so called Anaconda Distribution, which you have probably heard about and that is one of the most used and extended platforms to perform data science. Anaconda is free and open-source, and is conceived to run Python and R code for data analysis and machine learning. This is a great tool for computational scientists and if you plan to follow this book we recommend you to install the complete Anaconda Distribution on your computer[8]. In addition to Spyder and Jupyter Notebooks (and RStudio!), you will also get a set of pre-installed packages often used in data science (Pandas, Numpy, Scipy, Numba, Dask, Boreh, HoloViews, Datashader, Matplotlib, ScikitLearn, TensorFlow, etc.). Moreover, it will include the *conda*, which is an environment management system that will help you to install and update other libraries or dependencies. Once Anaconda is installed, you may say that you have on your local machine the most important software to perform computational analysis of communication. In the next chapter, we will address notebooks again as a good practice for a computational scientist.

3.2. About objects and data types

Now that you have all the necessary software in you own computer we can move on to the very basics of data analysis in R and Python[9]. In both languages, you write a *script* or *program* containing the commands for the computer to do data processing or other tasks. Before we move on to data processing, it is important to take a quick look at how data is actually stored and at the basics of programming.

All data is stored in memory as *objects*, and each object has a name. You create these objects by assigning a value to a name. For example, the command `x = 10` creates a new object[10], named `x`, and stores the value 10 in it. This object is now

[8] https://www.anaconda.com/distribution/#download-section

[9] If you have not installed the require software yet and still want to follow this chapter, you may use available online platforms for RStudio (https://rstudio.cloud/) and Python (https://www.python.org/shell/).

[10] In both R and Python, the equals sign (`=`) can be used to assign values. In R, however,

stored in memory and can be used in later commands.

Note: A small note on terminology: In programming, a distinction is often made between an object (such as the number 10) and the variable in which it is store (such as `x`). The latter is also called a "pointer". However, this distinction is not very relevant for our usage. Morever, in statistics the word variable often refers to a column of data, rather than to the name of the data frame. For that reason, we will use the word *object* to refer to both the actual object or value and its name, and will avoid the word variable.

Objects can be simple values such as the number 10, but they can also be pieces of text, whole data frames, or analysis results. The computer keeps track of the *type*, or *class*, of each object. You can inspect the type of an object with the `type` (Python) or `class` function, as shown in example Example **??**.

Let' create an object that we call `a` (an arbitray name, you can use whatever you want) an assign a value, 100, to it (see Example **??**).

Python Code

```
a = 100
print(type(a))
```

R Code

```
a = 100
print(class(a))
```

Python Output

```
<class 'int'>
```

R Output

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

Example 3.1: Determining the type of an object

As you can see, R calls the number 'numeric', while Python reports it as being 'int', short for integer or whole number. Although they use different names, both languages offer very similar data types. Table **??** provides an overview of the data types we will

the traditional way of doing this is using an arrow (`<-`). In this book we will use the equals sign for assignment in both languages, but remember that for R, `x = 10` and `x <-10` are essentially the same.

Table 3.1: Most used data types in python and R

Python		R		Description
Name	Example	Name	Example	
int	`1`	integer	`1L`	whole numbers
float	`1.3`	numeric	`1.3`	Numbers with decimal
str	`"Spam", 'ham'`	character	`"Spam", 'ham'`	Textual data
bool	`True, False`	logical	`TRUE, FALSE`	The truth values

encounter most frequently in this book. As you can see in the table, both python and R have very similar types for simple data such as numbers, text, and truth values.

Let's have a closer look at the code in Example **??**.

The first line is mandatory to create the object *a* and store its value 100; and the second is illustrative and will give you the class of the created object, in this case "numeric". Notice that we are using two native functions of R, `print` and `class`, and including `a` as an argument of `class`, and the very same `class(a)` as an argument of `print`. The only difference between R and Python, here, is that the relevant Python function is called `type` instead of `class`.

This is the way we work in object-orientated programming and if you compare to other languages you will realize that is a very simple syntax indeed. Once created, you can now perform multiple operations with `a` and other values or new variables. For example, you could transform `a` by multiplying `a` by 2, create a new variable `b` of value 50 and then create another new object `c` with the result of `a + b` (Example **??**).

3.2.1. Storing single values: integers, floating-point numbers, booleans

When working with numbers, we distinguish betwee integers (whole numbers) and floating point numbers (numbers with a decimal point, called 'numeric' in R). Both

Python Code

```
a = 100
a = a*2   # equivalent to (shorter) a*=2
b = 50
c = a + b
print(a, b, c)
```

R Code

```
a = 100
a = a*2
b = 50
c = a + b
print(a)
print(b)
print(c)
```

Python Output

```
200 50 250
```

R Output

```
[1] 200
[1] 50
[1] 250
```

Example 3.2: Some simple operations

Python and R automatically determine the correct datatype when creating an object, but differ in their default behavior when storing a number that can be represented as an int: R will store it as float anyway and you need to force it to do otherwise, for Python it is the other way round (Example **??**). We can also convert between types later on, even though converting a float to an int might not be a too good idea, as you truncate your data.

We also have a data type that is even more restricted and can take only two values: true or false. It is called ‘logical’ (R) or ‘bool’ (Python). Just notice that True or False values are case sensitive and while in R you must capitalize the whole value (TRUE, FALSE), in Python we only capitalize the first letter: True, False. As you can see in Example **??**, such an object behaves exactly as an integer that is only allowed to be 0 or 1, and it can easily be converted to an integer.

3.2.2. Storing text

As a computational analyst of communication you will usually work with text objects or string of characters, which are called character vector objects in R, and you can

Python Code

```python
d = 20
print(type(d))
# forcing python to ctreat 20 as a float
d2 = 20.0
print(type(d2))

e = int(20.7)
print(type(e))
print(e)

f = True
print(type(f))
print(int(f))
print(int(False))
```

R Code

```r
d = 20
print(class(d))
# forcing R to treat 20 as an int
d2 = 20L
print(class(d2))

e = as.integer(20.7)
print(class(e))
print(e)

f = TRUE
print(class(f))
print(as.integer(f))
print(as.integer(FALSE))
```

Python Output

```
<class ’int’>
<class ’float’>
<class ’int’>
20
<class ’bool’>
1
0
```

R Output

```
[1] "numeric"
[1] "integer"
[1] "integer"
[1] 20
[1] "logical"
[1] 1
[1] 0
```

Example 3.3: Floating point numbers, integers, and boolean values

create them just by adding single or double quotes to the value of the variable. Just think of a tweet or a Facebook post that you want to load into your workspace for natural language processing analysis.

Python Code

```python
sometext = "I like Python!"
print(type(sometext))
somebytes = b"I like Python!"
print(type(somebytes))

print("’Hi!’ said he.")
print(’"Hi!" said he.’)
```

R Code

```r
sometext = "I am happy learning R!"
print(class(sometext))
somebytes= charToRaw(sometext)
print(class(somebytes))

print("’Hi!’ said he.")
print(’"Hi!" said he.’)
```

Python Output

```
<class ’str’>
<class ’bytes’>
’Hi!’ said he.
"Hi!" said he.
```

R Output

```
[1] "character"
[1] "raw"
[1] "’Hi!’ said he."
[1] "\"Hi!\" said he."
```

Example 3.4: Strings and bytes

As we will discuss in Section ??, there are several ways of how textual characters can be stored as bytes. You probably will not encounter this too often, but sometimes we may need a low-level approach in which we care about the exact bytes underlying a string. Example ?? shows how to create string and raw (byte) strings.

As you see in Example ??, a string is denoted by quotation marks. You can use either double or single quotation marks, but you need to use the same mark to begin and end the string. This can be useful if you want to use quotationmarks within a string, then you can use the other type to denote the beginning and end of the string.

3.2.3. Combining multiple values: lists, vectors, and friends

Until now, we have focused on the basic, inital datatypes or "vector objects", as they are called in R. Often, however, we want to group multiple of these objects. For example, we do not want to manually create thousands of objects called tweet0001, tweet0002, . . . tweet9999 – we'd rather have one list called tweets that contains all of them. You will encounter several names for such combined data structures: lists, vectors, one-dimensional arrays, series, and maybe even more. Because the typical usage differs a bit between R and Python, we will discuss them one by one, starting with R.

Table 3.2: Classes of vector objects in R

Data type	Example	Code
Numeric	1, 2.5, 100, 2500.38	a = 100
Integer	20L, 100L	d = 20L
Character	'a', "I am happy learning R!!!", "200", 'R', "FALSE"	tweet = "I am happy learning R!!!"
Raw	"Any text" stored as: 41 6e 79 20 74 65 78 74	raw_string = charToRaw("Any text")
Logical	TRUE, FALSE	logical_operator = TRUE

Table **??** summarizes vector objects available in R. By bringing together a set of same-class vector objects we can create a vector. A vector is a basic structure of R and holds different elements of the same type (numeric, integer, complex, character, logical, raw) in a one-dimensional array. You will use vectors very often to run computation and we can easily create them using the `c` function. For example, we can create a numeric vector with the scores of a class of 10 students or a character vector of three countries (Example **??**).

Vectors in R

```
scores = c(8, 8, 7, 6, 9, 4, 9, 2, 8, 5)
print(class(scores))
countries = c(’Netherlands’, ’Germany’, ’Spain’)
print(class(countries))

scores2 = c(8, 8, 7, 6, 9, 4, 9, 2, 8, 5, ’b’)
print(class(scores2))
print(scores2)
print(scores)
```

As you see, the data types will correspond to only one class, either numeric or character, since vectors contain only one class. If we create the vector with two different data types, R will recognize one class forcing some elements to be transformed into the dominant class. For example, if you re-build the vector of scores (a new set of values will be loaded into the workspace) with a new student who has been graded with the letter *b* instead of a number, your vector will become a character vector. If you print it, you will see that the values are now displayed surrounded by `"`.

Having a vector means that you can operate with it in many ways. The basic is *indexing*, which is a technique that you will use with other data types (list, arrays, matrices, data frames) and in other languages such as Python. Indexing helps you to locate any given element or group of elements within a vector using its or their positions. In R all positions begin in the number 1 (you will see that in Python and other languages they will begin on 0) and you can index using the square brackets [] over the vector (Example **??**).

Slicing vectors and converting data types

```
scores2[11]
scores2[c(1, 10)]
scores2[1:10]

# taking the first 10 elements from scores2 and convert them back into numbers
scores3 = as.numeric(scores[1:10])
class(scores3)
scores3
```

In the first case, we asked for the score of the 11th student ("b"); in the second we asked for the 1st and 10th position ("8" "5"); and finally for all the elements between the 1st and 10th position ("8" "8" "7" "6" "9" "4" "9" "2" "8" "5"). Note that while we can directly indicate a range by using a `:`, but if we want to pass multiple single index values, we need to create a vector of these indices by using `c()` (Example **??**).

As the vector `scores2` was forced to be a string, all the elements – even those that could be represented as numeric – are now characters. Indexing is very useful to access elements and also to create new objects from a part of another one. Imagine you want to create a new numeric vector by *slicing* the vector scores with just the numeric values and change its class using the function as.numeric (Example **??**).

And voilà, we have a new object called scores3 with the original values of the vector scores and of the class numeric.

Some more vector operations

```
# appending a new value to a vector
scores3 = c(scores3, 7)
# alternative approach
scores3 = append(scores3, 7)

# Of course, we could also store it in a different object instead of overwriting the original vector
scores4 = c(scores3, 7)
# or
scores4 = append(scores3, 7)

# removing an entry from a vector
scores3 = scores3[-11]

# creating a vector containing the values 1,2,3,...20 (or -5, -4, ... 5)
range1 <- 1:20
range2 <- -5:5

# more advanced: 0, 0.2, 0.4, ... 1.0
my_sequence = seq(0,1, by=0.2)
length(my_sequence)
```

We can do many other things like adding a value to an existing vector or creating a vector from scratch by using a function. Example **??** illustrates how to include the new numeric score of 7 (imagine we translate the grade *b* to the number 7) to our vector by two procedures, or how to remove an element from your vector. For example, for dropping the 11th student's score recently included you just index negatively the value of the vector: `[-11]`.

Rather than just typing over a lot of values by hand, we often might wish to create a vector from an operator or a function, without typing each value. Using the operator ':', we can create numeric vectors with a range of numbers, from 1 to 20 or from -5 to 5 (Example **??**). You can also use a function to create more complex vectors. For example the function `seq` will help you to generate regular sequences by providing the number of points of an interval. Imagine you want to create a numeric vector than begins in point 0 and finishes in point 1, including all possible values that separate those two points by a step of 0.2. Let's create that vector and check the amount of created elements by using the function `length` (Example **??**).

A similar R object is the factor, which might be very useful when working with categorical data. When dealing with factors, you can store the values of a vector with their corresponding labels, which in turn will produce levels. If you create a factor, it will automatically read the labels from the values of the vector, but you may manually change those levels. Let’s have a look at Example **??**. The first line will create again a vector of scores of ten students. Then we automatically convert the vector into a factor (reading the very same values as labels) and finally we create the labels (“fail" or “pass") for each of the possible grades. The factor-type object factor_scores_2 will be probably more useful to deal with a bigger amount of students. While we have only two levels (which can be identified with the function `nlevels`), we can have many values to connect with those labels.

Factors can also store labels

```
scores <- c(8, 8, 7, 6, 9, 4, 9, 2, 8, 5)
factor_scores <- factor(scores)
factor_scores_2 <- factor(scores, levels = c(1,2,3,4,5,6,7,8,9,10), labels = c("fail", "fail", "fail", "fail", "fail",
"pass", "pass", "pass", "pass", "pass"))
```

While vectors are probably the most common one-dimensional data structure in R, there are other ones. In particular, we have *lists.* A list is similar to vectors but may contain different type of elements, functions and even other lists. We create lists in a similar way to vectors, except that we have to add the word `list` before declaring the values. Let’s build a list with four different kind of elements, a numeric object, a character object, a square root function (`sqrt`) and a numeric vector (Example **??**). In fact, you can use any of the elements in the list through indexing – even the function `sqrt` that you stored in there to get the square root of 16!

Lists can store multiple data types

```
my_list <- list(33, ’Twitter’, sqrt, c(1,2,3,4))
class(my_list)

# You can check this is a list of four elements and you can use any of these elements (even the function to get the square root of 16!) using the indexing:
my_list[[3]](16)
```

In Python, such lists are, in fact, the most common type for creating a one-dimensional collection of basic data types.

Lists are mutable sequences of different types of elements. In other words, they are ordered collections of objects with no fixed size. We build this Python object using square brackets [], instead of parentheses (), as we did in R. Pay attention to this detail because in Python we will create tuples if you use the parentheses. Tuples are similar to lists, except that they are immutable (which means that they are constant or unchangeable and you can not freely change their values, just as numbers or strings). We can now create our first list and tuple, and get their sizes with the native function len (Example **??**). We have created two sequences of six objects (three integers and three strings) that represent common numeric and categorical values for sentiment analysis. The first sequence is a list and the second a tuple. We can get any value of the list or tuple by indexing its position, just keep in mind that indexing in Python begins in 0 and not in 1 like in R.

Lists and tuples in Python

```
my_list = [1, 0, -1, 'positive', 'neutral', 'negative']
my_tuple = (1, 0, -1, 'positive', 'neutral', 'negative')
print(len(my_list), len(my_tuple))

# indexing
print(my_list[4])
print(my_tuple[3:5])

#let's add another element, and also change 'neutral' to 'informative
my_list.append('something')
my_list[4] = 'informative'

# both of the following will cause an error, because tuples are immutable
my_tuple.append(4)
my_tuple[4] = 'informative'
```

Imagine now that you decide that the value ‘neutral’ should be substituted by the value ‘informative’ in the created sequences. You can update them by just assigning a new value to an item that is identified by its index. The list will be correctly updated, but the code you run to change the tuple will give you an error (with some message such as: "’tuple’ object does not support item assignment"). Thus, tuples are immutable and are worth for certain types of operations because they give stability to a particular computation. Notice that we can include any object into lists or tuples.

Note: While most of the time, you will not care about mutable or immutable objects, the fact that lists (and, later, data frames) are mutable can be really crucial if you. Example **??** illustrates what happens if you create a new object referring to a mutable object, which – in contrast to what you may think – does not create a copy of the object itself. Rather, we now get two names for the same object.

The (for many unexpected) behavior of mutable objects

```
l1 = [1,2,3,4]
l2 = l1  # in R, this would create a copy, but here, we just create a reference!
l2.append(5)
# l1 is changed as well!!!
print(l1)

# if you want to create a copy of a *mutable* object, use .copy() instead:
l1 = [1,2,3,4]
l2 = l1.copy()
l2.append(5)
print(l1)


# strings, instead, are *immutable*, which means that a copy is created anyway:
s1 = "hi"
s2 = s1 + "you"
# s1 is still unchanged
print(s1)
```

Output

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

We also have another Python object called sets. A set is a mutable collection of *unique* elements (you cannot repeat a value) with no order. As it is not properly ordered, you cannot run any indexing or slicing operation on it. The immutable version of sets are called frozensets.

Unordered collections with only unique values sets

```
colours = set([’blue’, ’yellow’, ’red’])
# or simply:
colours2 = {’blue’, ’yellow’, ’red’}

colours3 = frozenset([’blue’, ’yellow’, ’red’])

print(type(colours), type(colours2), type(colours3))
```

While lists give you a lot of flexibility – e.g., they happily accept entries of very different types –, you sometimes may want a stricter structure like R’s vector. This may be especially interesting for high-performance calculations, and therefore, such a structure is avalable from the *numpy* (*nu*mbers in *py*thon) package: the numpy array (Example **??**).

Numpy arrays behave more like R vectors

```
import numpy as np
scores_as_list = [8, 8, 7, 6, 9, 4, 9, 2, 8, 5, ’b’]
print(scores_as_list)
scores_as_array = np.array(scores_as_list)
print(scores_as_array)
```

chapter03/1darray7.py.out

```
[8, 8, 7, 6, 9, 4, 9, 2, 8, 5, ’b’]
[’8’ ’8’ ’7’ ’6’ ’9’ ’4’ ’9’ ’2’ ’8’ ’5’ ’b’]
```

3.2.4. Dictionaries

Almost all data structures in R have a similar equivalent in Python and vice versa. There is one exception, though: *dictionaries.* A dictionary, also known as associative array or hash table, is a very popular data structure in Python, but does not exist in R.

Dictionaries contain unordered and mutable collections of objects that contain certain information in another object. Python generates this data type in form of `{key : value}` in order to map any object by its key and not by its relative position in the collection. This means that you will index using the key. You will find dictionaries very useful in your journey as a computational scientist or practitioner, since they are flexible ways to store and retrieve structured information. We can create them using the curly brackets and including each key-value pair as an element of the collection (Example **??**).

Dictionaries store key-value pairs

```
sentiments = {"positive" : 1, "neutral" : 0, "negative" : -1}
print(type(sentiments))

grades = {}
grades[’A’] = 4
grades[’B’] = 3
grades[’C’] = 2
grades[’D’] = 1
print(grades)
print(sentiments[’positive’])
print(grades[’A’])
```

chapter03/dict.py.out

```
<class ’dict’>
{’A’: 4, ’B’: 3, ’C’: 2, ’D’: 1}
1
4
```

As in other objects you can index its elements, by keep in mind that you do it using the key and not the position in the sequences. For example, we want to get the values of the object ’positive’ in the dictionary *sentiments* and of the object ’A’ in the dictionary *grades* (Example **??**).

A good analogy for a dictionary is a telephone book (imagine a paper one, but it actually often holds true for digital phone books as well): The names are the keys, and the associated phone numbers the values. If you know someone’s name (the key), it is *very easy* to look up the corresponding values: even in a phone book of thousands of pages, it takes you maybe 10 or 20 seconds to look up the name (key). But if you know someones phone number (the value) instead and want to look up the name, that's very inefficient: you need to read the whole phone book until you find the number.

Just as the elements of a list can be of *any* type, and you can have lists of lists, you can also nest dictionaries to get dicts of dicts. Think of our phone book example: rather than storing just a phone number as value, we could store another dict with the keys 'office phone', 'mobile phone', etc. This is very often done, and you will come across many examples dealing with such data structures.

3.2.5. From 1D to 2D (and higher): matrices and n-dimensional arrays

Matrices are two-dimensional rectangular data sets that include values in rows and columns. This is the kind of data you will have to deal with in many analyses shown in this book, such as those related to machine learning. Often, we can generalize to higher dimensions.

In Python, the easiest representation is to simply construct a list of lists. This is, in fact, often done, but has the disadvantage that there are no easy ways to get, for instance, the dimensions (the shape) of the table, or to print it in a neat(-er) format. To get all that, one can transform the list of list into an `array`, a datastructure provided by the package *numpy*.

To create a matrix in R, you have to use the function `matrix` and create a vector of values with the indication of how many rows and columns will be on it. We also have to tell R if the order of the values is determined by the row or not. In Example **??**, we create two matrices in which we vary the `byrow` argument to be TRUE and FALSE, respectively, to illustrate how it changes the values of the matrix, even when the shape (2x3) remains identical. As you may imagine, we can operate with matrices, such as adding up two of them.

A powerful data type are the arrays (very much like the *numpy* arrays described

Python Code

```
import numpy as np

matrix = [[1, 2, 3], [4, 5, 6], [7,8,9]]
print(matrix)

array2d = np.array(matrix)
print(array2d)

array3d = np.array(([[’green’, ’green’, ’green’], [’
        yellow’, ’yellow’, ’yellow’], [’red’, ’red’, ’red
        ’]],
          [[’green’, ’green’, ’green’], [’yellow’, ’yellow
                  ’, ’yellow’], [’red’, ’red’, ’red’]],
          [[’green’, ’green’, ’green’], [’yellow’, ’yellow
                  ’, ’yellow’], [’red’, ’red’, ’red’]]]))
print(array3d.shape)
print(array3d)
```

R Code

```
my_matrix <- matrix( c(0,0,1,1,0,1), nrow = 2, ncol = 3,
        byrow = TRUE)
print(dim(my_matrix))
print(my_matrix)

my_matrix2 <- matrix( c(0,0,1,1,0,1), nrow = 2, ncol = 3,
         byrow = FALSE)
print(my_matrix2)

my_matrix3 = my_matrix + my_matrix2
print(my_matrix3)

array3d <- array(c(’green’, ’yellow’, ’red’),dim = c
        (3,3,3))
print(dim(array3d))
print(array3d)
```

Python Output

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
(3, 3, 3)
[[[’green’ ’green’ ’green’]
  [’yellow’ ’yellow’ ’yellow’]
  [’red’ ’red’ ’red’]]

 [[’green’ ’green’ ’green’]
  [’yellow’ ’yellow’ ’yellow’]
  [’red’ ’red’ ’red’]]

 [[’green’ ’green’ ’green’]
  [’yellow’ ’yellow’ ’yellow’]
  [’red’ ’red’ ’red’]]]
```

R Output

```
[1] 2 3
     [,1] [,2] [,3]
[1,]    0    0    1
[2,]    1    0    1
     [,1] [,2] [,3]
[1,]    0    1    0
[2,]    0    1    1
     [,1] [,2] [,3]
[1,]    0    1    1
[2,]    1    1    2
[1] 3 3 3
, , 1

     [,1]     [,2]     [,3]
[1,] "green"  "green"  "green"
[2,] "yellow" "yellow" "yellow"
[3,] "red"    "red"    "red"

, , 2

     [,1]     [,2]     [,3]
[1,] "green"  "green"  "green"
[2,] "yellow" "yellow" "yellow"
[3,] "red"    "red"    "red"

, , 3

     [,1]     [,2]     [,3]
[1,] "green"  "green"  "green"
[2,] "yellow" "yellow" "yellow"
[3,] "red"    "red"    "red"
```

Example 3.5: Working with two- or n-dimensional arrays

above), which might be similar to matrices but without the restriction of the two-dimensional rectangular space. This means that you create your own dimensions, indicating (in order) the number of rows, columns and matrices of the array. Your will need vectors as inputs and dimensions as parameters. For example, three-dimensional arrays are fundamental when writing pixel values and coding images.

3.2.6. Making life easier: dataframes

Finally, we have data frames. This is the friendly type of data that you find in SPSS or Excel that will help you in a wide range of statistical analysis (though in more advanced exercises such in machine learning you will not be able to use them any more!). A data frame is a tabular data object that includes rows (usually the instances) and columns (the variables). In a three-column data frame, the first variable can be *numeric*, the second *character* and the third *logic*, but the important thing is that each variable is a vector and that all these vectors must be of the same length. We create data frames from scratch using the data.frame() function. Let’s generate a simple data frame of three instances (each case is an author of this book) and three variables of the types numeric (*age*), character (*country* where he obtained his master degree) and logic (*living abroad*, wether he currently lives out of his born country) (Example **??**).

Python Code

```
import pandas as pd
authors = pd.DataFrame({’age’: [38, 36, 39], ’countries’:
         [’Netherlands’, ’Germany’, ’Spain’], ’
        living_abroad’: [False, True, True]})
print(authors)
```

R Code

```
authors = data.frame(age = c(38, 36, 39), countries = c(’
        Netherlands’, ’Germany’, ’Spain’), living_abroad=
         c(FALSE, TRUE, TRUE))
print(authors)
```

Python Output

```
   age   countries living_abroad
0   38  Netherlands         False
1   36      Germany          True
2   39        Spain          True
```

Example 3.6: Creating a simple dataframe

Notice that you have the label of the variables at the top of each column and that it creates an automatic numbering for indexing the rows.

3.3. Simple control structures: loops and conditions

3.3.1. Loops

Having a clear understanding of what an object is allows us to comprehend how object-orientated languages such as R and Python work, but now we need to get some literacy of how to write code and interact with the computer using these programs. Learning a programming language is just like learning any new language. Imagine you want to speak Italian or you want to learn how to play the piano. First thing will be to learn some words or musical notes, and to get familiarized with some examples or basic structures. This is what we have done so far in the last sections and chapters by providing real examples of computational analysis of communication and explaining you the main types of objects you will have to deal with in R and Python. In the case of Italian or piano, you would have to learn next some grammar: How to form sentences in different times or how play some accords and reproduce patterns, respectively. And this is exactly how we should now move on to acquiring computational literacy; by learning some rules to make the computer do exactly what you want.

Remember that you can interact with R and Python directly on their consoles just by typing any given command on the shell. However, when you begin to use several of these commands and combine them with a specific syntax you will need to put all these instructions into a script that can run partially or entirely. Scripts are very useful as the syntax of our code becomes more complex and you move to many levels of *indentation* (logical empty spaces – 2 in R and 4 in Pyhton – depicting blocks and sub-blocks on your code structure) that can result messy on a shell. Thus, to learn how to create functions and simple control structures you should create and save documents (i.e. .r or .py) that we call scripts.

And this is the way computer programs work: you give them some *statements* (directly or from a script) to which they react. Statements are just written instructions that are executed by the console. We call a sequence of these statements a *computer program*. In the previous section, when we created objects or *variables* (i.e. `a = 100`) we already dealt with the basic statement, which is called *assignment statement*. But the statements can be more complex.

Fortunately, we have great syntactic tools to write code. Both, R and Python have *loops* and *conditional statements*, which will make your coding journey much easier and with more sophisticated results because you can control the way your statements are executed. By controlling the flow of instructions you can deal with a lot of challenges in computer programming such as iterating over unlimited cases or executing part of your code as a function of new inputs.

Firstly, loops are sequences that are executed once, indefinitely or until a certain condition is reached. This means that you can operate over a set of objects as many times as you want just by giving one instruction. The most common types of loops are *for*, *while* and *repeat* (do-while). Let's understand this concept by explaining the for-loops. Imagine you have a list of headlines as an object either on R or Python and you want simple code to print out the length of each message. Of course you can go headline by headline using the indexing, but you will get bored or will not have time enough if you have thousands of cases. Thus, the idea is to operate a loop in the list so you can get all the results, from the first until the last element, with just one instruction. The syntax of the for-loop in R is:

```
for (val in sequence)
{
statement
}
```

In Python, the syntax of the for-loop is:

```
for <var> in <iterable>:
    <statement(s)>
```

Python Code

```
#This script creates in Python a list of headlines and
        shows two ways to count the number of characters
        of each string
#We create a list of headlines (strings):
headlines = ('US condemns South new terrorist attacks', '
        New elections forces UK to go back to the UE', '
        Venezuelan president is dismissed')
#The first way is manual:
print('manual results:')
print(len(headlines[0]))
print(len(headlines[1]))
print(len(headlines[2]))
#and the second is using a for-loop
print('for-loop results:')
for x in headlines:
    print(len(x))
```

R Code

```
#This script creates in R a list of headlines and shows
        two ways to count the number of characters of each
         string
#The first way is manual
headlines <- list ('US condemns South new terrorist
        attacks', 'New elections forces UK to go back to
        the UE',
'Venezuelan president is dismissed')
print('manual results: ')
print(nchar(headlines[1]))
print(nchar(headlines[2]))
print(nchar(headlines[3]))
#and the second is using a for-loop
print('for-loop results:')
for (x in headlines){
  print(nchar(x))
}
```

Python Output

```
manual results:
40
44
33
for-loop results:
40
44
33
```

R Output

```
[1] "manual results: "
[1] 40
[1] 44
[1] 33
[1] "for-loop results:"
[1] 40
[1] 44
[1] 33
```

Example 3.7: for loops let you repeat operations

As Example **??** illustrates, every time you find yourself *repeating* something, for instance printing each element from a list, you can get the same results easier by *iterating* or *looping* over the elements of the list, in this case. Notice that you have same results, but with the loop you can automatize your operation writing few lines of code. As we will stress along this book, a good practice in coding is to be efficient and harmonious in the amount of code we write, which is another justification for using loops.

Another way to iterate in Python is using list comprehensions (not available natively in R), which are a stylish way to create list of elements automatically even with conditional clauses. This is the syntax:

```
newlist  = [ expression for item in list if conditional ]
```

In Example **??** we provide a simple example (without any conditional clause) that creates a list with the number of characters of each headline. As this example illustrates, list comprehensions allow you to essentially write a whole for-loop in one line. Therefore, list comprehensions are very popular in Python.

List comprehensions are very popular in Python

```python
len_headlines= [len(x) for x in headlines]
print(len_headlines)

# The code above is a really useful shorthand and considered very ’pythonic’.
# It is equivalent to the more verbose code below:
len_headlines = []
for x in headlines:
    len_headlines.append(len(x))
print(len_headlines)
```

3.3.2. Conditional statements

Conditional statements will allow you to control the flow and order of the statements written on your code. This means you can mandate the machine to do this or that, depending on a given circumstance. These statements use logic operators to test *if* your condition is met (True) or not (False) and execute an instruction accordingly. Both in R and Python, we use the clauses *if*, *else if* (*elif* in Python) and *else* to write the syntax of the conditional statements. Let’s begin showing you the basic structure of the conditional statement in R:

```
if (condition) {
    Statement1
} else if{
    Statement2
}
else {
    Statement3
}
```

The syntax in Python is even simpler:

```
if condition_1:

    statement_1

elif condition_2:

    statement_2

else:

    statement_3
```

Imagine you want to print the headlines of Example **??** only if the text is less than 40 characters long. We can include the conditional statement in the loop adding the next code to the script:

Python Code

```
#Print the headline if it is < 40 characters
for x in headlines:
  if len(x)<40:
    print(x)
```

R Code

```
#Print the headline if it is < 40 characters
for (x in headlines){
  if (nchar(x)<40) {
    print(x)}
  }
```

Python Output

```
Venezuelan president is dismissed
```

Example 3.8: A simple conditional control structure

We could also make it a bit more complicated: First check whether the length is smaller than 40, then check whether it is exactly 44 (`elif` / `else if`), and finally specify what to do if none of the conditions was met (`else`).

In Example **??**, we will print the headline if it is shorter than 40 characters, print the string "What a surprise!" if it is exactly 44 characers, and print "NaN" in all other cases.

Notice that we have included the clause *elif* in the structure (in R it is noted *else if*). *elif* is just another clause which statement must be executed only if an earlier condition was not satisfied. However, the reasoning behind the logic operators remains

Python Code

```
#Print the headline if it is < 40 characters and print '
        NaN' in the other cases
for x in headlines:
  if len(x)<40:
    print(x)
  elif len(x) == 44:
    print("What a coincidence!")
  else :
    print ('NaN')
```

R Code

```
#Print the headline if it is < 40 characters
for (x in headlines){
  if (nchar(x)<40) {
    print(x)}
  else if (nchar(x)==44) {print("What a coincidence!")}
else {print("NaN")}
  }
```

Python Output

```
NaN
What a coincidence!
Venezuelan president is dismissed
```

Example 3.9: A more complex conditional control structure

the same.

3.4. Functions and methods

Functions and *methods* are fundamental concepts in writing code in object-orientated programming. Both are objects that we use to store a set of statements and operations that we can later use without writing the whole syntax again. This makes our code simpler and more powerful.

We have already used some built-in functions, such as `length` and `class` (R) and `len` and `type` (Python) to get the length of an object and the class to which it belongs. But, as you will learn in this chapter, you can also write your own functions. In essence, a function takes some input (the *arguments* supplied between brackets) and returns some output. Methods and functions are very similar concepts. The difference between them is that the functions are defined independently from the object, while methods are created based on a class, meaning that they are associated to an object. For example, in Python, each string has an associated method `lower`, so that writing `'HELLO'.lower()` will return 'hello'. In R, in contrast, one uses a function, `tolower('HELLO')`. For now, it is not really important to know why some things are implemented as a method and some are implemented as a function. You can read more about it HERE ADD REFERENCE TO WHERE WE TALK A BIT ABOUT CLASSES, OR ALTERNATIVELY TO ANOTHER BOOK.

Note: Because methods are associated with an object, you have a very useful trick at your disposal to find out which methods (and other properties of an object) there are: TAB completion. In Jupyter, just type the name of an object followed by a dot (e.g., `a.|` in case you have an object called a) and hit the TAB key. This will open a drop-down menu to choose from.

We will illustrate how to create simple functions in R and Python, so you will have a better understating of how they work. Imagine you want to create two functions: one that computes the 60% of any given number and another that estimates this percentage only if the given argument is above the threshold of 5. The structure of a function in R is:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

In Python it is:

```
def function_name(arg_1, arg_2, ...):
  statement
```

Example **??** shows how to write our function and how to use it.

The power of functions, though, lies in scenarios where they are used repeatedly. Imagine that you have a list of 5 (or 5 million!) scores and you wish to apply the function `por_60_cond` to all the scores at once using a loop. This costs you only two extra lines of code (Example **??**).

So far you have taken your first steps as a programmer, but there many more advanced things to learn that are out of the scope of this book. You can find a lot

Python Code

```python
#This script in Python creates two simple functions to estimate the 60% of any given number

#The first function computes the 60% always
def por_60(x):
  return x*0.6
print (por_60(10))
print (por_60(4))

#The second function computes the 60% ONLY IF the argument is above 5,
# and the other cases the function returns the very same argument
def por_60_cond(x):
  if x>5:
    return x*0.6
  else:
    return x
print (por_60_cond(10))
print (por_60_cond(4))
```

R Code

```r
#This scripts in R creates two simple functions to estimate the 60% of any given number

#The first function computes the 60% always
por_60 <- function(x) {
  return(x*0.6)
}
print (por_60(10))
print (por_60(4))

#The second function computes the 60% ONLY IF the argument is above 5,
# and the other cases the function returns the very same argument

por_60_cond <- function(x) {
  if (x>5) {
    return(x*0.6)
  } else {
    return(x)
  }
}
print (por_60_cond(10))
print (por_60_cond(4))
```

Python Output

```
6.0
2.4
6.0
4
```

Example 3.10: Writing functions

Python Code

```python
#Create the list with the scores and then apply a simple loop
scores = [3,4,5,6,7]
for x in scores:
  print(por_60_cond(x))
```

R Code

```r
#Create the list with the scores and then apply a simple loop
scores <- list(3,4,5,6,7)
for (x in scores)
{
  print(por_60_cond(x))
}
```

Python Output

```
3
4
5
3.5999999999999996
4.2
```

Example 3.11: Functions are particular useful when used repeatedly