R Coding Basics

An Introduction to the Basics of Coding in R

Gaston Sanchez

Table of contents

About

1st Edition: November 2022 2nd Edition: December 2023

This text seeks to give you an introduction to programming in R.

In a nutshell, I discuss the basics of R, covering properties of data objects, such as vectors, factors, matrices, lists, and data frames. Likewise, I describe fundamental notions of programming (e.g. functions, conditionals, iterations).

About You

I am assuming that you have both R or RStudio installed in your computer. If this is not the case, you can take a look at **Breaking the Ice with R**

https://www.gastonsanchez.com/R-ice-breaker

Citation

You can cite this work as:

Sanchez, G. (2022) An Introduction to the Basics of Coding in R. https://www.gastonsanchez.com/R-coding-basics

---

My Series of R Tutorials

This document is part of a series of texts that I've written about Programming and Data Analysis in R:

- **Breaking the Ice with R: Getting Started with R and RStudio** https://www.gastonsanchez.com/R-ice-breaker
- **Tidy Hurricanes: Analyzing Tropical Storms with Tidyverse Tools** https://www.gastonsanchez.com/R-tidy-hurricanes

- **R Coding Basics: An Introduction to the Basics of Coding in R** https://www.gastonsanchez.com/R-coding-basics
- **Rolling Dice: Exploring Simulations in Games of Chance with R** https://www.gastonsanchez.com/R-rolling-dice
- **Web Technologies in R: A Short Introduction to Web Technologies in R** https://www.gastonsanchez.com/R-web-technologies

---

Donation

As a Data Science and Statistics educator, I love to share the work I do. Each month I spend dozens of hours curating learning materials like this resource. If you find any value and usefulness in it, please consider making a one-time donation—via paypal—in any amount (e.g. the amount you would spend inviting me a cup of coffee or any other drink). Your support really matters.

License

Part I

Vectors

1 First Contact with Vectors

In order to enjoy and exploit R as a computational tool, one of the first things you need to learn is about 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.

1.1 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 to ask is:

> 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:

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

In this example, you deposit \$1000, and the bank pays you 2% of \$1000 = \$20, 12 months from now.

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
```

1.1.1 Comments

One thing to note in all the previous commands is the use of the informative text preceded by the pound or hash `#` symbol, for example: `# in three years`. This kind of text is not a command but rather a **comment**.

```
# this is a comment
# and so is this

# in four years
1061.208 * (1.02)
```

```
[1] 1082.432
```

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

1.1.2 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
```

1.1.3 Assignment Statements

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

```
object <- value

# equivalent to
object = value
```

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

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
```

1.1.4 Case Sensitive

Recall that R is case sensitive. This means that `a1` is not the same as `A1`. Similarly, `money` is not the same as `Money` or `MONEY`

```
# case sensitive
money = 10
Money = 100
MONEY = 1000

money + Money
```

```
[1] 110
```

```
MONEY - Money
```

```
[1] 900
```

1.1.5 Use Descriptive Names

While the names of objects such as `d`, `r`, `a1`, etc, are good for a computer, they can be a bit cryptic for a human being. Right now you may not have an issue understanding what those names refer to. You may even find these names to be quite convenient: they are short and easy to type. Moreover, they match the algebraic notation used in the compound interest formula. What's not to like about them?

Well, the issue is that this kind of names are too short. Mathematically there's nothing wrong with them. Computationally, from the point of view of the programming language (R), there is also nothing inherently bad about them. However, from the human (i.e. the reader or the code reviewer) standpoint, it is much better if we use more descriptive names, for example:

- `deposit` instead of `d`
- `rate` instead of `r`
- `amount1` instead of `a1`

The longer and more descriptive names are good for a computer and also for a human being (that reads English).

```
# 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 idea of using descriptive names has to do with a broader topic known as *literate programming*. This term, coined by computer scientist Donald Knuth, involves the core idea of creating programs as being works of literature. As Donald puts it:

> *"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."*
>
> Donald Knuth (1984)

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, which most of the times it's going to be your future self. 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.

1.1.6 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**. You'll learn about the basic properties of vectors in the next chapter.

1.2 Exercises

1) Area of a rectangle.

As you know, the area of a rectangle is the product of its length and width:

$$\text{area of rectangle} = \text{length} \times \text{width}$$

Write R code to compute the area of a rectangle of a certain `length` and a certain `width`. You can give `length` and `width` numeric values of your preference. The computed area should be stored in an object `area`.

```
length = 3
width = 4

area_rect = length * width
```

2) Area of a circle.

As you know, the area of a circle of radius r is given by

$$\text{area of circle} = \pi \times r^2$$

Write R code to compute the area of a circle of radius $r = 5$. Write code in a way that you create a `radius` object, and an `area` object. By the way, R comes with a built-in constant `pi` for the number π.

```
radius = 5

area_circ = pi * radius^2
```

3) In this chapter we introduced the formula of Future Value. There is also 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. Its equation is given by:

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

where:

- FV = Future Value
- r = Rate of return

- n = Number of periods

Write R code to compute the Present Value of $2,200 one year from now, knowing that the annual rate of return is $r = 3\%$. Try using descriptive names for the objects created to obtain the present value.

```
FV = 2200
rate = 0.03
years = 1

PV = FV / ((1 + rate)^years)
```

4) Consider the monthly bills of an undergraduate student:

- cell phone $90
- transportation $30
- groceries $580
- gym $15
- rent $1700
- other $85

a) Use assignments to create variables `phone`, `transportation`, `groceries`, `gym`, `rent`, and `other` with their corresponding amounts.

```
cellphone = 90
transportation = 30
groceries = 580
gym = 15
rent = 1700
other = 85
```

b) Create a `total` object with the sum of the expenses.

```
total = cellphone + transportation + groceries + gym + rent + other
```

c) Assuming that the student has the same expenses every month, how much would she spend during a school "term"? (assume the term involves five months).

```
semester_exps = total * 5
```

2 Properties of 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 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.

2.1 Motivation

In the preceding chapter we ended up creating six main objects, which I'm bringing back for you in the following code chunk:

```
# 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 in a single object
amounts = c(amount1, amount2, amount3)
amounts
```

```
[1] 1020.000 1040.400 1061.208
```

As we said, all these objects are **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:

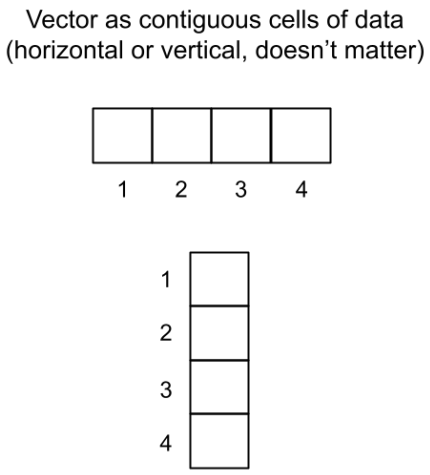


Figure 2.1: Think of vectors as contiguous cells of data.

Because this is supposed to be a "mental representation" of a vector, you can think of a vector either horizontally or vertically oriented. It does not really matter which orientation you prefer to visualize since R has no notion of a vector's orientation.

Regardless of how you decide to picture a vector in your head, an important trait of this class of object, and of any other data object in R, is that the starting position or index is always number 1.

2.2 Vectors are Atomic Objects

The first thing you should learn about R vectors is that they are considered 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`

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

```r
# 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. Also, notice that these logical values are specified with upper case letters. Recall that R is case sensitive, so if you type `True` or `False` R will not recognize them as logical values.

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 `"yosemite"` or single quotes `'yosemite'`. The important thing is to have an opening and a closing quote of the same kind.

The following diagram summarizes the four common data types of vectors.

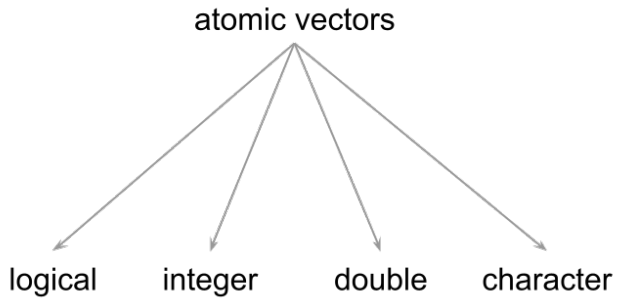


Figure 2.2: Common data types of atomic vectors.

2.2.1 Complex and Raw Types

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

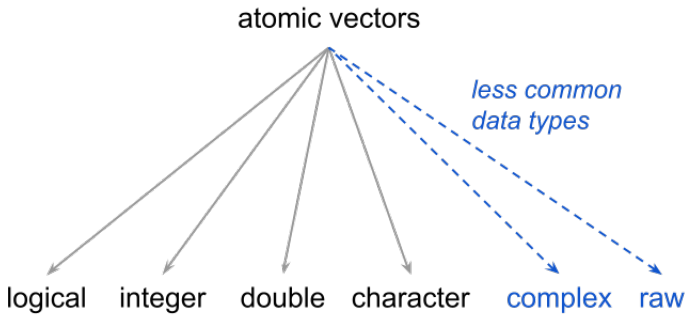


Figure 2.3: The 4 most common, and 2 less common, data types of atomic vectors.

I got my first contact with R back in 2001, and I've been using it almost on a daily basis since 2005. I have never had the need to use `raw` vectors; R may be using them in the background but not me (at least not explicitly or consciously). As for `complex` numbers, I've seen them every once in a while when doing certain matrix algebra computations. But again, I haven't had the need to intentionally work with complex values. I can say the same thing about most of my colleagues: they rarely use these data types. And I'm willing to bet that you will rarely use them too. But now you know, if you ever need to work with `complex` and/or `raw` values, they are available in R.

2.3 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()`
- `storage.mode()`
- `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)
```

```
[1] "double"
```

```
typeof(rate)
```

```
[1] "double"
```

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*. To further complicate matters, there is not just one but two functions related to the concept of mode: `storage.mode()` and `mode()`.

```
storage.mode(deposit)
```

```
[1] "double"
```

```
mode(deposit)
```

```
[1] "numeric"
```

Both `storage.mode()` and `mode()` rely on the output of `typeof()`, and as their name indicate, provide information about the storage mode. So what is the difference between these functions?

For practical purposes, `storage.mode()` is similar to `typeof()`, in the sense that they both return the same output:

```
typeof(deposit)
```

```
[1] "double"
```

```
storage.mode(deposit)
```

```
[1] "double"
```

In turn, `mode()` behaves a bit different. `mode()` groups together types `"double"` and `"integer"` into a single mode called `"numeric"` because both data types are numeric values.

```
typeof(deposit)
```

```
[1] "double"
```

```
mode(deposit)
```

```
[1] "numeric"
```

The following diagram depicts the `numeric` mode of types `integer` and `double`:

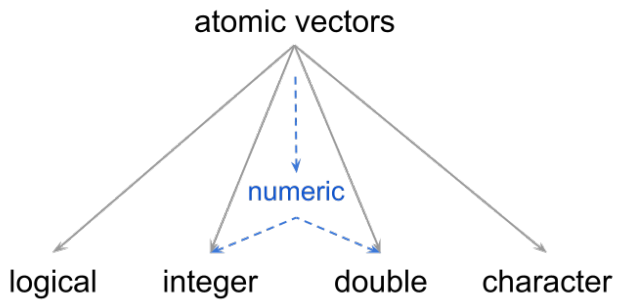


Figure 2.4: Data types "integer" and "double" correspond to "numeric" mode.

R also comes with a set of testing functions to determine the type and mode of a given value:

- `is.logical()` to test if a value is of type logical
- `is.integer()` to test if a value is of type integer
- `is.double()` to test if a value is of type double
- `is.numeric()` to test if a value is of numeric mode: either integer or double
- `is.character()` to test if a value is of type character

We discuss more details about how R handles data types in chapter More About Vectors. In the meantime, the following table shows basic examples of each common data type, and the output of associated functions that give you information about the type and mode of vectors:

Example	`typeof()`	`storage.mode()`	`mode()`
`TRUE`	`"logical"`	`"logical"`	`"logical"`
`1L`	`"integer"`	`"integer"`	`"numeric"`
`1`	`"double"`	`"double"`	`"numeric"`
`"one"`	`"character"`	`"character"`	`"character"`

2.4 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). The way I like to think of this is as "nothing"; that is, an object that represents "nothing".
- `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`

R also comes with a set of testing functions to determine if a given value is of a special kind:

- `is.null()` to test if a value is `NULL`
- `is.na()` to test if a value is `NA`
- `is.nan()` to test if a value is `NaN`
- `is.infinite()` to test if a value is `Inf` or `-Inf`

2.5 Length of Vectors

Another important property about vectors is that they have **length**, which refers to the number of elements or cells that they contain. If it helps, you can think of length as the "size" of a vector.

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. To find the length of a vector you use the function `length()`

```
length(deposit)
```

```
[1] 1
```

```
length(amounts)
```

```
[1] 3
```

By the way, vectors can be of any length, including 0. We talk more about these special vectors in the next chapter.

No Scalars in R

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. This means that when you type `5` R handles this data with a vector of length one.

2.6 Vector elements can have names

Another feature of vectors is that their elements can have names. For example, we have the `amounts` vector that contains the savings amounts at the end of years 1, 2, and 3:

```
amounts
```

```
[1] 1020.000 1040.400 1061.208
```

We can give names to the elements in `amounts` by using the `names()` function. We apply `names()` to `amounts` and we assign a vector of names like so:

```
names(amounts) = c("year1", "year2", "year3")
amounts
```

```
   year1    year2    year3
1020.000 1040.400 1061.208
```

2.7 Exercises

Consider the following three types of financial products:

a) joint **savings account** that pays 2% annual return, during 2 years,

b) joint **money market account** that pays 3% annual return, during 3 years

c) individual **certificate of deposit** that pays 4% annual return, during 4 years

Questions below.

1) Use the `c()` function to create a character vector `accounts` such that when printed it displays the values shown below:

```
[1] "savings"      "money market" "certificate"
```

```
accounts = c("savings", "money market", "certificate")
```

2) Use the `c()` function to create a vector `joint` of **logical** values such that when printed it displays the values shown below:

```
[1]  TRUE  TRUE FALSE
```

```
joint = c(TRUE, TRUE, FALSE)
```

3) Use the `c()` function to create a vector `rates` such that when printed it displays the values shown below:

```
[1] 0.02 0.03 0.04
```

```
rates = c(0.02, 0.03, 0.04)
```

4) Use the function `names()` to give names to the elements of your vector `rates`. When printing `rates` you should get following output:

```
savings  market  certif
   0.02    0.03    0.04
```

```
names(rates) = c('savings', 'market', 'certif')
```

5) Use the `c()` function to create a vector `years` of **integer** values such that when printed it displays the values shown below. Recall that integer values are of the form `1L`.

```
[1] 2 3 4
```

```
years = c(2L, 3L, 4L)
```

6) Use the `typeof()` function to inspect the data type of `accounts`, `joint`, `rates`, and `years`. Confirm that their output is `"character"`, `"logical"`, `"double"` and `"integer"`, respectively.

```
typeof(accounts)
typeof(joint)
typeof(rates)
typeof(years)
```

7) Use the `length()` function to inspect the length of `accounts`, `joint`, `rates`, and `years`.

```
length(accounts)
length(joint)
length(rates)
length(years)
```

8) Explain, in your own words, why vectors are said to be **atomic** objects.

9) Explain, in your own words, the difference between the output of functions `typeof()` and `mode()`—when applied to vectors.

3 Creating Vectors

In the preceding chapter you started learning about the basic properties of vectors. We focused on the common flavors of vectors (e.g. logical, integer, double, and character), reviewed special values (e.g. `NULL`, `NA`), talked about the length or size of a vector, and we also mentioned that elements in a vector can have names.

Likewise, you have seen two basic ways to create simple vectors:

- creation of one-element vectors, e.g. `money = 100`, `rate = 0.02`, `account = "savings"`
- creation of more than one-element, yet small, vectors with the combine function `c()`, e.g. `years = c(1L, 2L, 3L)`.

In this chapter I discuss two broad topics: 1) a review of various functions and ways to **create vectors**, and 2) a description of the **coercion** notion. Why should you learn about various forms of vector creation in R? Because as I said, R is—to a large extent—a vector-based language, and you have to be ready to create multiple kinds of vectors, and to take advantage of some of the mechanisms that R provides for doing this. As for the topic of coercion, you also need to understand the behavior of R when working with vectors of different data types.

3.1 Creating vectors with `c()`

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

```
# 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)
```

We've also seen the basic use of the **combine** function `c()` to create a vector containing several elements:

```r
# combine amounts in a single vector
amounts = c(amount1, amount2, amount3)
amounts
```

```
[1] 1020.000 1040.400 1061.208
```

The `c()` function is one of the primary functions to create vectors of length greater than one. Here's another example for how to create a vector `flavors` with some ice-cream flavors:

```r
flavors <- c("lemon", "vanilla", "chocolate")

flavors
```

```
[1] "lemon"     "vanilla"   "chocolate"
```

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

If your vector has only one element, you don't need to call the `c()` function.

```r
# no need to use c() to create a one-element vector
lemon = c("lemon")

# instead just do this
lemon = "lemon"
```

One more thing that you can do when using `c()` is to give names to the elements of the created vector. This is done by joining pairs of values of the form: `'name' = value`, where `'name'` is the name given to the `value` of an element. For instance, you can create the vector `amounts` and give names to each element like this:

```r
# give names to elements when using c()
# (names specified with quotes)
amounts = c(
  "year1" = amount1,
  "year2" = amount2,
  "year3" = amount3)

amounts
```

```
   year1    year2    year3
1020.000 1040.400 1061.208
```

As you can tell, the names of each element are specified as `character` values: `"year1"`, `"year2"`, and `"year3"`. Interestingly, you can also specify names without quoting them:

```
# give names to elements when using c()
# (names unquoted)
amounts2 = c(
  year1 = amount1,
  year2 = amount2,
  year3 = amount3)

amounts2
```

```
   year1    year2    year3
1020.000 1040.400 1061.208
```

This way of giving names to the elements of a vector can feel a bit surprising, especially to users that have previous programming experience but are new to the R syntax. Personally, I don't really care about having 2 different—and apparently confusing—ways to give names to elements when using functions like `c()`. Having said that, I can perfectly understand the initial shock and confusion that this may cause to non-experienced useRs.

To be consistent with most other languages, and also to play defensively, I tend to recommend quoting the values that are supposed to be the names of the elements in a vector. Again, this is my personal biased suggestion, and it is not a rule by any means.

3.2 Default Vectors

R comes with a set of functions to initialize vectors of a specific data type. The generic function is `vector()` but there are also type-specific versions:

- `vector()`
- `logical()`
- `integer()`
- `double()` and `numeric()`
- `character()`

The function `vector()`, as the name indicates, lets you create a vector of a given `mode` and of a certain `length`. By default, `vector()` creates a `"logical"` vector of `length = 0`.

```
log = vector()
log
```

```
logical(0)
```

```
length(log)
```

```
[1] 0
```

Notice what happens when you print `log`, the output displayed is: `logical(0)`. This is the notation that R uses to indicate that a vector is of length zero. The previous call is equivalent to:

```
vector(mode = "logical", length = 0)
```

A common question that some useRs have when they encounter things like `logical(0)` is "when do you use zero-length vectors"? The quick answer is: you can use zero-length vectors to initialize a vector that will later be populated with more elements. This typically happens when you know that a vector of certain type is needed to store several values, but you don't know in advance how many elements will be computed.

All the other functions, e.g. `logical()`, `integer()`, etc, take just one argument `length` to indicate the number of elements of the output vector. Keep in mind that the value(s) of the initialized vector cannot be changed:

```
logical(length = 1)
```

```
[1] FALSE
```

```
integer(length = 2)
```

```
[1] 0 0
```

```
double(length = 3)
```

```
[1] 0 0 0
```

```
character(length = 4)
```

```
[1] "" "" "" ""
```

3.3 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()`

3.3.1 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 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 one-unit steps. For example:

```
dbls = 1.5:5.5
dbls
```

```
[1] 1.5 2.5 3.5 4.5 5.5
```

```
typeof(dbls)
```

```
[1] "double"
```

Run the following commands to see 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 **descending** sequence by starting with a value on the left-hand side of `:` that is greater than the value on the right-hand side:

```
# descending (reversed) sequence
10:1
```

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

```
# this also applies to negative numbers
-10:-1
```

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

3.3.2 Sequences with `seq()`

The colon operator `:` can be very useful but it has its limitations. Its main downside is that the generated sequences are of one-unit steps. But what if you want a sequence with steps different from one-unit? For instance, what if you are interested in something like: `2, 4, 6, 8, ...`?

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, then specify `by = 2`:

```
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`

```r
seq(from = 2, length.out = 6, by = 2)
```

```
[1]  2  4  6  8 10 12
```

3.3.3 Sequences with `seq_len()` and `seq_along()`

`seq()` comes with sibling functions such as `seq.int()`, `seq_len()` and `seq_along()`. These are more specialized functions than the generic `seq()`, and they can be more efficient to generate certain sequences.

The function `seq.int()` is designed to generate integer sequences. The difference against `seq()` is that `seq.int()` is more efficient:

```r
# equivalent to seq(from = 5, to = 10), but more efficient
seq.int(from = 5, to = 10)
```

```
[1]  5  6  7  8  9 10
```

If you want a sequence of consecutive positive integers starting at 1, `seq_len()` is your friend:

```r
# equivalent to seq(from = 1, to = 10), but more efficient
seq_len(10)
```

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

The third type of sequence function is `seq_along()`. This function takes a vector of any length, and it produces a sequence of consecutive positive integers of the same length as the input vector.

```r
accounts = c("savings", "checking", "brokerage", "retirement")
seq_along(accounts)
```

```
[1] 1 2 3 4
```

If the input vector has length zero, then `seq_along()` returns zero

```r
null = NULL  # length(null) is zero
seq_along(null)
```

```
integer(0)
```

3.4 Replicated Vectors

Some times you need to create vectors containing repeated elements. To do this you can use the function `rep()`. This function takes a vector as the main input, and then it optionally takes various arguments: `times`, `length.out`, and `each` that let you control the way in which the elements of the input vector should be repeated.

```r
rep(1, times = 5)        # repeat 1 five times
```

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

```r
rep(c(1, 2), times = 3)  # repeat 1 2 three times
```

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

```r
rep(c(1, 2), each = 2)   # each element repeated twice
```

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

```r
rep(c(1, 2), length.out = 5)  # repeat until length of 5
```

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

Here are two less simple examples:

```r
rep(c(3, 2, 1), times = 3:1)
```

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

```
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
```

3.5 Coercion

One of the basic properties of vectors that you learned in the preceding chapter is that vectors are *atomic* objects. This is just the fancy way to say that all the elements of a vector have to be of the same data type. If I show you the following vectors, and ask you about their data types, you should have no problem answering this question:

```
one = c(TRUE, FALSE)
two = 2:4
three = c(11, 22, 33)
four = c("one", "two", "three")
```

If you have doubts about the data type of any of the above vectors, recall that you can use `typeof()` to get the answer.

But what if I create vectors by mixing elements of different data types? For example:

```
uno = c(FALSE, 1L)     # logical & integer
dos = c(1L, 2L, 3)     # integer & double
tres = c(1, 2, "3")    # double & character
```

Enter **coercion** principles!

Coercion is another fundamental concept that you should learn about vectors. This has to do with the mechanisms 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

3.5.1 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:

```r
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

Also, when certain operations are applied to certain data types, R may apply its coercion rules. An example of this behavior is when you have a logical vector on which you apply arithmetic operations:

```r
# logical vector
logs = c(TRUE, FALSE, TRUE)

# addition (creates integers)
logs2 = logs + logs
typeof(logs2)
```

```
[1] "integer"
```

```r
# multiplication (creates doubles)
logs3 = logs * 3
typeof(logs3)
```

```
[1] "double"
```

3.5.2 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 such as:

- `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:

```r
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

```r
is.integer(deposit)
```

```
[1] FALSE
```

```r
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
```

3.6 Exercises

1) 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")`

```
x = c(TRUE, FALSE)
y = c(x, 10)
z = c(y, 10, "a")

typeof(x)
typeof(y)
typeof(z)
```

2) 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)`

```r
x <- c('1', '2', '3', '4')
y <- (x == 1)
z <- y + 0
w <- c(x, "5.5")
yz1 <- c(y, z, pi)

typeof(x)
typeof(y)
typeof(z)
typeof(w)
typeof(yz1)
```

3) 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	TRUE

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

```r
planet = c("Mercry", "Venus", "Earth", "Mars")
gravity = c(3.7, 8.9, 9.8, 3.7)
```

```
daylength = c(4222.6, 2802, 24, 24.7)
temp = c(167L, 464L, 15L, -65L)
moons = c(0L, 0L, 1L, 2L)
haswater = c(FALSE, FALSE, TRUE, TRUE)
```

4) 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, i.e. using the function `c()`, the following:

a) `planets` with `gravity`

b) `planets` with `temp`

c) `planets` with `haswater`

d) `gravity` with `daylength`

e) `gravity` with `temp`

f) `temp` with `moons`

g) `temp` with `haswater`

5) Figure out how to 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
```

```
seq(from = 1, to = 2, by = 0.1)
```

6) Figure out how to use the function `seq()` to create the following vector

```
 [1] 1000  900  800  700  600  500  400  300  200  100
```

```
seq(from = 1000, to = 100, by = -100)
```

7) Figure out how to use the colon operator `:` to create the following vector

```
 [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

```
5:-5
```

8) Figure out how to use the colon operator `:` to create the following vector

```
[1] 9.25 8.25 7.25 6.25 5.25 4.25 3.25 2.25 1.25
```

```
9.25:1.25
```

9) Find out how to use the function `rep()` and the input vector `1:3` to create the following vector:

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

```
rep(1:3, each = 2)
```

10) Find out how to use the function `rep()` and the input vector `1:3` to create the following vector:

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

```
rep(1:3, times = 2)
```

11) Find out how to use the function `rep()` and the input vector `1:4` to create the following vector:

```
 [1] 1 2 2 3 3 3 4 4 4 4
```

```
rep(1:4, each = 1:4)
```

12) Use the `seq()` function to create vectors for each of the following parts, and find their `sum()`.

a) What is the sum of the first 100 positive odd numbers?

```
x = seq(from = 1, by = 2, length.out = 100)
sum(x)
```

b) Find the sum of the first 64 terms of the arithmetic series: $3 + 9 + 15 + 21 + ...$

```
x = seq(from = 3, by = 6, length.out = 64)
sum(x)
```

c) Find the partial sum of the arithmetic series below: $7 + 12 + 17 + 22 + \cdots + 187$

```r
x = seq(from = 7, to = 187, by = 5)
sum(x)
```

4 More About Vectors

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

4.1 Motivation: Future Value

Let's bring back the savings example from the previous chapters: 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.

4.1.1 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:

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

- FV = future value (how much you'll have)
- PV = present value (the initial deposit)
- r = rate of return (e.g. annual rate of return)
- n = 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!

4.2 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.

So what is 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, in R parlance, we call it *vectorization*.

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)
```

```
 [1] 31.93744 32.25523 32.57619 32.90034 33.22771 33.55834 33.89227 34.22951
 [9] 34.57011 34.91410
```

Be careful. Not every function that takes in a vector is necessarily *vectorized*. An example of a function that does not perform vectorization is the `mean()` function:

```
mean(amounts)
```

```
[1] 1116.872
```

As expected, `mean()` returns the average or mean value of all the numeric values in the vector `amounts`. It is not vectorized because it does compute the mean of every element of the input vector.

So be careful, just because a function does a computation with an input vector, it does not mean that its vectorized. Vectorization happens when the same function or action is applied to every element of a vector.

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).

4.3 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 its **recycling principle**, 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
```

4.3.1 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:

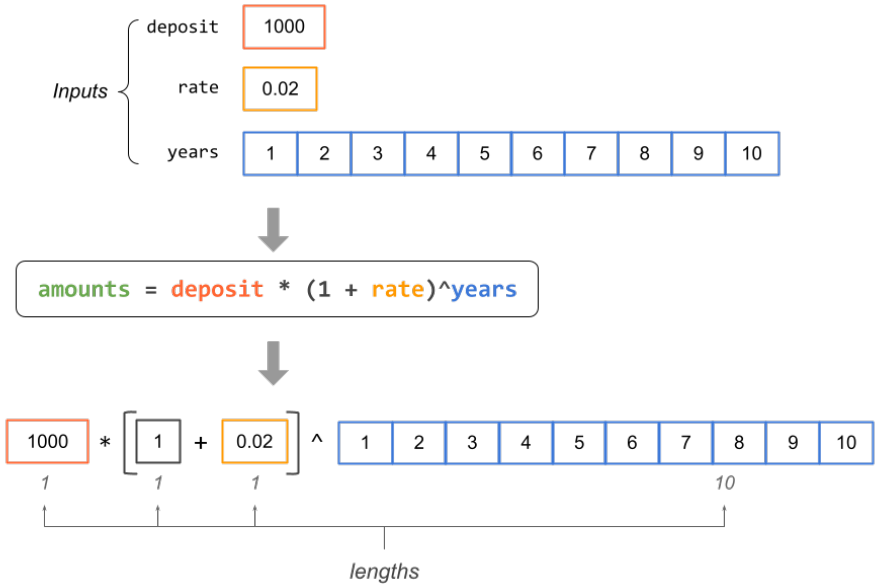
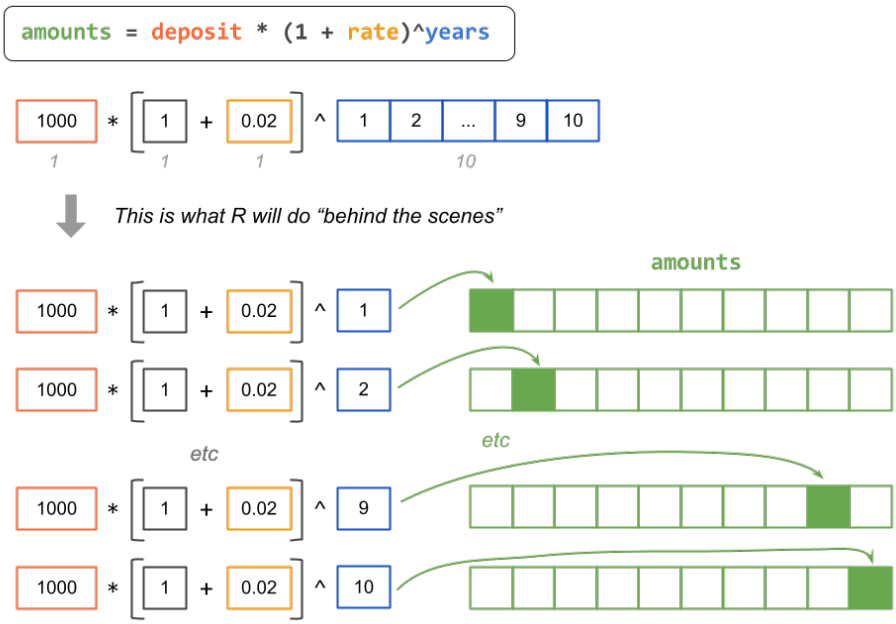


Figure 4.1: Diagram depicting vectors of different lengths.

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.



4.4 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 we use to extract and also replace 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 an indexing vector which could be a numeric vector, a logical vector, and sometimes a character vector. Let's see these options in more detail in the following subsections.

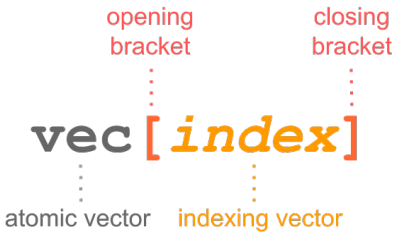


Figure 4.2: Use of brackets for subscripting vectors.

4.4.1 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 a single number (which is a vector of length one).

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

```
[1] 1020
```

The numeric indexing 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
```

4.4.2 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
```

4.4.3 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 or equal 5, and `TRUE` indicates that the number is greater than 5, 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 logical comparison:

```
x[x > 3 & x <= 6]
```

```
[1] 4 6
```

4.4.4 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)
```

4.5 Exercises

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

```
# Vectorization (or vectorized code) is when R applies calculations or
# operations to all the elements of a vector (element-wise)
```

2) Explain, in your own words, the *recycling principle* in R.

```
# Recycling is what R does when you perform a calculation with vectors
# of different length
```

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

```r
x[1:5]
head(x, 5)
```

4) 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`.

```r
y <- c(1, 4, 9, 16, 25)

z <- sqrt(y)
```

5) 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)`

```r
# command e)
```

6) For each the following parts, state whether the given function or operator is vectorized, and provide a code example to support your claim.

a) `length()`

b) `*`

c) `log()`

d) `max()`

e) `trunc()`

```r
# a) not vectorized
# b) vectorized
# c) vectorized
# d) not vectorized
# e) vectorized
```

7) 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)

```
name[seq(from = 2, to = length(name), by = 2)]
```

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

```
mpg[seq(from = 1, to = length(mpg), by = 2)]
```

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

```
mpg[seq(from = 5, to = length(mpg), by = 5)]
```

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

```
name[seq(from = length(name), to = 2, by = -2)]

# equivalently
rev(name[seq(from = 2, to = length(name), by = 2)])
```

8) 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

```
name[mpg < 15]
```

b) name of cars that have 6 cylinders

```
name[cyl == 15]
```

c) largest mpg value of all cars with 8 cylinders

```
max(mpg[cyl == 8])
```

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

```
name[mpg == median(mpg)]
```

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

```
name[mpg <= 22 & cyl >= 6]
```

Part II

More Atomic Objects

5 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. Luckily, 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 background in statistics, this is one of R's idiosyncrasies that seems disconcerting 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

5.1 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
```

5.2 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
```

```r
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:

```r
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
```

```r
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:

```r
# factors are not vectors
is.vector(first_factor)
```

```
[1] FALSE
```

```r
# factors are factors
is.factor(first_factor)
```

```
[1] TRUE
```

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

```r
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"`).

```r
# 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 characters `"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"`.

5.3 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.

5.3.1 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
```

5.3.2 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
```

5.4 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
```

```r
is.ordered(second_factor)
```

```
[1] FALSE
```

6 Matrices and Arrays

In the previous four 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.

6.1 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 n:

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

where:

- FV = future value
- PV = present value
- r = annual interest rate
- n = 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
```

```r
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? Let's find out.

6.2 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.

6.2.1 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.

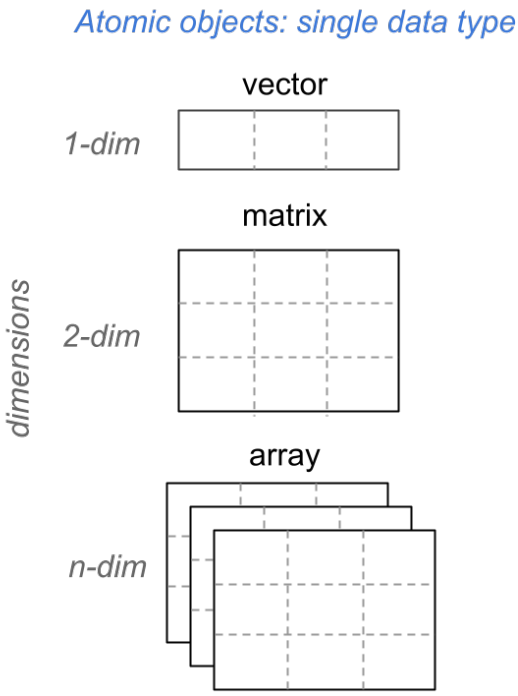


Figure 6.1: Triad of atomic data objects in R.

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.

6.3 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.

6.3.1 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; think of this as the total number of cells or entries in the matrix.
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:

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

R then compares the length of the input vector 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 input 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 vector
# does not 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)
```

```
Warning in matrix(1:12, nrow = 3, ncol = 2): data
length differs from size of matrix: [12 != 3 x 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 the matrix: 6. However, 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 nor a sub-multiple of the size of the matrix. In these cases 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.

6.3.2 Giving names to rows and columns

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

```
# matrix of savings amounts
savings = matrix(c(years, amounts), nrow = 6, ncol = 2)

# row and columns names
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
```

6.3.3 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
```

```r
# money market
moneymkt = 2000 * (1 + 0.025)^(0:2)
moneymkt
```

```
[1] 2000.00 2050.00 2101.25
```

```r
# 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 for each type of account:

```r
# savings account
sav_mat = cbind(0:4, savings)
```

```r
# money market
mm_mat = cbind(0:2, moneymkt)
```

```r
# certificate of deposit
cd_mat = cbind(0:3, certificate)
```

Single matrix

Alternatively, we can stack everything into a single matrix:

```r
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 about mixing data types?

What if you want some table like this:

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

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

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

investments
```

```
     [,1]       [,2] [,3]
[1,] "savings"  "0"  "1000"
[2,] "savings"  "1"  "1020"
```

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

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

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

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" by 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))

typeof(investments)
```

```
[1] "character"
```

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

6.4 Exercises

1) Use `matrix()` to create a matrix `mat1` (see below) from the input vector `x = letters[1:15]`:

```
# mat1
"a"  "d"  "g"  "j"  "m"
"b"  "e"  "h"  "k"  "n"
"c"  "f"  "i"  "l"  "o"
```

```
x = letters[1:15]
mat1 = matrix(x, nrow = 3, ncol = 5)
```

2) Look at the documentation of `matrix()` and find how to use it for obtaining the matrix `mat2` (see below) from the input vector `x = letters[1:15]`:

```
# mat2
"a"  "b"  "c"  "d"  "e"
"f"  "g"  "h"  "i"  "j"
"k"  "l"  "m"  "n"  "o"
```

```
x = letters[1:15]
mat2 = matrix(x, nrow = 3, ncol = 5, byrow = TRUE)
```

3) Find out how to use the functions `rownames()` and `colnames()` to give names to the rows and the columns of matrix `mat1`. Choose any names you want, and display matrix `mat1`.

```
rownames(mat1) = c('r1', 'r2', 'r3')
colnames(mat1) = c('c1', 'c2', 'c3', 'c4', 'c5')
```

4) Use `matrix()` to create a matrix `mat3` (see below) from the input vector `y = month.name`:

```
# mat3
"January" "February" "March"
"April"   "May"      "June"
"July"    "August"   "September"
"October" "November" "December"
```

```
y = month.name
mat3 = matrix(y, nrow = 4, ncol = 3, byrow = TRUE)
```

5) Use `matrix()`—and its recycling principle—to create a matrix `mat4` (see below) from the input vector `a = c(3, 6, 9)`:

```
# mat4
3    3    3
6    6    6
9    9    9
```

```
a = c(3, 6, 9)
mat4 = matrix(a, nrow = 3, ncol = 3)
```

6) Use `matrix()`—and its recycling principle—to create a matrix `mat5` (see below) from the input vector `a = c(3, 6, 9)`:

```
# mat5
3    3    3
6    6    6
9    9    9
3    3    3
6    6    6
9    9    9
```

```
a = c(3, 6, 9)
mat5 = matrix(a, nrow = 6, ncol = 3)
```

7) Consider the following vectors `a` and `b`

```
a = c(2, 4, 6)
b = c(1, 3, 5)
```

Use the row-binding function `rbind()`, with inputs `a` and `b`, to create a matrix `mat6` displayed below:

```
# mat6
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
mat6 = rbind(b, a)
```

8) Consider the following vectors `u` and `v`

```
u = c(2, 4, 6, 8)
v = c(1, 3, 5, 7)
```

Use the column-binding function `cbind()`, with inputs `u` and `v`, to create a matrix `mat7` displayed below:

```
# mat7
```

```
     [,1] [,2] [,3]
[1,]    2    1    2
[2,]    4    3    4
[3,]    6    5    6
[4,]    8    7    8
```

```
mat7 = cbind(u, v, u)
```

9) Find out how to use the `diag()` function to create an **identity matrix** of dimensions 4 rows and 4 columns (see below). BTW: An identity matrix is a matrix with the same number of rows and columns, has ones in the diagonal, and zeroes off-diagonal.

```
1    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1
```

```
diag(1, nrow = 4)
```

10) Refer to matrices `mat4` and `mat7`. Use both `cbind()` and `rbind()` to attempt binding these two matrices. If one of the binding operations fails, explain why.

```
# cbind(mat4, mat7) fails because these matrices have different number of rows

# rbind(mat4, mat7) works
```

7 More About Matrices

Now that you have seen some ways to create matrices, let's discuss a number of basic manipulations of matrices. I 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.

7.1 Basic Operations with Matrices

- Selecting 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
- Adding a new row
- Deleting a column
- Deleting a row
- Renaming a column
- Moving a column

Let's say you have a matrix `mat` with the following content:

```
# inputs
deposit = 1000
rate_savings = 0.02
rate_moneymkt = 0.025
rate_certificate = 0.03
years = 0:5

# future values
savings = deposit * (1 + rate_savings)^years
moneymkt = deposit * (1 + rate_moneymkt)^years
```

```r
certificate = deposit * (1 + rate_certificate)^years

# matrix
mat = matrix(c(years, savings, moneymkt, certificate), nrow = 6, ncol = 4)

# row and columns names
rownames(mat) = 1:6
colnames(mat) = c("year", "savings", "moneymkt", "certificate")

mat
```

```
  year  savings moneymkt certificate
1    0 1000.000 1000.000    1000.000
2    1 1020.000 1025.000    1030.000
3    2 1040.400 1050.625    1060.900
4    3 1061.208 1076.891    1092.727
5    4 1082.432 1103.813    1125.509
6    5 1104.081 1131.408    1159.274
```

7.1.1 Selecting elements

The matrix `mat` is a 2-dimensional object: the 1st dimension corresponds to the rows, while the 2nd dimension corresponds to the columns. Because `mat` has two dimensions, the bracket notation involves working with data frames in this form: `mat[ , ]`.

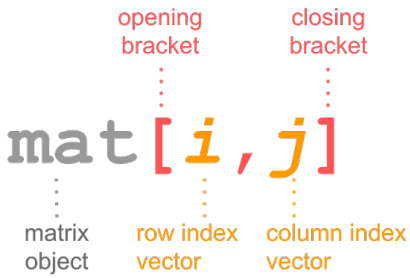


Figure 7.1: Bracket notation in matrices

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

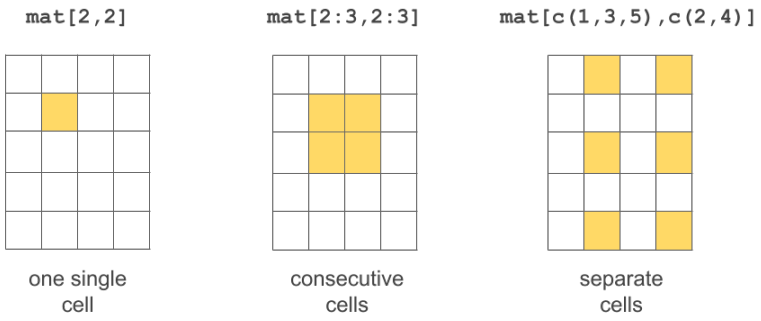


Figure 7.2: Several ways to select cells

Selecting cells

```
# select value in row 1 and column 1
mat[1, 1]
```

```
[1] 0
```

```
# select value in row 2 and column 3
mat[2, 3]
```

```
[1] 1025
```

```
# select values in these cells
mat[1:2, 3:4]
```

```
  moneymkt certificate
1     1000        1000
2     1025        1030
```

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

Selecting 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.

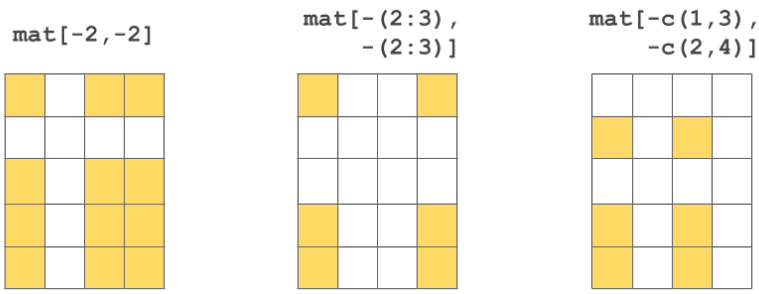


Figure 7.3: Several ways to exclude cells

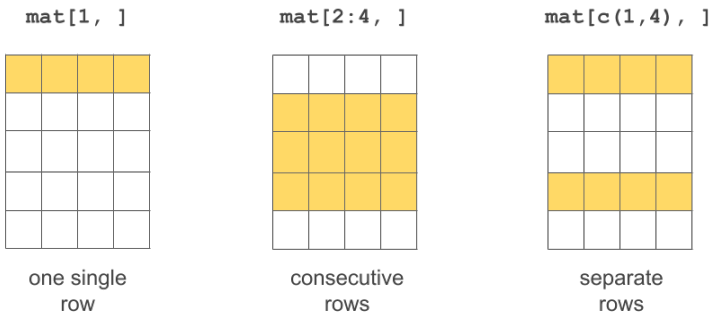


Figure 7.4: Several ways to select rows

```
# selecting first row
mat[1, ]
```

```
    year     savings    moneymkt certificate
       0        1000        1000        1000
```

```
# selecting third row
mat[3, ]
```

```
    year     savings    moneymkt certificate
   2.000    1040.400    1050.625    1060.900
```

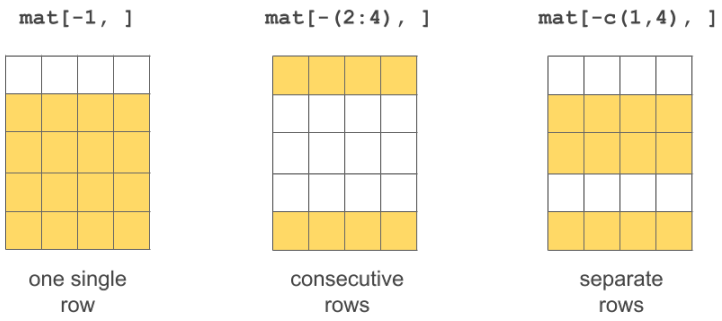


Figure 7.5: Several ways to exclude rows

Selecting columns

```
# selecting second column
mat[ ,2]
```

```
      1       2       3       4       5       6
1000.000 1020.000 1040.400 1061.208 1082.432 1104.081
```

```
# selecting columns 2 to 4
mat[ ,2:4]
```

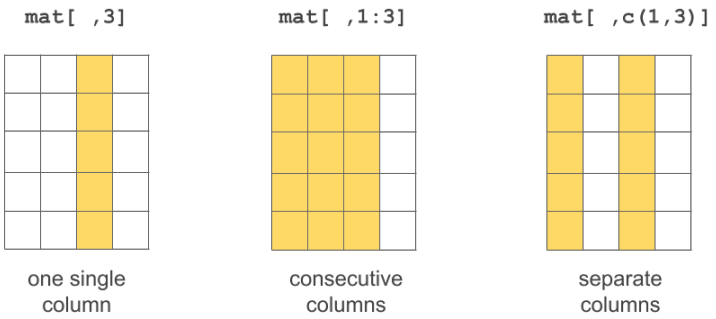


Figure 7.6: Several ways to select columns

```
   savings moneymkt certificate
1 1000.000 1000.000    1000.000
2 1020.000 1025.000    1030.000
3 1040.400 1050.625    1060.900
4 1061.208 1076.891    1092.727
5 1082.432 1103.813    1125.509
6 1104.081 1131.408    1159.274
```

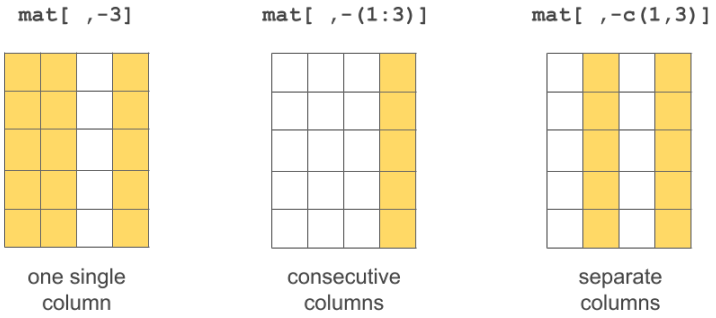


Figure 7.7: Several ways to exclude columns

7.1.2 Adding a column

To add a column, use the column-bind function `cbind()`

```
# vector
vec <- c(2, 4, 6, 8, 10, 12)
```

```r
# adding weights to mat
mat <- cbind(mat, vec)
mat
```

```
  year  savings moneymkt certificate vec
1    0 1000.000 1000.000    1000.000   2
2    1 1020.000 1025.000    1030.000   4
3    2 1040.400 1050.625    1060.900   6
4    3 1061.208 1076.891    1092.727   8
5    4 1082.432 1103.813    1125.509  10
6    5 1104.081 1131.408    1159.274  12
```

7.1.3 Deleting a column

What if you want to delete a column? Simple: use a negative index to exclude the undesired column(s)

```r
# deleting fifth column
mat <- mat[ ,-5]
mat
```

```
  year  savings moneymkt certificate
1    0 1000.000 1000.000    1000.000
2    1 1020.000 1025.000    1030.000
3    2 1040.400 1050.625    1060.900
4    3 1061.208 1076.891    1092.727
5    4 1082.432 1103.813    1125.509
6    5 1104.081 1131.408    1159.274
```

7.1.4 Moving a column

What if you want to move one or more columns to a different position? For example, what if you want to move `year` to the last position (last column)? One option is to create a vector of column indices in the desired order, and then use this vector (for the index of columns) to reassign the matrix like this:

```r
reordered_cols <- c(2:4, 1)
mat <- mat[ ,reordered_cols]
mat
```

```
   savings moneymkt certificate year
1 1000.000 1000.000    1000.000    0
2 1020.000 1025.000    1030.000    1
3 1040.400 1050.625    1060.900    2
4 1061.208 1076.891    1092.727    3
5 1082.432 1103.813    1125.509    4
6 1104.081 1131.408    1159.274    5
```

Part III

Non-Atomic Objects

8 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

8.1 Motivation

In the chapter about Matrices and Arrays we considered a small portfolio consisting of the following three investments:

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

b) $2000 in a **money market** account that pays 2.5% annual return, during 2 years

c) $5000 in a **certificate of deposit** that pays 3% annual return, during 3 years

Let's pay attention to the first investment: the savings account. Suppose I'm interested in creating an object to store the specifications of this investment, that is, I would like to have an R object with four elements

- the initial deposit: `1000`
- the annual rate of return: `0.02`
- the number of years: `4L`
- and the type of account: `"savings"`

What kind of object could I use? With all the things we've discussed so far, a natural decision would be to store these values in a vector:

```
investment1_specs = c(
  "deposit" = 1000,      # double
  "rate" = 0.02,         # double
  "years" = 4L,          # integer
```

```
  "account" = "savings"  # character
)
```

Notice that I'm creating `investment1_specs` by mixing elements of different data types: a couple of double types, an integer type, and a character type. Thus, a very pertinent question is: What kind of vector is `investment1_specs`? If your answer to the preceding question was *character*, then congrats! By now, I expect that you can correctly answer this question without any trouble. If that is not the case then go back to chapter Creating Vectors and reread the section on Coercion.

One simple way to confirm that `investment1_specs` is indeed of `"character"` type is by simply inspecting its contents:

```
investment1_specs
```

```
deposit      rate     years   account
 "1000"    "0.02"       "4" "savings"
```

While the `investment1_specs` object "technically" is storing the specifications of the savings account, all the initial numeric values have been coerced into characters, which may not be the best way to store this information. The solution to this limitation that vectors and other atomic objects have is to employ another kind of object in R: **lists**.

8.2 Lists

An R list is the most generic kind of data object in R in the sense that you can combine elements of different data types without them being coerced.

The primary function to create lists is the homonym function `list()`. To give you an example of a basic list let us again pay attention to the specifications of the first investment

- the initial deposit: `1000`
- the annual rate of return: `0.02`
- the number of years: `4L`
- and the type of account: `"savings"`

Instead of using a vector, we can create a list to store these values. All we have to do is use `list()` instead of `c()`:

```
specs1 = list(
  "deposit" = 1000,      # double
```

```
  "rate" = 0.02,          # double
  "years" = 4L,           # integer
  "account" = "savings"   # character
)

specs1
```

```
$deposit
[1] 1000

$rate
[1] 0.02

$years
[1] 4

$account
[1] "savings"
```

Observe the way R prints a list with named elements. In contrast to the way elements of a vector are displayed—in a contiguous form—the elements of a list are displayed in a non-contiguous manner. Also, note how the names of the elements are listed with a preappended dollar sign. For example, the first element is `$deposit`, the second element is `$rate`, and so on.

What about the data type for each element of the list? From the visual inspection of the elements in `specs1`, you can tell that all the numeric values are not being coerced into strings. Which is what we were looking for. We were interested in obtaining an object in which each of its elements gets to keep its data type.

If you try to use `typeof()` on a list in an attempt to get the data types of its elements, I'm afraid this won't work the way you expect it:

```
typeof(specs1)
```

```
[1] "list"
```

Applying `typeof()` on a list results in a not very interesting output `"list"`.

Getting ahead of myself momentarily, let me show you how to use the dollar operator `$` to refer to an named element of a list and check its data type:

```
typeof(specs1$deposit)
```

```
[1] "double"
```

```
typeof(specs1$account)
```

```
[1] "character"
```

We'll discuss the different ways in which you can subset elements of a list later in this chapter.

8.3 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.

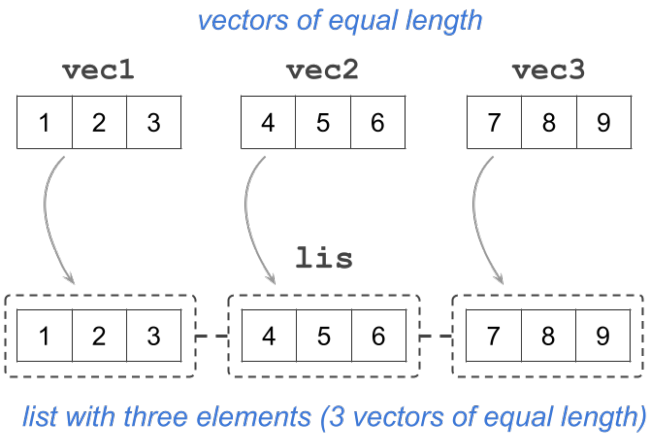


Figure 8.1: A list containing three unnamed elements (numeric vectors of length 3)

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

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):

```r
# 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:

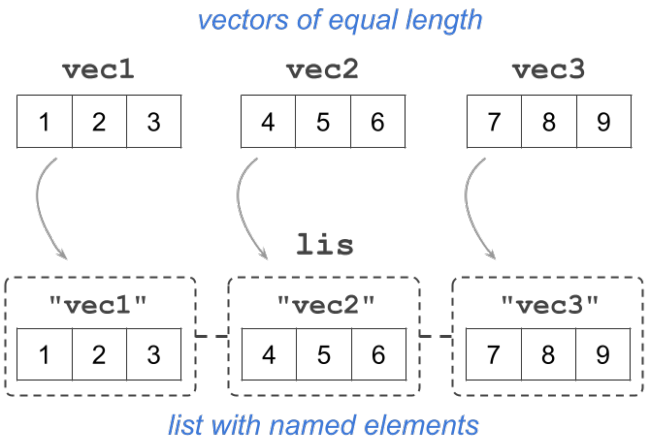


Figure 8.2: A list with named elements (numeric vectors of length 3)

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:

```r
lst <- list(
  c("savings", "money_mkt", "certificate"),
  matrix(1:6, nrow = 2, ncol = 3),
  factor(c("yes", "no", "no", "no", "yes")),
```

```
  list(1000, 2000, 5000)
)

lst
```

```
[[1]]
[1] "savings"     "money_mkt"   "certificate"

[[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] 1000

[[4]][[2]]
[1] 2000

[[4]][[3]]
[1] 5000
```

Whenever possible, I 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("savings", "money_mkt", "certificate"),
  second = matrix(1:6, nrow = 2, ncol = 3),
  third = factor(c("yes", "no", "no", "no", "yes")),
  fourth = list(1000, 2000, 5000)
)
```

8.4 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]]`).

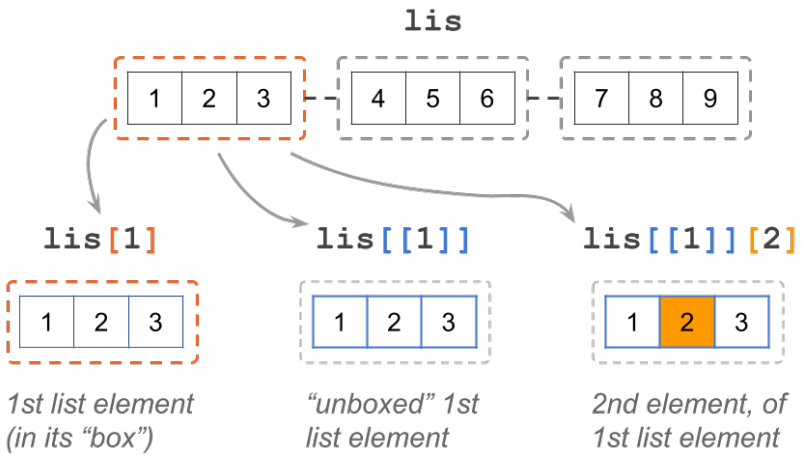


Figure 8.3: Bracket notation with lists

8.4.1 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 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".

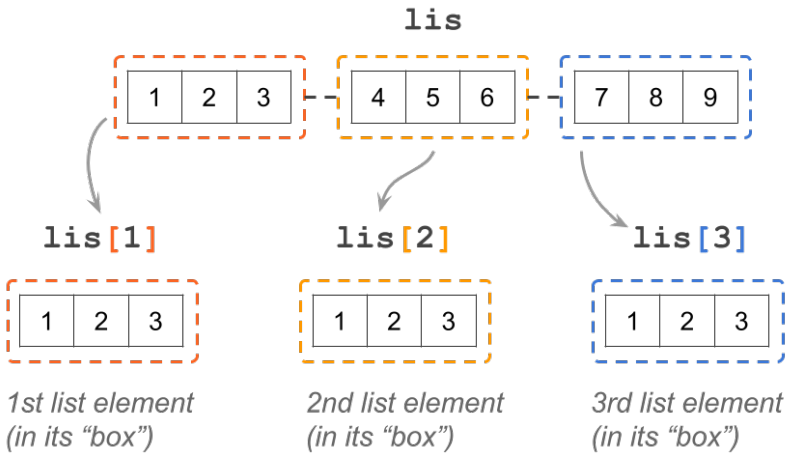


Figure 8.4: Use single brackets to select an element

8.4.2 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 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
```

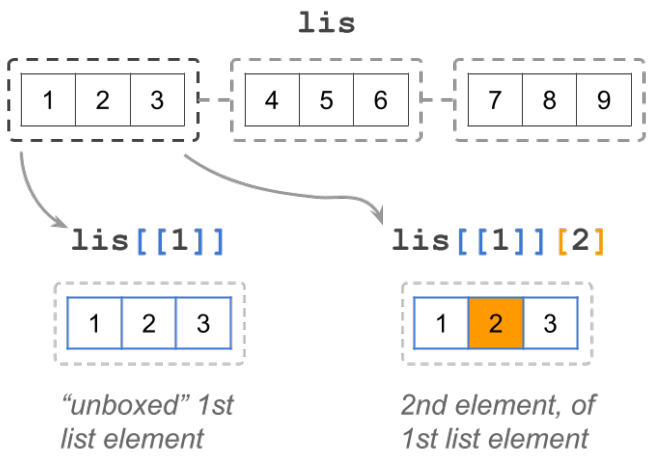


Figure 8.5: Use double brackets to extract an element

8.4.3 Dollar signs

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

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.

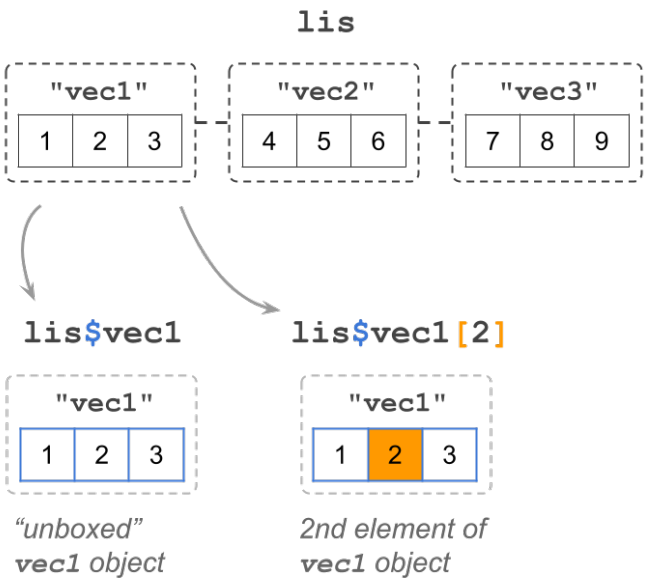


Figure 8.6: Dollar notation with lists

8.4.4 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"
```

8.4.5 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"
```

8.5 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"
```

```
lis = list(
  first = "Gaston",
  middle = NULL,
  last = "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]`

```
# a) values in column "letter" are given by:
# option ii)
# option iii)
```

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]])`

```
# b) length of "major_minor" is given by:
# option ii)
# option iv)
```

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")])`

```
# c) values in column "score" are given by:
# option i)
# option iii)
```

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)]]`

```
# d) value in "name" is given by:
# option ii)
# option iv)
```

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

```
# a) length of obj$hws is 3
# False
```

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

```
# b) data type of obj$exams could be double
# False
```

c) `obj$grades` cannot be a factor

```
# c) obj$grades cannot be a factor
# True
```

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

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

e) `obj$hws` could be a matrix

```
# e) obj$hws could be a data frame
# False
```

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

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

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

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

$gpa
[1] 4

$major_minor
             major1              major2               minor
     "jedi studies"      "sith studies" "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), ]`

9 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".

9.1 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.

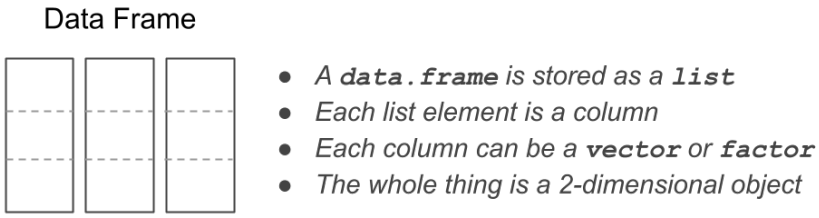


Figure 9.1: Abstract view of a data.frame

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 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.

9.2 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"   "qsec" "vs"   "am"   "gear"
[11] "carb"

$row.names
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
```

```
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"

$class
[1] "data.frame"
```

9.3 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:

```r
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
```

In old versions of R, to prevent `data.frame()` from converting strings into factors, you had to use the argument `stringsAsFactors = FALSE`

```r
# strings as strings, not as factors
# (for R ver 3.1.0 or older)
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
```

9.4 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 column
- Renaming a column
- Moving a column
- Transforming a column

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

```r
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
```

9.4.1 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[ , ]`.

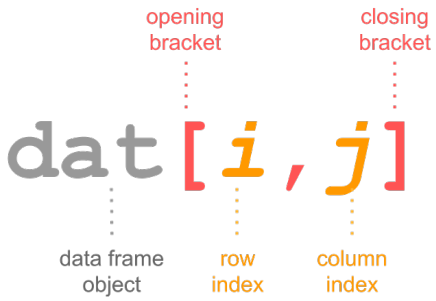


Figure 9.2: Bracket notation in data frames

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

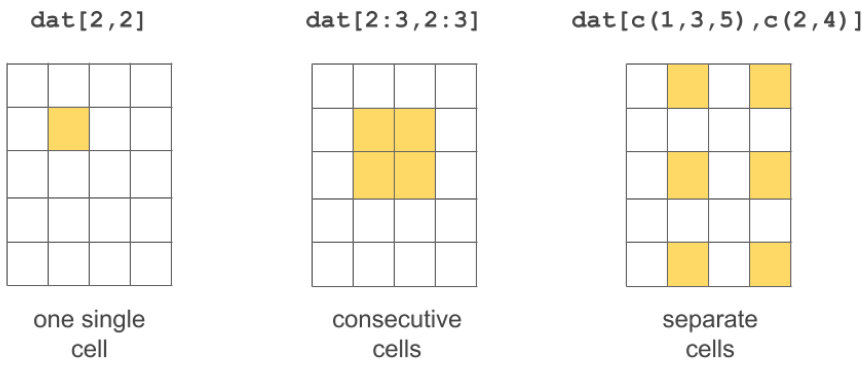


Figure 9.3: Several ways to select cells

Selecting 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:

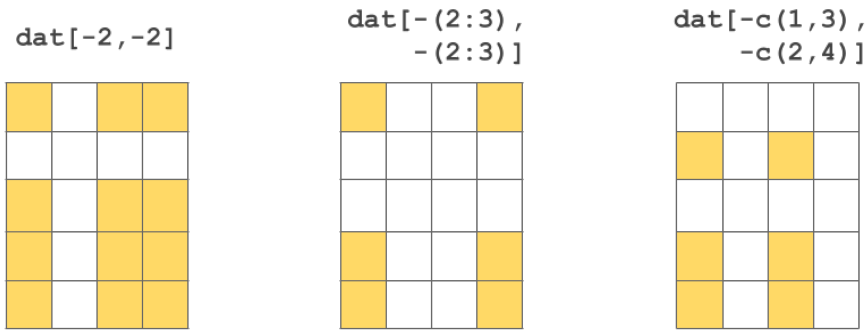


Figure 9.4: Several ways to exclude cells

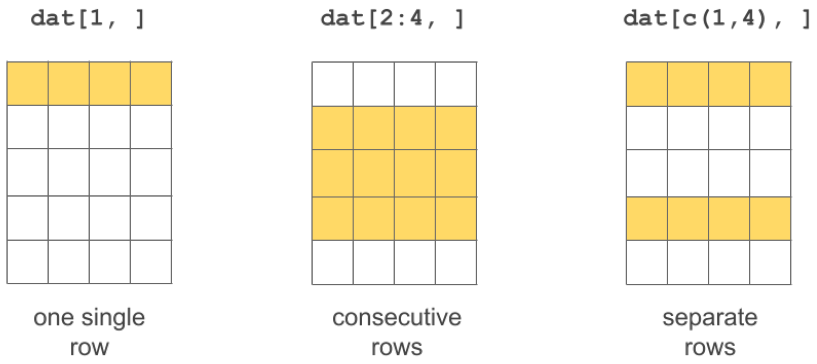


Figure 9.5: Several ways to select rows

Selecting 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
```

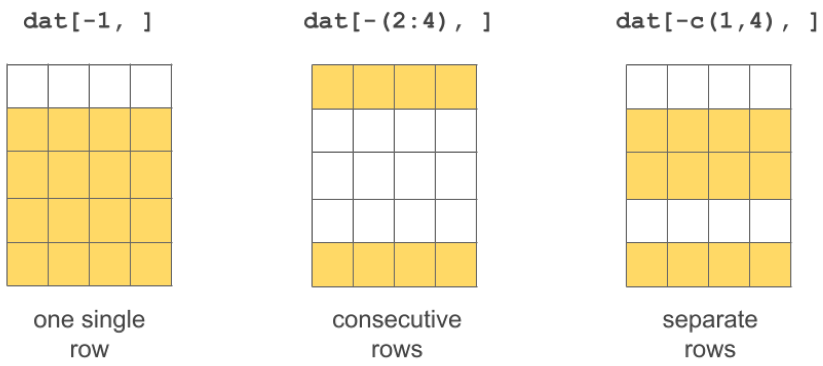


Figure 9.6: Several ways to exclude rows

Selecting columns

```
# selecting second column
dat[ ,2]
```

```
[1] "female" "male"   "male"
```

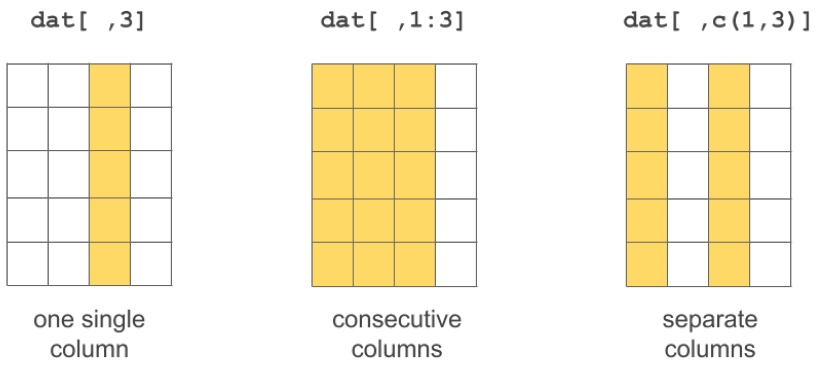


Figure 9.7: Several ways to select columns

```
# 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
```

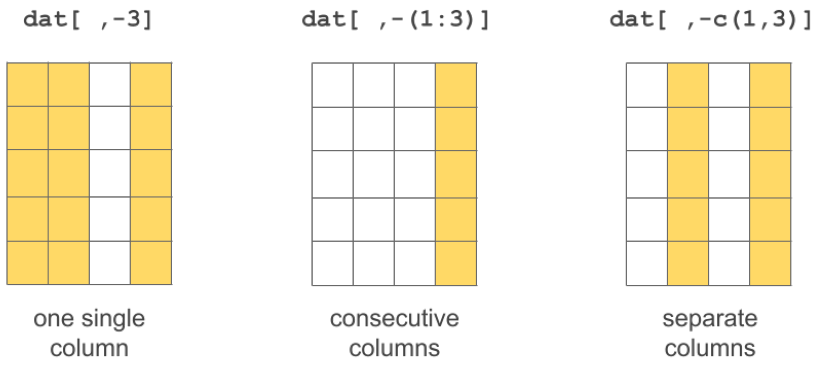


Figure 9.8: Several ways to exclude columns

More Options to Access Columns

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

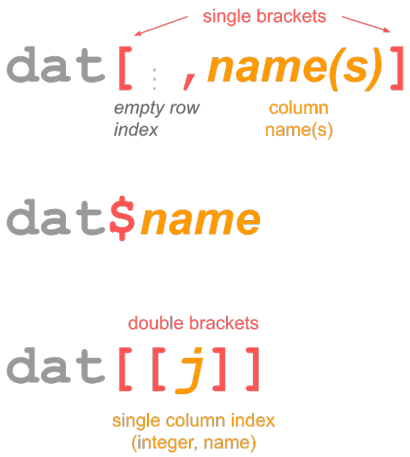


Figure 9.9: Other options to select columns of a data frame

```
mtcars$mpg
```

```
 [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 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`
```

9.4.2 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
```

9.4.3 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
```

9.4.4 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.

9.4.5 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
```

9.4.6 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
```

9.5 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]`

```
# a) these options:
# (i) 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)`

```
# b) this option:
# (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
```

```
# c) The command "df[1:3, 1 & 2]" displays all columns
# because "1 & 2" is a logical comparison that returns
# "TRUE", and therefore all columns are selected.
```

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

```
# c) any of these:
# df[1:3, 1:2]
# df[1:3, c(1, 2)]
# df[1:3, c('first', 'last')]
```

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
```

```
# d) any of these
# df[ ,c('spell', 'first')]
# df[ ,c(5,1)]
```

2) Consider the following data frame `dat`

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

```
# There are five errors:
#
# dat$gpa = max_gpa should be dat$gpa == max_gpa
#
# dat$first(which_max_gpa) should be dat$first[which_max_gpa]
#
# dat$gpa[dat[ ,title] = "lord"], should use quotations
# for "title" and also use double equals in:
# dat$gpa[dat[ ,"title"] == "lord"]
#
# median_females <- median(dat$gpa[which_males]) should
# be: median_females <- median(dat$gpa[which_females])
```

Part IV

Programming

10 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.

10.1 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?

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

- FV = future value (how much you'll have)
- PV = present value (the initial deposit)
- r = rate of return (e.g. annual rate of return)
- n = 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!

10.2 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):

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

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)^n
```

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
n = 10

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

# 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 `n` 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`
- `n`

Process:

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

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 `n`, 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
n

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

# 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, n) {
  fv = pv * (1 + r)^n
  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, n) {
  fv = pv * (1 + r)^n
  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:

```
# vectorized years
FV(1000, 0.02, 1:5)
```

```
[1] 1020.000 1040.400 1061.208 1082.432 1104.081
```

```
# vectorized rates
FV(1000, seq(0.01, 0.02, by = 0.005), 1)
```

```
[1] 1010 1015 1020
```

```
# 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.

10.2.1 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:

```
# future value function with default arguments
FV = function(pv = 100, r = 0.01, n = 1) {
  fv = pv * (1 + r)^n
  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, n = 3, pv = 1000)
```

```
[1] 1061.208
```

10.3 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
```

10.3.1 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)

10.3.2 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:

```r
# 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 planning 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.

10.4 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:

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

- PV = present value (the initial deposit)
- FV = future value (how much you'll have)
- r = rate of return (e.g. annual rate of return)
- n = number of periods (e.g. number of years)

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

```
present_value = function(future, rate, year) {
  present = future / (1 + rate)^year
  return(present)
}
```

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
```

```
fv_list <- function(present, rate, year) {
  future = present * (1 + rate)^(0:year)
  list(year = 0:year, amount = future)
}
```

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:

```r
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.

```r
fv_table <- function(present, rate, year) {
  future = present * (1 + rate)^(0:year)
  data.frame(year = 0:year, amount = future)
}
```

11 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.

11.1 R Expressions

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

11.1.1 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.

11.1.2 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 kind of expression plays an important role but it is typically used together with other programming structures (e.g. functions, conditionals, loops).

11.1.3 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 have no 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.

11.1.4 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 expression 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.

12 Conditionals: If-Else

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.

12.1 Motivation

So far we have extensively used a simple savings-investing scenario in which $1000 are deposited in a savings account 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?

12.1.1 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:

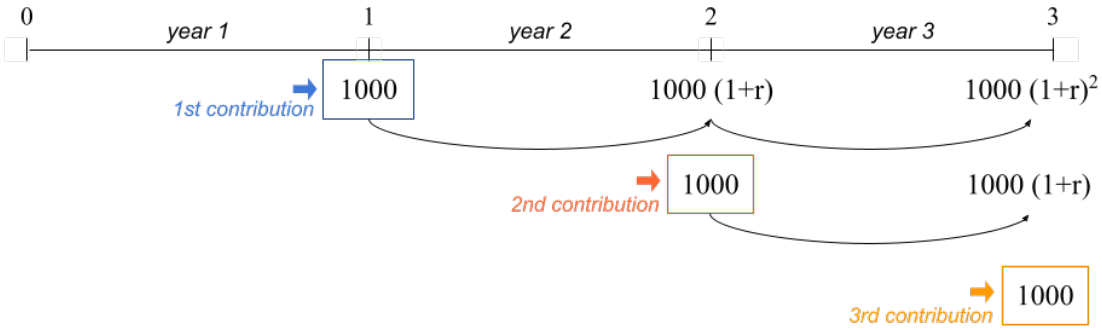


Figure 12.1: Timeline of an ordinary annuity: contributions made at the end of each year.

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 way, we may type the following 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
```

12.1.2 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 let's assume 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
```

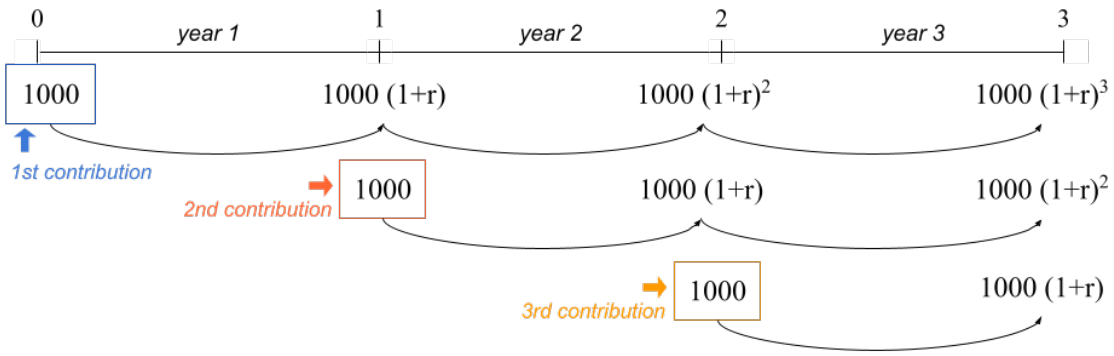


Figure 12.2: Timeline of an annuity due: contributions made at the beginning of each year.

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

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

- 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 to obtain the amounts at the end of every year:

```
# 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
```

12.2 Conditionals

Let's do a quick recap. We know there are two types of annuity:

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

And we also have their corresponding future value formulas:

- **future value of an ordinary annuity**:

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

- **future value of an annuity due**:

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

Let us now consider the possibility of having an input object (or variable) called `type` that can take two values: `type = "ordinary"` or `type = "due"`. Depending on the value of `type`, we could write code to compute the appropriate kind of annuity, something more or less like the code below:

```
# 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
```

12.2.1 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 precending 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

```r
# 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
```

12.2.2 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:

```r
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`:

```r
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.

```r
type <- "ordinary"

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

Let's dissect the conditional statement

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.

```r
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 → single TRUE
                                      → single FALSE
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 {                  What to do if condition is TRUE
  # compute annuity due
}                         What to do if condition is FALSE
```

12.2.3 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)
}
```

12.2.4 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"
```

12.3 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
```

12.3.1 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:

```
# 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:

```
# 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"
```

12.4 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:

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

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

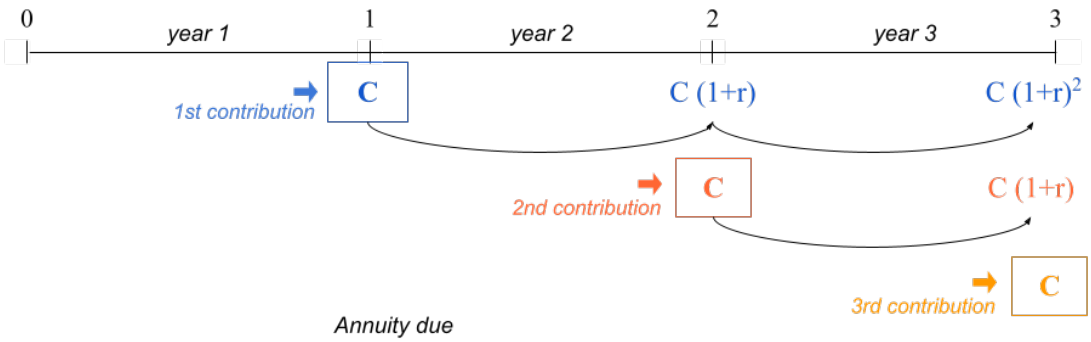


Figure 12.3: Timeline of an ordinary annuity

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

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:

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

13 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.

13.1 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.

13.1.1 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

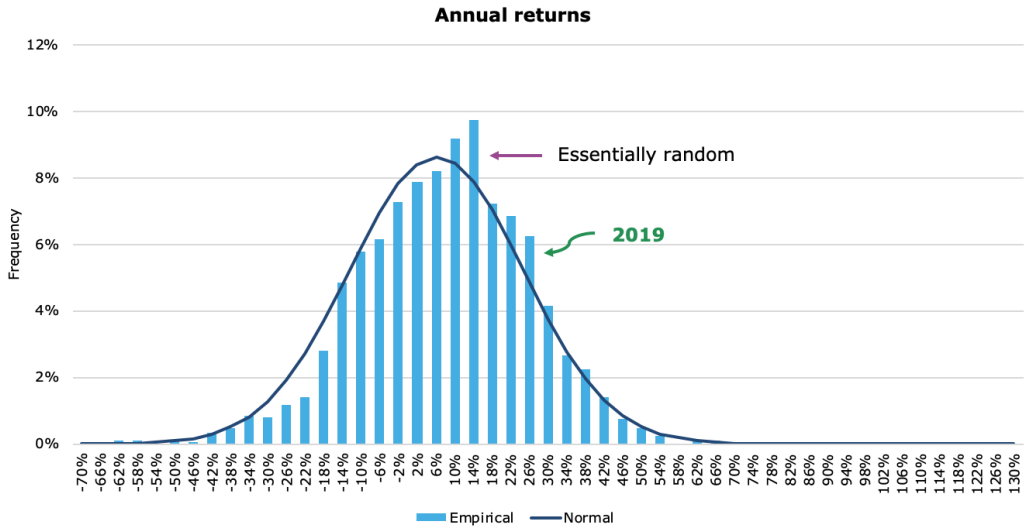


Figure 13.1: Distribution of Annual Returns (by Joachim Klement)

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.

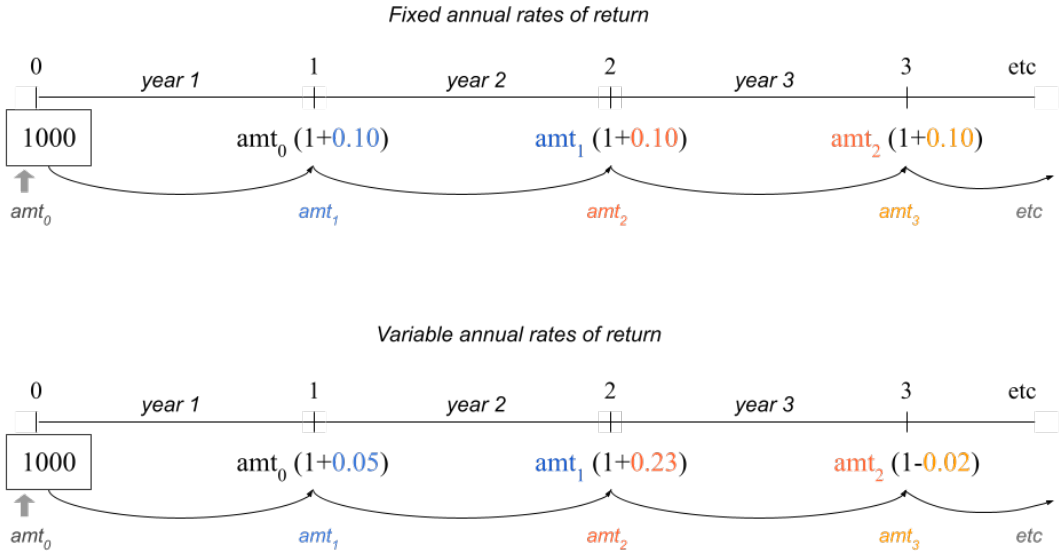


Figure 13.2: Rates of Return: Fixed -vs- Variable

13.2 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)

```r
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 $\sigma = 0.18$

```r
rates = rnorm(n = 3, mean = 0.10, sd = 0.18)
rates
```

```
[1] 0.04968742 0.07093758 0.04769262
```

13.2.1 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 ($t = 1, 2, 3$)

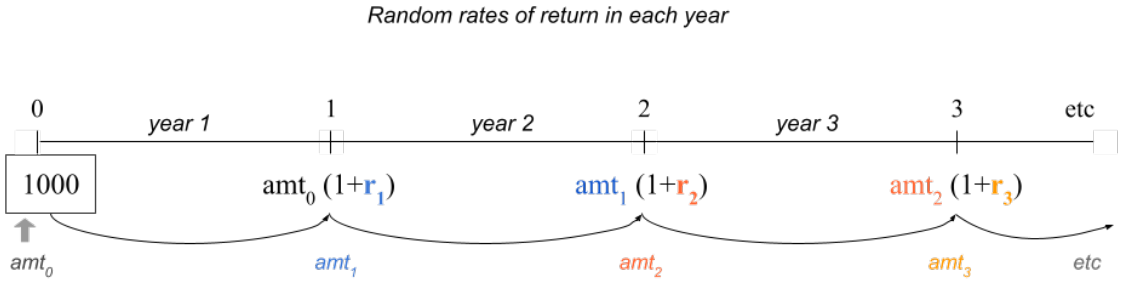


Figure 13.3: Random rates of return following a normal distribution

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

```r
# 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?

13.2.2 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?

13.3 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.

```r
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:

```r
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.

13.4 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

```r
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
```

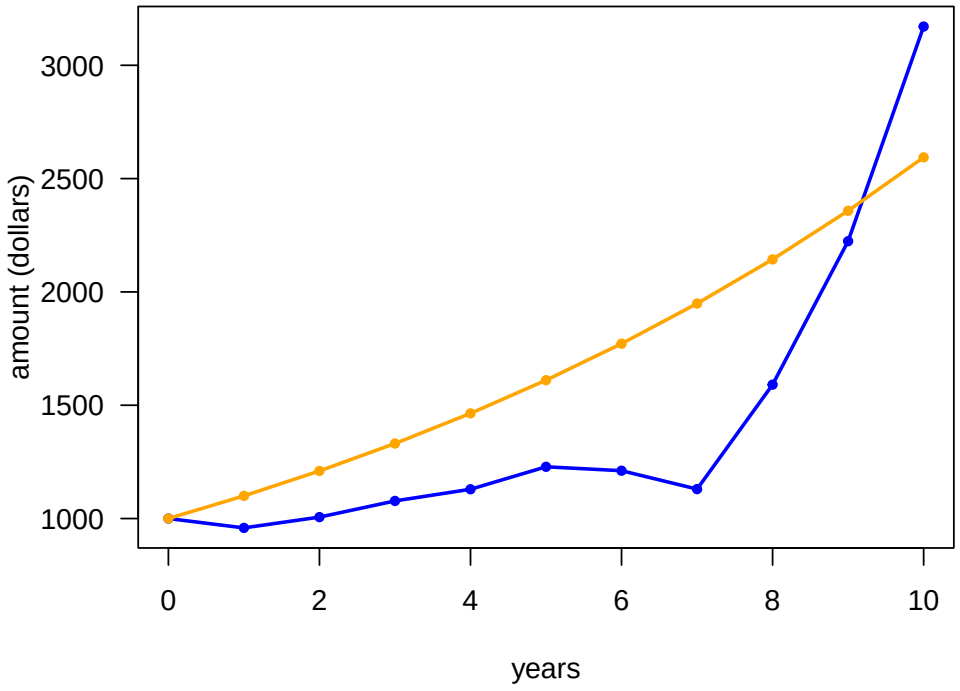
There are a couple of important things worth noticing:

- 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)
```



13.5 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 element `vec[j]`, adding 1 to it, and printing the outcome with `print()` so that we 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
}
```

13.5.1 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. For example, in the following code the auxiliary iterator is `i` and the vector of times is the numeric sequence `1:5`.

```
value <- 2

for (i in 1:5) {
  value <- value * 2
  print(value)
}
```

```
[1] 4
[1] 8
[1] 16
[1] 32
[1] 64
```

As you can tell, `print()` is used to display the updated `value` at each iteration. This print statement is used here for illustration purposes; in practice you rarely need to print any output in a loop.

The vector of *times* does NOT have to be a numeric vector; it can be **any** vector. The important thing about this vector is its length, which R uses to determine the number of iterations in the `for` loop.

```
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
```

13.5.2 For Loops and Next statement

Sometimes we need to skip a certain 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
```

13.5.3 For Loops and Break statement

In addition to skipping certain iterations, 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
```

13.5.4 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
```

13.5.5 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.

14 Iterations: While Loop

In the previous chapter you got introduced to your first iterative construct: `for` loops. You use this type of loop when you know how many times a given computation needs to be repeated. But what about those situations in which you have to repeat a process without necessarily knowing how many times this repetition will take place? This is where we need a more general type of loop, namely, the **while** loop.

14.1 Motivation

Let's begin with the same toy example discussed in the previous chapter. Say you have a vector `vec <- c(3, 1, 4)`, and suppose you want to obtain a new vector `vec2` that adds 1 to every element in `vec`. You know that this can easily be achieved using vectorized code:

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

vec2 <- vec + 1
vec2
```

```
[1] 4 2 5
```

Again, in order to explain the concept of a `while` loop, I am going to ask you to pretend that R does not have vectorized code.

What would you need to do in order to add 1 to the elements in `vec`? As we mentioned in the preceding chapter, you would need to do something like this:

```
# new vector to be updated
vec2 <- rep(0, 3)

# repetitive steps
vec2[1] <- vec[1] + 1
vec2[2] <- vec[2] + 1
vec2[3] <- vec[3] + 1
```

That is, take the first element in `vec` and add 1, then take the second element in `vec` and add 1, and finally the third element in `vec` and add 1. Basically, you are performing the same type of operation several times: selecting an element in `vec` and adding 1 to it. But there's a lot of (unnecessary) repetition.

We've seen how to write a `for` loop to take care of the addition computation. Alternatively, we can also approach this problem from a slightly different perspective by considering a **stopping condition** to decide when to terminate the repetitive process of adding 1 to the elements in `vec`.

What stopping condition can we use? Well, one example may involve: "let's keep selecting a single element in `vec` and adding 1 to it, until we *exhaust* all elements in `vec`". In other words, let's keep iterating until we reach the last element in `vec`.

As usual, the first step involves identifying the common structure of the repetitive steps. We can make the repetitive code a bit more general by referring to each position as `pos`:

```
vec2[pos] <- vec[pos] + 1
```

Once we have the correct abstraction for the code that needs to be repetead, then we can encapsulate it with a `while` loop. Let me first show you an example and then we'll examine it in detail:

```
# input vector
vec <- c(3, 1, 4)

# initialize output vector
vec2 <- rep(0, 3)

# declare auxiliary iterator
pos <- 1

# while loop
while (pos <= length(vec)) {
  vec2[pos] <- vec[pos] + 1
  pos <- pos + 1  # update iterator
}
```

The first thing that I should mention is that writing an R `while` loop is a bit more complex than writing a `for` loop. The complexity has to do with some of the things that R does not automatically take care of in a `while` loop.

One main difference between a `for` loop and a `while` loop is that in the latter we must explicit declare the auxiliary iterator and give it an initial value: `pos <- 1`.

Next we have the `while` statement. This statement is technically a function, but I prefer to think of it, and call it, a statement (like the `if` and the `for` statements). What you pass inside parenthesis of the `while` declaration is a **condition**. This is basically any piece of code that R will evaluate and coerce it into a logical condition that is `TRUE` or `FALSE`. The `while` loop iterates as long as the condition is `TRUE`. If the condition becomes `FALSE` then the loop is terminated.

The code of the repetitive steps consists of an R expression `{ ... }`. This is where we indicate what to do at each step. Often, an important piece of code that we need to include here involves increasing the value of the auxiliary iterator: `pos <- pos + 1`. In this particular example, if we don't increase the iterator `pos`, the loop would iterate forever.

Note that the condition is the **stopping condition**, which in turn depends on the auxiliary iterator: `pos <= length(vec)`. You can think of this condition as: "let's keep iterating until we reach the last element in `vec`".

14.2 Anatomy of a While Loop

Now that you've seen a first example of a `while` loop, I can give you a generic template for this kind of iterative construct:

```
iterator <- initial

while (condition) {
  do_something
  iterator <- iterator + 1
}
```

What's going on?

- you need to declare the auxiliary iterator with some initial value
- you declare the `while` statement by giving a condition inside parenthesis
- the condition must be a piece of code that gets evaluated into a single logical value: `TRUE` or `FALSE`
- the condition is used as the stopping condition: if the condition is `TRUE` the loop keeps iterating; when the condition becomes `FALSE` the loop is terminated
- we use an R compound expression `{ ... }` to embrace the code that will be repeated at each iteration

- inside the loop, you typically need to increase the value of the `iterator`; even if the `condition` does not depend on the `iterator`, it's a good idea to keep track of the number of iterations in the loop

14.3 Another Example

Let's see a more interesting example.

Say we generate a vector with 10 different integer numbers between 1 and 100, arranged in increasing order. To make things more interesting, we are going to generate these numbers in a random way using the `sample.int()` function that allows us to get a random sample of `size = 10` integers, sampling without replacement (`replace = FALSE`):

```
set.seed(234)  # for replication purposes

# vector of 10 random integers between 1 and 100
random_numbers = sample.int(n = 100, size = 10, replace = FALSE)
random_numbers = sort(random_numbers)
random_numbers
```

```
 [1]  1 18 31 34 46 56 68 92 97 98
```

What are we going to do with these `random_numbers`? We are going to compute a cumulative sum until its value becomes greater than 100. And we are going to consider these two questions:

- What is the value of the cumulative sum?
- How many numbers were added to reach the sum's value?

My recommendation is to always start with baby steps. Simply put, start writing code for a couple of concrete steps so that you understand what kind of computations will be repeated, and what things they have in common:

```
# initialize output sum
total_sum = 0

# accumulate numbers
total_sum = total_sum + random_numbers[1]
total_sum = total_sum + random_numbers[2]
total_sum = total_sum + random_numbers[3]
# ... keep adding numbers as long as total_sum <= 100
```

There are three important aspects to keep in mind:

- we need an object to store the cumulative sum: `total_sum`
- we need an iterator to move through the elements of `random_numebrs`
- and of course we need to determine a stopping-condition: `total_sum <= 100`

Here's the code:

```
# initialize object of cumulative sum
total_sum = 0

# declare iterator
pos = 0

# repetitive steps
while (total_sum <= 100) {
  pos = pos + 1
  total_sum = total_sum + random_numbers[pos]
}

# what is the value of the cumulative sum?
total_sum
```

```
[1] 130
```

```
# how many iterations were necessary?
pos
```

```
[1] 5
```

Observe that in this example, we declared `pos = 0`. Then, at each iteration, we increase its value `pos = pos + 1`, and then we added `random_numbers[pos]` to the previous `total_sum` value, effectively updating the cumulative sum.

For comparison purposes, consider this other `while` loop. It looks extremely similar to the preceding loop but there is an important difference.

```
# initialize object of cumulative sum
total_sum = 0
```

```
# declare iterator
pos = 1

# repetitive steps
while (total_sum <= 100) {
  total_sum = total_sum + random_numbers[pos]
  pos = pos + 1
}

# what is the value of the cumulative sum?
total_sum
```

```
[1] 130
```

```
# how many iterations were necessary?
pos
```

```
[1] 6
```

Can you see the difference between these two `while` loops?

In this second loop, the iterator is declared as `pos = 1`, and its value is increased after updating the cumulative sum. While `total_sum` has the correct value, `pos` does not indicate anymore the right number of iterations.

I wanted to show you this second example to make a point: in a `while` loop you not only need to declare the iterator before entering the loop, but you also need to carefully think what initial value you'll use, as well as when to increase its value inside the loop. Some times the very first thing to do in each iteration is to increase the value of the iterator; some times that's the last thing to do. It all depends on the specific way you are approaching a given iterative task.

14.3.1 While Loops and Next statement

Sometimes we need to skip a certain iteration if a given condition is met, this can be done with the `next` statement. The following code chunk contains an abstract template that uses `next`:

```
iterator <- initial

while (condition) {
  do_something
  if (skip_condition) {
    next
  }
  iterator <- iterator + 1
}
```

As a less abstract example, let's bring back the while loop of the cumulative sum of random numbers, but this time say we want to skip any numbers between 30 and 39. This means we need an `if-else` statement to check whether a given element of `random_numbers` is between 30 and 39. If yes, we should skip that element and go to the `next` iteration. Here is how to do it:

```
total_sum = 0
pos = 0

while (total_sum <= 100) {
  pos = pos + 1
  if (random_numbers[pos] %in% 30:39) {
    next
  }
  total_sum = total_sum + random_numbers[pos]
}

total_sum
```

```
[1] 121
```

```
pos
```

```
[1] 6
```

14.3.2 While Loops and Break statement

In addition to skipping certain iterations, sometimes we need to stop a loop from iterating if a given condition is met. This can be done with the `break` statement, which is shown below in an abstract code template:

```
while (condition) {
  expr1
  expr2
  if (stop_condition) {
    break
  }
  expr3
  expr4
}
```

Let's go back to the cumulative sum example. Say we want to stop iterating if numbers are greater than or equal to 40. Like we did previously, we need again an `if-else` statement to check whether a given element of `random_numbers` is greater than or equal to 40. If yes, we stop the loop from iterating by using the `break` statement as follows:

```
total_sum = 0
pos = 0

while (total_sum <= 100) {
  pos = pos + 1
  if (random_numbers[pos] >= 40) {
    break
  }
  total_sum = total_sum + random_numbers[pos]
}

total_sum
```

```
[1] 84
```

```
pos
```

```
[1] 5
```

15 More About Functions

In this chapter you will learn more aspects about creating functions in R.

15.1 Functions Recap

Consider a toy example with a function that squares its argument:

```
square = function(x) {
  x * x
}
```

- the function name is `"square"`
- it has one argument: `x`
- the function body consists of one simple expression
- it returns the value `x * x`

`square()` works like any other function in R:

```
square(10)
```

```
[1] 100
```

In this case, `square()` is also vectorized:

```
square(1:5)
```

```
[1]  1  4  9 16 25
```

Why is `square()` vectorized?

Once defined, functions can be used in other function definitions:

```
sum_of_squares = function(x) {
  sum(square(x))
}
sum_of_squares(1:5)
```

```
[1] 55
```

15.1.1 Simple Expressions

Functions with a body consisting of a **simple expression** can be written with no braces (in one single line!):

```
square = function(x) x * x
square(10)
```

```
[1] 100
```

However, as a general coding rule, you should get into the habit of writing functions using braces.

15.1.2 Nested Functions

We can also define a function inside another function:

```
getmax = function(a) {
  # nested function
  maxpos <- function(u) which.max(u)
  # output
  list(position = maxpos(a),
       value = max(a))
}
getmax(c(2, -4, 6, 10, pi))
```

```
$position
[1] 4

$value
[1] 10
```

15.2 Function Output

The value of a function can be established in two ways:

- As the last evaluated simple expression (in the body of the function)
- An explicitly **returned** value via `return()`

Here's a basic example of a function in which the output is the last evaluated expression:

```
add = function(x, y) {
  x + y
}

add(2, 3)
```

```
[1] 5
```

Here's another version of `add()` in which the output is the last evaluated expression:

```
add = function(x, y) {
  z = x + y
  z
}

add(2, 3)
```

```
[1] 5
```

Be careful with the form in which the last expression is evaluated:

```
add = function(x, y) {
  z = x + y
}

add(2, 3)
```

In this case, it looks like `add()` does not work. If you run the previous code, nothing appears in the console. Can you guess why? To help you answer this question, assign the invocation to an object and then print the object:

```
why <- add(2, 3)
why
```

`add()` does work. The issue has to do with the form of the last expression. Nothing gets displayed in the console because the last statement `z <- x + y` is an assignment (that does not print anything).

15.2.1 The `return()` command

More often than not, the `return()` command is included to explicitly indicate the output of a function:

```
add = function(x, y) {
  z <- x + y
  return(z)
}

add(2, 3)
```

```
[1] 5
```

I've seen that many users with previous programming experience in other languages prefer to use `return()`. The main reason is that most programming languages tend to use some sort of *return* statement to indicate the output of a function.

So, following good language-agnostic coding practices, we also recommend that you use the function `return()`. In this way, any reader can quickly scan the body of your functions and visually locate the places in which a *return* statement is being made.

15.2.2 White Spaces

- Use a lot of it
- around operators (assignment and arithmetic)
- between function arguments and list elements
- between matrix/array indices, in particular for missing indices
- Split long lines at meaningful places

Avoid this

```
a<-2
x<-3
y<-log(sqrt(x))
3*x^7-pi*x/(y-a)
```

Much Better

```
a <- 2
x <- 3
y <- log(sqrt(x))
3*x^7 - pi * x / (y - a)
```

Another example:

```
# Avoid this
plot(x,y,col=rgb(0.5,0.7,0.4),pch='+',cex=5)

# okay
plot(x, y, col = rgb(0.5, 0.7, 0.4), pch = '+', cex = 5)
```

Another readability recommendation is to limit the width of line: they should be broken/wrapped around so that they are less than 80 columns wide

```
# lines too long
histogram <- function(data){
hist(data, col = 'gray90', xlab = 'x', ylab = 'Frequency', main = 'Histogram of x')
abline(v = c(min(data), max(data), median(data), mean(data)),
col = c('gray30', 'gray30', 'orange', 'tomato'), lty = c(2,2,1,1), lwd = 3)
}
```

Lines should be broken/wrapped around so that they are less than 80 columns wide

```
# lines with okay width
histogram <- function(data) {
  hist(data, col = 'gray90', xlab = 'x', ylab = 'Frequency',
       main = 'Histogram of x')
  abline(v = c(min(data), max(data), median(data), mean(data)),
         col = c('gray30', 'gray30', 'orange', 'tomato'),
         lty = c(2,2,1,1), lwd = 3)
}
```

- Spacing forms the second important part in code indentation and formatting.
- Spacing makes the code more readable
- Follow proper spacing through out your coding
- Use spacing consistently

```
# this can be improved
stats <- c(min(x), max(x), max(x)-min(x),
  quantile(x, probs=0.25), quantile(x, probs=0.75), IQR(x),
  median(x), mean(x), sd(x)
)
```

Don't be afraid of splitting one long line into individual pieces:

```
# much better
stats <- c(
  min(x),
  max(x),
  max(x) - min(x),
  quantile(x, probs = 0.25),
  quantile(x, probs = 0.75),
  IQR(x),
  median(x),
  mean(x),
  sd(x)
)
```

You can even do this:

```
# also OK
stats <- c(
  min    = min(x),
  max    = max(x),
  range  = max(x) - min(x),
  q1     = quantile(x, probs = 0.25),
  q3     = quantile(x, probs = 0.75),
  iqr    = IQR(x),
  median = median(x),
  mean   = mean(x),
  stdev  = sd(x)
)
```

- All commas and semicolons must be followed by single whitespace

- All binary operators should maintain a space on either side of the operator
- Left parenthesis should start immediately after a function name
- All keywords like `if`, `while`, `for`, `repeat` should be followed by a single space.

All binary operators should maintain a space on either side of the operator

```
# NOT Recommended
a=b-c
a = b-c
a=b - c;

# Recommended
a = b - c
```

All binary operators should maintain a space on either side of the operator

```
# Not really recommended
z <- 6*x + 9*y

# Recommended (option 1)
z <- 6 * x + 9 * y

# Recommended (option 2)
z <- (7 * x) + (9 * y)
```

Left parenthesis should start immediately after a function name

```
# NOT Recommended
read.table ('data.csv', header = TRUE, row.names = 1)

# Recommended
read.table('data.csv', header = TRUE, row.names = 1)
```

All keywords like `if`, `while`, `for`, `repeat` should be followed by a single space.

```
# not bad
if(is.numeric(object)) {
  mean(object)
}

# much better
if (is.numeric(object)) {
```

```
  mean(object)
}
```

15.3 Indentation

- Keep your indentation style consistent
- There is more than one way of indenting code
- There is no "best" style that everyone should be following
- You can indent using spaces or tabs (but don't mix them)
- Can help in detecting errors in your code because it can expose lack of symmetry
- Do this systematically (RStudio editor helps a lot)

Don't write code like this:

```
# no indentation
# Don't do this!
if(!is.vector(x)) {
stop('x must be a vector')
} else {
if(any(is.na(x))){
x <- x[!is.na(x)]
}
total <- length(x)
x_sum <- 0
for (i in seq_along(x)) {
  x_sum <- x_sum + x[i]
}
x_sum / total
}
```

Instead, write with indentation

```
# better with indentation
if (!is.vector(x)) {
  stop('x must be a vector')
} else {
  if (any(is.na(x))) {
    x <- x[!is.na(x)]
```

```
  }
  total <- length(x)
  x_sum <- 0
  for (i in seq_along(x)) {
    x_sum <- x_sum + x[i]
  }
  x_sum / total
}
```

There are several Indenting Styles

```
# style 1
find_roots <- function(a = 1, b = 1, c = 0)
{
  if (b^2 - 4*a*c < 0)
  {
    return("No real roots")
  } else
  {
    return(quadratic(a = a, b = b, c = c))
  }
}
```

My preferred style is like this:

```
# style 2
find_roots <- function(a = 1, b = 1, c = 0) {
  if (b^2 - 4*a*c < 0) {
    return("No real roots")
  } else {
    return(quadratic(a = a, b = b, c = c))
  }
}
```

Benefits of code indentation:

- Easier to read
- Easier to understand
- Easier to modify
- Easier to maintain
- Easier to enhance

15.3.1 Meaningful Names

Choose a consistent naming style for objects and functions

- `someObject` (lowerCamelCase)
- `SomeObject` (UpperCamelCase)
- `some_object` (underscore separation)
- `some.object` (dot separation)

Avoid using names of standard R objects, for example:

- `vector`
- `mean`
- `list`
- `data`
- `c`
- `colors`

If you're thinking about using names of R objects, prefer something like this

- `xvector`
- `xmean`
- `xlist`
- `xdata`
- `xc`
- `xcolors`

Better to add meaning like this

- `mean_salary`
- `input_vector`
- `data_list`
- `data_table`
- `first_last`
- `some_colors`

Prefer Pronounceable Names

```
# avoid cryptic abbreviations
DtaRcrd102 <- list(
  nm = 'John Doe',
  bdg = 'Valley Life Sciences Building',
  rm = 2060
)
```

```
# prefer pronounceable names
Customer <- list(
  name = 'John Doe',
  building = 'Valley Life Sciences Building',
  room = 2060
)
```

15.3.2 Syntax: Parentheses

Use parentheses for clarity even if not needed for order of operations.

```
a <- 2
x <- 3
y <- 4

a/y*x

# better
(a / y) * x
```

another example

```
# confusing
1:3^2
```

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

```
# better
1:(3^2)
```

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

15.4 Recommendations

- Functions are tools and operations
- Functions form the building blocks for larger tasks
- Functions allow us to reuse blocks of code easily for later use
- Use functions whenever possible
- Try to write functions rather than carry out your work using blocks of code
- Don't write long functions
- Ideal length between 2 and 4 lines of code
- No more than 10 lines
- Should not exceed the size of the text editor window
- Functions shouldn't be longer than one visible screen (with reasonable font)
- Rewrite long functions by converting collections of related expression into separate functions
- Smaller functions are easier to debug, easier to understand, and can be combined in a modular fashion
- Separate small functions
- Smaller functions are easier to reason about and manage
- Smaller functions are easier to test and verify they are correct
- Smaller functions are more likely to be reusable
- Think about different scenarios and contexts in which a function might be used
- Can you generalize it?
- Who will use it?
- Who is going to maintain the code?
- Use descriptive names
- Readers (including you) should infer the operation by looking at the call of the function
- be modular (having a single task)
- have meaningful name
- have a comment describing their purpose, inputs and outputs
- Functions should not modify global variables
- except connections or environments
- should not change global `par()` settings

Part V

Import & Export

16 Introduction

This part of the book is dedicated to describe the various mechanisms available in R to import and export "data" and other "resources". Here I'm using the term "resources" in an informal sense to refer to various types of files such as script files, binary files, text files, image files, and things like that. Likewise, I'm using the term "data" in a loosely way to indicate R data objects (e.g. vectors, arrays, lists, data-frames), graphics, code, as well as content of any kind of file.

You should know upfront that there's a wide range of ways and options to import/export "data" in R. To go beyond the topics discussed in the book, and to know more about many of the technicalities behind importing and exporting resources in R, the authoritative document to look at is the manual **R Data Import/Export** available at:

https://cran.r-project.org/doc/manuals/r-release/R-data.html

16.1 Importing and Exporting Resources

R has a large number of functions and packages to import and to export a wide variety of resources and files (e.g. script files, binary files, text files, image files).

In the following figure, I'm depicting a conceptual diagram to show a couple of common examples for importing and exporting some resources.

Perhaps the most common type of importing operation is when we have some data table that we want to read in. For instance, we may have a data file called `table.csv`, located in our session's working directory. Assuming that the format of this file is a comma-separated-value (CSV), we could try to import it in R with the help of the `read.csv()` function:

```
# hypothetical data-table importing example
dat = read.csv(file = "table.csv")
```

Likewise, we could have one or more functions in an R script file called `script.R` that we want to import. Assuming that this file is also in our session's working directory, we could use the function `source()` to source-in such functions:

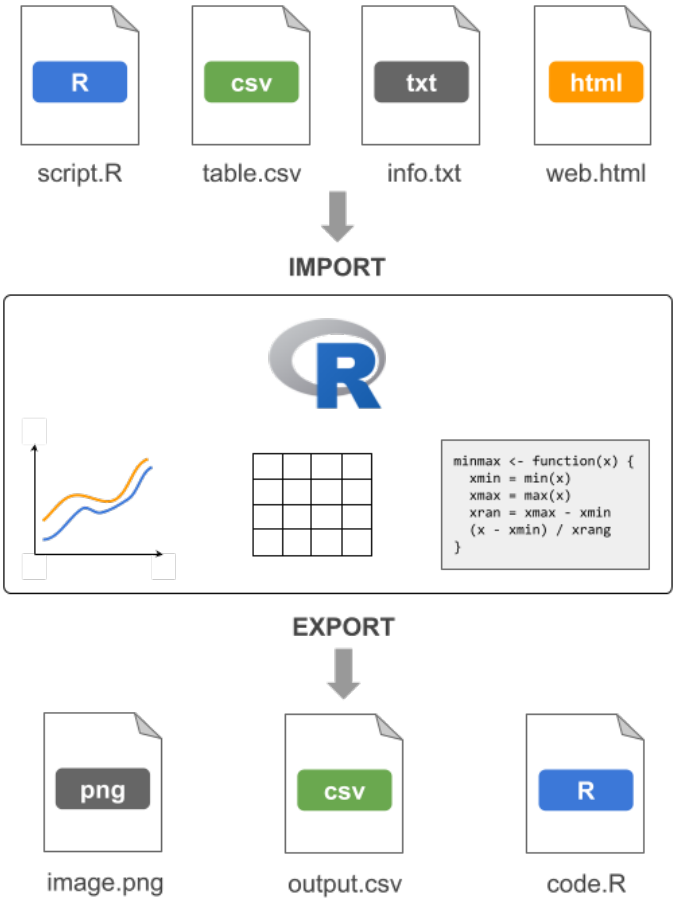


Figure 16.1: Conceptual diagram illustrating some importing and exporting resources.

```
# hypothetical R-script importing example
source(file = "script.R")
```

What about exporting resources from R to some external resource? For example, consider a typical situation in which we have a data object, say a data frame, called `tbl` that we want to export into a text file. To be more precise, say we want to export the data frame `tbl` to a CSV file called `output.csv` to be located in our session's directory. In this case, we may want to use the function `write.csv()`

```
# hypothetical data-table exporting example
write.csv(x = tbl, file = "output.csv")
```

16.1.1 Behind import/export functions

Before I tell you more about some of the common—and not so common—import and export operations available in R, I first need to take you down a rabbit hole to explain some technicalities behind the related functions for importing and exporting resources in R.

When importing information from an external file, we need to use a certain data-import function, e.g. `read.table()`, `readLines()`, `scan()`, etc. As you might expect, each data-import function has specific arguments that let you choose further options for how R should handle this task. Despite their different arguments, all these functions have one thing in common which is the mandatory argument: the name of the input file.

The same thing can be said about the data-export functions such as `write.table()`, `writeLines()`, `cat()`, `png()`, etc. Even though each of these functions differs in its structure and arguments, they all have one thing in common: the mandatory argument consisting of the name of the output file.

The figure below illustrates both types of generic situations: importing from an input file, and exporting to an output file. In this diagram, I'm using the names `import()` and `export()` as generic labels for data-import and data-export functions.

As I said, the primary argument to the import and export functions is the "name" of the external file. Here the term "name" refers to the *file path*. In other words, the "name" is not just the name of the file but also its location on the file-system where the file is located.

The external file can be located in your computer or somewhere else, like in the "cloud", which at the end of the day it's going to be another computer remotely located.

Another thing to notice in the diagram has to do with the so-called **connections**. What I'm trying to indicate in the diagram is that all import and export functions use—under the hood—an internal `connection()` function which plays a fundamental role in this type of operations. What is a connection? Let's find out.

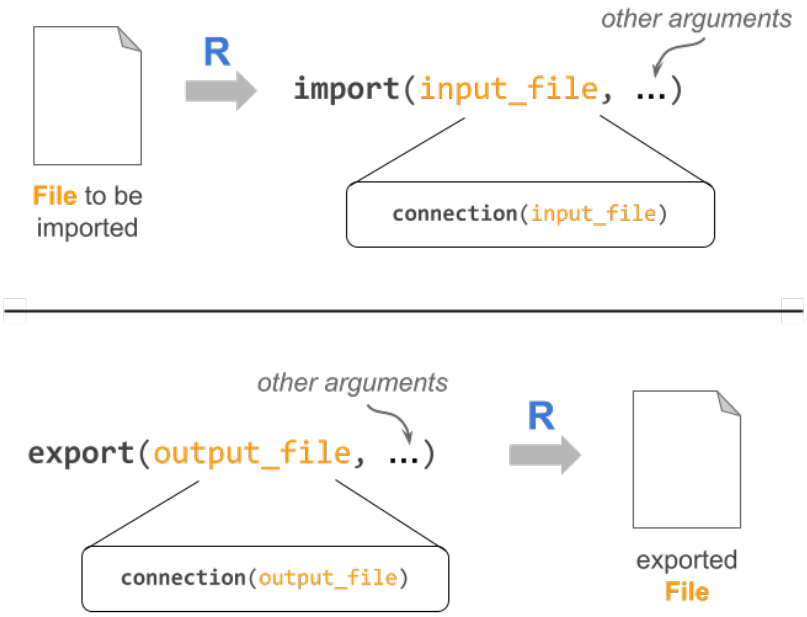


Figure 16.2: Conceptual diagram illustrating generic import and export tasks.

16.2 Connections

In order for R to be able to import external resources (e.g. data files, script files, image files), it needs to have a way to communicate with the outside world. The same applies to any resource-exporting activity: R needs to open its doors to let resources enter and exit its territory.

The mechanism used by R to establish channels of communication with the outside world is given by the so-called **connections**. To explain this concept let me give you an analogy.

In this analogy I'm going to play the role of R. My family, friends, coworkers and students will play the generic role of resources. For example, if I want to communicate with my teaching assistants (TA), I need a mechanism to reach out to them. One way to communicate with my TAs could be talking to them in-person. Another way of communication could be via email (or old-fashion mail). Another possibility may involve me sending them a text-message, or maybe calling them by phone. In summary, there are several ways for me to *connect* with my TAs. This is precisely the main idea behind R connections.

Formally speaking, a connection is the mechanism used by R to establish a line of communication to external resources or files. Technically, connections are implemented by a set of functions, such as `file()`, `url()`, and `gzfile()`—to mention but a few—that allow us to

create, open and close connections. You can find more information about these functions in their help documentation page:

```
help(connections)
```

Let's go back to my analogy. Say I want to reach out to one of my teaching assistants, and I decide to communicate with a text message sent from my cellphone. In this case we can say that the connection consists of sending a text message. If you think about this exporting operation I need to:

- use a text-messaging application; this would be the equivalent of an `export()` function
- specify the phone number to which my message will be sent; this would be the equivalent of the *file path*
- write the content of the message; this would be the equivalent of the "data" to be exported

Now, the text-messaging application will take the phone number and "do its magic" to send my message. This "magical" part is the equivalent of the internal connection function.

It turns out that you rarely need to explicitly call any of the connection functions in R. If you were using other programming languages, chances are you may very well need to explicitly call a connection function (or its equivalent) to tell the program the kind of operation that you want to perform: for instance open a file in reading mode, or open a file in writing mode, or closing a file. Most of the time in R, though, we don't need to specify this kind of low-level communication.

So why bother talking about connections?

I just want you to know that behind any function that allows you to import from a file, and export to a file, there is a connection function. Typically you don't need to do anything with these internal functions. But if you want or need to take full control over all the details in an importing/exporting operation, connection functions are there for you.

17 Importing Data Tables

The most common type of data-import operation in R has to do with importing data tables. Because of this, I have decided to exclusively dedicate this chapter to review this type of operation. In a nutshell, I describe various standard ways to import tabular data. By "standard" I mean using base functions such as `read.table()` and friends. This is in contrast to alternative functions from packages such as `"readr"`, and other packages.

17.1 Motivation

Suppose we have data about the accounts of an individual's portfolio (see table below). There are five different accounts, the type of bank (brick-and-mortar or online) associated to the account, the annual rate of return, and the balance amount (in dollars).

account	bank	rate	balance
savings	brick-n-mortar	0.020	1000
money market	online	0.025	2000
certificate	brick-n-mortar	0.030	3000
brokerage	online	0.070	5000
retirement	online	0.050	9000

A data table like this one could be stored in various types of files. For example, it could be stored in a *Comma Separated Value* (CSV) file, which is a very common kind of text file format used to store tables. Another possible way in which this table could be stored is in a spreadsheet (e.g. Excel, Google Sheets, Mac Numbers).

17.1.1 Data in Text Files

Let's assume that the table is stored in a text file. Because the term text file is used in slightly different ways, let me be a bit more specific about what I mean by *text* file:

- By text file I mean a plain text file that can be read and manipulated with a text editor (not to be confused with a word-processor)

- Plain text as an umbrella term for any file that is in a human-readable form, with comon file extensions such as `.txt`, `.csv`, `.xml`, `.html`.
- Text file implying that its content is stored as a sequence of characters
- Each character stored as a single byte of data
- Data is arranged in rows, with several values stored on each row

17.2 Character Delimited Text Files

A common way to store data tables is via text files. But how is this actually done? How can a data table be stored in a text file?

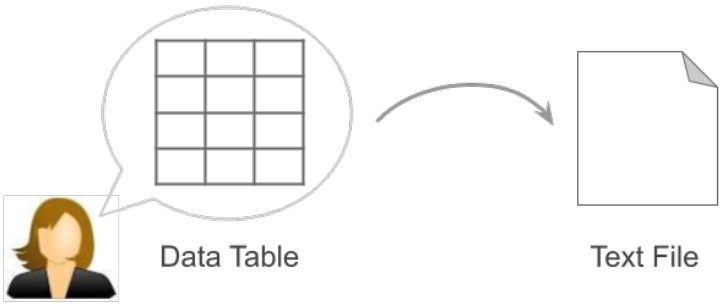


Figure 17.1: How are data tables stored in text files?

You may not have thought about this before. This was certainly the case for me back when I was an undergraduate student. I could perfectly picture a data table in my head, with a bunch of rows and columns forming its grid-like structure. But I never considered a table's storage format in a text file.

Think about it: a table is made of cells that come from the intersection of rows and columns. So the two fundamental questions when storing data in a text file are: 1) how to represent rows?, and 2) how to represent columns? Or in other words: how to convey the notion of cells in a text file?

The answer to this question comes from realizing what a data cell does. A cell is basically a placeholder for a data value. Cells let us separate or delimit one data value from another. So all we have to do is to come up with a **delimiter** or **separator** to differentiate from one data point to the next one. The way this is done is by choosing a certain character as the delimiter. As for the notion of rows, we can simply use newlines.

Here is an example of how the accounts data table can be store in a text file, using commas `","` as the character to separate values:

```
account,bank,rate,balance
savings,bricknmor,0.020,1000
monmarket,online,0.025,2000
certificate,bricknmor,0.030,3000
brokerage,online,0.070,5000
retirement,online,0.050,9000
```

17.2.1 Common Delimiters

As you can tell, the idea of storing data tables via text files is very simple and clever. More formally, this storage option is commonly referred to as **field-delimiter** formats. The term "field" is a synonym of *variable* or *column*. The term "delimiter" indicates that a certain character is used to delimit the values of every column.

Common examples of such delimiters are blank space characters `" "`, tab characters `"\t"`, or comma characters `","`.

Delimiter	Description
`" "`	white space
`","`	comma
`"\t"`	tab
`";"`	semicolon

As a matter of fact, you can come up with your own delimiter. For instance, I can use vertical bars `"|"` as delimiters:

```
account|bank|rate|balance
savings|bricknmor|0.020|1000
monmarket|online|0.025|2000
certificate|bricknmor|0.030|3000
brokerage|online|0.070|5000
retirement|online|0.050|9000
```

Or even use double colons as delimiters:

```
account::bank::rate::balance
savings::bricknmor::0.020::1000
monmarket::online::0.025::2000
certificate::bricknmor::0.030::3000
brokerage::online::0.070::5000
retirement::online::0.050::9000
```

To be honest, if you have to choose a certain character for delimiting purposes, I would recommend not to be creative, and instead to stick with one of the conventional delimiters. Below are some examples of what the content of a text file—storing the accounts table—could look like when using different delimiters.

Space Delimited

Example of a **space** delimited file (common file extension `.txt`)

```
account bank rate balance
savings bricknmor 0.020 1000
monmarket online 0.025 2000
certificate bricknmor 0.030 3000
brokerage online 0.070 5000
retirement online 0.050 9000
```

Tab Delimited

Example of a **tab** delimited file (common file extensions `.txt` or `.tsv`)

```
account   bank    rate    balance
savings   bricknmor   0.020   1000
monmarket   online    0.025   2000
certificate   bricknmor   0.030   3000
brokerage   online    0.070   5000
retirement    online    0.050   9000
```

Semicolon Delimited

Example of a **semicolon** delimited file (common file extension `.csv`)

```
account;bank;rate;balance
savings;bricknmor;0.020;1000
monmarket;online;0.025;2000
certificate;bricknmor;0.030;3000
brokerage;online;0.070;5000
retirement;online;0.050;9000
```

17.2.2 Delimiters in Data Values

What happens if the chosen delimiter is actually part of a data value? As an example of this type of situation, let's modify the accounts data table. In particular, suppose that the values in column `balance` contain a comma to better distinguish numbers in thousands of dollars:

account	bank	rate	balance
savings	brick-n-mortar	0.020	1,000
money market	online	0.025	2,000
certificate	brick-n-mortar	0.030	3,000
brokerage	online	0.070	5,000
retirement	online	0.050	9,000

If commas are also used as delimiters, we will have a conflict because the commas in balance values are not intended to be delimiters:

```
account,bank,rate,balance
savings,bricknmor,0.020,1,000
monmarket,online,0.025,2,000
certificate,bricknmor,0.030,3,000
brokerage,online,0.070,5,000
retirement,online,0.050,9,000
```

When something like this happens, to avoid a conflict between the delimiting character and its usage as part of a data value, the convention is to surround the values within quotes (typically double quotes), like this:

```
account,bank,rate,balance
savings,bricknmor,0.020,"1,000"
monmarket,online,0.025,"2,000"
certificate,bricknmor,0.030,"3,000"
brokerage,online,0.070,"5,000"
retirement,online,0.050,"9,000"
```

17.2.3 Fixed-Width Format Files

In addition to character-delimited files, there is another kind of plain text format that can also be used to store data tables: the so-called **fixed-width formats** or *fwf* for short.

In a fixed-width format file, the way in which values are separated is not with a character. Instead, values in a column are given a fixed width expressed in a certain number of characters.

Here's an example of a **fixed width** delimited file (common file extension `.txt`). For sake of illustration, I'm adding a sequence of digits at the top. In real life, the first line of digits is not supposed to be part of the file.

```
12345678901234567890123456789012345
account     bank      rate  balance
savings     bricknmor 0.020 1000
monmarket   online    0.025 2000
certificate bricknmor 0.030 3000
brokerage   online    0.070 5000
retirement  online    0.050 9000
```

All the data of the first column `account` is given a width of 12 characters (i.e. digits `123456789012`). In turn, all the data in column `bank` fits within a width of 10 characters (i.e. digits `3456789012`), and so on.

17.2.4 Summary

To summarize:

- A common way to store data in tabular form is via text files
- To store the data we need a way to separate data values
- Each line represents a "row"
- The idea of "columns" is conveyed with delimiters
- Fields within each line are separated by the delimiter
- Quotation marks are used when the delimiter character occurs within one of the fields

17.3 Importing Data Tables

Now that we have talked about how data tables are stored in text files, and the various formats in which this is done, let's review the set of functions to import tables in R, as well as important considerations to keep in mind when importing data in general.

Considerations before importing a data table in R:

- What is the character used as field delimiter?
- Does the file contain names of columns?
- Does the file contain a column for row names?
- Are there any missing values? If yes, how are missing values encoded?
- Do you want to read in all rows, or just some of them?
- Do you want to read in all columns, or just some of them?
- Do you need to convert delimiter characters? (e.g. from space to comma)
- Can you determine the data-type of each column?
- Are there any uninformative numbers?
- Can you convert those uninformative numbers to informative labels?

17.3.1 Function `read.table()` and friends

The most common way to read and import tables in R is by using `read.table()` and friends such as `read.csv()`, `read.delim()`, etc. If you take a look at the arguments of `read.table()` you'll see that this function has more than 20 of them. Actually, if you inspect the help documentation, `read.table()` comes with the following usage:

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
           dec = ".", row.names, col.names,
           as.is = !stringsAsFactors,
           na.strings = "NA", colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE,
           fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#",
           allowEscapes = FALSE, flush = FALSE,
           stringsAsFactors = default.stringsAsFactors(),
           fileEncoding = "", encoding = "unknown", text,
           skipNul = FALSE)
```

As I mention in the foregoing chapter, the mandatory input for `read.table()` and friends is `file` which is the name of the file which the data are to be read from. On the more technical side, recall that there is a `connection()` function used by R related to the `file` argument.

The following table lists some `read.table()` arguments and their meaning:

Argument	Description
`file`	Name of file
`header`	Whether column names are in 1st line
`sep`	Character used as field separator
`quote`	Quoting characters
`dec`	Character for decimal point
`row.names`	Optional vector of row names
`col.names`	Optional vector of column names
`na.strings`	Characters treated as missing values
`colClasses`	Optional vector of data types for columns
`nrows`	Maximum number of rows to read in
`skip`	Number of lines to skip before reading data
`check.names`	Check valid column names
`stringsAsFactors`	Should characters be converted to factors

In versions of R `< 4.0.0`, `read.table()` and friends convert character strings into factors by default.

In addition to `read.table()`, there's a handful of sibling functions to read other types of text files (see table below). By the way, all these functions are actually wrappers of `read.table()`:

Function	Description
`read.csv()`	import comma separated values
`read.csv2()`	import semicolon separated values (Europe)
`read.delim()`	import tab separated values
`read.delim2()`	import tab separated values (Europe)

17.3.2 Reading space-separated files

Let's review some examples that illustrate the use of the read-table functions. For simplicity's sake, we'll assume that all data files are located in your working directory.

Suppose you have the following data in a file: `accounts.txt`

```
account bank rate balance
savings bricknmor 0.020 1000
monmarket online 0.025 2000
certificate bricknmor 0.030 3000
brokerage online 0.070 5000
retirement online 0.050 9000
```

Example 1. Importing table in blank separated file

```
# using read.table()
dat <- read.table(
  file = "accounts.txt",
  header = TRUE)
```

The imported data frame `dat` will be:

```
       account       bank   rate  balance
1      savings  bricknmor  0.020     1000
2    monmarket     online  0.025     2000
3  certificate  bricknmor  0.030     3000
4    brokerage     online  0.070     5000
5   retirement     online  0.050     9000
```

Example 2. Limit the number of rows to read in (first 2 rows):

```
dat <- read.table(
  file = "accounts.txt",
  header = TRUE,
  nrows = 2)
```

The imported data frame `dat` will be:

```
     account       bank   rate  balance
1    savings  bricknmor  0.020     1000
2  monmarket     online  0.025     2000
```

Example 3. Skip the first row (no header) and limit the number of rows to read in (4 rows)

```
dat <- read.table(
  file = "accounts.txt",
  header = FALSE,
  skip = 1,
  nrows = 4)
```

The imported data frame `dat` will be:

```
            V1          V2     V3    V4
1      savings  bricknmor  0.020  1000
2    monmarket      online  0.025  2000
3  certificate  bricknmor  0.030  3000
4    brokerage      online  0.070  5000
```

Example 4. Skip importing the second and third columns

```
dat <- read.table(
  file = "accounts.txt",
  header = TRUE,
  colClasses = c(
    "character",
    "NULL",
    "NULL",
    "numeric"))
```

The imported data frame `dat` will be:

```
       account  balance
1      savings     1000
2    monmarket     2000
3  certificate     3000
4    brokerage     5000
5   retirement     9000
```

17.3.3 Reading comma-separated files with `read.csv()`

Let's now consider a data table stored in a CSV file: `accounts.csv`

```
account,bank,rate,balance
savings,bricknmor,0.020,1000
monmarket,online,0.025,2000
certificate,bricknmor,0.030,3000
brokerage,online,0.070,5000
retirement,online,0.050,9000
```

Example 1. Data in comma separated value (CSV) file

We can use `read.table()`

```r
# using read.table()
dat <- read.table(
  file = "accounts.csv",
  header = TRUE,
  sep = ",")
```

Or more conveniently, we can use `read.csv()`

```r
# using read.csv()
dat <- read.csv(file = "accounts.csv")
```

The imported data frame `dat` will be:

```
       account       bank   rate  balance
1      savings  bricknmor  0.020     1000
2    monmarket     online  0.025     2000
3  certificate  bricknmor  0.030     3000
4    brokerage     online  0.070     5000
5   retirement     online  0.050     9000
```

17.4 Importing with `scan()`

To finish this chapter, I want to briefly talk to you about the `scan()` function. This is another function that you can use to read data into a vector or list from the console or from a file. As a matter of fact, `scan()` is under the hood of `read.table()` and friends. Because of this, I often refer to `scan()` as a low-level function.

Let's see an example for how to use `scan()` with a **space** delimited file. Suppose that we have the data in the file `accounts.txt`, located in our working directory.

```
account bank rate balance
savings bricknmor 0.020 1000
monmarket online 0.025 2000
certificate bricknmor 0.030 3000
brokerage online 0.070 5000
retirement online 0.050 9000
```

`scan()` is what I call a "low-level function". What I mean by that is that we typically need to write more code when using this kind of functions. To be more precise, we usually have to

specify more arguments when calling `scan()` than when calling other high-level functions like `read.table()` and friends.

Like all other data-import functions, the first argument of `scan()` is the the name of the file (i.e. file path) to be imported. The second argument is called `what`, and this has to do with the *type of data* to be read.

If you look at the content of `accounts.txt`, the first line contains the column names: `account bank rate balance`. Because the data in this first line is to be used as names, their data-type is `"character"`.

The rest of the lines have to do with the actual data of the accounts. Each line contains four pieces of data:

- the kind of `account`, to be encoded as `"character"`
- the kind of `bank`, to be encoded also as `"character"`
- the `rate` of return, to be encoded as `"double"` (or `"numeric"`)
- the `balance` amount, to be encoded as `"integer"` or if you prefer as `"double"` (or generically as `"numeric"`)

In order to import the `accounts.txt` data into R, we must use a two-step process:

1) first we import the column names into a character vector
2) then we import the rest of the lines into a list

Here is how to do it:

```
# step 1) scan column names
# ins a character vector
header = scan(
  file = "accounts.txt",
  what = list("", "", "", ""),
  nmax = 1)

header
```

```
[1] "account" "bank"    "rate"    "balance"
```

```
# step 2) scan the rest of the lines
# into a list
dat_list = scan(
  file = "accounts.txt",
```

```
    what = list(character(), character(), double(), integer()),
    skip = 1)

  dat_list
```

```
[[1]]
[1] "savings"     "monmarket"   "certificate" "brokerage"   "retirement"

[[2]]
[1] "bricknmor" "online"    "bricknmor" "online"    "online"

[[3]]
[1] 0.020 0.025 0.030 0.070 0.050

[[4]]
[1] 1000 2000 3000 5000 9000
```

Once we have the vector of column names `header` and the list with the data for each of the columns, we can then assemble the data into a data table using the `as.data.frame()` function as follows:

```
  dat = as.data.frame(dat_list, col.names = header)
  dat
```

```
      account      bank  rate balance
1     savings bricknmor 0.020    1000
2   monmarket    online 0.025    2000
3 certificate bricknmor 0.030    3000
4   brokerage    online 0.070    5000
5  retirement    online 0.050    9000
```

17.5 In Summary

When importing a data table, you should keep in mind the following considerations.

What is the field separator?

- space `" "`
- tab `"\t"`

- comma `","`
- semicolon `";"`
- other?

Does the data file contains:

- row names?
- column names?
- missing values?
- special characters?

So far …

- There are multiple ways to import data tables
- The workhorse function is `read.table()`
- But you can use the other wrappers, e.g. `read.csv()`
- The output is a `"data.frame"` object

18 Exporting Data

One common task in most data analysis projects involves exporting data to external files.

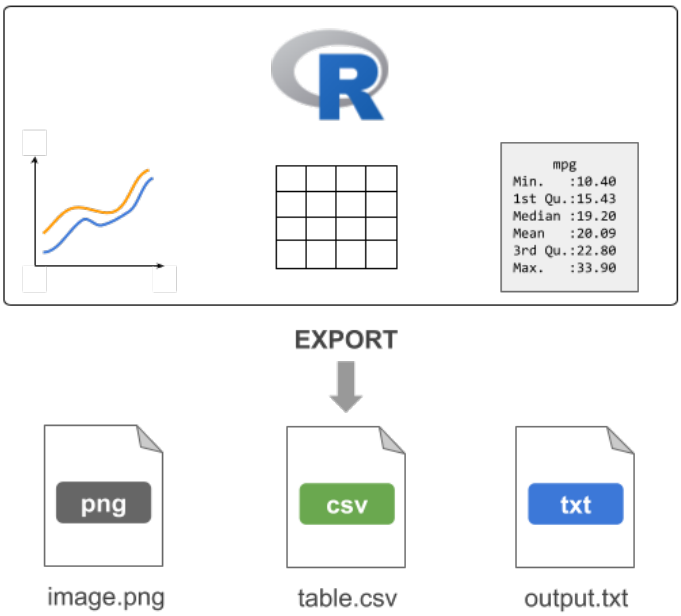


Figure 18.1: Conceptual diagram illustrating some exporting resources

18.1 Exporting Tables

One common task in most data analysis projects involves exporting derived data tables (e.g. clean data sets, or processed tables). To accomplish this task you can use any of the write-table functions such as `write.table()`, `write.csv()`, etc.

```
# blank separated (default)
write.table(mtcars, file = 'mtcars.txt', row.names = FALSE)

# tab-separated value
```

```r
write.table(mtcars, file = 'mtcars.tsv', sep = "\t", row.names = FALSE)

# comma-separated value
write.csv(mtcars, file = 'mtcars.csv', row.names = FALSE)
```

18.2 Exporting Text

Another type of data-exporting operation has to do with "unstructured" text, and text output in general. By "unstructured" I mean non-tabular data.

Consider the following piece of code, which writes the elements of a character vector `some_text`, one element per line (via a file connection) to the file `mytext.txt` in the local working directory:

```r
# create a connection to a file
# (assuming output file in working directory)
txt <- file("mytext.txt")

# write contents to the file
writeLines(text = some_text, con = txt)
```

Note: Calling `file()` just creates the connection object but it does not open it. The function `writeLines()` is the one that opens the connection, writes the content to the file `mytext.txt`, and then closes the connection on exiting.

The previous code can be compactly written with one command without the need to explicitly use the connection function `file()` as follows:

```r
# write contents to the file
writeLines(text = some_text, con = "mytext.txt")
```

18.3 Sending output with `cat()`

You can use `cat()` to concatenate and print information to a file. For instance, say you are interested in some descriptive statistics about the column `mpg` (miles per gallon) from the `mtcars` data frame:

```r
# summary statistics of mpg
min(mtcars$mpg)
max(mtcars$mpg)
median(mtcars$mpg)
mean(mtcars$mpg)
sd(mtcars$mpg)
```

Suppose the goal is to generate a file `mpg-statistics.txt` with the following contents:

```
Miles per Gallon Statistics

Minimum: 10.4
Maximum: 33.9
Median : 19.2
Mean   : 20.09062
Std Dev: 6.026948
```

How can this goal be achieved? We can create objects for all the summary statistics, and then involke `cat()` as many times as there are lines of text to be exported in the output file.

```r
# summary statistics of mpg
mpg_min <- min(mtcars$mpg)
mpg_max <- max(mtcars$mpg)
mpg_med <- median(mtcars$mpg)
mpg_avg <- mean(mtcars$mpg)
mpg_sd <- sd(mtcars$mpg)

# name of output file
outfile <- "mpg-statistics.txt"

# first line of the file
cat("Miles per Gallon Statistics\n\n", file = outfile)
# subsequent lines appended to the output file
cat("Minimum:", mpg_min, "\n", file = outfile, append = TRUE)
cat("Maximum:", mpg_max, "\n", file = outfile, append = TRUE)
cat("Median :", mpg_med, "\n", file = outfile, append = TRUE)
cat("Mean   :", mpg_avg, "\n", file = outfile, append = TRUE)
cat("Std Dev:", mpg_sd, "\n", file = outfile, append = TRUE)
```

Notice that the first call to `cat()` exports the text that is supposed to be in the first line of content in the output file `outfile`. The subsequent calls to `cat()` use the argument `append`

= TRUE so that the next lines of content are appended to the existing outfile. If we don't use append = TRUE, R will override the existing contents in outfile.

18.3.1 Sending output with cat()

To make the content in "mpg-statistics.txt" look "prettier", limiting the number of decimals digits to just 2, you may consider using sprintf(). This function allows you to print strings using C-style formatting.

```
cat("Miles per Gallon Statistics\n\n", file = outfile)
cat(sprintf("Minimum: %0.2f", mpg_min), "\n", file = outfile, append = TRUE)
cat(sprintf("Maximum: %0.2f", mpg_max), "\n", file = outfile, append = TRUE)
cat(sprintf("Median : %0.2f", mpg_med), "\n", file = outfile, append = TRUE)
cat(sprintf("Mean   : %0.2f", mpg_avg), "\n", file = outfile, append = TRUE)
cat(sprintf("Std Dev: %0.2f", mpg_sd), "\n", file = outfile, append = TRUE)
```

Look at the call of sprintf() in the second line: the one that prints the minimum value of miles-per-gallon:

```
sprintf("Minimum: %0.2f", mpg_min)
```

What does sprintf() do in this example? It takes the string "Minimum: %0.2f" and the numeric value mpg_min. Observe that the provided string contains some weird-looking characters: %0.2f. This set of characters have a meaning and they are used as a place holder to be replaced with the formatted value mpg_min. Specifically, the notation %0.2f indicates two decimal digits of a double precision value. In other words, a numeric value 10.4 will be printed using two decimal digits as: 10.40. Likewise, a number such as 20.09062 will be printed as 20.09. If you are curious about the various types of *C-style string formatting* place-holders, check the documentation of sprintf()

```
help(sprintf)
```

18.4 Redirecting output with sink()

Another interesting function is sink(). This function is very useful when you want to export R output **as it is displayed in R's console**.

For example, consider the following output from summary() applied on three columns of mtcars

```
summary(mtcars[ ,c('mpg', 'hp', 'cyl')])
```

```
      mpg              hp             cyl
 Min.   :10.40   Min.   : 52.0   Min.   :4.000
 1st Qu.:15.43   1st Qu.: 96.5   1st Qu.:4.000
 Median :19.20   Median :123.0   Median :6.000
 Mean   :20.09   Mean   :146.7   Mean   :6.188
 3rd Qu.:22.80   3rd Qu.:180.0   3rd Qu.:8.000
 Max.   :33.90   Max.   :335.0   Max.   :8.000
```

To be able to keep the same output displayed by R, you must use `sink()`. This function will **divert** R output to the specified file.

```
# sink output
sink(file = "mtcars-stats.txt")

# summary statistics
summary(mtcars[ ,c('mpg', 'hp', 'cyl')])

# stops diverting output
sink()
```

The use of `sink()` is a bit different from other data-exporting functions. As you can tell from this example, we start by invoking `sink()` and specifying the name of the output file. Then we include the commands whose outputs will be redirected to the target file. In order to stop the redirecting mechanism, we need to invoke `sink()` again, without specifying any arguments.

Consider one more example involving the use of `sink()` to send the output from running a linear regression of `mpg` on `hp` with the function `lm()`. To make things more interesting, we will also export the results returned by `summary()` on the regression object. And not only that; we will also run a t-test between `am` and `hp` with `t.test()`, and export the results of such test.

```
# sink output
sink(file = "regression-output.txt")

# regression of mpg onto hp
reg = lm(mpg ~ hp, data = mtcars)
summary(reg)

# t-test
```

```
t.test(hp ~ am, data = mtcars)

# stop sinking process
sink()
```

18.5 Exporting R's Binary Data

R also allows you to save objects in R's binary format with the functions `save()` and `save.image()`. It is customary to use the `RData` extension for the files created by `save()` and `save.image()`. You may also find users specifying the old extension `.rda` or some other variation.

You can use `save()` to save specific objects from your current session. For example, here is how to save the data frame `mtcars3`

```
mtcars3 = mtcars[ ,c('mpg', 'hp', 'cyl')]
save(mtcars3, file = 'mtcars3.RData')
```

The difference between `save()` and `save.image()` is that the latter saves all the objects in your current session. This is actually the function that is run behind the scenes every time you quit R and accept to save the so-called *workspace image*.

You can share `mtcars3.RData` with any other R user, regardless of the operating system that they use. To read in binary R files, use `load()`.

18.6 Exporting Images

In addition to exporting tables and/or a combination of text-numeric output in the form of several objects, another typical data-exporting activity involves saving graphics, and images in general.

R comes with a handful of functions to export graphics in various formats:

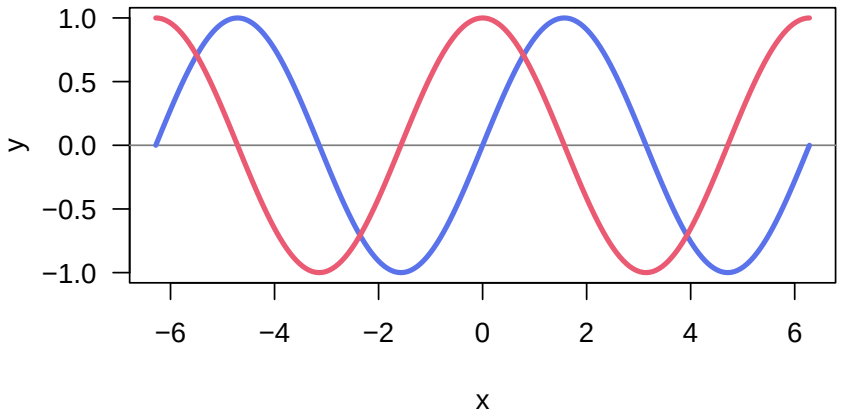
Function	Description
`png()`	Portable Network Graphics
`jpeg()`	Joint Photographic Experts Group
`pdf()`	Portable Document Format
`bmp()`	Bitmap
`tiff()`	Tag Image File Format

Function	Description
`svg()`	Scalable Vector Graphics

For example, say you have a plot like the following one:

```
x = 2*pi * seq(from = -1, to = 1, by = 0.01)
sin_x = sin(x)
cos_x = cos(x)

plot(x, sin_x, type = "n", las = 1, ylab = "y")
abline(h = 0, col = "gray50")
lines(x, sin_x, lwd = 3, col = "#5A73EB")
lines(x, cos_x, lwd = 3, col = "#eb5a73")
```



To export the graphic into a PNG image file, we use `png()` and `dev.off()`

```
png(file = "myplot.png", bg = "transparent")
plot(x, sin_x, type = "n", las = 1, ylab = "y")
abline(h = 0, col = "gray50")
lines(x, sin_x, lwd = 3, col = "#5A73EB")
lines(x, cos_x, lwd = 3, col = "#eb5a73")
dev.off()
```

The usage of `png()` and friends is similar to `sink()`. By default, when you use any of the graphing functions such as e.g. `plot()`, `barplot()`, `boxplot()`, etc, the image is rendered in the graphics device that comes in R (or RStudio). By using `png()`, the rendering mechanism is diverted into an external file. This is why we need to use the function `dev.off()` to shut down the device used when we are done exporting an image to an external file.