elements

From time to time, you will want to add one or more elements to an existing list. For instance, consider a list `lst` with two elements:

```
lst <- list(1:3, c('A', 'B', 'C'))

lst
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "A" "B" "C"
```

Say you want to add a logical vector as a third element to `lst`. One option to do this is with double brackets, specifying a new index position to which you

assign the new element:

```
lst[[3]] <- c(TRUE, FALSE, TRUE, FALSE)

lst
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "A" "B" "C"
#>
#> [[3]]
#> [1]  TRUE FALSE  TRUE FALSE
```

Another option is to use the dollar operator by giving a new name to which you assign the new element. Even though the previous elements in `lst`are unnamed, the new added element will have an associated label:

```
lst$new_elem <- 'nuevo'

lst
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "A" "B" "C"
#>
#> [[3]]
#> [1]  TRUE FALSE  TRUE FALSE
#>
#> $new_elem
#> [1] "nuevo"
```

9.2.5 Removing elements

Just like you will want to add new elements in a list, you will also find occasions in which you need to remove one or more elements. Take the previous list `lst` with four elements, and say you want to remove the third element (containing the logical vector)

```
lst
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "A" "B" "C"
#>
```

```
#> [[3]]
#> [1]  TRUE FALSE  TRUE FALSE
#>
#> $new_elem
#> [1] "nuevo"
```

To remove the third element, which is unnamed, you use double brackets and assign a value `NULL` to that position:

```
lst[[3]] <- NULL
lst
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "A" "B" "C"
#>
#> $new_elem
#> [1] "nuevo"
```

As for those named elements, such as `lst$new_elem`, you do the same and assign a `NULL` value, but this time using dollar notation:

```
lst$new_elem <- NULL
lst
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "A" "B" "C"
```

9.3 Exercises

1) How would you create a list with your first name, middle name, and last name? For example, something like:

```
$first
[1] "Gaston"

$middle
NULL

$last
[1] "Sanchez"
```

2) Consider an R list `student` containing the following elements:

```
$name
[1] "Luke Skywalker"

$gpa
[1] 3.8

$major_minor
              major               minor
     "jedi studies" "galactic policies"

$grades
        course letter
1 light-sabers      B
2    force-101      A
3  jedi-poetry     C+
```

a) Which of the following commands gives you the values of column `letter` (i.e. column of element `grades`)? Mark all valid options.
   i) `student$grades[ ,letter]`
   ii) `student$grades[ ,2]`
   iii) `student[[4]][ ,2]`
   iv) `student[4][ ,2]`

b) Which of the following commands gives you the length of `major_minor`? Mark all valid options.
   i) `length(student[3])`
   ii) `length(student$major_minor)`
   iii) `length(student[c(FALSE, FALSE, TRUE, FALSE)])`
   iv) `length(student[[3]])`

c) Which of the following commands gives you the number of rows in `grades`? Mark all the valid options.
   i) `nrow(student[["grades"]])`
   ii) `nrow(student[4])`
   iii) `nrow(student$grades)`
   iv) `nrow(student[ ,c("course", "letter")])`

d) Which of the following commands gives you the value in `name`? Mark all the valid options.
   i) `student(1)`
   ii) `student$name`
   iii) `student[[name]]`
   iv) `student[[-c(1,2,3)]]`

3) I have created an R object called `obj`, which looks like this when printed on the console:

```
$exams
midterm   final
    100      90
```

```
Levels: 90 100

$grades
midterm   final
   "A+"    "A-"

$hws
  topic points
1     x    151
2     y    154
3     z    159
```

Indicate whether each of the following statements is True or False.

a) length of `obj$hws` is 3

b) data type of `obj$exams` could be double (i.e. real)

c) `obj$grades` cannot be a factor

d) `obj$hws` could be a data frame

e) `obj$hws` could be a matrix

f) column `points` in `obj$hws` could be a factor

4) Consider an R list `apprentice` containing the following elements:

```
$name
[1] "Anakin Skywalker"

$gpa
[1] 4

$major_minor
             major1              major2
     "jedi studies"      "sith studies"
              minor
"galactic policies"

$grades
        course score
1    force-101   9.3
2    podracing  10.0
3 light-sabers   8.5
```

Without running the commands in R, write down what will appear at the console when such commands are executed:

a) `length(apprentice$major_minor)`

b) `apprentice$gpa < 2.5`

c) `names(apprentice$major_minor)`

d) `rep(apprentice$grades[[2]][2], apprentice$gpa)`

e) `apprentice$grades[order(apprentice$grades$score), ]`

Chapter 10

Data Frames

The most common format/structure for a data set is a tabular format: with rows and columns (like a spreadsheet). When your data is in this shape, most of the time you will work with R **data frames** (or similar rectangular structures like a `"matrix"`, `"table"`, `"tibble"`, etc).

Learning how to manipulate data frames is among the most important *data computing* skills in R. Nowadays, there are two primary approaches for manipulating data frames. One is what I call the "traditional" or "classic" approach which is what I present in this chapter. The other is the "tidy" approach which you can think of as a modern version based on the *tidy data* framework mainly developed by Hadley Wickham. We leave the discussion of this alternative approach for later.

To make the most of the content covered in the next sections, I am assuming that you are familiar with the rest of data objects covered in the previous chapters of part "II Data Objects in R".

10.1 R Data Frames

A data frame is a special type of R list. In most cases, a data frame is internally stored as a list of vectors or factors, in which each vector (or factor) corresponds to a column. This implies that columns in a data frame are typically atomic structures: all elements in a given column are of the same data type. However, since a data frame is a list, you can technically have any kind of object as a column. In practice, though, having data frames with columns that are not vectors or factors is something that does not make much sense.

From the data manipulation point of view, data frames behave like a hybrid object. On one hand, they are lists and can be manipulated like any other list using double brackets `dat[[ ]]` and dollar operator `dat$name`. On the other

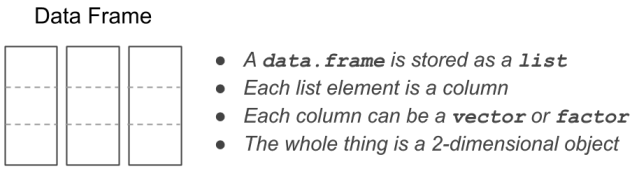


Figure 10.1: Abstract view of a data.frame

hand, because data frames are designed as tabular or 2-dimensional objects, they also behave like two-dimensional arrays or matrices, admitting bracket notation `dat[ , ]`. For these reasons, there is a wide array of functions that allows you to manipulate data frames in very convenient ways. But to the inexperienced user, all these functions may feel overwhelming.

10.2 Inspecting data frames

One of the basic tasks when working with data frames involves inspecting its contents. Specially in the early stages of data exploration, when dealing for the first time with a new data frame, you will need to inspect things like its overall structure, which includes its dimensions (number of rows and columns), the data types of its columns, the names of columns and rows, and also be able to take a peak to some of its first or last rows, and usually obtain a summary of each column.

Let's see an example with one of the built-in data frames in R: `mtcars`. Just a few rows and columns of `mtcars` are displayed below:

```
#>                    mpg cyl disp  hp drat    wt
#> Mazda RX4         21.0   6  160 110 3.90 2.620
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875
#> Datsun 710        22.8   4  108  93 3.85 2.320
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440
```

The main function to explore the *structure* of not just a data frame, but of any kind of object, is `str()`. When applied to data frames, `str()` returns a report of the dimensions of the data frame, a list with the name of all the variables, and their data types (e.g. `chr` character, `num` real, etc).

```
str(mtcars, vec.len = 1)
#> 'data.frame':    32 obs. of  11 variables:
#>  $ mpg : num  21 21 ...
#>  $ cyl : num  6 6 ...
#>  $ disp: num  160 160 ...
#>  $ hp  : num  110 110 ...
```

```
#>  $ drat: num  3.9 3.9 ...
#>  $ wt  : num  2.62 ...
#>  $ qsec: num  16.5 ...
#>  $ vs  : num  0 0 ...
#>  $ am  : num  1 1 ...
#>  $ gear: num  4 4 ...
#>  $ carb: num  4 4 ...
```

The argument `vec.len = 1` is optional but we like to use it because it indicates that just the first elements in each column should be displayed. Observe the output returned by `str()`. The first line tells us that `mtcars` is an object of class `'data.frame'` with 32 observations (rows) and 11 variables (columns). Then, the set of 11 variables is listed below, each line starting with the dollar `$` operator, followed by the name of the variable, followed by a colon `:`, the data mode (all numeric `num` variables in this case), and then a couple of values in each variable.

It is specially useful to check the data type of each column in order to catch potential issues and avoid disastrous consequences or bugs in subsequent stages.

Here's a list of useful functions to inspect a data frame:

- `str()`: overall structure
- `head()`: first rows
- `tail()`: last rows
- `summary()`: descriptive statistics
- `dim()`: dimensions
- `nrow()`: number of rows
- `ncol()`: number of columns
- `names()`: names of list elements (i.e. column names)
- `colnames()`: column names
- `rownames()`: row names
- `dimnames()`: list with column and row names

On a technical side, we should mention that a data frame is a list with special attributes: an attribute `names` for column names, an attribute `row.names` for row names, and of course its attribute `class`:

```
attributes(mtcars)
#> $names
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"
#>  [7] "qsec" "vs"   "am"   "gear" "carb"
#>
#> $row.names
#>  [1] "Mazda RX4"           "Mazda RX4 Wag"
#>  [3] "Datsun 710"          "Hornet 4 Drive"
#>  [5] "Hornet Sportabout"   "Valiant"
#>  [7] "Duster 360"          "Merc 240D"
```

```
#>  [9] "Merc 230"            "Merc 280"
#> [11] "Merc 280C"           "Merc 450SE"
#> [13] "Merc 450SL"          "Merc 450SLC"
#> [15] "Cadillac Fleetwood"  "Lincoln Continental"
#> [17] "Chrysler Imperial"   "Fiat 128"
#> [19] "Honda Civic"         "Toyota Corolla"
#> [21] "Toyota Corona"       "Dodge Challenger"
#> [23] "AMC Javelin"         "Camaro Z28"
#> [25] "Pontiac Firebird"    "Fiat X1-9"
#> [27] "Porsche 914-2"       "Lotus Europa"
#> [29] "Ford Pantera L"      "Ferrari Dino"
#> [31] "Maserati Bora"       "Volvo 142E"
#>
#> $class
#> [1] "data.frame"
```

10.3 Creating data frames

Most of the (raw) data tables you will be working with will already be in some data file. However, from time to time you will face the need to create some sort of data table in R. In these situations, you will likely have to create such table with a data frame. So let's look at various ways to "manually"" create a data frame.

Option 1: The primary option to build a data frame is with `data.frame()`. You pass a series of vectors (or factors), of the same length, separated by commas. Each vector (or factor) will become a column in the generated data frame. Preferably, give names to each column like in the example below:

```
dat <- data.frame(
  name = c('Anakin', 'Padme', 'Luke', 'Leia'),
  gender = c('male', 'female', 'male', 'female'),
  height = c(1.88, 1.65, 1.72, 1.50),
  weight = c(84, 45, 77, 49)
)

dat
#>     name gender height weight
#> 1 Anakin   male   1.88     84
#> 2  Padme female   1.65     45
#> 3   Luke   male   1.72     77
#> 4   Leia female   1.50     49
```

Option 2: Another way to create data frames is with a `list` containing vectors or factors (of the same length), which you then convert into a data frame with

`data.frame()`:

```
# another way to create a basic data frame
lst <- list(
  name = c('Anakin', 'Padme', 'Luke', 'Leia'),
  gender = c('male', 'female', 'male', 'female'),
  height = c(1.88, 1.65, 1.72, 1.50),
  weight = c(84, 45, 77, 49)
)

tbl <- data.frame(lst)

tbl
#>     name gender height weight
#> 1 Anakin   male   1.88     84
#> 2  Padme female   1.65     45
#> 3   Luke   male   1.72     77
#> 4   Leia female   1.50     49
```

Remember that a `data.frame` is nothing more than a `list`. So as long as the elements in the list (vectors or factors) are of the same length, we can simply convert the list into a data frame.

Keep in mind that in old versions of R (3.1.0 or older), `data.frame()` used to convert character vectors into factors. You can always check the data type of each column in a data frame with `str()`:

```
str(tbl)
#> 'data.frame':    4 obs. of  4 variables:
#>  $ name  : chr  "Anakin" "Padme" "Luke" "Leia"
#>  $ gender: chr  "male" "female" "male" "female"
#>  $ height: num  1.88 1.65 1.72 1.5
#>  $ weight: num  84 45 77 49
```

To prevent `data.frame()` from converting strings into factors, you must use the argument `stringsAsFactors = FALSE`

```
# strings as strings (not as factors)
dat <- data.frame(
  name = c('Anakin', 'Padme', 'Luke', 'Leia'),
  gender = c('male', 'female', 'male', 'female'),
  height = c(1.88, 1.65, 1.72, 1.50),
  weight = c(84, 45, 77, 49),
  stringsAsFactors = FALSE
)

str(dat)
#> 'data.frame':    4 obs. of  4 variables:
```

```
#>  $ name  : chr  "Anakin" "Padme" "Luke" "Leia"
#>  $ gender: chr  "male" "female" "male" "female"
#>  $ height: num  1.88 1.65 1.72 1.5
#>  $ weight: num  84 45 77 49
```

10.4 Basic Operations with Data Frames

Now that you have seen some ways to create data frames, let's discuss a number of basic manipulations of data frames. We will show you examples of various operations, and then you'll have the chance to put them in practice with some exercises listed at the end of the chapter.

- Selecting table elements:
  - select a given cell
  - select a set of cells
  - select a given row
  - select a set of rows
  - select a given column
  - select a set of columns
- Adding a new column
- Deleting a new column
- Renaming a column
- Moving a column
- Transforming a column

Let's say you have a data frame `dat` with the following content:

```
dat <- data.frame(
  name = c('Leia', 'Luke', 'Han'),
  gender = c('female', 'male', 'male'),
  height = c(1.50, 1.72, 1.80),
  jedi = c(FALSE, TRUE, FALSE),
  stringsAsFactors = FALSE
)

dat
  name gender height  jedi
1 Leia female   1.50 FALSE
2 Luke   male   1.72  TRUE
3  Han   male   1.80 FALSE
```

10.4.1 Selecting elements

The data frame `dat` is a 2-dimensional object: the 1st dimension corresponds to the rows, while the 2nd dimension corresponds to the columns. Because `dat`

has two dimensions, the bracket notation involves working with data frames in this form: `dat[ , ]`.

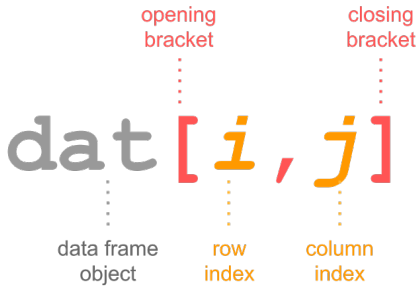


Figure 10.2: Bracket notation in data frames

In other words, you have to specify values inside the brackets for the 1st index, and the 2nd index: `dat[index1, index2]`.

Selecting cells

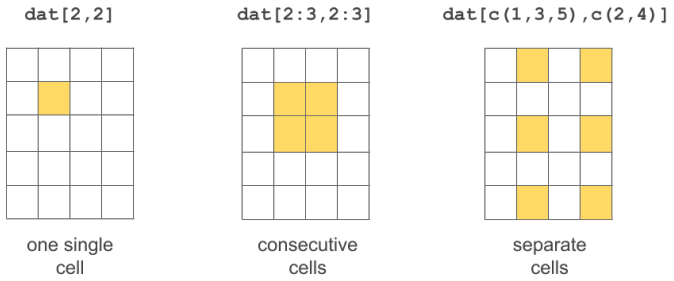


Figure 10.3: Several ways to select cells

```
# select value in row 1 and column 1
dat[1,1]
#> [1] "Leia"

# select value in row 2 and column 3
dat[2,3]
#> [1] 1.72

# select values in these cells
dat[1:2,3:4]
#>   height  jedi
```

```
#> 1   1.50 FALSE
#> 2   1.72  TRUE
```

It is also possible to exclude certain rows-and-columns by passing negative numeric indices:

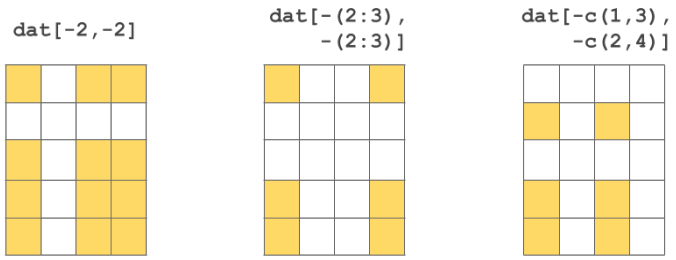


Figure 10.4: Several ways to exclude cells

Selecting rows

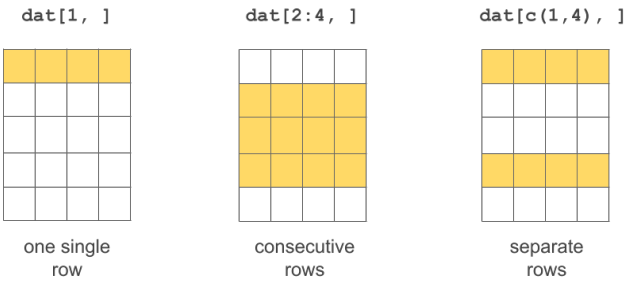


Figure 10.5: Several ways to select rows

If no value is specified for `index1` then all rows are included. Likewise, if no value is specified for `index2` then all columns are included.

```
# selecting first row
dat[1, ]
#>   name gender height  jedi
#> 1 Leia female    1.5 FALSE

# selecting third row
dat[3, ]
#>   name gender height  jedi
#> 3  Han   male    1.8 FALSE
```

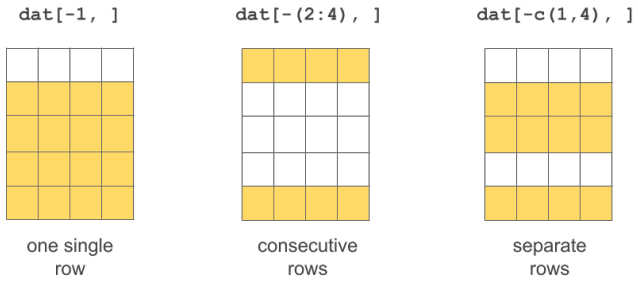


Figure 10.6: Several ways to exclude rows

Selecting columns

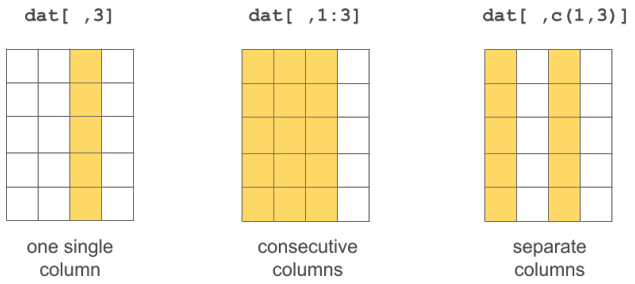


Figure 10.7: Several ways to select columns

```
# selecting second column
dat[ ,2]
#> [1] "female" "male"   "male"

# selecting columns 2 to 4
dat[ ,2:4]
#>   gender height  jedi
#> 1 female   1.50 FALSE
#> 2   male   1.72  TRUE
#> 3   male   1.80 FALSE
```

More Options to Access Columns

The dollar sign also works for selecting a column of a data frame using its name

```
mtcars$mpg
#>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8
#> [10] 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7 32.4
```

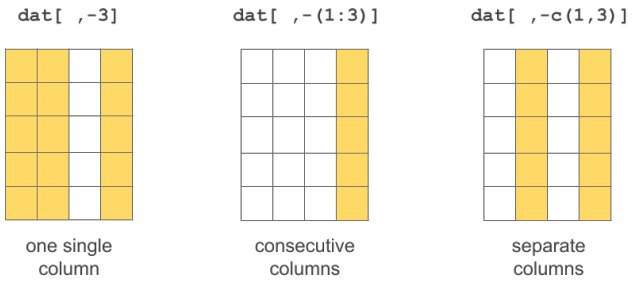


Figure 10.8: Several ways to exclude columns

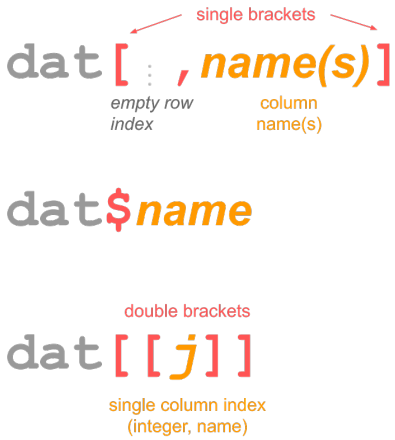


Figure 10.9: Other options to select columns of a data frame

```
#> [19] 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0
#> [28] 30.4 15.8 19.7 15.0 21.4
```

You don't need to use quote marks, but you can if you want. The following calls are equivalent.

```
mtcars$'mpg'
mtcars$"mpg"
mtcars$`mpg`
```

10.4.2 Adding a column

Perhaps the simplest way to add a column is with the dollar operator `$`. You just need to give a name for the new column, and assign a vector (or factor):

```
# adding a column
dat$new_column <- c('a', 'e', 'i')
dat
#>   name gender height  jedi new_column
#> 1 Leia female   1.50 FALSE          a
#> 2 Luke   male   1.72  TRUE          e
#> 3  Han   male   1.80 FALSE          i
```

Another way to add a column is with the *column binding* function `cbind()`:

```
# vector of weights
weight <- c(49, 77, 85)

# adding weights to dat
dat <- cbind(dat, weight)
dat
#>   name gender height  jedi new_column weight
#> 1 Leia female   1.50 FALSE          a     49
#> 2 Luke   male   1.72  TRUE          e     77
#> 3  Han   male   1.80 FALSE          i     85
```

10.4.3 Deleting a column

The inverse operation of adding a column consists of **deleting** a column. This is possible with the `$` dollar operator. For instance, say you want to remove the column `new_column`. Use the `$` operator to select this column, and assign it the value `NULL` (think of this as *NULLifying* a column):

```
# deleting a column
dat$new_column <- NULL
dat
#>   name gender height  jedi weight
#> 1 Leia female   1.50 FALSE     49
```

```
#> 2 Luke   male   1.72  TRUE     77
#> 3  Han   male   1.80 FALSE     85
```

10.4.4 Renaming a column

What if you want to rename a column? There are various options to do this. One way is by changing the column `names` attribute:

```
# attributes
attributes(dat)
#> $names
#> [1] "name"   "gender" "height" "jedi"   "weight"
#>
#> $row.names
#> [1] 1 2 3
#>
#> $class
#> [1] "data.frame"
```

which is more commonly accessed with the `names()` function:

```
# column names
names(dat)
#> [1] "name"   "gender" "height" "jedi"   "weight"
```

Notice that `dat` has a list of attributes. The element `names` is the vector of column names.

You can directly modify the vector of `names`; for example let's change `gender` to `sex`:

```
# changing rookie to rooky
attributes(dat)$names[2] <- "sex"

# display column names
names(dat)
#> [1] "name"   "sex"    "height" "jedi"   "weight"
```

By the way: this approach of changing the name of a variable is very low level, and probably unfamiliar to most useRs.

10.4.5 Moving a column

A more challenging operation is when you want to move a column to a different position. What if you want to move `salary` to the last position (last column)? One option is to create a vector of column names in the desired order, and then use this vector (for the index of columns) to reassign the data frame like this:

```
reordered_names <- c("name", "jedi", "height", "weight", "sex")
dat <- dat[ ,reordered_names]
dat
#>   name  jedi height weight    sex
#> 1 Leia FALSE   1.50     49 female
#> 2 Luke  TRUE   1.72     77   male
#> 3  Han FALSE   1.80     85   male
```

10.4.6 Transforming a column

A more common operation than deleting or moving a column, is to transform the values in a column. This can be easily accomplished with the `$` operator. For instance, let's say that we want to transform `height` from meters to centimeters:

```
# converting height to centimeters
dat$height <- dat$height * 100
dat
#>   name  jedi height weight    sex
#> 1 Leia FALSE    150     49 female
#> 2 Luke  TRUE    172     77   male
#> 3  Han FALSE    180     85   male
```

Likewise, instead of using the `$` operator, you can refer to the column using bracket notation. Here's how to transform weight from kilograms to pounds (1 kg = 2.20462 pounds):

```
# weight into pounds
dat[ ,"weight"] <- dat[ ,"weight"] * 2.20462
dat
#>   name  jedi height   weight    sex
#> 1 Leia FALSE    150 108.0264 female
#> 2 Luke  TRUE    172 169.7557   male
#> 3  Han FALSE    180 187.3927   male
```

There is also the `transform()` function which transform values *interactively*, that is, temporarily:

```
# transform weight to kgs
transform(dat, weight = weight / 0.453592)
#>   name  jedi height   weight    sex
#> 1 Leia FALSE    150 238.1576 female
#> 2 Luke  TRUE    172 374.2476   male
#> 3  Han FALSE    180 413.1305   male
```

`transform()` does its job of modifying the values of `weight` but only temporarily; if you inspect `dat` you'll see what this means:

```
# did weight really change?
dat
#>   name  jedi height   weight    sex
#> 1 Leia FALSE    150 108.0264 female
#> 2 Luke  TRUE    172 169.7557   male
#> 3  Han FALSE    180 187.3927   male
```

To make the changes permanent with `transform()`, you need to reassign them to the data frame:

```
# transform weight to inches (permanently)
dat <- transform(dat, weight = weight / 0.453592)
dat
#>   name  jedi height   weight    sex
#> 1 Leia FALSE    150 238.1576 female
#> 2 Luke  TRUE    172 374.2476   male
#> 3  Han FALSE    180 413.1305   male
```

10.5 Exercises

1) Consider the following data frame `df`:

```
      first      last  gender  born         spell
1     Harry    Potter    male  1980  sectumsempra
2  Hermione   Granger  female  1979     alohomora
3       Ron   Weasley    male  1980    riddikulus
4      Luna  Lovegood  female  1981       episkey
```

a) What commands will fail to return the data of individuals born in 1980?

i) `df[c(TRUE, FALSE, TRUE, FALSE), ]`

ii) `df[df[,4] == 1980, ]`

iii) `df[df$born == 1980]`

iv) `df[df$born == 1980, ]`

v) `df[ ,df$born == 1980]`

b) Select the command that does **not** provide information about the data frame `df`:

i) `head(df)`

ii) `str(df)`

iii) `tail(df)`

iv) `rm(df)`

v) `summary(df)`

c) Your friend is trying to display the first three rows on columns 1 (`first`) and 2 (`last`), by unsuccessfully using the following command. Why does the command print all columns?

```
df[1:3, 1 & 2]
#>      first    last gender born        spell
#> 1    Harry  Potter   male 1980 sectumsempra
#> 2 Hermione Granger female 1979    alohomora
#> 3      Ron Weasley   male 1980   riddikulus
```

d) Write a command that would correctly display the first two columns.

e) Write a command that would give you the following data from `df`.

```
        spell    first
1 sectumsempra    Harry
2    alohomora Hermione
3   riddikulus      Ron
4      episkey     Luna
```

2) Consider the following data frame `dat`

```
       first        last   gender      title
1        Jon        Snow     male       lord
2       Arya       Stark   female   princess
3     Tyrion   Lannister     male     master
4   Daenerys   Targaryen   female   khaleesi
5       Yara     Greyjoy   female   princess
    gpa
1   2.8
2   3.5
3   2.9
4   3.7
5    NA
```

One of your friends wrote the following R code. Help your friend find all the errors and **explain what's wrong**.

```
# value of 'first' associated to maximum 'gpa'
max_gpa <- max(dat$gpa, na.rm = TRUE)
which_max_gpa <- dat$gpa = max_gpa
dat$first(which_max_gpa)

# gpa of title lord
dat$gpa[dat[ ,title] = "lord"]

# median gpa (of each gender)
which_males <- dat$gender == 'male'
```

```
which_females <- dat$gender == 'female'
median_females <- median(dat$gpa[which_males])
median_males <- median(dat$gpa[which_males])
```

Part III

Programming Basics

Chapter 11

Intro to Functions

R comes with many functions and packages that let us perform a wide variety of tasks, and so far we've been using a number of them. In fact, most of the things we do in R is by calling some function. Sometimes, however, there is no function to do what we want to achieve. When this is the case, we may very well want to write our own functions.

In this chapter we'll describe how to start writing small and simple functions. We are going to start covering the "tip of the iceberg", and in the following chapters we will continue discussing more aspects about writing functions, and describing how R works when you invoke (call) a function.

11.1 Motivation

We've used the formula of **future value**, given below, which is useful to answer questions like: If you deposit $1000 into a savings account that pays an annual interest of 2%, how much will you have at the end of year 10?

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

- 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)

R has a large number of functions—e.g. `sqrt()`, `log()`, `mean()`, `sd()`, `exp()`, etc—but it does not have a built-in function to compute future value.

Wouldn't it be nice to have a `future_value()` function—or an `fv()` function—that you could call in R? Perhaps something like:

```
future_value(present = 1000, rate = 0.02, year = 10)
```

Let's create such a function!

11.2 Writing a Simple Function

This won't always be the case, but in our current example we have a specific mathematical formula to work with (which makes things a lot easier):

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

Like other programming languages that can be used for scientific computations, we can take advantage of the syntax in R to write an expression that is almost identical to the algebraic formulation:

```
fv = pv * (1 + r)^t
```

We will use this simple line of code as our starting point for creating a future value function. Here is how to do it "logically" step by step.

Step 1: Start with a concrete example

You should always start with a **small and concrete example**, focusing on writing code that does the job. For example, we could write the following lines:

```
# inputs
pv = 1000
r = 0.02
t = 10

# process
fv = pv * (1 + r)^t

# output
fv
#> [1] 1218.994
```

When I say "small example" I mean working with objects containing just a few values. Here, the objects `pv`, `r`, and `t` are super simple vectors of size 1. Sometimes, though, you may want to start with less simple—yet small—objects containing just a couple of values. That's fine too.

Sometimes you may even need to start not just with one, but with a couple of concrete examples that will help you get a better feeling of what kind of objects, and operations you need to use.

As you get more experience creating and writing functions, you may want to start with a "medium-size" concrete example. Personally, I don't tend to start like this. Instead, I like to take baby-steps, and I also like to take my time, without rushing the coding. You know the old-saying: "measure twice, cut once."

An important part of starting with a concrete example is so that you can identify what the inputs are, what computations or process the inputs will go through, and what the output should be.

Inputs:

- `pv`
- `r`
- `t`

Process:

- `fv = pv * (1 + r)^t`

Output:

- `fv`

Step 2: Make your code more generalizable

After having one (or a few) concrete example(s), the next step is to make your code more generalizable, or if you prefer, to make it more abstract (or at least less concrete).

Instead of working with specific values `pv`, `r`, and `t`, you can give them a more algebraic spirit. For instance, the code below considers "open-ended" inputs without assigning them any values

```
# general inputs (could take "any" values)
pv
r
t

# process
fv = pv * (1 + r)^t

# output
fv
```

Obviously this piece of code is very abstract and not intended to be executed in R; this is just for the sake of conceptual illustration.

Step 3: Encapsulate the code into a function

The next step is to encapsulate your code as a formal function in R. I will show you how to do this in two logical substeps, although keep in mind that in practice you will merge these two substeps into a single one.

The encapsulation process involves placing the "inputs" inside the function `function()`, separating each input with a comma. Formally speaking, the inputs of your functions are known as the **arguments** of the function.

Likewise, the lines of code that correspond to the "process" and "output" are what will become the **body** of the function. Typically, you encapsulate the code of the body by surrounding it with curly braces `{ }`

```
# encapsulating code into a function
function(pv, r, t) {
  fv = pv * (1 + r)^t
  fv
}
```

The other substep typically consists of **assigning a name** to the code of your function. For example, you can give it the name `FV`:

```
# future value function
FV = function(pv, r, t) {
  fv = pv * (1 + r)^t
  fv
}
```

In summary:

- the inputs go inside `function()`, separating each input with a comma
- the processing step and the output are surrounded within curly braces `{ }`
- you typically assign a name to the code of your function

Step 4: Test that the function works

Once the function is created, you test it to make sure that everything works. Very likely you will test your function with the small and concrete example:

```
# test it
FV(1000, 0.02, 10)
#> [1] 1218.994
```

And then you'll keep testing your function with other less simple examples. In this case, because the code we are working with is based on vectors, and uses common functions for vectors, we can further inspect the behavior of the function by providing vectors of various sizes for all the arguments:

```r
# vectorized years
FV(1000, 0.02, 1:5)
#> [1] 1020.000 1040.400 1061.208 1082.432 1104.081
```

```r
# vectorized rates
FV(1000, seq(0.01, 0.02, by = 0.005), 1)
#> [1] 1010 1015 1020
```

```r
# vectorized present values
FV(c(1000, 2000, 3000), 0.02, 1)
#> [1] 1020 2040 3060
```

Notice that the function is vectorized, this is because we are using arithmetic operators (e.g. multiplication, subtraction, division) which are in turn vectorized.

In Summary

- To define a new function in R you use the function `function()`.
- Usually, you specify a name for the function, and then assign `function()` to the chosen name.
- You also need to define optional arguments (i.e. inputs of the function).
- And of course, you must write the code (i.e. the body) so the function does something when you use it.

11.2.1 Arguments with default values

Sometimes it's a good idea to add a default value to one (or more) of the arguments. For example, we could give default values to the arguments in such a way that when the user executes the function without any input, `FV()` returns the value of 100 monetary units invested at a rate of return of 1% for 1 year:

```r
# future value function with default arguments
FV = function(pv = 100, r = 0.01, t = 1) {
  fv = pv * (1 + r)^t
  fv
}

# default execution
FV()
#> [1] 101
```

An interesting side effect of giving default values to the arguments of a function is that you can also call it by specifying arguments in an order different from the order in which the function was created:

```
FV(r = 0.02, t = 3, pv = 1000)
#> [1] 1061.208
```

11.3 Writing Functions for Humans

When writing functions (or coding in general), you should write code not just for the computer, **but also for humans**. While it is true that R doesn't care too much about what names and symbols you use, your code will be used by a human being: either you or someone else. Which means that a human will have to take a look at the code.

Here are some options to make our code more human friendly. We can give the function a more descriptive name such as `future_value()`. Likewise, we can use more descriptive names for the arguments: e.g. `present`, `rate`, and `years`.

```
# future value function
future_value = function(present, rate, years) {
  future = present * (1 + rate)^years
  future
}

# test it
future_value(present = 1000, rate = 0.02, years = 10)
#> [1] 1218.994
```

Even better: whenever possible, as we just said, it's a good idea to give default values to the arguments (i.e. inputs) of the function:

```
# future value function
future_value = function(present = 100, rate = 0.01, years = 1) {
  future = present * (1 + rate)^years
  future
}

future_value()
#> [1] 101
```

11.3.1 Naming Functions

Since we just change the name of the function from `fv()` to `future_value()`, you should also learn about the rules for naming R functions. A function cannot have any name. For a name to be valid, two things must happen:

- the first character must be a letter (either upper or lower case) or the dot `.`
- besides the dot, the only other symbol allowed in a name is the underscore

`_` (as long as it's not used as the first character)

Following the above two principles, below are some valid names that could be used for the future value function:

- `fv()`
- `fv1()`
- `future_value()`
- `future.value()`
- `futureValue()`
- `.fv()`: a function that starts with a dot is a valid name, but the function will be a *hidden* function.

In contrast, here are examples of invalid names:

- `1fv()`: cannot begin with a number
- `_fv()`: cannot begin with an underscore
- `future-value()`: cannot use hyphenated names
- `fv!()`: cannot contain symbols other than the dot and the underscore (not in the 1st character)

11.3.2 Function's Documentation

Part of writing a human-friendly function involves writing its **documentation**, usually providing the following information:

- **title**: short title
- **description**: one or two sentences of what the function does
- **arguments**: short description for each of the arguments
- **output**: description of what the function returns

Once you are happy with the status of your function, include comments for its documentation, for example:

```
# title: future value function
# description: computes future value using compounding interest
# inputs:
# - present: amount for present value
# - rate: annual rate of return (in decimal)
# - years: number of years
# output:
# - computed future value
future_value = function(present = 100, rate = 0.01, years = 1) {
  future = present * (1 + rate)^years
```

```
  future
}
```

Writing documentation for a function seems like a waste of time and energy. Shouldn't a function (with its arguments, body, and output) be self-descriptive? In an ideal world that would be the case, but this rarely happens in practice.

Yes, it does take time to write these comments. And yes, you will be constantly asking yourself the same question: "Do I really need to document this function that I'm just planing to use today, and no one else will ever use?"

Yes!

I'll be the first one to admit that I've created so many functions without writing their documentation. And almost always—sooner or later—I've ended up regretting my laziness for not including the documentation. So do yourself and others (especially your future self) a big favor by including some comments to document your functions.

Enough about this chapter. Although, obviously, we are not done yet with functions. After all, this is a book about programming in R, and there is still a long way to cover about the basics and not so basics of functions.

11.4 Exercises

1) In the second part of the book we have talked about the Future Value (FV), and we have extensively used its simplest version of the FV formula. Let's now consider the "opposite" value: the Present Value which is the current value of a future sum of money or stream of cash flows given a specified rate of return.

Consider the simplest version of the formula to calculate the **Present Value** given by:

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

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

Write a function `present_value()` to compute the Present Value based on the above formula.

2) Write another function to compute the **Future Value**, but this time the output should be a **list** with two elements:

- vector `year` from 0 to provided year
- vector `amount` from amount at year 0, till amount at the provided year

For example, something like this:

```
fv_list(present = 1000, rate = 0.02, year = 3)
$year
[1] 0 1 2 3

$amount
[1] 1000.000 1020.000 1040.400 1061.208
```

3) Write another function to compute the **Future Value**, but this time the output should be a **"table"** with two columns: `year` and `amount`. For example, something like this:

```
fv_table(present = 1000, rate = 0.02, year = 3)
  year   amount
1    0 1000.000
2    1 1020.000
3    2 1040.400
4    3 1061.208
```

Note: by "table" you can use either a `matrix` or a `data.frame`. Even better, try to create two separate functions: 1) `fv_matrix()` that returns a matrix, and 2) `fv_df()` that returns a data frame.

Chapter 12

Expressions

In this chapter you will learn about *R expressions* which is a technical concept that appears everywhere in all R programming structures (e.g. functions, conditionals, loops). This chapter, by the way, is the shortest of the book. But its implications are fundamental to get a solid understanding of programming structures in R.

12.1 R Expressions

Before moving on with more programming structures we must first talk about R **expressions**.

12.1.1 Simple Expressions

So far you've been writing several lines of code in R, most of which have been *simple* expressions such as:

```
deposit = 1000
rate = 0.02
year = 3
```

The expression `deposit = 1000` is an assignment statement because we assign the number `1000` to the name `deposit`. It is also a simple expression. The same can be said about the expressions for `rate` and `year`.

Simple expressions are fairly common but they are not the only ones. It turns out that there is another class of expressions known as compound expressions.

12.1.2 Compound Expressions

R programs are made up of expressions which can be either *simple* expressions or *compound* expressions. Compound expressions consist of simple expressions separated by semicolons or newlines, and grouped within braces.

```
# structure of a compound expression
# with simple expressions separated by semicolons
{expression_1; expression_2; ...; expression_n}

# structure of a compound expression
# with simple expressions separated by newlines
{
  expression_1
  expression_2
  expression_n
}
```

Here's a less abstract example:

```
# simple expressions separated by semicolons
{"first"; 1; 2; 3; "last"}
#> [1] "last"

# simple expressions separated by newlines
{
  "first"
  1
  2
  3
  "last"
}
#> [1] "last"
```

Writing compound expressions like those in the previous example is not something common among R users. Although the expressions are perfectly valid, these examples are very dummy (just for illustration purposes). By the way, I strongly discourage you from grouping multiple expressions with semicolons because it makes it difficult to inspect things.

What does R do with a compound expression? When R encounters a compound expression, it handles everything inside of it as a single unit or a single block of code.

What is the purpose of a compound expression? This type of expressions play an important role but they are typically used together with other programming structures (e.g. functions, conditionals, loops).

12.1.3 Every expression has a value

A fundamental notion about expressions is that **every expression in R has a value**.

Consider this simple expression:

```
a <- 5
```

If I ask you: What is the value of `a`?, you should not have trouble answering this question. You know that `a` has the value 5.

What about this other simple expression:

```
b <- 1:5
```

What is the value of `b`? You know as well that the value of `b` is the numeric sequence given by `1 2 3 4 5`.

Now, let's consider the following compound expression:

```
x <- {5; 10}
```

Note that the entire expression is assigned to `x`. Let me ask you the same question. What is the value of `x`? Is it:

- 5?
- 10?
- 5, 10?

Let's find out the answer by taking a look at `x`:

```
x
#> [1] 10
```

Mmmm, this is interesting. As you can tell, `x` has a single value, and it's not `5` but `10`. Out of curiosity, let's also consider this other compound expression for `y` and examine its value:

```
y <- {
  15
  10
  5
}

y
#> [1] 5
```

Same thing, `y` has a single value, the number `5`, which happens to be the **last statement** inside the expression that was evaluated. This is precisely the essence of an R expression. Every R expression has a value, the value of the last statement that gets evaluated.

To make sure you don't forget it, repeat this mantra:

- Every expression in R has a value: the value of the last evaluated statement.
- Every expression in R has a value: the value of the last evaluated statement.
- Every expression in R has a value: the value of the last evaluated statement.

12.1.4 Assignments within Compound Expressions

It is possible to have assignments within compound expressions. For instance:

```
# simple expressions (made up of assignments) separated by newlines
{
  one <- 1
  pie <- pi
  zee <- "z"
}
```

This compound expression contains three simple expressions, all of which are assignments. Interestingly, when an R expressions contains such assignments, the values of the variables can be used in later expressions. In other words, you can refer later to `one` or `pie` or `zee`:

```
# simple expressions (made up of assignments) separated by newlines
{
  one <- 1
  pie <- pi
  zee <- "z"
}

one
#> [1] 1
pie
#> [1] 3.141593
zee
#> [1] "z"
```

Here's another example:

```
z <- { x = 10 ; y = x^2; x + y }

x
#> [1] 10
y
#> [1] 100
z
#> [1] 110
```

Now that we've introduced the concept of compound expressions, we can move on to next chapter where we introduce conditional structures.

Chapter 13

Conditionals

In the last two chapters you got your feet wet around programming structures. Specifically, you got your first contact with functions, and you also got introduced to the notion of R compound expressions. In this chapter you will learn about another common programming structure known as **conditionals**.

Every programming language comes with a set of structures that allows us to have control over how commands are executed. One of these structures is called **conditionals**, and as its name indicates, they are used to evaluate conditions. Simply put, conditional statements, commonly referred to as **if-else** statements, allow you to decide what to do based on a logical condition.

13.1 Motivation

So far we have extensively used a simple savings-investing scenario in which $1000 are deposited in a savings that pays an annual interest rate of 2%, and the future value formula is used to calculate the amount of money at the end of a certain number of years.

Let's now consider a less simplistic savings scenario.

Say you deposit $1000 into a savings account that gives you 2% annual return. The difference this time is that you will also make contributions of $1000 to this savings account every year. The question is still the same, for example:

> How much money will you have in 3 years?

13.1.1 Future Value of Ordinary Annuity

To make things more specific, consider the following scenario. Imagine you recently applied for a job, they hired you, and today is your first day of work.

So let's take this point in time as time 0, or equivalently the beginning of year 1 in your new job.

During this first year you manage to save $1000, and at the end of year 1 you deposit this sum of money into a savings account that pays 2% interest annually. During your second year of work, you manage again to save $1000, which you add to your savings account at the very end of year 2. The same thing happens during your third year of work: you save $1000 and contribute this amount to your savings account at the end of year 3. This is illustrated in the diagram below, with a generic rate of return:

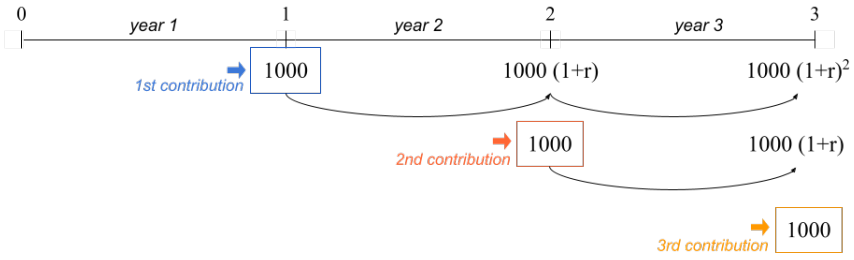


Figure 13.1: Timeline of an ordinary annuity: contributions made at the end of each year.

The balance in your savings account at the end of year 3 is given by:

$$\underbrace{1000(1+0.02)^2}_{\text{1st contribution}} + \underbrace{1000(1+0.02)}_{\text{2nd contrib.}} + \underbrace{1000}_{\text{3rd contrib.}} = 3060.4$$

which in R we can quickly calculate as:

```
1000 * (1.02)^2 + 1000 * (1.02) + 1000
#> [1] 3060.4
```

This example corresponds to what is formally called an **ordinary annuity**. It is an annuity because the same amount of money is contributed every year. It is ordinary because the contributions are made at the **end of each period** (e.g. end of each year).

The formula to calculate the **future value of an ordinary annuity** is given by:

$$\text{FV} = \text{C} \times \left[\frac{(1+r)^t - 1}{r} \right]$$

- FV = future value (how much you'll have)
- C = constant periodic contribution
- r = rate of return (e.g. annual rate of return)
- t = number of periods (e.g. number of years)

Writing code in a less quick and dirty, we may type these commands:

```
# at the end of year 3
contrib = 1000
year = 3
rate = 0.02

# FV of ordinary annuity
contrib * ((1 + rate)^year - 1) / rate
#> [1] 3060.4
```

13.1.2 Future Value of Annuity Due

It turns out that there is another type of annuity known as **annuity due**. The difference between the ordinary annuity and the annuity due is that in the latter the contributions are made at the **beginning of every year**. Here's an example.

Picture the same hypothetical situation. Today is your first day of work which corresponds to time 0, that is, the beginning of year 1 in your new job. In this scenario, though, let's say you already have $1000 at your disposal at this point in time. You go to the bank and deposit this sum of money into a savings account that pays an annual interest rate of 2%.

During this first year you manage to save $1000, and at the beginning of year 2 you make this contribution to your savings account. During your second year of work, you manage again to save $1000, which you add to your savings account at the beginning of year 3. This is illustrated in the diagram below, with a generic rate of return:

The balance in your savings account at the end of year 3 is given by:

$$\underbrace{1000(1+0.02)^3}_{\text{1st contribution}} + \underbrace{1000(1+0.02)^2}_{\text{2nd contrib.}} + \underbrace{1000(1+0.02)}_{\text{3rd contrib.}} = 3121.608$$

which in R we can quickly calculate as:

```
1000 * (1.02)^3 + 1000 * (1 + 0.02)^2 + 1000 * (1 + 0.02)
#> [1] 3121.608
```

The formula to calculate the **future value of an annuity due** is given by:

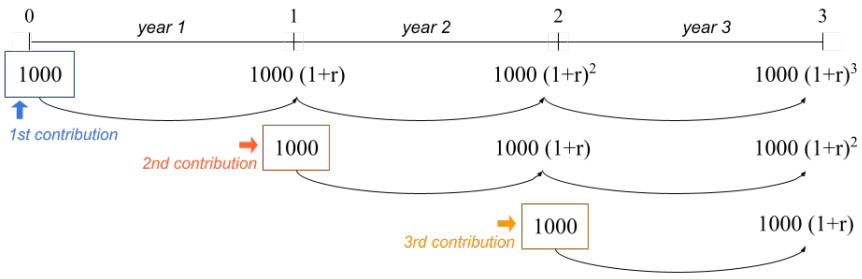


Figure 13.2: Timeline of an annuity due: contributions made at the beginning of each year.

$$\text{FV} = \text{C} \times \left[\frac{(1+r)^t - 1}{r} \right] \times (1+r)$$

- FV = future value (how much you'll have)
- C = constant periodic contribution
- r = rate of return (e.g. annual rate of return)
- t = number of periods (e.g. number of years)

In a less informal way, we can write the following lines of code:

```
# at the end of year 3
contrib = 1000
year = 3
rate = 0.02

# FV of annuity due
contrib * (1 + rate) * ((1 + rate)^year - 1) / rate
#> [1] 3121.608
```

As you know, we can also consider a vectorized option:

```
# over a 3 year period
contrib = 1000
years = 1:3
rate = 0.02

# FV of annuity due
contrib * (1 + rate) * ((1 + rate)^years - 1) / rate
#> [1] 1020.000 2060.400 3121.608
```

13.2 Conditionals

There are two types of annuity:

- **ordinary** (contributions at the end of each period)
- **due** (contributions at the beginning of each period)

What if you want to consider an input `type` that can take two values: `type = "ordinary"` or `type = "due"`?

```
# in 3 years
contrib = 1000
years = 1:3
rate = 0.02
```

Ordinary annuity:

```
# if type == "ordinary"
contrib * ((1 + rate)^years - 1) / rate
#> [1] 1000.0 2020.0 3060.4
```

Annuity due:

```
# if type == "due"
contrib * (1 + rate) * ((1 + rate)^years - 1) / rate
#> [1] 1020.000 2060.400 3121.608
```

13.2.1 If-else conditions

Perhaps the best way for you to get introduced to `if-else` statements is to see one for yourself, so here it is:

```
# ordinary annuity
contrib = 1000
years = 1:3
rate = 0.02
type = "ordinary"

# if-else statement
if (type == "ordinary") {
  fv = contrib * ((1 + rate)^years - 1) / rate
} else {
  fv = contrib * (1 + rate) * ((1 + rate)^years - 1) / rate
}

fv
#> [1] 1000.0 2020.0 3060.4
```

I hope that just by looking at the conditional statement you get the gist of what

is going on. If the type of annuity is the ordinary one (`type == "ordinary"`) then we apply the formula of the FV of ordinary annuity. Otherwise (`else`) we apply the formula of the FV of annuity due.

The same piece of code can be implemented with the other type of annuity

```
# annuity due
contrib = 1000
years = 1:3
rate = 0.02
type = "due"

if (type == "ordinary") {
  fv = contrib * ((1 + rate)^years - 1) / rate
} else {
  fv = contrib * (1 + rate) * ((1 + rate)^years - 1) / rate
}

fv
#> [1] 1020.000 2060.400 3121.608
```

13.2.2 Anatomy of if-else statements

The **if-then-else** statement makes it possible to choose between two (possibly compound) expressions depending on the value of a (logical) condition.

In R (as in many other languages) the if-then-else statement has the following structure:

```
if (condition) {
  # do something
} else {
  # do something else
}
```

In our working example with the two types of annuities, the condition that we are evaluating depends on the value of `type`:

```
if (type == "ordinary") {
  # compute ordinary annuity
} else {
  # compute annuity due
}
```

If `type == "ordinary"`, then we should compute the future value of annuity using its ordinary version. Otherwise, we should compute the future value using the annuity due formula.

Let's dissect the conditional statement

```
type <- "ordinary"

if (type == "ordinary") {
  # compute ord annuity
} else {
  # compute annuity due
}
```

An `if-else` statement always begins with the `if` clause. You basically refer to it as a function, that is, employing parenthesis `if( )`. The thing that goes inside parenthesis corresponds to the logical condition to be evaluated. Then we have the R expression, defined with the first pair of braces `{ }` that contains the code to be executed when the logical condition is true. Next, right after the closing brace of the expression associated to the if-clause, we have the `else` clause. This second clause involves another R expression, the one defined with a second pair of braces. This expression contains the code to be executed when the logical condition is false.

For readability purposes, and to match the syntax used in many other programming languages, when declaring the `if` clause I prefer to leave a blank space before the opening parenthesis: `if (condition)`.

Using the annuity example, let's recap the main parts of a typical if-else statement. In general, this kind of statement consists of the `if` clause and the `else` clause. Only the `if` clause uses parenthesis.

```
type <- "ordinary"
 if-else statement
if (type == "ordinary") {
  # compute ord annuity
} else {
  # compute annuity due
}
```

Inside the `if()` function, you specify a **condition** to be evaluated. This condition can be almost any piece of code that R will evaluate into a **logical** value. The important thing about this condition is that it must correspond to a single logical value, either a single `TRUE` or a single `FALSE`.

The *condition* is an expression that when evaluated returns a **logical** value of length one. In other words, whatever you pass as the input of the `if` clause, it has to be something that becomes `TRUE` or `FALSE`

```
type <- "ordinary"
                    Logical condition
if (type == "ordinary") {
  # compute ord annuity
} else {
  # compute annuity due
}
```


In general, an R expression—using braces `{ }`—is used for each clause: the first one with the code that tells R what to do when the evaluated condition is true; the second one for what to do when the condition is false.

```
type <- "ordinary"

if (type == "ordinary") {
  # compute ord annuity
} else {
  # compute annuity due
}
```


13.2.3 Minimalist If-then-else

`if-else` statements can be written in different forms, depending on the types of expressions that are evaluated. If the expressions of both the `if` clause and the `else` clause are **simple** expressions, the syntax of the `if-else` code can be simplified into one line of code:

```
if (condition) expression_1 else expression_2
```

Consider the following example that uses a conditional statement to decide between calculating the square root of the input *if* the input value is positive, or computing the negative square root of the negative input *if* the input value is negative:

```
x <- 10

if (x > 0) {
  y <- sqrt(x)
} else {
  y <- -sqrt(-x)
}
y
#> [1] 3.162278
```

Because the code in both clauses consists of simple expressions, the use of braces

is not mandatory. In fact, you can write the conditional statement in a single line of code, as follows:

```
x <- 10

# with simple expressions, braces are optional
if (x > 0) y <- sqrt(x) else y <- -sqrt(-x)

y
```

Interestingly, the previous statement can be written more succinctly in R as:

```
x <- 10

# you can assign the output of an if-else statement
# to an object
y <- if (x > 0) sqrt(x) else -sqrt(-x)

y
```

Again, even though the previous commands are perfectly okay, I prefer to use braces when working with conditional structures. This is a good practice that improves readability:

```
# embrace braces: use them as much as possible!
x <- 10

if (x > 0) {
  y <- sqrt(x)
} else {
  y <- -sqrt(-x)
}
```

13.2.4 Simple If's

There is a simplified form of if-else statement which is available when there is no expression in the `else` clause. In its simplest version this statement has the general form:

```
if (condition) expression
```

and it is equivalent to:

```
if (condition) expression else NULL
```

Here's an example in which we have two numbers, `x` and `y`, and we are interested in knowing if `x` is greater than `y`. If yes, we print the message `"x is greater than y"`. If not, then we don't really care, and we do nothing.

```
x <- 4
y <- 2

if (x > y) {
  print("x is greater than y")
}
#> [1] "x is greater than y"
```

13.3 Multiple If's

A common situation involves working with multiple conditions at the same time. You can chain multiple if-else statements like so:

```
y <- 1 # Change this value!

if (y > 0) {
  print("positive")
} else if (y < 0) {
  print("negative")
} else {
  print("zero?")
}
#> [1] "positive"
```

Working with multiple chained if's becomes cumbersome. Consider the following example that uses several if's to convert a day of the week into a number:

```
# Convert the day of the week into a number.
day <- "Tuesday" # Change this value!

if (day == 'Sunday') {
  num_day <- 1
} else {
  if (day == "Monday") {
    num_day <- 2
  } else {
    if (day == "Tuesday") {
      num_day <- 3
    } else {
      if (day == "Wednesday") {
        num_day <- 4
      } else {
        if (day == "Thursday") {
          num_day <- 5
        } else {
          if (day == "Friday") {
```

```
            num_day <- 6
          } else {
            if (day == "Saturday") {
              num_day <- 7
            }
          }
        }
      }
    }
  }
}

num_day
#> [1] 3
```

Working with several nested if's like in the example above can be a nightmare.

In R, you can get rid of many of the braces like this:

```
# Convert the day of the week into a number.
day <- "Tuesday" # Change this value!

if (day == 'Sunday') {
  num_day <- 1
} else if (day == "Monday") {
  num_day <- 2
} else if (day == "Tuesday") {
  num_day <- 3
} else if (day == "Wednesday") {
  num_day <- 4
} else if (day == "Thursday") {
  num_day <- 5
} else if (day == "Friday") {
  num_day <- 6
} else if (day == "Saturday") {
  num_day <- 7
}

num_day
#> [1] 3
```

13.3.1 Switch statements

But still we have too many if's, and there's a lot of repetition in the code. If you find yourself using many if-else statements with identical structure for slightly different cases, you may want to consider a **switch** statement instead:

```r
# Convert the day of the week into a number.
day <- "Tuesday" # Change this value!

switch(day, # The expression to be evaluated.
  Sunday = 1,
  Monday = 2,
  Tuesday = 3,
  Wednesday = 4,
  Thursday = 5,
  Friday = 6,
  Saturday = 7,
  NA) # an (optional) default value if there are no matches
#> [1] 3
```

Switch statements can also accept integer arguments, which will act as indices to choose a corresponding element:

```r
# Convert a number into a day of the week.
day_num <- 3 # Change this value!

switch(day_num,
  "Sunday",
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday")
#> [1] "Tuesday"
```

13.4 Derivation of FVOA

In case you are curious, here's the derivation of the formula for the Future Value of an Ordinary Annuity.

For the sake of illustration, we'll consider a time period of three years, but the formula can be easily generalized to any number of years.

The starting point is the following equation:

$$\mathrm{FV} = \mathrm{C} + \mathrm{C}(1+r) + \mathrm{C}(1+r)^2$$

Multiplying both sides by $(1+r)$ we get:

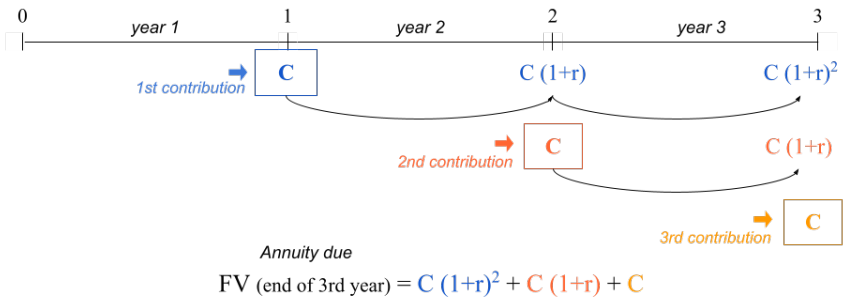


Figure 13.3: Timeline of an ordinary annuity

$$(1+r)\text{FV} = (1+r)\left[\text{C} + \text{C}(1+r) + \text{C}(1+r)^2\right]$$
$$(1+r)\text{FV} = (1+r)\text{C} + \text{C}(1+r)^2 + \text{C}(1+r)^3$$

Notice that:

$$(1+r)\text{FV} = \underbrace{\text{C}(1+r) + \text{C}(1+r)^2}_{\text{FV}-\text{C}} + \text{C}(1+r)^3$$

Doing more algebra we get the following:

$$\begin{aligned}
(1+r)\text{FV} &= \underbrace{\text{C}(1+r) + \text{C}(1+r)^2}_{\text{FV}-\text{C}} + \text{C}(1+r)^3 \\
(1+r)\text{FV} &= \text{FV} - \text{C} + \text{C}(1+r)^3 \\
\text{FV} - (1+r)\text{FV} &= \text{C} - \text{C}(1+r)^3 \\
\text{FV}\left[1-(1+r)\right] &= \text{C}\left[1-(1+r)^3\right] \\
\text{FV} &= \text{C}\frac{\left[1-(1+r)^3\right]}{\left[1-(1+r)\right]} \\
\text{FV} &= \text{C}\frac{\left[(1+r)^3-1\right]}{\left[(1+r)-1\right]} \\
\text{FV} &= \text{C}\left[\frac{(1+r)^3-1}{r}\right]
\end{aligned}$$

Therefore:

$$\text{FV} = \text{C} + \text{C}(1+r) + \text{C}(1+r)^2 = \text{C}\left[\frac{(1+r)^3-1}{r}\right]$$

Chapter 14

Iterations: For Loop

In the previous chapter you got introduced to conditional statements, better known as if-else statements. In this chapter we introduce another common programming structure known as **iterations**. Simply put *iterative procedures* commonly referred to as **loops**, allow you to repeat a series of common steps.

14.1 Motivation

Say you decide to invest $1000 in an investment fund that tracks the performance of the total US stock market. This type of financial asset, commonly referred to as a *total stock fund* is a mutual fund or an exchange traded fund (ETF) that holds every stock in a selected market. The purpose of a total stock fund is to replicate the broad market by holding the stock of every security that trades on a certain exchange. From the investing point of view, these funds are ideal for investors who want exposure to the overall equity market at a very low cost.

Examples of such funds are:

- VTSAX: Vanguard Total Stock Market Index Fund
- FSKAX: Fidelity Total Stock Market Index Fund
- SWTSX: Schwab Total Stock Market Index Fund

As we were saying, you decide to invest $1000 in a total stock market index fund (today).

How much money would you **expect** to get in 10 years?

If we knew the annual rate of return, we could use the future value formula (of compound interest)

$$\text{FV} = \$1000 \times (1 + r)^{10}$$

The problem, or the "interesting part"—depending on how you want to see it—is that *total stock funds* don't have a constant annual rate of return. Why not? Well, because the stock market is **volatile**, permanently moving up and down every day, with prices of stocks fluctuating every minute.

Despite the variability in prices of stocks, it turns out that in most (calendar) years, the annual return is positive. But not always. There are some years in which the annual return can be negative.

14.1.1 US Stock Market Historical Annual Returns

We can look at historical data to get an idea of the average annual return for investing in the total US stock market. One interesting resource that gives a nice perspective of the historical distribution of US stock market returns comes from amateur investor Joachim Klement

https://klementoninvesting.substack.com/p/the-distribution-of-stock-market

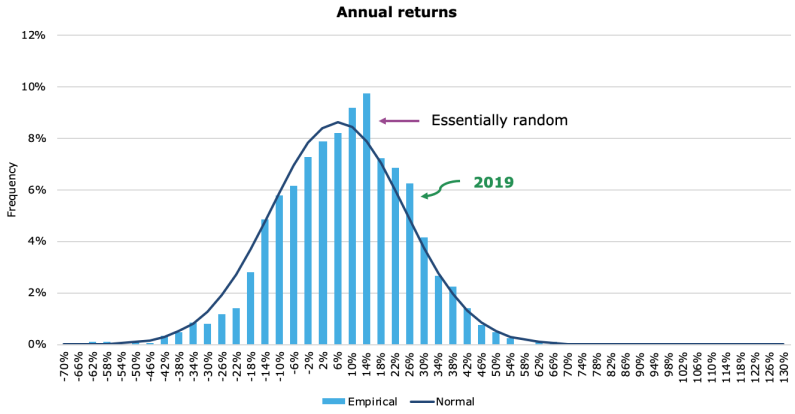


Figure 14.1: Distribution of Annual Returns (by Joachim Klement)

As you can tell, on an annual scale, market returns are basically random and follow the normal distribution fairly well.

So let us assume that annual rates of return for the total US Stock Market have a **Normal distribution** with mean $\mu = 10\%$ and standard deviation $\sigma = 18\%$.

Mathematically, we can write something like this:

$$r_t \sim \mathcal{N}(\mu = 0.10,\ \sigma = 0.18)$$

where r_t represents the annual rate of return in any given year t.

This means that, on average, we expect 10% return every year. But a return between 28% = 10% + 18%, and -2% = 10% - 18%, it's also a perfectly reasonable return in every year.

In other words, there is nothing surprising if you see years where the return is any value between -2% and 28%.

Admittedly, we don't know what's going to happen with the US Stock Market in the next 10 years. But we can use our knowledge of the long-term regular behavior of the market to guesstimate an expected return in 10 years. If we assume that the (average) annual return for investing in a total market index fund is 10%, then we could estimate the expected future value as:

$$\text{expected FV} = \$1000 \times (1 + 0.10)^{10} = \$2593.742$$

Keep in mind that one of the limitations behind this assumption is that we are not taking into account the volatility of the stock market. This is illustrated in the following figure that compares a theoretical scenario with constant 10% annual returns, versus a plausible scenario with variable returns in each year.

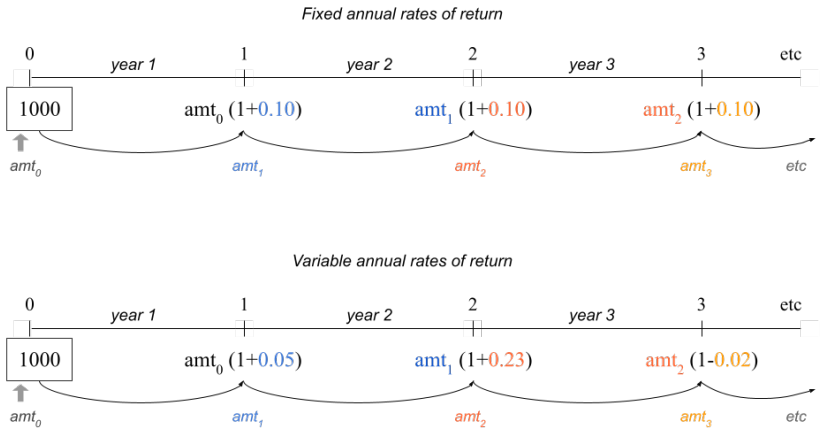


Figure 14.2: Rates of Return: Fixed -vs- Variable

14.2 Simulating Normal Random Numbers

Because the stock market is volatile, we need a way to simulate random rates of return. The good news is that we can use `rnorm()` to simulate generating numbers from a Normal distribution. By default, `rnorm()` generates a random number from a standard normal distribution (mean = 0, standard deviation = 1)

```
set.seed(345)  # for replication purposes
rnorm(n = 1, mean = 0, sd = 1)
#> [1] -0.7849082
```

Here's how to simulate three rates from a Normal distribution with mean $\mu = 0.10$ and $\sigma = 0.18$

```
rates = rnorm(n = 3, mean = 0.10, sd = 0.18)
rates
#> [1] 0.04968742 0.07093758 0.04769262
```

14.2.1 Investing in the US stock market during three years

To understand what could happen with your investment, let's focus on a three year horizon. For each yar, we need a random rate of return r_t $(t = 1, 2, 3)$

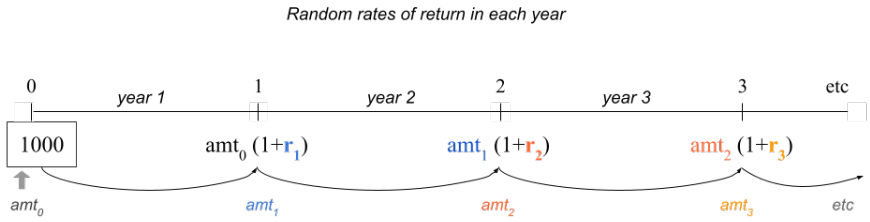


Figure 14.3: Random rates of return following a normal distribution

As we mentioned, we can use `rnorm()` to generate three random rates of return from a normal distribution with $\mu = 0.10$ and $\sigma = 0.18$

```
# inputs, consider investing during three years
set.seed(345)   # for replication purposes
amount0 = 1000
rates = rnorm(n = 3, mean = 0.10, sd = 0.18)
rates
#> [1] -0.04128347  0.04968742  0.07093758
```

With these rates, we can then calculate the amounts in the investment fund that we could expect to have at the end of each year:

```
# output: balance amount at the end of each year
amount1 = amount0 * (1 + rates[1])
amount2 = amount1 * (1 + rates[2])
amount3 = amount2 * (1 + rates[3])

c(amount1, amount2, amount3)
#> [1]  958.7165 1006.3527 1077.7409
```

Notice the common structure in the commands used to obtain an `amount` value. Moreover, notice that we are **repeating** the same operation three times. At this point you may ask yourself whether we could vectorize this code. Let's see if we can:

```
# vectorized attempt 1
amounts = amount0 * (1 + rates)
amounts
#> [1]  958.7165 1049.6874 1070.9376
```

If we use the vector `rates`, we obtain some `amounts` but these are not the values that we are looking for.

What if we vectorize both `rates` and years `1:3`?

```
# vectorized attempt 2
amounts = amount0 * (1 + rates)^(1:3)
amounts
#> [1]  958.7165 1101.8437 1228.2661
```

Once again, we obtain some `amounts` but these are not the values that we are looking for.

Can you see why the above vectorized code options fail to capture the correct compounding return?

14.2.2 Investing during ten years

Let's expand our time horizon from three to ten years, generating random rates of return for each year, and calculating a simulated amount year by year:

```
set.seed(345)   # for replication purposes
amount0 = 1000
rates = rnorm(n = 10, mean = 0.10, sd = 0.18)

amount1 = amount0 * (1 + rates[1])
amount2 = amount1 * (1 + rates[2])
amount3 = amount2 * (1 + rates[3])
amount4 = amount3 * (1 + rates[4])
amount5 = amount4 * (1 + rates[5])
amount6 = amount5 * (1 + rates[6])
amount7 = amount6 * (1 + rates[7])
amount8 = amount7 * (1 + rates[8])
amount9 = amount8 * (1 + rates[8])
amount10 = amount9 * (1 + rates[10])
```

Note: we know that this is too repetitive, time consuming, boring and error prone. Can you spot the error?

14.3 Iterations to the Rescue

Let's first write some "inefficient" code in order to understand what is going on at each step.

```
amount1 = amount0 * (1 + rates[1])
amount2 = amount1 * (1 + rates[2])
amount3 = amount2 * (1 + rates[3])
amount4 = amount3 * (1 + rates[4])
# etc
```

What do these commands have in common?

They all take an input amount, that gets compounded for one year at a certain rate. The output amount at each step is used as the input for the next amount.

Instead of calculating a single `amount` at each step, we can start with an almost "empty" vector `amounts()`. This vector will contain the initial amount, as well as the amounts at the end of every year.

```
set.seed(345)    # for replication purposes
amount0 = 1000
rates = rnorm(n = 10, mean = 0.10, sd = 0.18)

# output vector (to be populated)
amounts = c(amount0, double(length = 10))

# repetitive commands
amounts[2] = amounts[1] * (1 + rates[1])
amounts[3] = amounts[2] * (1 + rates[2])
# etc ...
amounts[10] = amounts[9] * (1 + rates[9])
amounts[11] = amounts[10] * (1 + rates[10])
```

From the commands in the previous code chunk, notice that at step `s`, to compute the next value `amounts[s+1]`, we do this:

```
amounts[s+1] = amounts[s] * (1 + rates[s])
```

This operation is repeated 10 times (one for each year). At the end of the computations, the vector `amounts` contains the initial investment, and the 10 values resulting from the compound return year-by-year.

14.4 For Loop Example

One programming structure that allows us to write code for carrying out these repetitive steps is a **for** loop, which is one of the iterative *control flow* structures in every programming language.

In R, a `for()` loop has the following syntax

```
for (s in 1:10) {
  amounts[s+1] = amounts[s] * (1 + rates[s])
}
```

- You use the `for` statement
- Inside parenthesis, you specify three ingredients separated by blank spaces:
  - an auxiliary iterator, e.g. `s`
  - the keyword `in`
  - a vector to iterate through, e.g. `1:10`
- The code for the repetitive steps gets wrapped inside braces

Let's take a look at the entire piece of code:

```
set.seed(345)    # for replication purposes
amount0 = 1000
rates = rnorm(n = 10, mean = 0.10, sd = 0.18)

# output vector (to be populated)
amounts = c(amount0, double(length = 10))

# for loop
for (s in 1:10) {
  amounts[s+1] = amounts[s] * (1 + rates[s])
}

amounts
#>  [1] 1000.0000  958.7165 1006.3527 1077.7409 1129.1412
#>  [6] 1228.3298 1211.0918 1129.9604 1590.9151 2223.8740
#> [11] 3170.9926
```

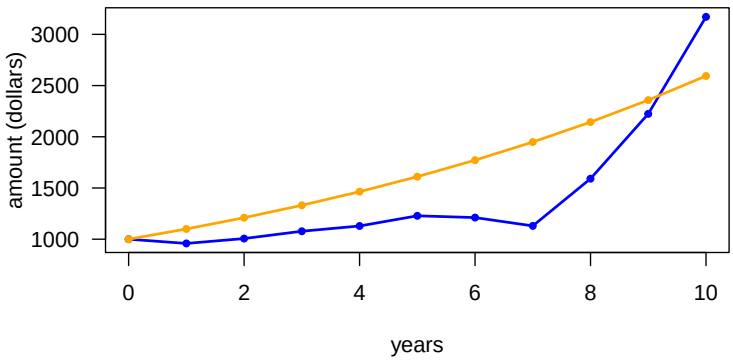
There are a couple of important things worth noticing:

- You don't need to declare or create the auxiliary iterator outside the loop
- R will automatically handle the auxiliary iterator (no need to explicitly increase its value)
- The length of the "iterations vector" determines the number of times the code inside the loop has to be repeated
- You use **for** loops when you know how many times a series of calculations need to be repeated.

Having obtained the vector of `amounts`, we can then plot a timeline to visualize the behavior of the simulated returns against the hypothetical investment with a constant rate of return:

```
# random annual rates of return (blue)
plot(0:10, amounts, type = "l", lwd = 2, col = "blue",
     xlab = "years", ylab = "amount (dollars)", las = 1)
points(0:10, amounts, col = "blue", pch = 20)

# assuming constant 10% annual return (orange)
lines(0:10, amount0 * (1+0.10)^(0:10), col = "orange", lwd = 2)
points(0:10, amount0 * (1+0.10)^(0:10), col = "orange", pch = 20)
```



14.5 About For Loops

To describe more details about `for` loops in R, let's consider a super simple example. Say you have a vector `vec <- c(3, 1, 4)`, and suppose you want to add 1 to every element of `vec`. You know that this can easily be achieved using vectorized code:

```
vec <- c(3, 1, 4)

vec + 1
#> [1] 4 2 5
```

In order to learn about loops, I'm going to ask you to forget about the notion of vectorized code in R. That is, pretend that R does not have vectorized functions.

Think about what you would need to do in order to add 1 to the elements in `vec`. This addition would involve taking the first element in `vec` and add 1, then taking the second element in `vec` and add 1, and finally the third element in `vec` and add 1, something like this:

```
vec[1] + 1
vec[2] + 1
vec[3] + 1
```

The code above does the job. From a purely arithmetic standpoint, the three lines of code reflect the operation that you would need to carry out to add 1 to

all the elements in `vec`.

From a programming point of view, you are performing the same type of operation three times: selecting an element in `vec` and adding 1 to it. But there's a lot of (unnecessary) repetition.

This is where loops come very handy. Here's how to use a `for ()` loop to add 1 to each element in `vec`:

```
vec <- c(3, 1, 4)

for (j in 1:3) {
  print(vec[j] + 1)
}
#> [1] 4
#> [1] 2
#> [1] 5
```

In the code above we are taking each `vec` element `vec[j]`, adding 1 to it, and printing the outcome with `print()` so you can visualize the additions at each iteration of the loop.

What if you want to create a vector `vec2`, in which you store the values produced at each iteration of the loop? Here's one possibility:

```
vec <- c(3, 1, 4)  # you can change these values
vec2 <- rep(0, length(vec))  # vector of zeros to be filled in the loop

for (j in 1:3) {
  vec2[j] = vec[j] + 1
}
```

14.5.1 Anatomy of a For Loop

The anatomy of a `for` loop is as follows:

```
for (iterator in times) {
  do_something
}
```

`for()` takes an **iterator** variable and a vector of **times** to iterate through.

```
value <- 2

for (i in 1:5) {
  value <- value * 2
  print(value)
}
#> [1] 4
#> [1] 8
```

```
#> [1] 16
#> [1] 32
#> [1] 64
```

The vector of *times* does NOT have to be a numeric vector; it can be **any** vector

```
value <- 2
times <- c('one', 'two', 'three', 'four')

for (i in times) {
  value <- value * 2
  print(value)
}
#> [1] 4
#> [1] 8
#> [1] 16
#> [1] 32
```

However, if the *iterator* is used inside the loop in a numerical computation, then the vector of *times* will almost always be a numeric vector:

```
set.seed(4321)
numbers <- rnorm(5)

for (h in 1:length(numbers)) {
  if (numbers[h] < 0) {
    value <- sqrt(-numbers[h])
  } else {
    value <- sqrt(numbers[h])
  }
  print(value)
}
#> [1] 0.6532667
#> [1] 0.4728761
#> [1] 0.8471168
#> [1] 0.9173035
#> [1] 0.3582698
```

14.5.2 For Loops and Next statement

Sometimes we need to skip a loop iteration if a given condition is met, this can be done with the `next` statement

```
for (iterator in times) {
  expr1
  expr2
  if (condition) {
```

```
    next
  }
  expr3
  expr4
}
```

Example:

```
x <- 2
for (i in 1:5) {
  y <- x * i
  if (y == 8) {
    next
  }
  print(y)
}
#> [1] 2
#> [1] 4
#> [1] 6
#> [1] 10
```

14.5.3 For Loops and Break statement

Sometimes we need to stop a loop from iterating if a given condition is met, this can be done with the `break` statement

```
for (iterator in times) {
  expr1
  expr2
  if (stop_condition) {
    break
  }
  expr3
  expr4
}
```

Example:

```
x <- 2
for (i in 1:5) {
  y <- x * i
  if (y == 8) {
    break
  }
  print(y)
}
#> [1] 2
```

```
#> [1] 4
#> [1] 6
```

14.5.4 Nested Loops

It is common to have nested loops

```
for (iterator1 in times1) {
  for (iterator2 in times2) {
    expr1
    expr2
    ...
  }
}
```

Example: Nested loops

Consider a matrix with 3 rows and 4 columns

```
# some matrix
A <- matrix(1:12, nrow = 3, ncol = 4)
A
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12
```

Suppose you want to transform those values less than 6 into their reciprocals (that is, dividing them by 1). You can use a pair of embedded loops: one to traverse the rows of the matrix, the other one to traverse the columns of the matrix:

```
# reciprocal of values less than 6
for (i in 1:nrow(A)) {
  for (j in 1:ncol(A)) {
    if (A[i,j] < 6) A[i,j] <- 1 / A[i,j]
  }
}
A
#>           [,1] [,2] [,3] [,4]
#> [1,] 1.0000000 0.25    7   10
#> [2,] 0.5000000 0.20    8   11
#> [3,] 0.3333333 6.00    9   12
```

14.5.5 About `for` Loops and Vectorized Computations

- R loops have a bad reputation for being slow.

- Experienced users will tell you: "tend to avoid `for` loops in R" (me included).
- It is not really that the loops are slow; the slowness has more to do with the way R handles the *boxing and unboxing* of data objects, which may be a bit inefficient.
- R provides a family of functions that are usually more efficient than loops (i.e. `apply()` functions).
- If you have NO programming experience, you should ignore any advice about avoiding loops in R.
- You should learn how to write loops, and understand how they work; every programming language provides some type of loop structure.
- In practice, many (programming) problems can be tackled using some loop structure.
- When using R, you may need to start solving a problem using a loop. Once you solved it, try to see if you can find a vectorized alternative.
- It takes practice and experience to find alternative solutions to `for` loops.
- There are cases when using `for` loops is not that bad.