Programming Primer | 1

by **Patrick Boily** and **Jen Schellinck**, with contributions from **Kevin Cheung**, **Aidan Crowther**, **Chunyun Ma**, and **Ehssan Ghashim**

Programming languages go in and out of style. To be a strong programmer, it is important to understand not just the ins and outs of a particular programming language, but how computer languages and computing infrastructure work more generally.

In this chapter, learners are first introduced to some of the core concepts of computer programming in a language-agnostic way, before being shown the basics of `R` and `Python`, two of the most common programming languages used in modern data analysis.

1.1 Programming Fundamentals

What are **computer code** and **computer programs**? Is there a difference between these two concepts? (see [8] for a discussion on the topic).

In a nutshell, a **computer program** is an **algorithm**, written in a **computer language**, providing **instructions** to a computer for carrying out a **series of operations**. An example of a computer program is provided in Figure 1.1.

Computer programs can be **compiled** or **interpreted** as a series of hardware operations, carried out by a computer's **electrical components**.

1.1.1 Compiled vs. Interpreted Languages

Compilers **translate** full source code programs, written in **high-level language** (i.e., using natural languages, only "understandable" by people, as in Figure 1.1) into **machine language** (i.e., binary code, only "understandable" by computers): they are basically **grammatical** (syntactic) checkers – if the source code is error-free, it is converted into machine code, which is eventually run by an executable file. Compiled code runs quickly, and is thus favoured for **deployment phase**. Commonly-used compiled languages include C/C++/C#, COBOL, Fortran, Pascal, and Julia.

Interpreters execute the source code directly: as long as an individual statement is error-free (in the context of the available workspace), it can be executed every time it is called, without regard for the overall syntax of the file. Interpreters are slower, generally, and are favoured during the **development phase**. Commonly-used interpreted languages include `R`,[1] Python, JavaScript, and Ruby.

1: Most programmers do not consider `R` to be a programming language. If they are feeling generous, they might dub it a scripting language, at best. But it gets the job done for data analysis purposes.

```c
#include <stdio.h>
int main()
{
        double firstNumber, secondNumber, temporaryVariable;

        printf("Enter first number: ");
        scanf("%lf", &firstNumber);

        printf("Enter second number: ");
        scanf("%lf",&secondNumber);

        // Value of firstNumber is assigned to temporaryVariable
        temporaryVariable = firstNumber;

        // Value of secondNumber is assigned to firstNumber
        firstNumber = secondNumber;

        // Value of temporaryVariable (which contains the initial value of firstNumber)
        secondNumber = temporaryVariable;

        printf("\nAfter swapping, firstNumber = %.2lf\n", firstNumber);
        printf("After swapping, secondNumber = %.2lf", secondNumber);

        return 0;
}
```

Figure 1.1: An example of a computer program written in the computer language C. What do you suppose this program does? (Programiz .)

1.1.2 Some Fundamental Concepts

We have been using the terms "computer language" and "algorithm" as though they were everyday words. Let us take the time to ensure that their meanings are clear.

Formal Language

In a **formal language**, *words* are created by combining *letters* from a pre-defined **alphabet**, according to the rules provided by a *formal grammar*. Everything that is formed according to the rules is an acceptable word; anything else is not.

Example: Consider the formal language defined with

- **alphabet**: {a, b, C, D, !}
- **grammatical rules**:
    1. letters may only be placed immediately to the left or to the right of another letter
    2. a letter instance must always be accompanied by another instance of the same letter at some location either to its left and/or to its right (or both)
    3. an upper case letter must always be accompanied by a lower case letter immediately to its left or to its right

Thus, `aa` is a word in this formal language (rules 2 and 3 are clearly satisfied; rule 3 is satisfied vacuously), as is `bCaCab`, but `!aC!`, `DDaa`, and `Patrick` are not (why?).

Formal languages can sometimes seem ridiculous – of course letters may only be placed to the left or to the right of other letters... where else would they go? Well, *rule 1* officially (and formally) eliminates letters piling up on top of one another, for starter, but also *spaces* between words (for that language, the space ␣ is not in the alphabet of letters).

Human languages, of the sort deemed **natural** (in contrast with artificial or constructed languages) are formal, in theory. In practice, their grammars tend to be **flexible** (more so with English than French, say) – syntax evolves with cultures (in time and in space), and semantics (meaning) can be retained even when the grammar is mangled.[2]

2: But only up to a point, of course.

Computer Language

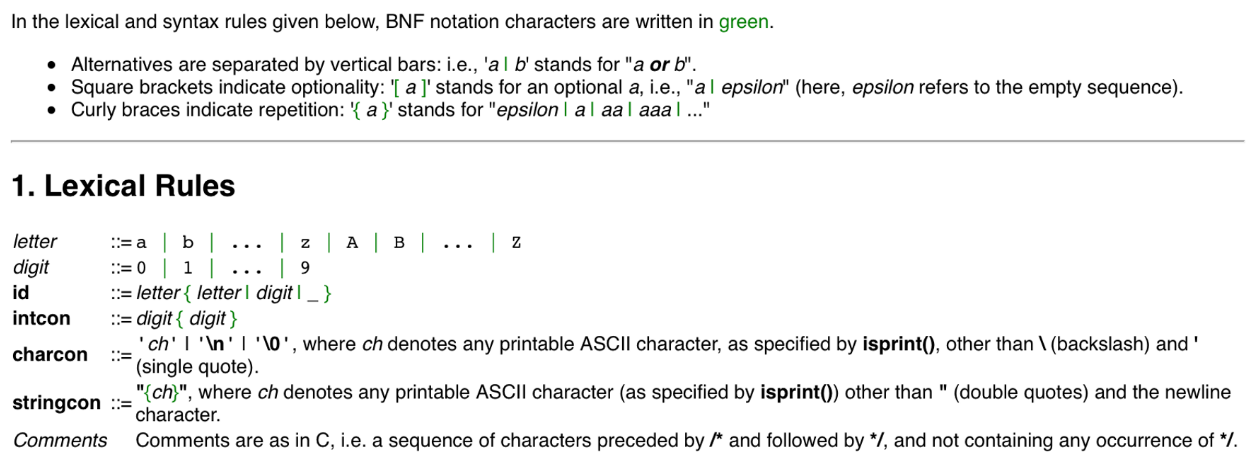
Computer languages are languages constructed to provide instructions **to a computer**, in such a way that they can be compiled into low-level instructions that the computer processor can execute.

Computer languages are also called **programming languages**, for reasons that will soon become obvious. They are **formal** languages because if the grammatical rules are not followed *to the letter*, the program cannot be executed – computers cannot guess or infer what the programmer really meant when the syntax is out of sorts.

The **structure** of the formal definition of a computer language contains the following sections:

1. **Lexical Rules**
2. **Syntax Rules**
   - *Grammar Productions*
   - *Operator Associativities and Precedences*
3. **Typing Rules**
   - *Declarations*
   - *Type Consistency Requirements* (Function Definitions, Expressions, Statements)
4. **Operational Characteristics**
   - *Data* (Scalars, String Constants, Arrays)
   - *Expressions* (Order of Evaluation, Type Conversion, Array Indexing)
   - *Assignment Statements* (Order of Evaluation, Type Conversion)
   - *Functions* (Evaluation of Actuals, Parameter Passing, Return From a Function)
5. **Program Execution**

As an illustration, the **lexical rules** of C are shown in Figure 1.2.



In the lexical and syntax rules given below, BNF notation characters are written in green.

- Alternatives are separated by vertical bars: i.e., '*a* | *b*' stands for "*a* ***or*** *b*".
- Square brackets indicate optionality: '[*a*]' stands for an optional *a*, i.e., "*a* | *epsilon*" (here, *epsilon* refers to the empty sequence).
- Curly braces indicate repetition: '{ *a* }' stands for "*epsilon* | *a* | *aa* | *aaa* | ..."

1. Lexical Rules

letter	::= a | b | ... | z | A | B | ... | Z
digit	::= 0 | 1 | ... | 9
id	::= *letter* { *letter* | *digit* | _ }
intcon	::= *digit* { *digit* }
charcon	::= ' *ch* ' | **'\n'** | **'\0'**, where *ch* denotes any printable ASCII character, as specified by **isprint()**, other than **** (backslash) and **'** (single quote).
stringcon	::= **"**{*ch*}**"**, where *ch* denotes any printable ASCII character (as specified by **isprint()**) other than **"** (double quotes) and the newline character.
Comments	Comments are as in C, i.e. a sequence of characters preceded by **/*** and followed by ***/**, and not containing any occurrence of ***/**.

Figure 1.2: Lexical rules of the programming language C Debray .

Algorithm

Computer programs are **algorithms**, which is to say, sequences of instructions with (at least) one well-defined **stopping point** (an instruction that tells the program when to stop running).

Algorithms are not always mathematical or computer-based. In some sense, we could think of recipes as algorithms as well: the baking/cooking steps are presented in sequence, and some last step that must be completed before the end product can be eaten.

For instance, here is an algorithm to make **muffins**:[3]

3: Delicious!

1. Pour 1/2 cup of flour into a bowl.
2. Break one egg into the bowl.
3. Pour 3 tablespoons of oil into the bowl.
4. Pour 1 teaspoon of baking powder into the bowl.
5. Pour 1/4 cup of sugar into the bowl.
6. Mix with spoon until smooth.
7. Pour the mixture into muffin tins.
8. Bake for 15 minutes at 350 degrees Fahrenheit.
9. Let cool before eating.
10. Enjoy!

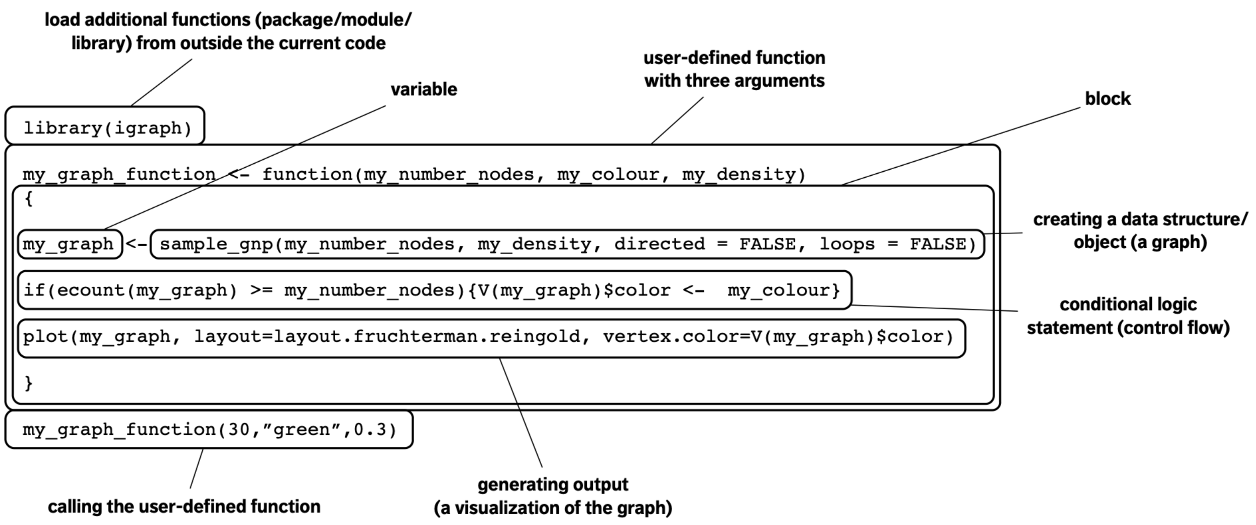
What is the **stopping point**? What is the **outcome**?

1.1.3 Code Components

Various sets of instructions, conventions, and structures are so fundamental to computer programming aims that they can be found in nearly all computer languages.

These **fundamental code elements** include:

- Variables
- Data Structures
- Operators
- Statements and Expressions
- Blocks (and Scope)



```
library(igraph)
my_graph_function <- function(my_number_nodes, my_colour, my_density)
{
my_graph <- sample_gnp(my_number_nodes, my_density, directed = FALSE, loops = FALSE)
if(ecount(my_graph) >= my_number_nodes){V(my_graph)$color <-  my_colour}
plot(my_graph, layout=layout.fruchterman.reingold, vertex.color=V(my_graph)$color)
}
my_graph_function(30,"green",0.3)
```

load additional functions (package/module/ library) from outside the current code
variable
user-defined function with three arguments
block
creating a data structure/ object (a graph)
conditional logic statement (control flow)
calling the user-defined function
generating output (a visualization of the graph)

Figure 1.3: Computer code elements in action, for the scripting language R.

- Functions
- Logical (Control) Flow
- Libraries/Packages/Modules
- Inputs/Outputs
- Interpreters/Compilers

How these components mesh with one another depends on the syntax of the programming language under consideration (or its dialect).

In Figure 1.3, we see how this could be done in base R, for instance. This particular chunk of code uses the

> igraph **library** (specifically, its pre-compiled **functions** `plot()`, `sample_gnp()`, `ecount()`, and `V()`),

and builds the

> user-defined **function** `my_graph_function()` *via* a **code block**,

which takes in as

> **inputs** the **variables** `my_number_nodes`, `my_colour`, and `my_-density`.

This function creates a

> graph **data structure** `my_graph`,

and colours the graph's vertices using `my_colour` as long as

> some conditional **logic statement** relating to the number of edges in the graphs and `my_number_nodes` is satisfied.

The function generates a visualization of the graph as an

> **output**,

which is displayed when the **function call** is issued.

The code is seen in action below: it creates and displays a 30-node, green-coloured, non-directed, loop-free graph with probability 0.3 of there being an edge between two arbitrary nodes (we will discuss what these concepts represent in Chapter **??**, *(Social) Network Data Analysis*).[4]

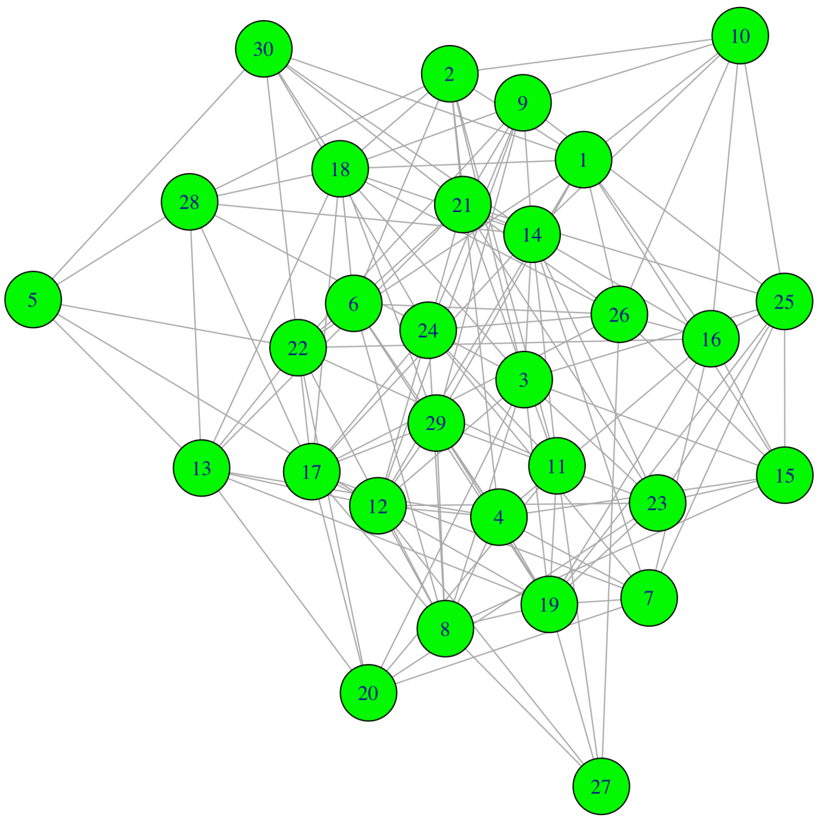
4: The seed enforces replicability.

Creating a random graph

```
library(igraph)

my_graph_function <- function(my_number_nodes,
                              my_colour,
                              my_density) {
            my_graph = sample_gnp(my_number_nodes,
                                  my_density,
                                  directed=FALSE,
                                  loops = FALSE)
            if(ecount(my_graph) >= my_number_nodes) {
               V(my_graph)$color <- my_colour
            }
         plot(my_graph,
              layout = layout.fruchterman.reingold,
              vertex.color = V(my_graph)$color)
}

set.seed(0)
my_graph_function(30,"green",0.3)
```



If all of this seems mysterious and opaque at first, it is important to remember that mastering a computer language requires time and practice.

Some languages are **dialects** or variants of other languages;[5] proficiency in one can make it is easier to become proficient in another. But not all programming languages follow the same paradigm: **imperative** languages (object-oriented programming) live in a different "linguistic family" than **declarative** languages (functional programming) languages.

5: Or at least, are mutually intelligible, like Swedish and Danish, say.

1.1.4 Designing With Pseudo-Code

Before we can start thinking about writing code (in whatever programming language whose syntax we have mastered), we need to think about what it means to **design an algorithm** (or a computer program). From a mathematical perspective, an **algorithm** is a (stochastic) function. We thus need to specify:

- the algorithm's **inputs**;
- its **outputs**, and
- the **procedure** to transform the inputs into the outputs.[6]

6: In the muffin recipe above, the ingredients are the inputs, the muffins themselves are the outputs, and the recipe instructions describe the transformation.

It is good programming practice to avoid typing up programs on the fly – code needs to be **planned**: we need to know what the program will do and how it will go about doing it before we commit it to a file, **independently of the language in which it will be implemented**.

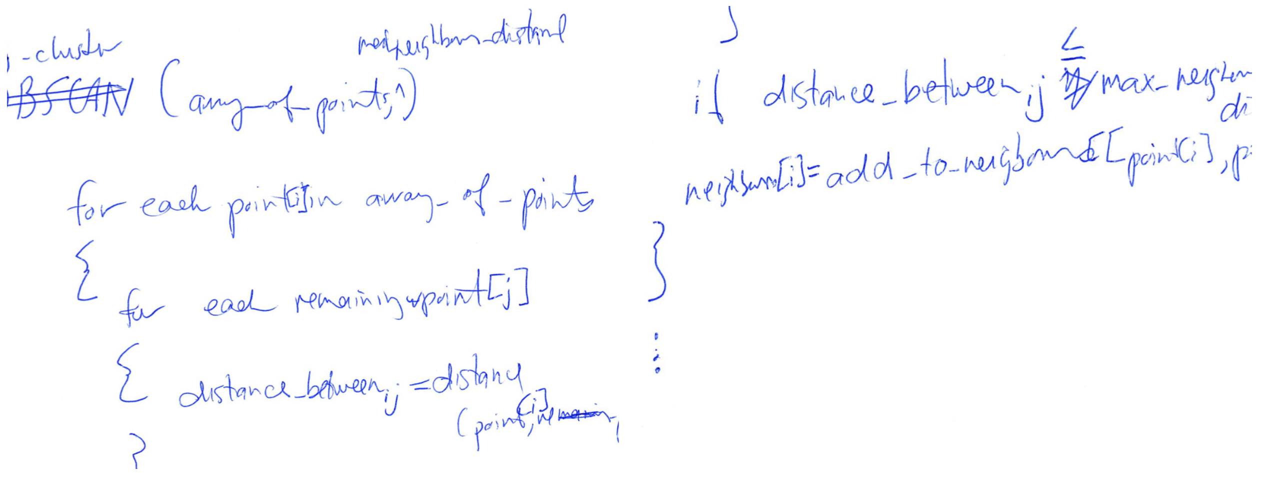
"**Pseudo-code**" is a term used to describe a **rough sketch** of the algorithm, which indicates its expected inputs, outputs, and steps, while leaving the specifics of its functionality in "black boxes". Pseudo-code is usually designed with the main elements of code (e.g., variables, functions, logical flow, etc.), in a **language-agnostic** (i.e., human readable) manner.

Example: we might be interested in building an algorithm that would cluster the observations in a dataset, using a maximum number of "local" observations (see Chapters 19, *Machine Learning 101*, and 22, *Spotlight on Clustering*, for an in-depth discussion of this topic).

What might the following chunk of pseudo-code (which is part of the bigger clustering picture) do?

Chunk of pseudo-code

```
find_neighbours(array_of_points, max_n_neighbour_distance)
{
 for each point[i] in array_of_points
 {
  for each remaining point[j] in array_of_points
  {
   distance_between_ij = distance(point[i], point[j])
   if distance_between_ij <= max_n_neighbour_distance
   then neighbours[i] = add_to_neighbrs(point[i],point[j])
  }
 }
}
```



```
-cluster                    max_neighbour_distance
~~DBSCAN~~ (array_of_points, ^)

    for each point[i] in array_of_points
    {
        for each remaining point[j]
        {
            distance_between_ij = distance
                                  (point[i], re~~main~~
        }
                                                          )
                                      if distance_between_ij ≤ ~~>~~ max_neighbour
                                                                          di
                                      neighbours[i] = add_to_neighbours[point[i], p
    }
    ⋮
```

Figure 1.4: The first stage of pseudo-coding, in all its chicken scratch glory.

This is what is happening:

- the algorithm `find_neighbours()` takes as **inputs** a dataset `array_of_points` and a quantity `max_n_neighbour_distance`;
- for each observation `point[i]` in the dataset (`i` indexes the observations), it considers all other observations `point[j]` and computes their distances to the initial observation `point[i]` (one by one);
- when these distances are smaller than the input threshold `max_n_neighbour_distance`, it considers that the corresponding observation `point[j]` is a neighbour of observation `point[i]`, and adds the former to the neighbours of observation `point[i]`.

Evidently, this chunk of pseudo-code defines the **neighbourhood** of each observation in the dataset. Note the **black box** functions `distance()` and `add_to_neighbrs()`: their specifics are not provided,[7] but what they represent is clear. That is the **power** of pseudo-code.[8]

7: Their eventual implementation may change depending on the computer language selected to write the program.

8: Of course, in practice, we also do not sit down and write pseudo-code on the fly... that too must be planned (see Figure 1.4).

Getting a feel for the right level of pseudo code detail takes practice: should we drill down into what `add_to_neighbrs()` does? Do we need to describe what <= does? How much utility should be sacrificed in favour of understanding?

The answers to these questions depends on the **level of abstraction** of the programming language used to implement the algorithm:

- **high-level languages** (such as `R` and `Python`) contain tons of built-in functions, which allow for programming at higher levels of abstraction, whereas
- many details and functions must be programmed "by hand" in **low-level languages** (such as assembly and machine languages), which require lower levels.

The **strategy** to write useful pseudo-code is deceptively simple:

1. define the available inputs;
2. define the desired outputs, and
3. identify (and write down) a set of programmatic instructions (procedure) to transform the inputs into the outputs.[9]

9: This is easier said than done, obviously, and it looks an awful lot like the definition of an algorithm we provided previously, but remember that parts of the pseudo-code can be "**black boxed**", which is to say, that functionality can be described at a **high level**.

1.1.5 From Pseudo-Code to Code That Runs

Once we are satisfied that the pseudo-code provides a decent path to solving the problem at hand,[10] , we can start thinking about how to implement it into **real code** ("code that runs"):

10: The proposed solution does not need to be final.

1. we start by determining the **appropriate syntax** for the computer language that will be used and we re-write the pseudo-code as syntactically correct code in this language;
2. we replace all "black box" functions with real code, and
3. we determine how to connect the real code (the **software**) to the computer, so that it can be compiled/interpreted, and run by the computer (receiving inputs and generating outputs).

It might take multiple tries before this is done successfully. That is to be expected. It takes time, even for the most gifted programmer, to become an expert in a new language. The urge to feel defeated if (when?) the first few attempts fail is completely natural; as always, practice is the answer.

The process of taking the high-level code (which is really a text file) and getting it to run on a computer without a hitch requires a certain amount of infrastructure to be in place:

- libraries
- input/output + file system
- compilers/interpreters

In these notes, we are taking care of much of these issues by setting up the `R`/`Python` examples internally and running them locally (using our infrastructure); this works well for illustrating the concepts and working with pedagogical datasets, but the infrastructure conundrum must be tackled and solve before it becomes possible to produce useful and actionable data analysis results (see Chapter 17, *Data Engineering and Data Management*, for more details).

In general, there is no **single authoritative reference manual** describing how to use a particular computer language and/or how to make code run on particular hardware configurations, in no small part because coding and computer references become obsolete in the blink of an eye.[11]

11: Consider the change from `Python` 2 to `Python` 3 as a cautionary tale.

Successfully coders must be embedded in a **community of coders**. Luckily, this is getting to be easier to do every day – most questions anybody could ever have about specific aspects of coding have already been answered somewhere online. Stack Exchange and similar sites can be quite useful in that regard.[12]

12: **Fair warning:** some coder communities can be ... let us say, not overly welcoming of neophytes. It is not unusual for the answer to a question to be some variation on "look it up in the documentation". While this can be true in a general sense, such an answer is useless. We all know that things can be looked up in the documentation. And we all know that some users ask questions without taking the time to think about things, or in the hope that somebody else will do their work for them. It is in the best interest of learners to seek communities that make a concerted effort to be healthy and inclusive, to recognize that not every user has reached the same proficiency level. Such communities are plentiful online; do not waste any time and energy on gatekeepers.

As a last remark on the topic, keep in mind that in the real coding world, **there is no such thing as cheating**: the objectives are to make happen the things you want to see happen. Getting help along the way is emphatically not prohibited (mind you, it is good practice to cite or acknowledge such help).

Crucially, though, we should not use code when we do not understand what it does – borrowed code may make complete sense in the context for which it was written, but may have **unintended ramifications** in a different context: **be careful!**

1.1.6 Debugging

PROGRAMMERS DRINKING SONG:
99 little bugs in the code,
99 bugs in the code,
fix one bug, compile it again,
141 little bugs in the code.
141 little bugs in the code. . . ..
(Repeat until bugs = 0)

Mistakes WILL happen. What do we do about that?

In the development phase, coding is about getting all the moving pieces to fit together, yes, but it is also about fixing the **bugs** , an "error in the source code that causes a program to produce unexpected results or crash altogether". Fixing these bugs (**debugging**) is mainly about revealing what is in memory at different points in the control flow of the code, to determine if it is actually doing what we think it ought to be doing.

As the quote at the start of the section implies, debugging is a bit of an art form, requiring the programmer to become a detective and a zen master (see The Tao of Programming). It teaches perseverance and humility, and it really helps us perfect our understanding of the language, of the code itself, and of the task at hand.

Debugging tools can help with all of this; at our level, debugging often requires running the code line-by-line until we can identify the chunk of code that is the culprit. Debugging is a **necessary** part of coding, no matter how experienced you are.

1.1.7 `R`/`Python`

There is only so much that can be said about programming **in general**; at some point, we need to select a computer language and get going in earnest.

At a foundational level, most programming languages are roughly **equivalent** (Turing-complete or Turing-equivalent), in the sense that anything that can be done with one can also (more or less) be done with another. But that does not mean that they are all **equally useful**.

Some are better suited to certain tasks, whether because they are less memory-intensive, or more elegant, or more intuitive, and so on. Even in the data analysis world, there are competing paradigms. In these notes, we will use two of the most popular languages (although by no mean the only ones): `R` and `Python`.

In the examples we provide, `R` code appears in blue boxes:

```
... some R code ...
```

Whereas `Python` code appears in green boxes:

```
... some Python code ...
```

Object-Oriented Languages vs. Procedural Languages `R` and `Python` are **objected-oriented languages**, as opposed to **procedural languages**.

> The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into objects that expose behavior (methods) and data (members or attributes) using interfaces. The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an "object", which is an instance of a class, operates on its "own" data structure. [9]

This will make more sense if we first understand the concepts of:

- data types
- data structures
- functions

Languages have a set of built-in basic **variable types**, such as:

- integer: 5
- character: 'm'
- list: (5, 3, 9)

Other variables types can be built up out of these basic types, such as

- strings, which are list of characters: ('t', 'a', 'b', 'l', 'e')

We can also define related variables – a **data structure**:

- `struct myNames = {string firstName, string middleName, string lastName}`
- `jenNames` might be a variable of type `myNames`, with `firstName = Jen`, `middleName = Adele`, `lastName = Schellinck`.

In addition a programmer might want to be able to carry out a set of predefined instructions, or **functions**, on that data structure:

- `jenNames.print_middleName` or
- `jenNames.string_length_lastName`, say (what these functions do should be clear from their name).

Loosely speaking, an **object** is a user-defined data structure, together with a set of functions that are specific to that structure.

The **data frame object** in `R` is structured similarly to a spreadsheet:

- it has rows and columns, with associated row and column names, and
- we can carry out predefined operations (mean, count, etc.) on specific values, on selected rows, or selected columns, or the data frame as a whole.

Learners that are familiar with databases and/or languages that are more vector-focused (e.g. Java) might find the data frame implementation in `R` frustrating; those who are familiar with matrices and other mathematical concepts used in data analysis, less so.

1.2 Introduction to `R`

`R` is a powerful language that is widely-used for data analysis and statistical computing. It was developed in the early 90s by Ross Ihaka and Robert Gentleman, as a successor to `S`, a **statistical programming language**.

The inclusion of sophisticated packages (such as `dplyr`, `tidyr`, `readr`, `data.table`, `SparkR`, `ggplot2`, etc.) has made `R` both more powerful and more useful, allowing for smart data manipulation, visualization, and computation, using its built-in data structures and functionality.

Notably, it has gained prominence as a free and open source alternative to expensive statistical software.

1.2.1 Why Use `R`

Here are some benefits that potential users might note:

- the style of coding is intuitive;
- `R` is open source and free;
- more than 18,500 packages, customized for various computation tasks, are available (as of February 2022);
- the `R` community is overwhelmingly welcoming and useful to new users and experienced users alike;[13]
- high performance computing experience is possible (with the appropriate packages), and
- is is one of the highly sought skills by analytics and data science companies.

13: You can browse and ask questions at StackOverflow, and consult worked-out examples on R-bloggers, for instance.

1.2.2 Installing `R` / RStudio

Note: If you have a pre-existing installation of `R` and/or RStudio, you may skip this part. However, we highly recommend that both of these applications be upgraded to the most recent version, if they have not been upgraded for a while.[14]

14: Note that these instructions can quickly become obsolete; we will do what we can to stay on top of them, but you may need to consult other sources or search for "Installing R and RStudio" online. Consult *Upgrading* `R` *and/or RStudio* on 17 for details.

Data analysis can be conducted using the **vanilla** (base) version of `R`, but also using RStudio provides a better coding experience, in our opinion.

The following steps will allow you to install `R` and RStudio.

1. Download and install `R` at https://cloud.r-project.org.
   - *Windows* users should click on **Download `R` for Windows**, then click on **base**, then click on the **Download `R X.X.X` for Windows** link, where `R X.X.X` is the version number. For example, the latest version of `R` as of 2022-02-07, was `R 4.1.2`;
   - *macOS* users should click on **Download `R` for macOS**, then on **R-X.X.X.pkg** (under "**Latest release:**:"), where `R-X.X.X` is the version number. If the Mac has an Arm-based M1 chip, choose **R-X.X.X-arm64.pkg** instead;
   - *Linux* users should click on **Download `R` for Linux** and choose the specific distribution for more information on installing `R` for their actual setup.

2. Download and install RStudio at posit.co/download/rstudio-desktop/#download .
   - look for the big blue button that says **DOWNLOAD RSTUDIO DESKTOP FOR ...**, where . . . represents the desired OS;
   - click on the button to start downloading;
   - Once downloading has completed, double-click the file to open it, and follow the installation instruction.
3. **(for *macOS* users only):** Download and install XQuartz.[15]
   - go to https://www.xquartz.org . Under "Quick Download", click on "XQuartz-2.8.1.dmg";
   - save the `.dmg` file, double-click it to open, and follow the installation instructions (you may need to restart your computer).
   - **Reminder**: you will need to re-install XQuartz when upgrading your macOS to a new major version.
4. Even with both `R` and RStudio installed, we will refrain from working directly with the `R` interface, given that RStudio provides such a "nice" **shell** over the engine that is `R`.

15: What is XQuartz and why does macOS users need it?

Once RStudio is opened, the **graphic user interface** (GUI) displays 4 panes, as in Figure 1.5.

- **Console:** bottom left; this area shows the output of code that has been run (either from the command line in the console or from the script window);
- **Script:** top left; as the name suggests, this is the area one would typically use to write code. Lines can be run by first selecting them (right-clicking) and pressing `ctrl + enter` (win) or `cmd + enter` (mac) simultaneously. Alternatively, you can click on the little 'Run' button located at the top right corner of the script window;
- **Environment:** top right; this space displays the set of external elements that have been added. This includes data set, variables, vectors, functions etc. This area allows the user to verify that data has been loaded properly;
- **Graphical Output:** bottom right; this space display the graphs created during exploratory data analysis, or embedded help on package functions from `R`'s official documentation.

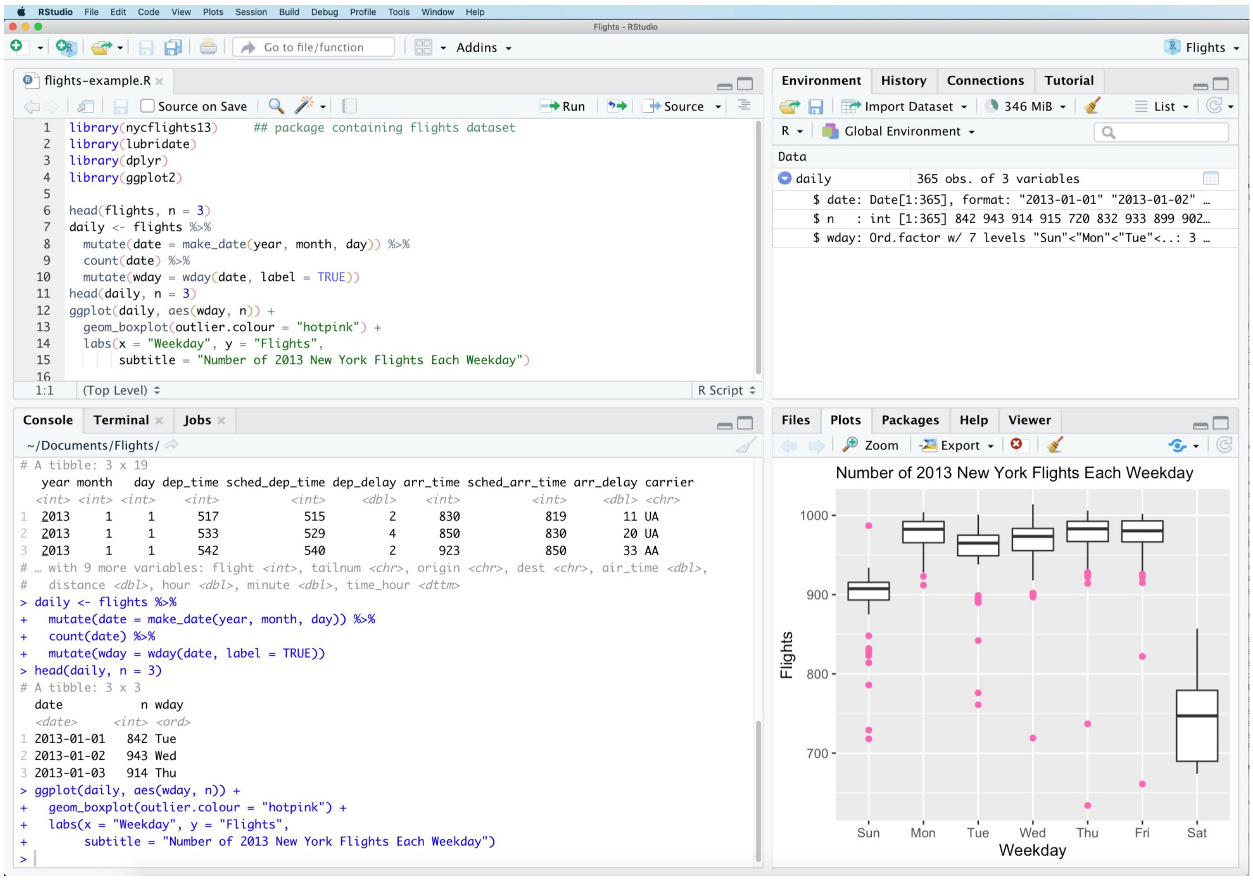
1.2.3 Test, Test, Test!

To make sure you have installed both `R` and RStudio properly, type a simple command in the console. For example, place your cursor in the pane labelled `Console`, type `x <- 2 + 2` at the prompt, followed by `enter` or `return`, then type `x`, again followed by `enter` or `return`.

Testing R

```
x <- 2 + 2
x
```

You should see the value 4 printed to the screen.



```r
library(nycflights13)    ## package containing flights dataset
library(lubridate)
library(dplyr)
library(ggplot2)

head(flights, n = 3)
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  count(date) %>%
  mutate(wday = wday(date, label = TRUE))
head(daily, n = 3)
ggplot(daily, aes(wday, n)) +
  geom_boxplot(outlier.colour = "hotpink") +
  labs(x = "Weekday", y = "Flights",
       subtitle = "Number of 2013 New York Flights Each Weekday")
```

1:1 (Top Level) R Script

```
Data
daily          365 obs. of 3 variables
   $ date: Date[1:365], format: "2013-01-01" "2013-01-02" …
   $ n   : int [1:365] 842 943 914 915 720 832 933 899 902…
   $ wday: Ord.factor w/ 7 levels "Sun"<"Mon"<"Tue"<..: 3 …
```

~/Documents/Flights/

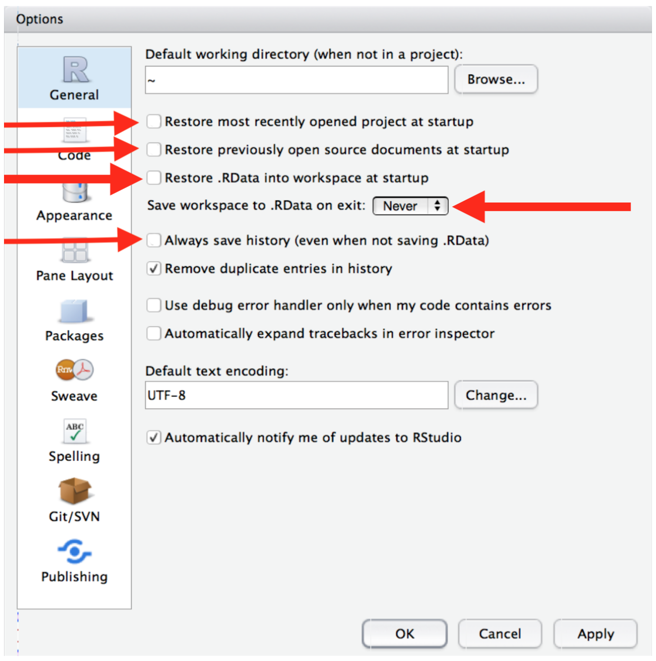
```
# A tibble: 3 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
1  2013     1     1      517            515         2      830            819        11 UA
2  2013     1     1      533            529         4      850            830        20 UA
3  2013     1     1      542            540         2      923            850        33 AA
# … with 9 more variables: flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
> daily <- flights %>%
+   mutate(date = make_date(year, month, day)) %>%
+   count(date) %>%
+   mutate(wday = wday(date, label = TRUE))
> head(daily, n = 3)
# A tibble: 3 x 3
  date           n wday
  <date>     <int> <ord>
1 2013-01-01   842 Tue
2 2013-01-02   943 Wed
3 2013-01-03   914 Thu
> ggplot(daily, aes(wday, n)) +
+   geom_boxplot(outlier.colour = "hotpink") +
+   labs(x = "Weekday", y = "Flights",
+        subtitle = "Number of 2013 New York Flights Each Weekday")
> 
```

Figure 1.5: RStudio interface, with 4 default windows: Console, Script, Environment, and Graphical Output.

1.2.4 Customizing RStudio

We would like to suggest the following settings for your R/RStudio installation, following [1, ch.8].16 In RStudio, go to `Tools >> Global Options`, and make the changes described below:

16: Feel free to ignore the suggestion as you wish.



Default working directory (when not in a project):
~

☐ Restore most recently opened project at startup
☐ Restore previously open source documents at startup
☐ Restore .RData into workspace at startup
Save workspace to .RData on exit: Never
☐ Always save history (even when not saving .RData)
☑ Remove duplicate entries in history

☐ Use debug error handler only when my code contains errors
☐ Automatically expand tracebacks in error inspector

Default text encoding:
UTF-8

☑ Automatically notify me of updates to RStudio

> [These settings] will cause you some short-term pain, because now when you restart RStudio it will not remember the results of the code that you ran last time. But this short-term pain will save you long-term agony because it forces you to capture all important interactions in your *source code*. There's nothing worse than discovering three months after the fact that you've only stored the *results* of an important calculation in your workspace, not the calculation itself in your source code. [1]

Optionally, you could also adjust the font size via `Tools >> Global Options >> Appearance >> Editor font size`.[17]

17: By default, it is set at 12, but a larger font size may be easier on the eyes.

1.2.5 Upgrading R / RStudio

We suggest always working with the latest version of R and RStudio.

- To upgrade R, find out the current version of R running on your computer. You can do so from within the RStudio Console:

  R version

  ```
  R.version.string
  ```

  ```
  [1] "R version 4.1.3 (2022-03-10)"
  ```

  As of January 2023, the most recent version of R is `4.2.2`. If you have an older version installed on your computer, go to cloud.r-project.org and follow the steps described in on p. 14 (*Installing R / RStudio*) to install the latest version of R. You can confirm that the upgrade was successful by restarting RStudio and typing `R.version.string` in the console again.

- To upgrade RStudio from within RStudio, go to `Help > Check for Updates` to install newer version of RStudio (if available). Once both R and RStudio have been upgraded, test by typing some simple command in the console (as on p. 15, *Test, Test, Test!*).

1.2.6 Basics of R

How are the elements of code (introduced in *Code Components* on p. 6) implemented in R? How do they mesh with one another to form interpretable code? First, we should mention that while R is technically object-oriented, this tends to be hidden in practice; the language is thus especially well-suited for **quick**, **interactive**, and **intuitive** scripting and data exploration.

Note as well that it uses special built-in notation for statistical models, which would not usually be found in other languages (hence the "statistical programming" moniker). Some of the examples and explanations provided in the text are modified from [1, 10–15].

The rest of this section contain information on the basic use of R; more examples are available in Section 1.3 (*More About Programming in* R) and throughout the course notes.

Simple Computations in R We will get familiar with the R coding environment, we start by showing how the console can be used as an interactive calculator.

Type the first line of each group in your console, followed by a carriage return to confirm that R works as we would expect of a calculator:

```
2 + 3
```

```
[1] 5
```

```
(3*8)/(2*3)
```

```
[1] 4
```

```
log(12)
```

```
[1] 2.484907
```

```
sqrt(121)
```

```
[1] 11
```

You can experiment with various combinations of calculations.

Should you want to modify or repeat a prior calculation, press the `Up Arrow` when the cursor is in the console to cycle through previously executed commands; pressing `Enter` re-runs the selected computation.

On the other hand, you can avoid scrolling through a wall of computations by creating a **variable**. In R, this is done *via* the variable assignment symbols `<-` or `=`.[18] Once a variable exists in memory, the output does not get printed directly unless it is called directly at the prompt, or if the variable assignment is surrounded with a pair of parentheses.

18: There are 3 others such symbols, but no language needs 5 assigners, let alone 2, so we will not introduce them here.

```
x <- 8 + 17
x
```

```
[1] 25
```

```
(y <- 8 + 17)
```

```
[1] 25
```

Variables can be named using any combination of alphanumeric symbols, but the name has to start with a letter (`a-z`, `A-Z`) and cannot contain spaces and punctuation marks other than periods and dashes.

R Packages **Packages** (or libraries) contain pre-compiled functions and objects that could be useful in specific settings.

To install a package, simply type:

```
install.packages("package_name")
```

Take note of the quotation marks. You can type this code directly in the console, followed by a carriage return, or enter it in the script window and click Run in the menu at the top.

The base distribution already comes with some high-priority add-on packages, namely:

KernSmooth	MASS	boot	class
foreign	lattice	mgcv	nlme
rpart	spatial	survival	base
grDevices	graphics	grid	methods
stats	stats4	tcltk	tools
cluster	nnet	datasets	splines

These packages implement standard statistical functionality, for example linear models, classical tests, a huge collection of high-level plotting functions, and tools for survival analysis. Once a package is installed, it needs to be **loaded** before its objects (datasets, functions) can be used. This can be done by typing:[19]

19: Since entering instructions is always done in one of the ways described above, we will stop specifying where and how it must be done.

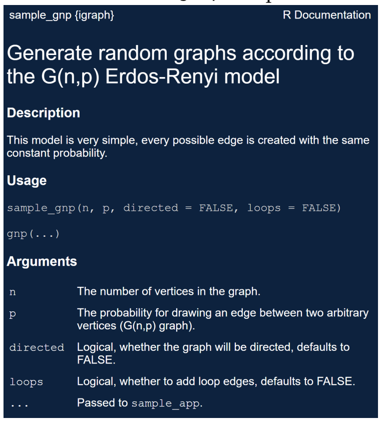
```
libary(package_name)
```

Note the absence of the quotation marks.

For instance, in *Code Components* (see p. 6), we loaded the `igraph` package to take advantage of the pre-compiled functions `sample_gnp()`, `ecount()`, `V()`, and `plot()`. The first 3 functions are not in the base distribution; the last function `plot()` does exist, but it would not know how to handle graph objects without the special instructions provided by `igraph`.

The **help file** for compiled functions can be displayed in the graphical output window by using the reserved character "?", as below (assuming that the `igraph` library has been loaded).[20]

20: Extract of the `igraph` help file below:



sample_gnp {igraph} R Documentation

Generate random graphs according to the G(n,p) Erdos-Renyi model

Description

This model is very simple, every possible edge is created with the same constant probability.

Usage

```
sample_gnp(n, p, directed = FALSE, loops = FALSE)

gnp(...)
```

Arguments

`n`	The number of vertices in the graph.
`p`	The probability for drawing an edge between two arbitrary vertices (G(n,p) graph).
`directed`	Logical, whether the graph will be directed, defaults to FALSE.
`loops`	Logical, whether to add loop edges, defaults to FALSE.
`...`	Passed to `sample_app`.

```
?sample_gnp
```

In more sophisticated code, it is conceivable that we would want to load multiple libraries; because we might forget which function is associated with which library, or even that different libraries use the same name for different functions, it is **good practice** to forego explicitly loading a library in favour of directly fetching the required functionality (the package must be installed first, however). In R, this is done as follows:

```
package_name::function_name(function_parameters)
```

For instance, the graph code from above can be replaced by the following chunk:

```
my_graph_function <- function(my_number_nodes,
                              my_colour,
                              my_density) {
  my_graph = igraph::sample_gnp(my_number_nodes,
                                my_density,
                                directed=FALSE,
                                loops = FALSE)
  if(igraph::ecount(my_graph) >= my_number_nodes) {
           igraph::V(my_graph)$color <- my_colour
  }
  plot(my_graph, vertex.color = igraph::V(my_graph)$color)
}

my_graph_function(30,"green",0.3)
```

Note, however, that this strategy is not always optimal (in particular, when using the **pipeline operator**, see p. 43).

R Essentials Everything you see or create in R is an **object**: vectors, matrices, data frames, even variables (and functions) are objects.

R allows 5 basic classes of objects:

- Character
- Numeric (real numbers)
- Integer (whole numbers)
- Complex
- Logical (True / False)

Each of these classes has **attributes**. An object can have the following attributes:

- names, dimension names
- dimensions
- class
- length
- etc.

An object's various attributes can be accessed using the `attributes()` function. We will have more to say on this topic.

The most basic R object is the **vector**. An empty vector can be created using `vector()`. A vector contains various objects, but all must be of the same class.[21]

21: That can cause unforeseen difficulties as it is not always easy to visually distinguish between a real number (*numeric*) and an *integer*. Furthermore, the digits of a number can be represented as character strings in some cases.

Vectors can also often created using the **combine** (or concatenate) operator `c()` (which makes it a singularly bad idea to use `c` as a variable name).

```
(a <- c(1.8, 4.5))                    # numeric
(b <- c(1 + 2i, 3 - 6i))              # complex
(d <- c(23, 44))                      # integer
```

```
(e <- vector("logical", length = 5))  # logical
(f <- c("abc","def"))                 # character
```

```
[1] 1.8 4.5
[1] 1+2i 3-6i
[1] 23 44
[1] FALSE FALSE FALSE FALSE FALSE
[1] "abc" "def"
```

Comments can be introduced in R code *via* the # symbol: all characters following a pound symbol are ignored by R until the next line of code (so the classes in the example above would not be part of the code proper).

R Data Types and Objects There are various types of R objects.

Vectors As mentioned above, a **vector** contains objects of the same class. We may have a need to mix objects of different classes in a list – this can be done to a vector by **coercion**. This has the effect of 'converting' objects of different types to the same class. For instance:

```
# coercion to character
(vec <- c("Time", 25,TRUE,"retro", 2.22))
# coercion to numeric
(bbb <- c(FALSE, 11))
# coercion to character
(i.a <- c(215,"October"))
```

```
[1] "Time"  "25"    "TRUE"  "retro" "2.22"
[1]  0 11
[1] "215"     "October"
```

We can verify the class of these objects using the `class()` function.

```
class(vec)
class(bbb)
class(i.a)
```

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

To convert the class of a vector, we can use the `as.` command.

```
g <- 10:16        # create a vector of 7 integers
class(g)          # find bar's class
as.numeric(g)     # convert to numeric
class(g)
as.character(g)   # convert to character
```

```
class(g)
```

```
[1] "integer"
[1] 10 11 12 13 14 15 16
[1] "integer"
[1] "10" "11" "12" "13" "14" "15" "16"
[1] "integer"
```

We can change the class of any vector using a similar approach. But be careful – while we can convert a numeric vector into a character one, going the other way will introduce `NA`s (conversion is subject to `R`'s internal class rules).

Lists A **list** is a special type of object which can contain elements of different data types.

```
my.list <- list(254,"abab", TRUE, 0 - 3i)
my.list
```

```
[[1]]
[1] 254

[[2]]
[1] "abab"

[[3]]
[1] TRUE

[[4]]
[1] 0-3i
```

The output of a list differs from that of a vector, since all the objects are of different types. The double bracket `[[1]]` shows the index of the first element and so on. The elements of a list can be extracted by using the appropriate index:

```
my.list[[3]]
```

```
[1] TRUE
```

The single single bracket `[ ]` also has a role: it returns the list element with its index number, instead of the result above.

```
my.list[3]
```

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

Matrices A vector for which rows and columns are explicitly identified is a **matrix**, a 2-dimensional data structure. All the entries of a matrix have to be of the same class. The following code produces a 6 by 3 matrix consisting of the first 18 integers.

```
my.matrix <- matrix(1:18, nrow=6, ncol=3)
my.matrix
```

```
     [,1] [,2] [,3]
[1,]    1    7   13
[2,]    2    8   14
[3,]    3    9   15
[4,]    4   10   16
[5,]    5   11   17
[6,]    6   12   18
```

The dimensions of a matrix can be obtained using either the `dim()` or `attributes()` commands (the matrix dimensions are a matrix's only attributes in R).

```
dim(my.matrix)
attributes(my.matrix)
```

```
[1] 6 3
$dim
[1] 6 3
```

To extract a particular element from a matrix, simply use the appropriate indices. What might you expect to see from the following commands?

```
my.matrix[5,2]          # row 5, col 2
my.matrix[c(1,2,4),2]   # col 2, rows 1, 2, 4
my.matrix[4,2:3]        # row 4, cols 2, 3
my.matrix[,2]           # col 2
my.matrix[4,]           # row 4
my.matrix[c(1,1,4),2]   # col 2, rows 1, 1, 4
```

```
[1] 11
[1]  7  8 10
[1] 10 16
[1]  7  8  9 10 11 12
[1]  4 10 16
[1]  7  7 10
```

As an aside, it is straightforward to create a matrix from any vector, by assigning the dimensions using `dim()`.

For instance, we start by reading in a vector of ages:

```
age <- c(23, 8, 5, 44, 15, 12, 31, 19, 16)
age
```

```
[1] 23  8  5 44 15 12 31 19 16
```

Then reshape the vector as a 3 x 3 matrix:

```
dim(age) <- c(3,3)
age
class(age)
```

```
     [,1] [,2] [,3]
[1,]   23   44   31
[2,]    8   15   19
[3,]    5   12   16
[1] "matrix" "array"
```

Matrices can also be created by joining two vectors (with matching dimensions) using `cbind()` or `rbind()`:

```
x <- c(1, -2, 3, -4, 5, -6)
y <- c(200, 300, 400, 500, 600, 700)
cbind(x, y)
rbind(x,y)
```

```
      x   y
[1,]  1 200
[2,] -2 300
[3,]  3 400
[4,] -4 500
[5,]  5 600
[6,] -6 700
  [,1] [,2] [,3] [,4] [,5] [,6]
x    1   -2    3   -4    5   -6
y  200  300  400  500  600  700
```

```
class(x)
class(y)
class(cbind(x, y))
class(rbind(x, y))
```

```
[1] "numeric"
[1] "numeric"
[1] "matrix" "array"
[1] "matrix" "array"
```

We will discuss how `R` implements regular matrix operations (transpose, multiplication, addition, etc.) in Chapter 3 (*Overview of Linear Algebra*).

Data Frames The **data frame** is R's most commonly-used (and most convenient) data type, especially for data analysis tasks.

Like matrices, we can use data frames to store tabular (rectangular) data, but unlike matrices, a data frame can accommodate lists of vectors of different classes: each column of a data frame acts like a list.

When data is read into R, it is first stored as a data frame.

The following bit of code, for instance, creates a data frame with two columns, name and age:

```
df <- data.frame(
  name = c("Patrick","Brownyn","Elowyn",
           "Llewellyn","Gwynneth"),
  age = c(45,41,19,8,5)
)
df
```

```
       name age
1   Patrick  45
2   Brownyn  41
3    Elowyn  19
4 Llewellyn   8
5  Gwynneth   5
```

Here are some of df attributes:

```
dim(df)
str(df)
nrow(df)
ncol(df)
```

```
[1] 5 2
'data.frame':   5 obs. of  2 variables:
 $ name: chr  "Patrick" "Brownyn" "Elowyn" "Llewellyn" ...
 $ age : num  45 41 19 8 5
[1] 5
[1] 2
```

In the code above, df is the name of data frame, dim() returns its dimensions, str() its structure (i.e., the list of variables stored in the data frame), and nrow() and ncol(), the number of rows and number of columns in the data frame, respectively.

Reading Data and Writing Reading data into a statistical system for analysis, and exporting the results to some other system for report writing, can be frustrating tasks that take far more time than the statistical/data analysis itself, but the former task is required if the latter is to be undertaken in earnest.

We describe the import/export facilities available in R itself or via packages available from Comprehensive R Archive Network (CRAN).

R comes with a few **data reading** functions:

- `read.table()`, `read.csv()` for tabular data;
- `readLines()` for lines of a text file;
- `source()`, `dget()` to read R code files (inverse of `dump()` and `dput()`, respectively);
- `load()` to read-in saved workspaces;
- `unserialize()` to read single R objects in binary form.

There are, of course, numerous R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

`read.table()` The `read.table()` function is one of the most commonly-used functions for reading data. The help file[22] is worth reading if only because the function gets so much use. Its main arguments are:

22: Run `?read.table` in the console.

- `file`, the name of a file, or a connection;
- `header`, logical indicating if the file has a header line;
- `sep`, string indicating how the columns are separated;
- `colClasses`, character vector indicating the class of each column in the dataset;
- `nrows`, number of rows in the dataset;[23]
- `comment.char`, character string indicating comments;[24]
- `skip`, the number of lines to skip from the beginning of the file;
- `stringsAsFactors`, whether character variables are coded as factors or as strings.[25]

23: By default `read.table()` will read the entire file.

24: Defaults to “#”.

25: Defaults to `TRUE` because back in the old days, strings represented levels of a categorical variable; now that text mining is an every day occurrence, that is not always the case.

For small to moderately sized datasets, you can usually call `read.table()` without specifying any other arguments

```
data <- read.table("foo.txt")
```

In this case, R will read in the file `foo.txt` an automatically:

- skip lines that begin with a #;
- figure out how many rows there are (and how much memory needs to be allocated), and
- figure what type of variable is in each column of the table.

Telling R all these things directly makes R run faster and more efficiently. The `read.csv()` function is identical to `read.table()` except that some of the defaults are set differently (such as the `sep` argument).

With much larger datasets, some things can be done to prevent R from choking on the data (a risk as R stores everything in RAM):

- read the help page for `read.table()`, which contains many hints;
- make a rough calculation of the memory required to store the dataset (see on the next page for an example); if the dataset is larger than the amount of RAM on your computer, it is best to stop here;
- set `comment.char = ""` if all lines in the file are uncommented;
- use the `colClasses` argument – specifying this option can make `read.table()` run MUCH faster, often twice as fast.[26] We can figure out the column classes *via* the following code:

26: In order to use this option, we must know the class of each column in the data frame; if all of the columns are “numeric”, for example, then we would simply set `colClasses = "numeric"`.

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                        colClasses = classes)]
```

- set `nrows` – this doesn't make R run faster but it helps with memory usage (a mild overestimate is okay; the Unix tool `wc` can be used to calculate the number of lines in the file).

In general, when using R with larger datasets, it is also useful to know a few things about the operating system:

- how much memory is available on the system?
- what other applications are in use?[27]
- are other users logged into the same system?
- what is the operating system? (some operating systems can limit the amount of memory a single process can access).

27: Close everything that is not required.

For example, suppose we have a data frame with 2,000,000 rows and 100 columns, all of which are numeric data. Roughly speaking, how much memory is required to store this data frame?

On most computers, numeric data is stored using 64 bits of memory (8 bytes). Given that information, we have:

$$\begin{aligned} 2,000,000 \times 100 \times 8 \text{ bytes} &= 1,600,000,000 \text{ bytes} \\ &\approx 1,600 \text{ MB} = 1.6 \text{ GB}. \end{aligned}$$

Reading in a large dataset for which one does not have enough RAM is an easy way to get the computer (or the R session) to freeze. This is usually an unpleasant experience that requires killing the R process, in the best case scenario, or rebooting the computer, in the worst case.

It is always a good idea to do a rough memory requirements calculation before reading in a large dataset.

`txt`, `csv`, and Other Formats

- **Fixed format text files**

```
# Windows only
df = read.table("folder\\file.txt", header=TRUE)
# all OS (including Windows)
df = read.table("folder/file.txt", header=TRUE)
```

The forward slash `/` is supported as a directory delimiter on all operating systems; the double backslash `\\` is only supported under Windows. If the first row of the file includes the name of the variables, these entries will be used to create appropriate names[28] for each of the columns in the dataset. If the first row does not include the names, the `header` option can be left off (or set to `FALSE`), and the variables will be named `V1`, `V2`, ..., `Vn`.

28: Reserved characters such as '$' are changed to '.'

A limit on the number of lines to be read can be specified through the `nrows` option. The `read.table()` function also supports using a URL as a filename or browsing files interactively using `read.table(file.choose())`.

Sometimes data arrives in irregularly-shaped data files (there may be a variable number of fields per line, or some data in the line may describe the remainder of the line). In such cases, a useful generic approach is to read each line into a single character variable, then use character variable functions to extract the contents.

```
df = readLines("file.txt")
df = scan("file.txt")
```

The `readLines()` function returns a character vector with length equal to the number of lines read. A limit on the number of lines to be read can be specified through the `nrows` option. The `scan()` function returns a vector, with entries separated by white space by default. These functions read from standard input, but can also read a file or a URL.

- **Comma-separated value (CSV) files:** the `read.csv()` function takes on much the same parameters as `read.table()`.

```
df = read.csv("folder/file.csv")
```

- **Read sheets from an Excel file:** if the data is available in an Excel file, various possibilities exist, depending on the spreadsheet format.

```
df.xls = gdata::read.xls("file.xls", sheet=1)
df.xlsx = xlsx::read.xlsx("file.xlsx", sheet=1)
```

The sheet can be provided as either a number or a name.[29]

29: The appropriate packages should have been installed beforehand, however.

- **Reading datasets in other formats:** the datasets of interest sometimes comes from another software. The `foreign` library is able to do a native import for some of the most common formats: Stata, Epi Info, Minitab, Octave, SPSS, Systat, and SAS files.[30]

30: The `read.ssd()` function will only work if SAS is installed locally, however.

```
df = foreign::read.dbf("filename.dbf")
df = foreign::read.epiinfo("filename.epiinfo")
df = foreign::read.mtp("filename.mtp")
df = foreign::read.octave("filename.octave")
df = read.ssd("filename.ssd")
df = read.xport("filename.xport")
df = read.spss("filename.sav")
df = read.dta("filename.dta")
df = read.systat("filename.sys")
```

There are analogous functions for **writing data** to files:

- `write.table()` writes tabular data to text files (i.e. CSV);
- `writeLines()`, to write character data line-by-line to a file;

- `dump()`, for dumping a textual representation of multiple R objects;
- `dput()`, for outputting a textual representation of an R object;
- `save()`, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file, and
- `serialize()`, for converting an R object into a binary format for outputting to a file.

There are numerous ways to store data, including structured text file formats like CSV or tab-delimited, or complex binary formats. It is important to take the time to explore the full range of functionality in order to achieve your specific aims.

1.3 More About Programming in R

Many software packages and libraries are available to the data analyst. R not only has the advantage that we can easily use its available packages, but it provides enough flexibility for the analyst who wants to get dirty with the data.

In this section, you will find examples and tips that highlight R's data manipulation features. It is not meant to be a complete introduction, or even necessarily a showcase of good programming practices.

1.3.1 Help and Documentation

R's various help files and demos can be accessed using the following commands (where function_name and search_term correspond to the desired function and/or term):

- `?function_name`
- `example(function_name)`
- `args(function_name)`
- `??search_term`

For instance, the following code would display the help file for the function `glm()` in the bottom graphical output window of RStudio:

```
?glm
```

Most help files contain examples showcasing the use of the function. These can be accessed *via* `example()`.

```
example(glm)
```

We can thus copy the code from the example file, and run it directly at the console.

```
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
print(d.AD <- data.frame(treatment, outcome, counts))
```

```
glm.D93 <- glm(counts ~ outcome + treatment,
               family = poisson())
anova(glm.D93)
summary(glm.D93)
```

```
  treatment outcome counts
1         1       1     18
2         1       2     17
3         1       3     15
4         2       1     20
5         2       2     10
6         2       3     20
7         3       1     25
8         3       2     13
9         3       3     12
Analysis of Deviance Table

Model: poisson, link: log

Response: counts

Terms added sequentially (first to last)


          Df Deviance Resid. Df Resid. Dev
NULL                          8    10.5814
outcome    2   5.4523         6     5.1291
treatment  2   0.0000         4     5.1291

Call:
glm(formula = counts ~ outcome + treatment, family = poisson())

Deviance Residuals:
       1          2          3          4          5
-0.67125   0.96272  -0.16965  -0.21999  -0.95552
                6          7          8          9
          1.04939   0.84715  -0.09167  -0.96656

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.045e+00  1.709e-01  17.815   <2e-16 ***
outcome2    -4.543e-01  2.022e-01  -2.247   0.0246 *
outcome3    -2.930e-01  1.927e-01  -1.520   0.1285
treatment2   1.338e-15  2.000e-01   0.000   1.0000
treatment3   1.421e-15  2.000e-01   0.000   1.0000
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 10.5814  on 8  degrees of freedom
Residual deviance:  5.1291  on 4  degrees of freedom
AIC: 56.761

Number of Fisher Scoring iterations: 4
```

Similarly, the function's arguments can be accessed via `args()`.

```
args(glm)
```

```
function (formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart, offset, control = list(...),
    model = TRUE, method = "glm.fit", x = FALSE, y = TRUE, singular.ok = TRUE,
    contrasts = NULL, ...)
NULL
```

1.3.2 Simple Data Manipulation

So what can we actually do with R?

Loading a Built-In Dataset We can obtain a list of such datasets in the `datsets` package by calling the following function:

```
data()
```

Or those available in all installed packages *via*:

```
data(package = .packages(all.available = TRUE))
```

Let us take a look at the `swiss` built in dataset.[31] We can display the dataset by simply calling it at the prompt, like so:

31: Type `?swiss` to see the help file.

```
swiss
```

Or we can take a look at its first or last `n` entries using the functions `head()` or `tail()`.

```
head(swiss,6)
```

```
             Fertility Agriculture Examination Education Catholic Infant.Mortality
Courtelary        80.2        17.0          15        12     9.96             22.2
Delemont          83.1        45.1           6         9    84.84             22.2
Franches-Mnt      92.5        39.7           5         5    93.40             20.2
Moutier           85.8        36.5          12         7    33.77             20.3
Neuveville        76.9        43.5          17        15     5.16             20.6
Porrentruy        76.1        35.3           9         7    90.57             26.6
```

Assigning Data We can create, assign, and display a vector consisting of a sequence of numbers like this:

```
(x<- c(1:3))
```

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

We can also assign non-sequential numbers:

```
(w <- c(12,-9))
```

```
[1] 12 -9
```

or mixed objects:

```
(v = c(w,"pomplamoose"))
```

```
[1] "12"          "-9"          "pomplamoose"
```

or matrices:

```
(u = t(matrix(1:10,ncol=5)))
```

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

Data Types and Conversion We can test whether objects are of a certain type or class:

```
is.numeric(x)
```

```
[1] TRUE
```

```
is.character(x)
```

```
[1] FALSE
```

```
is.vector(x)
```

```
[1] TRUE
```

```
is.matrix(x)
```

```
[1] FALSE
```

```
is.data.frame(x)
```

```
[1] FALSE
```

```
is.character(w)
```

```
[1] FALSE
```

```
is.character(v)
```

```
[1] TRUE
```

```
is.data.frame(swiss)
```

```
[1] TRUE
```

We can also set an object to be of a specific type:

```
as.numeric(x)
```

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

```
as.character(x)
```

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

```
as.vector(x)
```

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

```
as.matrix(x)
```

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

```
as.data.frame(x)
```

```
  x
1 1
2 2
3 3
```

Or combine two vectors into a single vector:

```
c(y,w)
```

```
[1] 200 300 400 500 600 700  12  -9
```

Or convert vectors to matrices or data frames:

```
cbind(x,y)
```

```
     x   y
[1,] 1 200
[2,] 2 300
[3,] 3 400
[4,] 1 500
[5,] 2 600
[6,] 3 700
```

```
rbind(x,y)
```

```
  [,1] [,2] [,3] [,4] [,5] [,6]
x    1    2    3    1    2    3
y  200  300  400  500  600  700
```

```
data.frame(x,y)
```

```
  x   y
1 1 200
2 2 300
3 3 400
4 1 500
5 2 600
6 3 700
```

Conversely, we can convert a matrix to a vector:

```
as.vector(u)
```

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

or a matrix to a data frame:

```
as.data.frame(u)
```

```
  V1 V2
1  1  2
2  3  4
3  5  6
4  7  8
5  9 10
```

or a data frame to a matrix:

```
swiss_matrix=as.matrix(swiss)
head(swiss_matrix)
```

```
             Fertility Agriculture Examination Education Catholic Infant.Mortality
Courtelary        80.2        17.0          15        12     9.96             22.2
Delemont          83.1        45.1           6         9    84.84             22.2
Franches-Mnt      92.5        39.7           5         5    93.40             20.2
Moutier           85.8        36.5          12         7    33.77             20.3
Neuveville        76.9        43.5          17        15     5.16             20.6
Porrentruy        76.1        35.3           9         7    90.57             26.6
```

Writing Functions One of R's most advantageous feature is its flexibility: what if we want to write our own functions? The template for all functions is a block of code that looks like:

```
my.function <- function(arg1,arg2, ..., argn) {
     # what my.function does
     # typically involving the arguments
}
```

Here are some (truly) simple examples: here is a function, `my.product()`, that computes the product of two arguments x and y.[32]

32: This is not a very interesting function as the standard multiplication $*$ is already defined in R, but this is just an illustration of the functionality.

```
my.product <- function (x,y) {
    x*y
}
```

Note that the function definition must be compiled (the code must be run) before it can be called in the code.

There are multiple ways to call `my.product()` for arguments `x=12` and `y=-2`.

```
my.product(x=12,y=-2)
my.product(y=-2,x=12)
my.product(12,-2)
my.product(-2,12)
```

```
[1] -24
[1] -24
[1] -24
[1] -24
```

The first two calls reflect better programming practices. The last of those is acceptable because multiplication is commutative, but it is risky to play with the arguments this way.

For instance, consider another simple function `my.quotient()`:

```
my.quotient <- function (x,y) {
    x/y
}
```

We call `my.quotient()` on `x=12` and `y=-2`.

```
my.quotient(x=12,y=-2)
my.quotient(y=-2,x=12)
my.quotient(12,-2)
```

```
[1] -6
[1] -6
[1] -6
```

but

```
my.quotient(-2,12)
```

```
[1] -0.1666667
```

When the parameters are not specified in the function call, their implied order reverts to the declared order in the definition (1st = x, 2nd = y).

And what might we expect to happen with this call?

```
my.quotient(12,0)
```

```
[1] Inf
```

1.3.3 Exploring Data

`R` is good tool for data exploration. Let us examine the `swiss` dataset in detail.

We start by displaying the first few rows of the dataset (3, in this case):

```
head(swiss,3)
```

```
             Fertility Agriculture Examination Education Catholic Infant.Mortality
Courtelary        80.2        17.0          15        12     9.96             22.2
Delemont          83.1        45.1           6         9    84.84             22.2
Franches-Mnt      92.5        39.7           5         5    93.40             20.2
```

We could also display the last few entries (6, here):

```
tail(swiss,6)
```

```
             Fertility Agriculture Examination Education Catholic Infant.Mortality
Neuchatel         64.4        17.6          35        32    16.92             23.0
Val de Ruz        77.6        37.6          15         7     4.97             20.0
ValdeTravers      67.6        18.7          25         7     8.65             19.5
V. De Geneve      35.0         1.2          37        53    42.34             18.0
Rive Droite       44.7        46.6          16        29    50.43             18.2
Rive Gauche       42.8        27.7          22        29    58.33             19.3
```

We can also get an idea as to the dataset's structure with `str()`:

```
str(swiss)
```

```
'data.frame':   47 obs. of  6 variables:
 $ Fertility       : num  80.2 83.1 92.5 85.8 76.9 76.1 83.8 92.4 82.4 82.9 ...
 $ Agriculture     : num  17 45.1 39.7 36.5 43.5 35.3 70.2 67.8 53.3 45.2 ...
 $ Examination     : int  15 6 5 12 17 9 16 14 12 16 ...
 $ Education       : int  12 9 5 7 15 7 7 8 7 13 ...
 $ Catholic        : num  9.96 84.84 93.4 33.77 5.16 ...
 $ Infant.Mortality: num  22.2 22.2 20.2 20.3 20.6 26.6 23.6 24.9 21 24.4 ...
```

We can extract the column names with the function `colnames()`:

```
colnames(swiss)
```

```
[1] "Fertility"        "Agriculture"      "Examination"      "Education"
[5] "Catholic"         "Infant.Mortality"
```

or display a specific column of the data frame, say `Education`, with the `$` operator:

```
swiss$Education
```

```
 [1] 12  9  5  7 15  7  7  8  7 13  6 12  7 12  5  2  8 28 20  9 10  3 12  6  1
[26]  8  3 10 19  8  2  6  2  6  3  9  3 13 12 11 13 32  7  7 53 29 29
```

This cannot be done with a matrix, however – the following code will provide an error message:

```
swiss_matrix$Education
```

```
Error in swiss_matrix$Education :
  $ operator is invalid for atomic vectors
```

To extract the `Education` column from a matrix, identify its column index and use this, instead:

```
swiss_matrix[,4]
```

```
  Courtelary     Delemont Franches-Mnt      Moutier   Neuveville
          12            9            5            7           15
   Porrentruy        Broye        Glane      Gruyere       Sarine
            7            7            8            7           13
   ...
     Le Locle    Neuchatel   Val de Ruz ValdeTravers V. De Geneve
           13           32            7            7           53
  Rive Droite  Rive Gauche
           29           29
```

Just as one would expect from the behaviour of `colnames()`, `rownames()` extracts the data frame's row names:

```
rownames(swiss)
```

```
 [1] "Courtelary"   "Delemont"     "Franches-Mnt" "Moutier"
     ...
[46] "Rive Droite"  "Rive Gauche"
```

The summary statistics (5-pt summary + mean + number of missing variables for numerical variables; frequency table for others) can be obtained for all data frame's variables simultaneously:

```
summary(swiss)
```

```
   Fertility      Agriculture     Examination      Education
 Min.   :35.00   Min.   : 1.20   Min.   : 3.00   Min.   : 1.00
 1st Qu.:64.70   1st Qu.:35.90   1st Qu.:12.00   1st Qu.: 6.00
 Median :70.40   Median :54.10   Median :16.00   Median : 8.00
 Mean   :70.14   Mean   :50.66   Mean   :16.49   Mean   :10.98
 3rd Qu.:78.45   3rd Qu.:67.65   3rd Qu.:22.00   3rd Qu.:12.00
 Max.   :92.50   Max.   :89.70   Max.   :37.00   Max.   :53.00
    Catholic       Infant.Mortality
 Min.   :  2.150   Min.   :10.80
 1st Qu.:  5.195   1st Qu.:18.15
 Median : 15.140   Median :20.00
 Mean   : 41.144   Mean   :19.94
 3rd Qu.: 93.125   3rd Qu.:21.70
 Max.   :100.000   Max.   :26.60
```

More in-depth statistics are available with psych's `describe()`:

```
psych::describe(swiss)
```

```
                 vars  n mean   sd   med trim  mad  min   max range skew kurt   se
Fertility           1 47 70.1 12.4  70.4 70.6 10.2 35.0  92.5 57.5 -0.46  0.2 1.82
Agriculture         2 47 50.6 22.7  54.1 51.1 23.8  1.2  89.7 88.5 -0.32 -0.8 3.31
Examination         3 47 16.4  7.9  16.0 16.0  7.4  3.0  37.0 34.0  0.45 -0.1 1.16
Education           4 47 10.9  9.6   8.0  9.3  5.9  1.0  53.0 52.0  2.27  6.1 1.40
Catholic            5 47 41.1 41.7  15.1 39.1 18.6  2.1 100.0 97.8  0.48 -1.6 6.08
Infant.Mortality    6 47 19.9  2.9  20.0 19.9  2.8 10.8  26.6 15.8 -0.33  0.7 0.42
```

The correlation matrix is obtained pretty much as one would expect:

```
cor(swiss)
```

```
                    F    A   Ex   Ed    C   IM
Fertility         1.0  0.3 -0.6 -0.6  0.4  0.4
Agriculture       0.3  1.0 -0.6 -0.6  0.4 -0.0
Examination      -0.6 -0.6  1.0  0.6 -0.5 -0.1
Education        -0.6 -0.6  0.6  1.0 -0.1 -0.0
Catholic          0.4  0.4 -0.5 -0.1  1.0  0.1
Infant.Mortality  0.4 -0.0 -0.1 -0.0  0.1  1.0
```

We can obtain the data frame's number of rows:

```
nrow(swiss)
```

```
[1] 47
```

or the summary of a single variable:

```
summary(swiss$Fertility)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  35.00   64.70   70.40   70.14   78.45   92.50 
```

We can also find all observations for which a feature takes on a value greater than a certain threshold, say:

```
swiss$Fertility>50
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

or provide summary information for the logical vector:

```
summary(swiss$Fertility>50)
```

```
   Mode   FALSE    TRUE 
logical       3      44 
```

```
table(swiss$Fertility>50)
```

```
FALSE  TRUE 
    3    44 
```

The logical vector can be used as an index: for instance, here is the dataset only for those observations where `Fertility` was greater than 50.

```
swiss[swiss$Fertility>50,]
```

with

```
nrow(swiss[swiss$Fertility>50,])
```

```
[1] 44
```

We could also replace the threshold; for instance, here is the dataset for observations data where `Fertility` is in the top 50%:

```
swiss[swiss$Fertility>median(swiss$Fertility),]
```

```
             Fertility Agriculture Examination Education Catholic
Courtelary        80.2        17.0          15        12     9.96
Delemont          83.1        45.1           6         9    84.84
...
Le Locle          72.7        16.7          22        13    11.22
Val de Ruz        77.6        37.6          15         7     4.97
             Infant.Mortality
Courtelary               22.2
Delemont                 22.2
...
Le Locle                 18.9
Val de Ruz               20.0
```

or, solely the `Fertility` and `Education` variables for observations where `Fertility` is in the top 50%:

```
swiss[swiss$Fertility > median(swiss$Fertility),
      c("Fertility","Education")]
```

```
             Fertility Education
Courtelary        80.2        12
Delemont          83.1         9
...                ...       ...
Le Locle          72.7        13
Val de Ruz        77.6         7
```

or those observations for which `Fertility` was maximal:

```
swiss[swiss$Fertility == max(swiss$Fertility),]
```

```
             Fertility Agriculture Examination Education Catholic Infant.Mortality
Franches-Mnt      92.5        39.7           5         5     93.4   20.2
```

1.3.4 A Word About NAs

`NA` values in `R` can create some havoc. Be careful!

To illustrate some of the issues, create a dataset by sampling 100 values (with replacement) among the values $\{1, 2, 3, 4, \text{NA}\}$.[33]

33: Your sample will be different.

```
test = sample(c(1:4,NA),100, replace=TRUE)
```

We can summarize `test` as follows:

```
summary(test)   # 5pt summary + mean + number of NAs
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  1.000   1.500   3.000   2.549   3.500   4.000      29
```

We can read the mean from the output, or we could try to compute it directly, using `mean()`:

```
mean(test)
```

```
[1] NA
```

What is happening? The function `mean()` does not know how it should handle the `NA` values; without further guidance, it elects to throw everything akimbo.

Compare with:

```
mean(test, na.rm=TRUE)
```

```
[1] 2.549296
```

1.3.5 Loops and Conditional Statements

R allows for **flow control** through loops and conditional statements:

- `if()` and `ifelse()` – when a condition holds, do thing 1, when it does not, do thing 2;
- `for()` – iterate a procedure for a fixed number of steps;
- `while()` – repeat steps as long as some condition holds.

High-level interpreted languages (like R) are **slower** than low-level and/or compiled languages. To get around this issue, interpreted languages will sometimes hand off[34] some operations to functions written in lower-level languages (like C).

34: "Behind the scenes", so to speak.

In order to take advantage of this, certain programming strategies are recommended when working with list, vectors, arrays, data frames, and so on, namely **vectorized** functions (see the family of `apply()` functions in R). In particular, we try to avoid cycling through each item of a list, and instead use special functions that map a chosen function or operation to every item in the list (in R, this can be done with the `apply` family of functions, among others).

This can run counter to habits gained when learning other languages, in which **for** and **while** loops, for instance, might have been emphasized. Consequently, we elect NOT to introduce such loops at this stage. The syntax is rather intuitive and will be easy to understand when we encounter it in examples.

The `ifelse()` statement is quite powerful and can speed-up and simplify data frame operations, however, and we take the time to illustrate how it can be used.

We can easily create a new `swiss` column determining whether the `Fertility` variable, say, is above a certain threshold (in which case it should take the value 1) or not (0):

```r
swiss$threshold <-ifelse(swiss$Fertility>50,1,0)
```

```
 [1] 1 1 1 ... 1 1 1
[45] 0 0 0
```

There will be other opportunities to use these functions; the best way to get the hang of R is to practice and debug.

1.4 The `tidyverse`

R is a **functional language**, which means that it uses nested parentheses, which can make code difficult to read.[35]

35: Exhibit A: everything up to now.

1.4.1 Pipeline Operator

The **pipeline operator** `|>` (formerly `%>%`) and the `dplyr` package can be used to remedy the situation. Hadley Wickham[36] provided an example to illustrate how it works:

36: See [1] for everything there is to know about pipelines and tidy data.

```
hourly_delay <- filter(
   summarise(
     group_by(
       filter(
         flights,
         !is.na(dep_delay)
       ),
       date, hour
     ),
     delay = mean(dep_delay),
     n = n()
   ),
   n > 10
)
```

Without necessarily knowing how each of the internal functions works, we can still get a sense for what the overall nested structure does, and realize (albeit, with a fair amount of work) that the basic object on which we operate is the `flights` data frame.

The pipeline operator `|>` removes the need for nesting function calls, in favor of passing data from one function to the next:

```
library(dplyr)
hourly_delay <- flights |>
    filter(!is.na(dep_delay)) |>
    group_by(date, hour) |>
    summarise(delay = mean(dep_delay),n = n()) |>
    filter(n > 10)
```

It is now obvious that the flights data frame is the base object, for instance – the **gap** between pseudo-code and “code that runs” is significantly reduced. The beauty of this approach is that the block of code can now be ‘read’ directly: the `flights` data frame is

1. filtered (to remove missing values of the `dep_delay` variable);
2. grouped by `hours` within `days`;
3. the mean `delay` is calculated within `groups`, and
4. the mean `delay` is returned for those hours with more than `n > 10` flights.

The **pipeline rules** are simple – the object immediately to the left of the pipeline is passed as the first argument to the function immediately to its right:

- `data |> function` is equivalent to `function(data)`
- `data |> function(arg=value)` is equivalent to `function(data, arg=value)`

For instance:

```
library(dplyr)
swiss |> summary()
```

```
   Fertility      Agriculture      Examination      Education    
 Min.   :35.00   Min.   : 1.20   Min.   : 3.00   Min.   : 1.00  
 1st Qu.:64.70   1st Qu.:35.90   1st Qu.:12.00   1st Qu.: 6.00  
 Median :70.40   Median :54.10   Median :16.00   Median : 8.00  
 Mean   :70.14   Mean   :50.66   Mean   :16.49   Mean   :10.98  
 3rd Qu.:78.45   3rd Qu.:67.65   3rd Qu.:22.00   3rd Qu.:12.00  
 Max.   :92.50   Max.   :89.70   Max.   :37.00   Max.   :53.00  
    Catholic       Infant.Mortality   threshold     
 Min.   :  2.150   Min.   :10.80    Min.   :0.0000  
 1st Qu.:  5.195   1st Qu.:18.15    1st Qu.:1.0000  
 Median : 15.140   Median :20.00    Median :1.0000  
 Mean   : 41.144   Mean   :19.94    Mean   :0.9362  
 3rd Qu.: 93.125   3rd Qu.:21.70    3rd Qu.:1.0000  
 Max.   :100.000   Max.   :26.60    Max.   :1.0000  
```

The `magrittr` vignette provides additional information on the `magrittr` package, on which `dplyr` is based.

1.4.2 Tidy Data

The pipeline operator is also compatible with the `tidyverse` suite of packages, championed by Wickham;[37] cheat sheets are available here
.

37: Including the ever popular `ggplot2` (see Section 16.5, *`ggplot2` Visualizations in R*).

Tidy data has a specific structure:

- each variable is a column;
- each observation is a row;
- each type of observational unit is a table.

Two `tidyr` functions are used to reshape tables to a tidy format: `gather()` and `spread()` – `gather()` requires:

- a data frame to reshape;
- a key column (against which to reshape);
- a value column (which will contain the new variable of interest), and
- the indices of the columns that need to be collapsed.

Consider the following dataset:

```
cities <- data.frame(
  city=c("Toronto","Montreal","Vancouver",
         "Ottawa","Calgary","Edmonton",
         "Quebec City","Winnipeg","Hamilton"),
  prov=c("Ontario","Quebec","BC",
    "Ontario","Alberta","Alberta",
    "Quebec","Manitoba","Ontario"),
  pop.2016=c(6202225,4291732,2642825,
             1488307,1481806,1418118,
             839311,834678,785184),
  pop.2011=c(5928040,4104074,2463431,
    1371576,1392609,1321441,
    806406,783099,747545)
)
cities
```

```
         city     prov pop.2016 pop.2011
1     Toronto  Ontario  6202225  5928040
2    Montreal   Quebec  4291732  4104074
3   Vancouver       BC  2642825  2463431
4      Ottawa  Ontario  1488307  1371576
5     Calgary  Alberta  1481806  1392609
6    Edmonton  Alberta  1418118  1321441
7 Quebec City   Quebec   839311   806406
8    Winnipeg Manitoba   834678   783099
9    Hamilton  Ontario   785184   747545
```

It is not presented in a tidy format, because populations show up in **two** columns. In tidy format, it would instead look like:

```
cities.tidy <- tidyr::gather(cities,"year","population",
                             3:4)
cities.tidy$year <- ifelse(cities.tidy$year=="pop.2016",
                             2016,2011)
cities.tidy
```

```
          city     prov year population
1      Toronto  Ontario 2016    6202225
2     Montreal   Quebec 2016    4291732
3    Vancouver       BC 2016    2642825
4       Ottawa  Ontario 2016    1488307
5      Calgary  Alberta 2016    1481806
6     Edmonton  Alberta 2016    1418118
7  Quebec City   Quebec 2016     839311
8     Winnipeg Manitoba 2016     834678
9     Hamilton  Ontario 2016     785184
10     Toronto  Ontario 2011    5928040
11    Montreal   Quebec 2011    4104074
12   Vancouver       BC 2011    2463431
13      Ottawa  Ontario 2011    1371576
14     Calgary  Alberta 2011    1392609
15    Edmonton  Alberta 2011    1321441
16 Quebec City   Quebec 2011     806406
17    Winnipeg Manitoba 2011     783099
18    Hamilton  Ontario 2011     747545
```

`spread()`, on the other hand, generates multiple columns from two columns; it requires a data frame to reshape; a **key** column, and values in the value column to become new values.

For instance, we could reverse the "tidying" of `cities.tidy` with:

```
cities.back.to.wide <- tidyr::spread(cities.tidy,year,
                                     population)
colnames(cities.back.to.wide) <- c("city","prov",
                                   "pop.2011","pop.2016")
cities.back.to.wide
```

```
         city     prov pop.2011 pop.2016
1     Calgary  Alberta  1392609  1481806
2    Edmonton  Alberta  1321441  1418118
3    Hamilton  Ontario   747545   785184
4    Montreal   Quebec  4104074  4291732
5      Ottawa  Ontario  1371576  1488307
6 Quebec City   Quebec   806406   839311
7     Toronto  Ontario  5928040  6202225
8   Vancouver       BC  2463431  2642825
9    Winnipeg Manitoba   783099   834678
```

Other useful wrangling functions include `separate()` and `unite()`. What do you think these do?[38]

38: How could you find out?

1.4.3 The `dplyr` Package

The `dplyr` package provides functions to transform tabular data. Its most useful functions are compatible with the pipeline operator `|>`:

- `select()`: to extract a subset of variables from the data frame;
- `filter()`: to extract a subset of observations from the data frame;
- `arrange()`: to sort the data frame;
- `mutate()`: to create new variables from existing variables;
- `summarise()`: to create so-called pivot tables;
- `group_by()`: ... self-evident?

We will showcase these functions with the help of various examples. Try to guess what the outputs would be before looking at them.[39]

39: We do not explicitly state the `dplyr::xyz` dependency since we already had to load the `dplyr` package to gain access to the pipeline operator `|>`.

```
cities |> select(prov,pop.2016)
```

```
      prov pop.2016
1  Ontario  6202225
2   Quebec  4291732
3       BC  2642825
4  Ontario  1488307
5  Alberta  1481806
6  Alberta  1418118
7   Quebec   839311
8 Manitoba   834678
9  Ontario   785184
```

```
cities |> select(-pop.2016)
```

```
         city     prov pop.2011
1     Toronto  Ontario  5928040
2    Montreal   Quebec  4104074
3   Vancouver       BC  2463431
4      Ottawa  Ontario  1371576
5     Calgary  Alberta  1392609
6    Edmonton  Alberta  1321441
7 Quebec City   Quebec   806406
8    Winnipeg Manitoba   783099
9    Hamilton  Ontario   747545
```

```
cities |> filter(pop.2016>1000000)
```

```
       city    prov pop.2016 pop.2011
1   Toronto Ontario  6202225  5928040
2  Montreal  Quebec  4291732  4104074
3 Vancouver      BC  2642825  2463431
4    Ottawa Ontario  1488307  1371576
5   Calgary Alberta  1481806  1392609
6  Edmonton Alberta  1418118  1321441
```

```
cities |> filter(pop.2016>1000000,
                 prov %in% c("Ontario","Quebec"))
```

```
      city    prov pop.2016 pop.2011
1  Toronto Ontario  6202225  5928040
2 Montreal  Quebec  4291732  4104074
3   Ottawa Ontario  1488307  1371576
```

```
cities |> mutate(pop.increase = pop.2016/pop.2011-1)
```

```
         city     prov pop.2016 pop.2011 pop.increase
1     Toronto  Ontario  6202225  5928040   0.04625222
2    Montreal   Quebec  4291732  4104074   0.04572481
3   Vancouver       BC  2642825  2463431   0.07282282
4      Ottawa  Ontario  1488307  1371576   0.08510721
5     Calgary  Alberta  1481806  1392609   0.06405028
6    Edmonton  Alberta  1418118  1321441   0.07316028
7 Quebec City   Quebec   839311   806406   0.04080451
8    Winnipeg Manitoba   834678   783099   0.06586524
9    Hamilton  Ontario   785184   747545   0.05035015
```

```
cities |> summarise(median.2011=median(pop.2011),
variance.2011=var(pop.2011))
```

```
  median.2011 variance.2011
1     1371576  3.209519e+12
```

```
cities |> summarise(mean.2016=mean(pop.2016),
                    sum.2016=sum(pop.2016), n=n())
```

```
  mean.2016 sum.2016 n
1   2220465 19984186 9
```

```
cities |> arrange(pop.2016)
```

```
         city     prov pop.2016 pop.2011
1    Hamilton  Ontario   785184   747545
2    Winnipeg Manitoba   834678   783099
3 Quebec City   Quebec   839311   806406
4    Edmonton  Alberta  1418118  1321441
5     Calgary  Alberta  1481806  1392609
6      Ottawa  Ontario  1488307  1371576
7   Vancouver       BC  2642825  2463431
8    Montreal   Quebec  4291732  4104074
9     Toronto  Ontario  6202225  5928040
```

```
cities |> arrange(desc(pop.2011))
```

```
         city     prov pop.2016 pop.2011
1     Toronto  Ontario  6202225  5928040
2    Montreal   Quebec  4291732  4104074
3   Vancouver       BC  2642825  2463431
4     Calgary  Alberta  1481806  1392609
5      Ottawa  Ontario  1488307  1371576
6    Edmonton  Alberta  1418118  1321441
7 Quebec City   Quebec   839311   806406
8    Winnipeg Manitoba   834678   783099
9    Hamilton  Ontario   785184   747545
```

```
cities |> arrange(prov,desc(pop.2016))
```

```
         city     prov pop.2016 pop.2011
1     Calgary  Alberta  1481806  1392609
2    Edmonton  Alberta  1418118  1321441
3   Vancouver       BC  2642825  2463431
4    Winnipeg Manitoba   834678   783099
5     Toronto  Ontario  6202225  5928040
6      Ottawa  Ontario  1488307  1371576
7    Hamilton  Ontario   785184   747545
8    Montreal   Quebec  4291732  4104074
9 Quebec City   Quebec   839311   806406
```

```
cities |> group_by(prov) |>
  summarise(mean.2016 = mean(pop.2016))
```

```
# A tibble: 5 × 2
  prov      mean.2016
  <chr>         <dbl>
1 Alberta    1449962
2 BC         2642825
3 Manitoba    834678
4 Ontario    2825239.
5 Quebec     2565522.
```

```
cities |> mutate(pop.increase = pop.2016/pop.2011-1) |>
  select(city, pop.increase) |>
  arrange(desc(pop.increase))
```

```
         city pop.increase
1      Ottawa   0.08510721
2    Edmonton   0.07316028
3   Vancouver   0.07282282
4    Winnipeg   0.06586524
5     Calgary   0.06405028
6    Hamilton   0.05035015
7     Toronto   0.04625222
8    Montreal   0.04572481
9 Quebec City   0.04080451
```

dplyr also comes with "database" functionality (bind_cols(), bind_-rows(),union(),intersect(),setdiff(),left_join(),inner_join(), semi_join(), anti_join(), etc.).

Do not hesitate to bookmark, consult, and borrow from the excellent [1] (and from the subsequent chapters) for more examples, and to practice, practice, practice: we learn programming by programming.

1.5 Basics of Python

Python is another object-oriented language (OOL). It was created in the early 90's but was not popularized until the 00's. It lends itself to writing structured, easy-to-read computer code.[40]

40: **Indentation** matters in Python: in some of the code boxes of the next two sections, we have been forced to sometimes introduce a carriage return in order for the code to fit the width of the available box – in instances where a new line starts with indentation, it is important to verify if that line is completing code from the previous line, in which case it should be entered as a single line at the prompt.

It is intended to be easier to understand and learn than other OOLs. One of its strength is that it has a **massive** base of open-source modules, which allow programmers to implement very sophisticated functionality simply by making a few function calls (not unlike R's packages).

More information is available from the Python Software Foundation, on Stack Exchange (and similar sites), and in reference manuals, such as Jake VanderPlas' *A Whirlwind Tour of Python* or the Python 3 documentation.

1.5.1 IDE for Python

Anaconda and Jupyter are popular data science Python **integrated development environments** (IDE); Rodeo, Spyder, PyCharm, Ninja (an others) also provide RStudio-like functionality for Python. Installation instructions are available on the respective websites.

1.5.2 Introduction to Python

The content of the next two sections is intended to help data analysts get a better sense of how Python could be used for data analysis. They are not designed to teach the ins and outs of Python programming. Instead, they illustrate typical tasks through examples.[41]

41: Note that these examples require Python 3.5 or higher.

Fundamentals Let us start with the basics.

Using Python as a Scientific Calculator Mathematical expressions can easily be evaluated numerically in Python. For scientific calculations, one should import the math module (package/library) which contains many mathematical functions.

It is important to note that Python also provides facilities for integer arithmetic which will be covered later. In this section, only floating-point calculations are used.

Modules can be imported using the import function.

```
import math
```

We can call pre-compiled functions in a module by prepending the module name (with a period) to the function name: module.function_-name() is the Python equivalent of package::function_name() in R.

For instance, there is a cos function in the math module: it is called using math.cos().

We can evaluate $\cos(\sqrt{\pi})$ with:

```
math.cos(math.sqrt(math.pi))
```

```
-0.20029354112337366
```

$\arctan(2^5/3)$ with

```
math.atan(2**5 / 3)
```

```
1.477319545636307
```

and $\ln(1+e^4)$ with

```
math.log(1 + math.exp(4))
```

```
4.0181499279178094
```

Using Variables to Hold Intermediate Results It could be helpful to break complex calculations into smaller steps. Variables can be used to store intermediate results. We will see later how variables are used in algorithmic settings.

For instance, we could break down the evaluation of $\exp(\sin(\sqrt{2}+2))$ into three parts:

- $x = \sqrt{2}$
- $y = \sin(x + 2)$
- $z = \exp(y)$

```
x = math.sqrt(2)
y = math.sin(x+2)
z = math.exp(y)
```

In order to display the values taken by the variables, we must call on them separately, as follows:

```
x,y,z
```

```
(1.4142135623731, -0.26925647329403, 0.7639472984402)
```

The variables are saved even when they are not displayed, however.

Numbers as Formatted Strings Quite often, we may want to control the way numbers are displayed (this can come in handy when reporting results). For example, we may wish to display no more than 4 decimal places for all real numbers, or we may want to pad numbers with zeros so that they all have a given width.

The following block illustrates a number of ways to obtain **formatted strings** of the number 12.3456789. For more details on the format specification mini-language, please consult the documentation .

Note that a string must be enclosed within double quotes or single quotes. We will discuss general string operations shortly.

```
x = 12.3456789
```

We can format the number as a string of width 10, with 2 decimal places:

```
"{:10.2f}".format(x)
```

```
'     12.35'
```

Or as a string with 4 decimal places:

```
"{:.4f}".format(x)
```

```
'12.3457'
```

or as a zero-padded string of width 5, with no decimal:

```
"{:05.0f}".format(x)
```

```
'00012'
```

Fixed Decimals **Floating-point numbers** are usually shunned as they are inherently inexact. For example, we might be bewildered to find out what the following sum amounts to:

```
2.2 + 1.1
```

```
3.3000000000000003
```

The result 3.3000000000000003 is definitely not what we would expect as a sum, namely, 3.3.

The `decimal` module allows us to express decimal numbers *exactly* (see the documentation for more information). Let's look at a few examples of working with `decimal` and `Decimal()`.

We start by defining `x` and `y` as the **fixed decimal** values 1.1 and 1.2, respectively. Note that the numbers must entered as strings.

```
import decimal
x = decimal.Decimal("1.1")
y = decimal.Decimal("2.2")
```

These computations behave as we would expect:

```
print(x+y)
print(y/x)
print(x**decimal.Decimal("3"))
```

```
3.3
2
1.331
```

If we do not enter the numbers as strings, they will be treated as floating-point numbers, and then be converted to a string, leading to unexpected results.

```
x = decimal.Decimal(1.1)
y = decimal.Decimal(2.2)

print(x+y )
```

```
3.300000000000000266453525910
```

Rounding works as one would expect when variables are correctly declared as fixed decimals:

```
z = decimal.Decimal("3.1416")
round(z, 3)
```

```
Decimal('3.142')
```

Once fixed decimals are used, we must use mathematical functions provided by the `decimal` module in order to stay within that module (unfortunately, trigonmetric functions are not available).

For instance, if:

```
a= decimal.Decimal("0.16")
```

then

```
print(a.sqrt())
print(a.ln())
print(a.log10())
```

```
0.4
-1.832581463748310130367054424
-0.7958800173440752191450444211
```

The same results could be obtained using the `math` module functions:

```
import math
print(math.sqrt(a))
print(math.log(a))
print(math.log10(a))
```

```
0.4
-1.8325814637483102
-0.7958800173440752
```

List and Tuples **Lists** and **tuples** are important objects in `Python` programming. Even though we will be mostly using `numpy` arrays and certain `pandas` objects instead of lists later on, it is useful to learn the basics of lists as some of the concepts are transferrable.

List Creation A **list** holds a sequence of objects, who do not all have to be the same type. One way to create a list is to enclose the elements, separated by commas, with square brackets.

Let us illustrate this concept with a simple list containing three objects.

```
x = [3,'a',5.1]
```

We can extract the elements using indices (note that the first element corresponds to index 0, the second to index 1, etc.):

```
x[0]
x[1]
x[2]
```

```
3
'a'
5.1
```

The type of each of the elements can be found using:

```
print(type(x[0]))
print(type(x[1]))
print(type(x[2]))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
```

We can also "multiply" an element and transform it into a longer list:

```
['Ho']*10
```

```
['Ho', 'Ho', 'Ho', 'Ho', 'Ho', 'Ho', 'Ho', 'Ho', 'Ho', 'Ho']
```

or create a list of integers ranging from 0 to $n-1$, or from a to $b-1$:

```
n = 5
list(range(n))

a=3
b=7
list(range(a,b))
```

```
[0, 1, 2, 3, 4]
[3, 4, 5, 6]
```

Tuples Tuples are list-like objects, but with the following differences:

- they are defined with parentheses instead of square brackets (sometimes, the parentheses can be omitted);
- they are **immutable** (once created, they cannot be modified).

For instance, if

```
t = (1,'a',4.5)
```

then we can obtain the length of `t` and print its 2nd element using

```
print(len(t))
print(t[1])
```

```
3
a
```

but we cannot change the value of the third element of `t` or append a new value to `t`: both commands in the next block of code are illegal:

```
t[2]=1
t.append(5)
```

although the same command applied to the list `x` would be legal:

```
x[2]=1
x.append(5)
print(x)
```

```
[3, 'a', 1, 5]
```

If we know the dimension of a tuple `t`, we can also use an **extract pattern** to extract the individual components, as the following examples illustrate.

```
t = (1, 'two', 3.0)
fst, snd, trd = t
print(fst, snd, trd )
```

```
two 3.0
```

We could use '_' (place holder) to extract the second component, say.

```
_, s, _ = t
print(s)
```

```
two
```

What do you think is happening on the next page?

```
days = [(0,"Sun"), (1, "Mon"), (2, "Tue"), (3, "Wed"),
        (4, "Thu"), (5, "Fri"), (6, "Sat")]
for n, d in days:
    print(d+" is represented by " + str(n))
```

```
Sun is represented by 0
Mon is represented by 1
Tue is represented by 2
Wed is represented by 3
Thu is represented by 4
Fri is represented by 5
Sat is represented by 6
```

List Comprehension **List comprehension** is a powerful way to create lists, based on set notation. Before we get into the technical details, let us look at some examples.

We start by importing solely the function `sqrt()` from the `math` module;[42] we also declare an index list `x`:

42: Doing so means that we will not require the prefix `math.` in order to invoke `sqrt()`.

```
from math import sqrt
x = [1, 4, 9, 16]
print(x)
```

```
[1, 4, 9, 16]
```

We can now build new lists from `x`, such as the list of the squares of the elements of `x`:

```
y = [a**2 for a in x]
print(y)
```

```
[1, 16, 81, 256]
```

the list of the square roots of the elements of `x` greater than 4:

```
z = [sqrt(b) for b in x if (b > 4)]
print(z)
```

```
[3.0, 4.0]
```

or the list of integers from 0 to 9 (equivalent to `range(10)`):

```
u = [ c for c in range(10) ]
print(u)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The most basic form of list comprehension is `[f(x) for x in l]`, where `l` is a list (or an **iterable**) and `f(x)` is an expression in `x`. It creates a list obtained by applying `f` to each element or iterate in `l`.[43]

43: `range` provides an example of an iterable. One way to think of an iterable is that it provides a mechanism for generating a sequence of elements one at a time. The benefit is that `range(100000)`, for example, does not take up much computation time since no actual element is generated until it is iterated over.

An optional **conditional** can also be present, giving the general form `[f(x) for x in l if g(x)]`, for some boolean expression `g` (taking on the values `True` or `False`) where generation of the list elements only applies to elements that satisfy the boolean expression.

Multiple lists or **iterables** can be specified in list comprehension. equal to either 'math' or 'stat'.

```
[(x,y,z) for x in [True, False] for y in range(4,7)
   for z in ['math','stat']]
```

```
[(True, 4, 'math'), (True, 4, 'stat'), (True, 5, 'math'),
 (True, 5, 'stat'), (True, 6, 'math'), (True, 6, 'stat'),
 (False, 4, 'math'), (False, 4, 'stat'), (False, 5, 'math'),
 (False, 5, 'stat'), (False, 6, 'math'), (False, 6, 'stat')]
```

We can mimic list comprehension with the help of **loops** (much less efficient); it is preferable to use the former to generate lists.

List Operations We illustrate various other operations that can be performed on **zero-indexed** lists in the following blocks: [44]

44: The first element in the list has index 0.

- sublisting
- changing values
- sorting values
- appending values
- concatenating lists
- deleting elements

Consider a given list `x`:

```
x = [3,1,7,2,5]
print(x)
```

```
[3, 1, 7, 2, 5]
```

We can find the length of the list or print the sublist from the second element to the fourth element, say:[45]

45: Remember, ordinals start with 0, cardinals with 1.

```
print(len(x))
print(x[1:4])
```

```
5
[1, 7, 2]
```

We could also modify the second element of the list (index 1), say:

```
x[1] = 4
print(x)
```

```
[3, 4, 7, 2, 5]
```

46: ... or at least, until it is modified again.

Note that `x` is now permanently changed;[46] if we want to modify the last entry but we are not sure about the length of the list, for instance, we could use:

```
x[-1] = 6
print(x)
```

```
[3, 4, 7, 2, 6]
```

If we are looking to change the third last element as well, we could use

```
x[-3] = 1
print(x)
```

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

Finally, we could sort the resulting list:

```
x.sort()
print(x)
```

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

A lot of `Python` methods are applied using the syntax `object.method()`, in contrast to the typical `R` syntax that would use `method(object)`; so it is `x.sort()` instead of `sort(x)`.

Let us create another list, this time with booleans:

```
y = [3, True, False]
print(y)
```

```
[3, True, False]
```

We can append a value, say 5, at the end of this list, as follows:

```
y.append(5)
print(y)
```

```
[3, True, False, 5]
```

It is also possible to concatenate lists, using the (somewhat confusing) addition notation:

```
z = x + y
print(z)
```

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

and delete the last element of this new list:

```
del z[-1] # Delete the last element from z
print(z)
```

```
[1, 2, 3, 4, 6, 3, True, False]
```

or delete a range of elements, say from the 3rd to the 6th, from the resulting list:

```
del z[2:6] # watch out for the indices
print(z)
```

```
[1, 2, True, False]
```

Flow Control We will take a brief look at two ways to alter the flow of control in `Python`: **conditional statements** and **loops**.

Conditional Statements `Python` supports `if-elif-else` statements in various forms.

In the following example, we let `x` be some random integer between 1 and 12 (using function `randint()` from module `random`) and see how the results are affected.

```
import random
x = random.randint(1,12)
print(x)
```

```
9
```

(which may change from one run to another). Perhaps we want to print the string 'Hello`ifx`' is less than 5, like so:

```
if x < 5:
    print('Hello')
```

We would see nothing here as `x` is 9 in this run. Perhaps we want to print 'Out of range' if `x` is less than 5 or greater than 9, and `Within range` otherwise?

```
if x < 5 or x > 9:
    print('Out of range')
else:
    print('Within range')
```

```
Within range
```

Finally, we might want to print 'Small' if x is positive and less than 5; otherwise, print 'Five' if x is 5; otherwise, print 'Six' if x is 6; otherwise, print +:

```
if 0 < x and x < 5:
    print('Small')
elif x == 5:
    print('Five')
elif x == 6:
    print('Six')
else:
    print('+')
```

```
+
```

Run this sequence of blocks a number of times to see the various outcomes.

Important: Note that the code block that follows an `if`, `else`, or `elif` statement must be **properly indented**. The custom is to use four spaces for indentation. The following example illustrates the effects of different indentations.

```
x = 4

if x < 5:
    print('Small')
else:
    print('This string will not be printed, because the
          else statement never triggers')
    print('Neither will this, for the same reason')
print('This will be printed no matter what x is, as it
      falls outside the if-else statement block')
```

```
Small
This will be printed no matter what x is, as it falls
   outside the if-else statement block
```

Loops Loops are useful for repeatedly executing a statement or a block. We first consider the **for loop**.

Let us start with a simple example: for each value in the list `[1,3,8]`, we print its square.

```
for i in [1,3,8]:
    print(i**2)
```

```
1
9
64
```

We could also compute sums with loops, such as $1 + 2 + \cdots + 8 + 9$:

```
sum = 0
for x in range(1,10):
    sum += x  # add the value of x to sum
print(sum)
```

```
45
```

Or print the first `n` even nonnegative integers

```
n = 5
for n in range(0,n):
    t = 2*n
    print(t)
```

```
0
2
4
6
8
```

If a for loop is used to create a list, it is probably best to rewrite it using list comprehension. The following time comparison (using `%%timeit`) illustrates the contrast when building a list of 100×1000 items.

Using a loop:

```
l = []
for i in range(100):
    for j in range(1000):
        l.append((i,j))
```

Using list comprehension:

```
l = [ (i,j) for i in range(100) for j in range(1000)]
```

While loops are useful for iterating until a certain condition is met. For instance, if we want to print the first 10 even positive integers, separated by a space, we could use the following block:

```
i = 0
while i < 10:  # Repeat the following block until i
               # reaches 10 or greater
    i += 1     # iterated index
    print(2*i, end=' ')
```

```
2 4 6 8 10 12 14 16 18 20
```

Or we could print the 26 lower case English alphabets letters on one line, with no separation:

```
i = 0;
while i < 26:
    print(chr(ord('a')+i), end='')
    i += 1
```

```
abcdefghijklmnopqrstuvwxyz
```

Note that `ord` returns the ordinal for a character; `chr` does the reverse.

Functions A **function** is a grouped sequence of code that can be called, such as `cos()` and `print()`. A function can have 0 or more **arguments**: `cos()` takes one argument, whereas `print()` can have up to five (see documentation for details).

Named Functions Functions facilitate code re-use. `Python` functions are defined *via* the `def` statement. In the next example, we define a function that returns a pair consisting of the sum and the product of its arguments.

```
def sumprod(x, y):
    return x+y, x*y
```

The parentheses around the tuple are optional in this context. The ouput for $x = 3$ and $y = 4$ can be obtained as follows (once the function is compiled):

```
print(sumprod(3,4))
```

```
(7, 12)
```

Functions can also have default argument values. In the following example, if the second argument is not supplied, it takes on the value 5.

```
def myIntegerList(start, end=5):
    return list(range(start, end+1))
```

Compare the results of the two calls below:

```
print(myIntegerList(2))
print(myIntegerList(7,9))
```

```
[2, 3, 4, 5]
[7, 8, 9]
```

Anonymous (Lambda) Functions Another way to define a function is with a **lambda statement**, which is used to define one-line functions.[47]

47: The function is anonymous because it has no name.

Anonymous functions are defined using the one-line notation:

```
lambda variables: output
```

For instance,

```
add = lambda u, v: u + v
multiply = lambda u, v: u*v
```

We can apply a bivariate function `func` to arguments `x` and `y`, in a general context, using:

```
def applyFunc(func, x, y):
    return func(x,y)
```

and apply in specific contexts (rule, inputs) as follows:

```
print(applyFunc(multiply, 3,4))
print(applyFunc(add, 7,20))
```

```
12
27
```

But we do not need to define the function prior to the call. This would also work:

```
print(applyFunc(lambda u, v: u*v, 3,4))
print(applyFunc(lambda u, v: u + v, 7,20))
```

```
12
27
```

Strings Text manipulation is an important part of data cleaning. Often, the raw data contains string fields that do not quite follow an expected format. For example, proper nouns could be incorrectly capitalized. Dates could have been entered under different conventions. Fortunately, `Python` offers many tools that make string manipulation rather painless. In this section, we look at some of the commonly-performed operations on strings.

Strings can be defined using single or double quotes; note that `Python` supports unicode strings.

```
a = 'First string'
b = "Second string"
c = '+*'
print(type(a), type(b), type(c))
```

```
<class 'str'> <class 'str'> <class 'str'>
```

We can use the multiplication syntax to define a string made up of identical copies of another string as illustrated below:

```
r1 = a*4
r2 = c*3

print(r1)
print(r2)
```

```
First stringFirst stringFirst stringFirst string
+*+*+*
```

Strings can be concatenated using the addition syntax:

```
d = a + c
e = r2 + a + b

print(d)
print(e)
```

```
First string+*
+*+*+*First stringSecond string
```

The character in position `i` (the **index**) of the string a can be accessed via `a[i]`. Remember that the first character's index is 0.

Negative indices can also be used:`a[-4]` returns the fourth character from the end, say. For instance, we can print the first, seventh, last, and fourth-last characters of `a` using:

```
print(a[0], a[6], a[-1], a[-4])
```

```
F s g r
```

We can obtain a **substring** of a string a using the syntax `a[i:j]` where `i` specifies the starting index and `j-1` the ending index. Note that `a[:j]` is equivalent to `a[0:j]`, and `a[i:]` is the substring starting at index `i` and reaching until the end of `a`.

```
print(a[2:4])
print(a[:3])
print(a[6:])
```

```
rs
Fir
string
```

For a string `x`, `x.split()` **splits** the string into a list of words separated by a space (by default). Note that a contiguous sequence of space characters including newline (`\n`), carriage return (`\r`), and tab (`\t`) is considered as one space.

We can also specify what separating characters to use for the splitting, instead of spaces. For example, `x.split(',')` splits `x` on commas and `x.split('--')` splits it on `--`.

Consider the examples below:

```
print('This is  a  \n\n   long   sentence with
   \r \t weird spaces separating the words.'.split())
```

```
['This', 'is', 'a', 'long', 'sentence', 'with', 'weird', 'spaces', 'separating', 'the', 'words.']
```

```
print('One,two, three ,four'.split(',')) # Note that
   # ' three ' is one of the words after separation.
```

```
['One', 'two', ' three ', 'four']
```

```
print('Five--six--ninety-four'.split('--'))
```

```
['Five', 'six', 'ninety-four']
```

In some case, it is helpful to remove leading and trailing space characters (**whitespace stripping**).

```
s = '  time   '
print(s)
print(s.strip())
```

```
  time
time
```

It is common to combine `strip()` with `split(',')`:

```
cs = 'One    , two,  three  '
print([s.strip() for s in cs.split(',')])
```

```
['One', 'two', 'three']
```

In fact, the `strip()` method can accept a string consisting of all characters to be stripped from anothe string, in any combination. For instance, we can strip any leading and trailing characters contained in `['&','#','-','.','!']` from any string as follows:

```
tostrip = '&#-.!'
t = '###.Hel#lo!?!&-'
print(t.strip(tostrip))
```

```
Hel#lo!?
```

The methods `upper()`, `lower()`, and `title()` are useful for **altering the case** of characters in a string. The following examples showcase their functionality.

```
x = "gArbagE collECtion"
print(x.upper())
print(x.lower())
print(x.title())
```

```
GARBAGE COLLECTION
garbage collection
Garbage Collection
```

The following example illustrates a function that takes a phrase and turns it into an acronym by concatenating the first letters of the words and capitalizing all the letters. Does the code make sense?

```
def acronymize(phrase):
    a = ''                     # start with empty string
    for w in phrase.split(): # iterate through words
        a += w[0]              # pick the first letter of
                               # the words and concatenate
    return a.upper()         # capitalize and return
```

```
acronymize("Be right back"), acronymize("Mr Pat Why?")
```

```
('BRB', 'MPW')
```

It can also be useful to **convert a string** representing a number to a number type, and vice versa. The following examples illustrate how these tasks can be achieved.

```
number = 12.345

s = str(number)
print( s, type(s))

f = float(s)
print(f, type(f))

i = int('345')
print(i, type(i))
```

```
12.345 <class 'str'>
12.345 <class 'float'>
345 <class 'int'>
```

We can also check if a string `t` is a substring of another string `s` via `t in s` (**pattern matching**).

```
t1 = "is"
t2 = "has"

s = "This is my car."

print(t1 in s)
print(t2 in s)
```

```
True
False
```

If we want to obtain the index at which a substring begins, we can use the `find()` method. If the substring is not found, -1 is returned.

```
print(s.find(t1))
print(s.find(t2))
```

```
2
-1
```

We shall revisit `Python` strings when we discuss *Natural Language Processing* (see Chapter **??**).

Dictionaries A **dictionary** is a data structure for **key-value pairs** (`k:v`). To define a dictionary, simply list the key-value pairs enclosed within braces ({,}), as shown in the following examples.

The simplest dictionary is the one that is empty:

```
d = {}  # This creates an empty dictionary

print(type(d))
```

```
<class 'dict'>
```

A more interesting dictionary could be the one below:

```
days = { 'Sun': 1, 'Mon': 2, 'Tue':3, 'Wed':4, 'Thu':5,
   'Fri':6, 'Sat':7 }
print(type(days))
```

```
<class 'dict'>
```

We can **access** the value for key `k` in dictionary `d` via `d[k]`. Note that an exception will be raised if `d` does not contain the key `k`.

We can check if a key `k` is in a dictionary `d` via `k in d`.

```
print(days['Wed'])
print('Aug' in days)
```

```
4
False
```

We can **add** a new key-value pair `k:v` to a dictionary `d` via `d[k] = v`.

```
d[1]=(1,2)
d[2]= 3.45
d['three']= 'string'
print(d)
```

```
{1: (1, 2), 2: 3.45, 'three': 'string'}
```

Conversely, we can delete key `k` and its associated value from dictionary `d` via del `d[k]`.

```
del d[2]
print(d)
```

```
{1: (1, 2), 'three': 'string'}
```

We can also iterate over the keys in a dictionary using a **for loop**.

```
for key in d:
    print(type(key), type(d[key]))
```

```
<class 'int'> <class 'tuple'>
<class 'str'> <class 'str'>
```

The following code gives the same output

```
for key, value in d.items():
    print(type(key), type(value))
```

```
<class 'int'> <class 'tuple'>
<class 'str'> <class 'str'>
```

1.5.3 NumPy and Arrays

NumPy is a Python module that supports numerical computation on multi-dimensional arrays. It comes with many useful mathematical functions.

It is the backbone to the scientific computing library SciPy and data analysis and manipulation library pandas. Even though it is possible to do basic statisical analysis using a comprehensive statistics package without direct manipulation of NumPy arrays, knowledge of NumPy is essential for performing custom operations.

In this section, we get a taste of NumPy arrays of dimension at most two. What is covered only scratches the surface of this powerful library. A handy cheat sheet can be found here .

It is customary to use the alias np when importing the module.

```
import numpy as np
```

Arrays Unlike lists, NumPy arrays cannot contain elements of different types. There are various ways to create such arrays.

We can create a 1D array from a list:

```
x = np.array([1,2,3,4])

print(x.shape)
```

```
(4,)
```

shape is the method that returns the array's dimensions. We can create a 2D array from a list of lists:

```
y = np.array([[1,2,3],[4,5,6]])

print(y.shape)
```

```
(2, 3)
```

If some of the elements are not of the “right” type, they are converted automatically:

```
c = np.array(['n','u','m',15])

print(c)
```

```
['n' 'u' 'm' '15']
```

We can also define a `NumPy` array out of a range using the `arange()` function:

```
np.arange(1,5)

print(c)
```

```
array([1, 2, 3, 4])
['n' 'u' 'm' '15']
```

yields the same result as `np.array([1,2,3,4])`, but it is more efficient, from a computational perspective.

We can also obtain special arrays, composed of zeros, or composed of ones, with the functions `zeros()` and `ones()`. Here is a 3x4 2D array of 0s:

```
z = np.zeros([3,4]) # A 3-by-4 array of 0's
print(z.shape)
```

```
(3, 4)
```

and 2x1x3 3D array of 1s:

```
f = np.ones([2,3,4]) # A 2x1x3 3D array of 1's
print(f.ndim)
```

```
3
```

Note the difference between the `shape` and `ndim` methods: the former gives the actual dimensions (number of rows, columns, etc.), the latter, the number of dimensions (axes).

We can also define `NumPy` arrays containing random values; for instance, here is a 1D array of 10 random values sampled from the standard normal distribution, using the function `random.normal()`:

```
r = np.random.normal(size=10)
print(r)
```

```
[-1.10501533 -0.69929125 -0.00882625  1.12738611  0.60354054
  1.50509863  1.07440466 -0.86260135  1.12680367 -0.01988042]
```

Arithmetic **Adding** and **subtracting** `NumPy` arrays of the same dimensions works as we would expect. Using `x` and `y` as above, and `x2` as below, we get:

```
w = np.array([-1,-2,-3,-4])
```

```
print(x+w)
```

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

```
print(x-w)
```

```
[2 4 6 8]
```

```
print(y+y)
```

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

Multiplication by a scalar also works as expected:

```
print(2*x)
```

```
[2 4 6 8]
```

However, note that **multiplication** and **division** via `*` and `/` (resp.) are applied component-wise:

```
print(x*w)
```

```
[ -1  -4  -9 -16]
```

as is **exponentiation**:

```
print(y**3)
```

```
[[  1   8  27]
 [ 64 125 216]]
```

Broadcasting allows addition and substraction to be performed between arrays that do not have the same shape. There are rules governing when such operations are valid and what the effects are. Here, we provide two simple examples:

```
x + 3.5
```

```
array([4.5, 5.5, 6.5, 7.5])
```

```
y - 1
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

Can you determine what broadcasting does from these examples?

Math Functions `NumPy` contain some useful methods mapping arrays to a scalar.

For instance, `sum` adds up the elements in the array.

```
x.sum()
```

```
10
```

(the same result could have been obtained with `np.sum(x)`).

The usual statistical descriptions are also available as methos:

```
print(x.std(),x.var(),x.mean())
```

```
1.118033988749895 1.25 2.5
```

`NumPy` also has a collection of mathematical functions that can be applied **component-wise**, such as `abs()` and `exp()`:

```
print(np.abs(r))
```

```
[1.10501533 0.69929125 0.00882625 1.12738611 0.60354054
 1.50509863 1.07440466 0.86260135 1.12680367 0.01988042]
```

```
print(np.exp(y))
```

```
[[  2.71828183   7.3890561   20.08553692]
 [ 54.59815003 148.4131591  403.42879349]]
```

`NumPy` functions are more efficient when it comes to array computations; they should be used whenever possible.

Logical Operations Operations over arrays of boolean values can also be performed efficiently in `NumPy`.

Let us create a boolean array `bx` of the same shape as `x`, with `bx[i] = True` if and only if `x[i] >= 2.5`, and a boolean array `by` of the same shape as `y`, with `by[i] = True` if and only if `y[i] >= 3.5`.

```
bx = x >= 2.5
by = y >= 3.5

print(bx)
print(by)
```

```
[False False  True  True]
[[False False False]
 [ True  True  True]]
```

Comparison of two `NumPy` arrays of the same shape results in a boolean array, yet again of the same shape. Note that comparison is performed component-wise:

```
x2 = np.array([2,1,3,0])

print(x == x2)
```

```
[False False  True False]
```

Comparisons use the symbols ==, <, and >:

```
print(x > x2)
```

```
[False  True False  True]
```

We can perform **boolean operations** (AND, OR, NEG) on boolean arrays:

```
b = np.array([True, False, True, True])
```

AND is computed using &:

```
b & bx
```

```
array([False, False,  True,  True])
```

OR with `|`:

```
b | bx
```

```
array([ True, False,  True,  True])
```

NEG with ~:

```
~b
```

```
array([False,  True, False, False])
```

We can also sum over the values of a boolean array (in this case, `True` is interpreted as 1 and `False` as 0):

```
np.sum(b)
```

```
3
```

1.6 Python for Data Science

While `Python` remains a bona fide programming language, it is as a data science tool that its popularity has soared. Let us take a look at some of its data functionality.

1.6.1 Pandas and Data Frames

The `Pandas` module provides `Python` with an equivalent of `R` data frames. Essentially, it is a two-dimensional tabular data structure in which each column can be of different value types.

In this section, we cover the basics of `Pandas` data frames (and introduce a dataset found in the `Seaborn` module.[48] Comprehensive references for doing data analysis with Python include [16–18]. The pandas cheat sheet could also prove handy.

48: Which is used for data visualization (see Chapter 16).

We start by importing the required modules, with the customary **aliases** `pd` and `sns`:

```
import pandas as pd
import seaborn as sns
```

Loading Data There are various ways to obtain data. One way is to use a pre-built sample dataset, such as `titanic` from `seaborn`.

```
titanic = sns.load_dataset("titanic")
type(titanic)
```

```
<class 'pandas.core.frame.DataFrame'>
```

Another way is to read a `csv` file using `pandas.read_csv()`. For instance, if the file `calculus.csv` is in the `data` folder, we would call:

```
calculus = pd.read_csv('data/calculus.csv')
```

The first rows are given using the `head()` method of a `DataFrame` object:

```
titanic.head()
```

```
   survived  pclass     sex   age  ...  deck  embark_town  alive  alone
0         0       3    male  22.0  ...   NaN  Southampton     no  False
...
4         0       3    male  35.0  ...   NaN  Southampton     no   True

[5 rows x 15 columns]
```

We can also look at the last rows using the `tail()` method,[49] such as:

49: The number of observations can also be specified in the `head()` method.

```
calculus.tail(6)
```

```
       ID Sex  Grade    GPA  Year
94  10095   F     69   6.49     1
95  10096   M     99  12.61     1
96  10097   M     40   4.17     2
97  10098   F     66   6.94     1
98  10099   M     83  10.09     1
99  10100   F     52   6.76     2
```

We get a quick **summary** of a `DataFrame` using the `describe()` method:

```
titanic.describe()
```

```
         survived      pclass         age       sibsp       parch        fare
count  891.000000  891.000000  714.000000  891.000000  891.000000  891.000000
mean     0.383838    2.308642   29.699118    0.523008    0.381594   32.204208
std      0.486592    0.836071   14.526497    1.102743    0.806057   49.693429
min      0.000000    1.000000    0.420000    0.000000    0.000000    0.000000
25%      0.000000    2.000000   20.125000    0.000000    0.000000    7.910400
50%      0.000000    3.000000   28.000000    0.000000    0.000000   14.454200
75%      1.000000    3.000000   38.000000    1.000000    0.000000   31.000000
max      1.000000    3.000000   80.000000    8.000000    6.000000  512.329200
```

We can also obtain a summary of a **subset** of the columns:

```
df1 = titanic[['survived', 'age', 'fare']]
df1.describe()
```

```
         survived         age        fare
count  891.000000  714.000000  891.000000
mean     0.383838   29.699118   32.204208
std      0.486592   14.526497   49.693429
min      0.000000    0.420000    0.000000
25%      0.000000   20.125000    7.910400
50%      0.000000   28.000000   14.454200
75%      1.000000   38.000000   31.000000
max      1.000000   80.000000  512.329200
```

Or specific summary statistics on the full objects or on a specific column:

```
df1.mean()
print()
df1['age'].median()
```

```
survived     0.383838
age         29.699118
fare        32.204208
dtype: float64

28.0
```

Data Frame Operations We continue with some basic operations on data frames. We will use another built-in dataset

```
crashes.head()
```

```
   total  speeding  alcohol  ...  ins_premium  ins_losses  abbrev
0   18.8     7.332    5.640  ...       784.55      145.08      AL
1   18.1     7.421    4.525  ...      1053.48      133.93      AK
2   18.6     6.510    5.208  ...       899.47      110.35      AZ
3   22.4     4.032    5.824  ...       827.34      142.39      AR
4   12.0     4.200    3.360  ...       878.41      165.63      CA

[5 rows x 8 columns]
```

New columns can be added to any data frame. In this example, we will generate a new column consisting of strings of the form `Cnnn` where nnn is a zero-padded three-digit number so that row 1, 2,. . . of `crashes` correspond to `C001`, `C002`, . . .

```
labels = ['C'+"{:03}".format(i+1) for
                   i in range(crashes.shape[0])]
crashes['label'] = labels
```

```
crashes.head()
```

```
   total  speeding  alcohol  ...  ins_losses  abbrev  label
0   18.8     7.332    5.640  ...      145.08      AL   C001
1   18.1     7.421    4.525  ...      133.93      AK   C002
2   18.6     6.510    5.208  ...      110.35      AZ   C003
3   22.4     4.032    5.824  ...      142.39      AR   C004
4   12.0     4.200    3.360  ...      165.63      CA   C005

[5 rows x 9 columns]
```

Quite often, a particular column in a `csv` file serves as the index column. We can set this column to be an index column via the `set_index()` method:

```
df = crashes.set_index('label')
df.head()
```

```
       total  speeding  alcohol  ...  ins_premium  ins_losses  abbrev
label                            ...
C001    18.8     7.332    5.640  ...       784.55      145.08      AL
C002    18.1     7.421    4.525  ...      1053.48      133.93      AK
C003    18.6     6.510    5.208  ...       899.47      110.35      AZ
C004    22.4     4.032    5.824  ...       827.34      142.39      AR
C005    12.0     4.200    3.360  ...       878.41      165.63      CA

[5 rows x 8 columns]
```

Note that `crashes` is not affected by `set_index()`. To make the change directly to `crashes`, we would need to replace

```
df = crashes.set_index('label')
```

with

```
crashes.set_index('label', inplace=True)
```

We can **subset** a data frame by rows and columns labels via `loc[]`, as in the examples below:

```
df.loc['C010':'C013',['speeding','total']]
```

```
       speeding  total
label
C010      3.759   17.9
C011      2.964   15.6
C012      9.450   17.5
C013      5.508   15.3
```

```
df.loc['C005':'C008',:]
```

```
       total  speeding  alcohol  ...  ins_premium  ins_losses  abbrev
label                            ...
C005    12.0     4.200    3.360  ...       878.41      165.63      CA
C006    13.6     5.032    3.808  ...       835.50      139.91      CO
C007    10.8     4.968    3.888  ...      1068.73      167.02      CT
C008    16.2     6.156    4.860  ...      1137.87      151.48      DE

[4 rows x 8 columns]
```

We can also extract using position values via `iloc[]`.

```
df.iloc[1:5,0:4]
```

```
       total  speeding  alcohol  not_distracted
label
C002    18.1     7.421    4.525          16.290
C003    18.6     6.510    5.208          15.624
C004    22.4     4.032    5.824          21.056
C005    12.0     4.200    3.360          10.920
```

We can **reset** the index in a data frame via the `reset_index()` method. This has the effect of turning `label` into a data column like all other columns in the data frame `df`, for instance:

```
df.reset_index(inplace=True)
df.head()
```

```
  label  total  speeding  ...  ins_premium  ins_losses  abbrev
0  C001   18.8     7.332  ...       784.55      145.08      AL
1  C002   18.1     7.421  ...      1053.48      133.93      AK
2  C003   18.6     6.510  ...       899.47      110.35      AZ
3  C004   22.4     4.032  ...       827.34      142.39      AR
4  C005   12.0     4.200  ...       878.41      165.63      CA

[5 rows x 9 columns]
```

It is possible to use the generator `iterrows` to yield both index and row of a data frame. For instance, the next block of code will print the labels corresponding to the first five rows.

```
for index, row in df[0:5].iterrows():
    print(row['label'])
```

```
C001
C002
C003
C004
C005
```

Columns and rows can be **dropped** from a data frame via the `drop()` method. In the example below, we drop the `label` column from `df` and assign the outcome to `df2` (but note `df` itself is not changed):

```
df2 = df.drop('label', axis=1)
df2.head()
```

```
   total  speeding  alcohol  ...  ins_premium  ins_losses  abbrev
0   18.8     7.332    5.640  ...       784.55      145.08      AL
1   18.1     7.421    4.525  ...      1053.48      133.93      AK
2   18.6     6.510    5.208  ...       899.47      110.35      AZ
3   22.4     4.032    5.824  ...       827.34      142.39      AR
4   12.0     4.200    3.360  ...       878.41      165.63      CA

[5 rows x 8 columns]
```

In contrast, the `total` column is dropped from `df` (and `df` is modified as a result):

```
df.drop('total', axis=1, inplace=True)
df.head()
```

```
  label  speeding  alcohol  ...  ins_premium  ins_losses  abbrev
0  C001     7.332    5.640  ...       784.55      145.08      AL
1  C002     7.421    4.525  ...      1053.48      133.93      AK
2  C003     6.510    5.208  ...       899.47      110.35      AZ
3  C004     4.032    5.824  ...       827.34      142.39      AR
4  C005     4.200    3.360  ...       878.41      165.63      CA

[5 rows x 8 columns]
```

We can **rename** the columns of a data frame via the `rename()` method:

```
df.rename(columns={'label':'case', 'abbrev':'abbr'},
                                     inplace=True)
df.head()
```

```
   case  speeding  alcohol  ...  ins_premium  ins_losses  abbr
0  C001     7.332    5.640  ...       784.55      145.08    AL
1  C002     7.421    4.525  ...      1053.48      133.93    AK
2  C003     6.510    5.208  ...       899.47      110.35    AZ
3  C004     4.032    5.824  ...       827.34      142.39    AR
4  C005     4.200    3.360  ...       878.41      165.63    CA

[5 rows x 8 columns]
```

What would we expect the following chunk of code to do?

```
newColumnNames = {}
for name in list(df):
    newColumnNames[name] = name.capitalize()

df2=df.rename(columns=newColumnNames)
```

Rows can be **filtered** according to a given condition. In the example below, `b` and `d` are Pandas series of booleans related to the `df` data frame:

```
b = df['ins_losses'] > 160
d = df['not_distracted'] < 12
```

If we want to return the rows of `df` for which `ins_losses` is greater than 160 **AND** `not_distracted` ia less than 12, we would simply call:

```
df[b & d]
```

```
    case  speeding  alcohol  ...  ins_premium  ins_losses  abbr
4   C005     4.200    3.360  ...       878.41      165.63    CA
6   C007     4.968    3.888  ...      1068.73      167.02    CT
20  C021     4.250    4.000  ...      1048.78      192.70    MD

[3 rows x 8 columns]
```

To return the rows of `db` for which `ins_losses` is greater than 160 **OR** `abbr` is equal to `AL`, we would call:

```
df[b | (df['abbr'] == 'AL')]
```

```
    case  speeding  alcohol  ...  ins_premium  ins_losses  abbr
0   C001     7.332    5.640  ...       784.55      145.08    AL
4   C005     4.200    3.360  ...       878.41      165.63    CA
6   C007     4.968    3.888  ...      1068.73      167.02    CT
18  C019     7.175    6.765  ...      1281.55      194.78    LA
20  C021     4.250    4.000  ...      1048.78      192.70    MD
36  C037     6.368    5.771  ...       881.51      178.86    OK

[6 rows x 8 columns]
```

1.6.2 Data Wrangling

We now take a look at some ways to **combine** and **clean** data frames.

Merging and Joins Consider a fictitious test score dataset. There are two sections in the class, contained in `testA.csv` and `testB.csv`. Each row consists of a student ID, a section, and a test mark. The file `gpa.csv` contains information on the students' GPAs and their current year of study.

We start by reading in the two test score files (recall that pd is the alias for the pandas module).

```
dfA = pd.read_csv('data/testA.csv')
dfB = pd.read_csv('data/testB.csv')
```

The first entries of each sets are shown below:

```
dfA.head()
```

```
      ID Section  Mark
0  10021       A    47
1  10073       A    83
2  10084       A    51
3  10102       A    57
4  10175       A    71
```

```
dfB.head()
```

```
      ID Section  Mark
0  10011       B    97
1  10063       B    63
2  10094       B    71
3  10110       B    77
4  10133       B    81
```

We now read in the GPA information.

```
gpa = pd.read_csv('data/gpa.csv')
gpa.head()
```

```
   Student ID   GPA  Year
0       10011  12.0   3.0
1       10021   NaN   3.0
2       10063   5.6   3.0
3       10073   9.8   3.0
4       10084   6.2   3.0
```

Note that the column title for student ID is different in the test score files and in gpa.csv.

We now **concatenate** the two data frames of test scores into a single object using the pandas function concat().

```
df = pd.concat([dfA,dfB])
```

We now merge the GPA data frame with this combined test score data frame.

```
df3 = pd.merge(gpa, df, left_on='Student ID', right_on='ID')
df3
```

```
   Student ID   GPA  Year     ID Section  Mark
0       10011  12.0   3.0  10011       B    97
1       10021   NaN   3.0  10021       A    47
2       10063   5.6   3.0  10063       B    63
3       10073   9.8   3.0  10073       A    83
4       10084   6.2   3.0  10084       A    51
5       10094   8.1   NaN  10094       B    71
6       10102   6.9   2.0  10102       A    57
7       10110   8.4   2.0  10110       B    77
8       10133  10.4   2.0  10133       B    81
9       10145   5.1   2.0  10145       B    41
10      10162   7.2   2.0  10162       B    68
11      10175   6.9   1.0  10175       A    71
12      10189   6.1   1.0  10189       B    68
13      10190  11.2   1.0  10190       A    91
14      10199   NaN   1.0  10199       A    56
```

`merge()` performs an **inner join**, but it can also perform **outer joins**.

Let us see what happens when we merge `gpa` with `dfA`.

```
pd.merge(gpa, dfA, left_on='Student ID', right_on='ID',
             how='outer').drop('Student ID', axis=1)
```

```
     GPA  Year       ID Section  Mark
0   12.0   3.0      NaN     NaN   NaN
1    NaN   3.0  10021.0       A  47.0
2    5.6   3.0      NaN     NaN   NaN
3    9.8   3.0  10073.0       A  83.0
4    6.2   3.0  10084.0       A  51.0
5    8.1   NaN      NaN     NaN   NaN
6    6.9   2.0  10102.0       A  57.0
7    8.4   2.0      NaN     NaN   NaN
8   10.4   2.0      NaN     NaN   NaN
9    5.1   2.0      NaN     NaN   NaN
10   7.2   2.0      NaN     NaN   NaN
11   6.9   1.0  10175.0       A  71.0
12   6.1   1.0      NaN     NaN   NaN
13  11.2   1.0  10190.0       A  91.0
14   NaN   1.0  10199.0       A  56.0
```

We can see that there is a row for every row in `gpa` and that only those rows for which `Student ID` is present in `dfA` have merged data (what happens if the `.drop('Student ID', axis=1)` is omitted?).

Data Cleansing Note that in the merged data frame `df3` (and in `gpa`), there are rows containing `NaN`. If we do not want any rows with such values, we can use the `dropna()` method.

```
df3.dropna()
```

```
    Student ID   GPA  Year     ID Section  Mark
0        10011  12.0   3.0  10011       B    97
2        10063   5.6   3.0  10063       B    63
3        10073   9.8   3.0  10073       A    83
4        10084   6.2   3.0  10084       A    51
6        10102   6.9   2.0  10102       A    57
7        10110   8.4   2.0  10110       B    77
8        10133  10.4   2.0  10133       B    81
9        10145   5.1   2.0  10145       B    41
10       10162   7.2   2.0  10162       B    68
11       10175   6.9   1.0  10175       A    71
12       10189   6.1   1.0  10189       B    68
13       10190  11.2   1.0  10190       A    91
```

We can also drop only the rows with NaN in specific columns. If we do not want to retain observations with `Year==NaN`, we would call:

```
gpa.dropna(subset=['Year'])
```

```
    Student ID   GPA  Year
0        10011  12.0   3.0
1        10021   NaN   3.0
2        10063   5.6   3.0
3        10073   9.8   3.0
4        10084   6.2   3.0
6        10102   6.9   2.0
7        10110   8.4   2.0
8        10133  10.4   2.0
9        10145   5.1   2.0
10       10162   7.2   2.0
11       10175   6.9   1.0
12       10189   6.1   1.0
13       10190  11.2   1.0
14       10199   NaN   1.0
```

Instead of dropping rows containing `NaN`, we could replace the unwanted values with some other chosen value instead (like 0, say).

```
gpa.fillna(0)
```

```
    Student ID   GPA  Year
0        10011  12.0   3.0
1        10021   0.0   3.0
2        10063   5.6   3.0
3        10073   9.8   3.0
4        10084   6.2   3.0
5        10094   8.1   0.0
6        10102   6.9   2.0
```

```
7        10110   8.4   2.0
8        10133  10.4   2.0
9        10145   5.1   2.0
10       10162   7.2   2.0
11       10175   6.9   1.0
12       10189   6.1   1.0
13       10190  11.2   1.0
14       10199   0.0   1.0
```

Note that all the NaNs are changed to 0.0. To change only the GPA volume, we can do the following (note that this will modify the original gpa data frame):

```
gpa.fillna({'GPA':0.0})
```

```
    Student ID   GPA  Year
0        10011  12.0   3.0
1        10021   0.0   3.0
2        10063   5.6   3.0
3        10073   9.8   3.0
4        10084   6.2   3.0
5        10094   8.1   NaN
6        10102   6.9   2.0
7        10110   8.4   2.0
8        10133  10.4   2.0
9        10145   5.1   2.0
10       10162   7.2   2.0
11       10175   6.9   1.0
12       10189   6.1   1.0
13       10190  11.2   1.0
14       10199   0.0   1.0
```

We can **apply** a function to a data frame column using the method map(). The following will add a Grade column to dfA, containing Pass or Fail based on the Mark column.

```
def markToGrade(x):
    res = 'Fail'
    if x >= 50:
        res = 'Pass'
    return res
dfA['Grade'] = dfA['Mark'].map(markToGrade)
dfA
```

```
      ID Section  Mark Grade
0  10021       A    47  Fail
1  10073       A    83  Pass
2  10084       A    51  Pass
3  10102       A    57  Pass
4  10175       A    71  Pass
5  10190       A    91  Pass
6  10199       A    56  Pass
```

1.6.3 Data Aggregation

Sometimes, the data in a dataset can be divided into **groups**. We might want to obtain **summary statistics** for each group. Analyses by groups and **aggregation** can help us obtain insight on groups.

Summaries by Groups We first illustrate obtaining simple statistics on groups using a dataset containing calculus marks (recall that `pd` is the `pandas` alias).

```
calc = pd.read_csv('data/calculus.csv')
calc.head()
```

```
      ID Sex  Grade   GPA  Year
0  10001   F     47  5.02     2
1  10002   M     57  3.82     1
2  10003   M     91  7.70     1
3  10004   M     71  4.82     1
4  10005   F     83  7.91     1
```

Suppose that we want to see separate mean grades and mean GPA based on the `Sex` variables. We can use the `groupby()` method to perform the task:

```
calc[['Sex','Grade','GPA']].groupby('Sex').mean()
```

```
         Grade       GPA
Sex
F    67.901961  6.539804
M    64.408163  5.609388
```

If we want descriptive statistics for `Grade` and `GPA` grouped by `Sex`, we can use the more general method `agg()`. Note that we first need to import `numpy` (alias `np`) to access these simple statistics functions.

```
calc[['Sex','Grade','GPA']].groupby('Sex').agg([np.mean,
                                     np.std, np.median])
```

```
         Grade                           GPA
          mean        std median        mean        std median
Sex
F    67.901961  20.162594   66.0  6.539804  3.008527   6.24
M    64.408163  16.237711   62.0  5.609388  2.756965   4.77
```

If we are interested in the `Grade` mean and the `GPA` median, grouped by `Sex`, we can use a dictionary to specify which function is applied to which column as follows:

```
calc[['Sex','Grade','GPA']].groupby('Sex').agg({'Grade':
                                        np.mean, 'GPA': np.median})
```

```
         Grade   GPA
Sex
F    67.901961  6.24
M    64.408163  4.77
```

We can also build **custom aggregate functions**. The following chunk of code computes the sum of squares for the `Grade` and `GPA` columns.

```
def sumOfSq(xs):
    return np.dot(xs,xs)

calc[['Sex','Grade','GPA']].groupby('Sex').agg(sumOfSq)
```

```
         Grade        GPA
Sex
F       255471  2633.7825
M       215928  1906.6374
```

Pivot Tables We could also have obtained the mean `Grade` and mean `GPA` for the `Sex` groups via `pivot_table()`, as below:

```
calc[['Sex','Grade','GPA']].pivot_table(index='Sex',
                                        aggfunc=np.mean)
```

```
       GPA     Grade
Sex
F  6.539804 67.901961
M  5.609388 64.408163
```

To obtain a **pivot table** displaying the number of students in each `Year` grouped by `Sex`, we can run the following code:

```
calc[['Sex','Year']].pivot_table(index='Sex',
   columns=['Year'],aggfunc=len, margins=False)
```

```
Year  1  2  3  4
Sex
F      33  11   6   1
M      32  11   2   4
```

We can also print the margins (totals) by changing to `margins=True`.

1.6.4 Combining Python with R

Ask most data scientist and they will tell you that they are a Python person or a R person (or perhaps less frequently a Julia person). Python might be best for **data processing** (in terms of efficiency, especially with large datasets), while R has a package (or three!) for pretty much any **statistical and data visualization** task under the sun, but that leaves a lot of data analysis real estate that is not spoken for; frankly, it makes much more sense to be conversant with both.[50]

50: And anything else that comes up from this point onward.

It is now possible to use Python within R through the reticulate package.[51] . The reticulate vignette contains detailed information on the process; for the time being, we will only give a small example detailing how this could be achieved, based on [19].

51: There are other means, see R Interface to Python and Five ways to work seamlessly between R and Python in the same project for more information), for instance

```
library(reticulate)
```

We start by creating a variable x in the Python session:

```
x = list(range(8))
```

Once that is done, we can access the Python variable x from R; it is a column in the (reserved) py data frame:

```
str(py)
py$x
```

```
Module(__main__)
[1] 0 1 2 3 4 5 6 7
```

We can also create new variables y in the Python session from R, and pass a data frame to y:

```
py$y <- head(AirPassengers) # a built-in R dataset
```

This variable can now be displayed in the the Python session, and operated on, as needed:

```
print(y)
```

```
[112.0, 118.0, 132.0, 129.0, 121.0, 135.0]
```

It is not difficult to imagine how to expand this back and forth to more complex data analysis situations, leaving us the option of picking whatever language is best suited to a specific task.

1.7 Getting Started with SQL

Structured Query Language (SQL) is the standard language used to retrieve, modify, and add data to a **relational database**. It is implemented by all *Relational Database Management Systems* (RDMS), such as:

- MySQL [20]
- MS Access
- Oracle
- Postgres
- etc.

SQL allows users to **query** a database and manipulate the stored data using a variety of parameters. SQL code can be **embedded** into other languages in order to enable storage and processing of large datasets in an efficient manner.

The toy database with which we will work is "implemented" in Aidan Crowther's github repository . Video instructions can be found at DUDADS – How to access the toy database (04:27) | A. Crowther .[52]

52: You will need to install git, docker, and MySQL Client, and know how to open a port on Windows, MacOS, or Linux (search online if necessary).

1.7.1 Basics

Table Structure The most common form of data organization in a **relational database** is known as a **table** – it is similar to a spreadsheet. Data is stored in a **record** (row), with individual observations aligned by **fields** (columns).

Records and Fields Rows consist of data that fall into the categories specified by each column and that either match the **field data type** or contain a `NULL` value,[53] the **absence of data** – it is not the same as a value of zero or an empty string; `NULL` can be matched to any data type.[54]

53: Similar to R's `NA`.

54: SQL syntax often uses `ALL CAPS` in its queries to make it easier to distinguish between commands and data.

Constraints Data can be further restricted by Table or Field **constraints**. These constraints define rules by which the data must abide. Most commonly, these constraints are used to identify **special fields** by which data can be uniquely identified, or to ensure data matches a pattern, such as being unique, or not allowing `NULL` entries.

Here are some of common constraints (and their meanings).

- `DEFAULT`: provides a predefined default value if none is specified
- `NOT NULL`: enforces that columns can not have a `NULL` value
- `UNIQUE`: ensures that all values in a column are different
- `PRIMARY KEY`: uniquely identifies a record within a table
- `FOREIGN KEY`: uniquely identifies a record in another table
- `CHECK`: ensures all data in a field matches a restriction
- `INDEX`: used to quickly retrieve and add data to a table

Notably, **primary** and **foreign** keys allow users to create relations between tables. In addition, every table must contain **no more than one** primary key; although they do not need to be defined with a primary key, doing so is considered **bad practice**.

Data Integrity Data entered into a table must follow some ensuring the latter's **integrity**. The following rules exist in every **Database Management System** (DBMS).

- **Entity Integrity:** there must not be any duplicate records within a table;
- **Domain Integrity:** enforces valid entries for all fields, following restrictions on data type, format, or range;
- **Referential Integrity:** rows used by other records can not be deleted.

Essentially, we cannot enter records that can cause a table to stop being able to uniquely identify and collect data. In addition, relations between tables must never be broken through the deletion of data.

1.7.2 SQL Syntax

The fundamental SQL unit is the **query**, a way to manipulate and output observations from a database by following a specific set of rules.

Generally, queries are used to request data from **tables** kept within a database, but they can also be used to **modify**, **remove**, and **add** data.

The "**sentence structure**" of a SQL query is a repeated pattern of a **command** followed by a **descriptor**; the end of a query being denoted by a **semicolon** (`;`).[55] More information on SQL (including its syntax) is available in [20–22].

55: SQL queries read rather naturally as regular English sentences, too.

We will illustrate the various SQL query parameters with the help of a toy database with 4 tables, whose structure is shown in Figure 1.6.

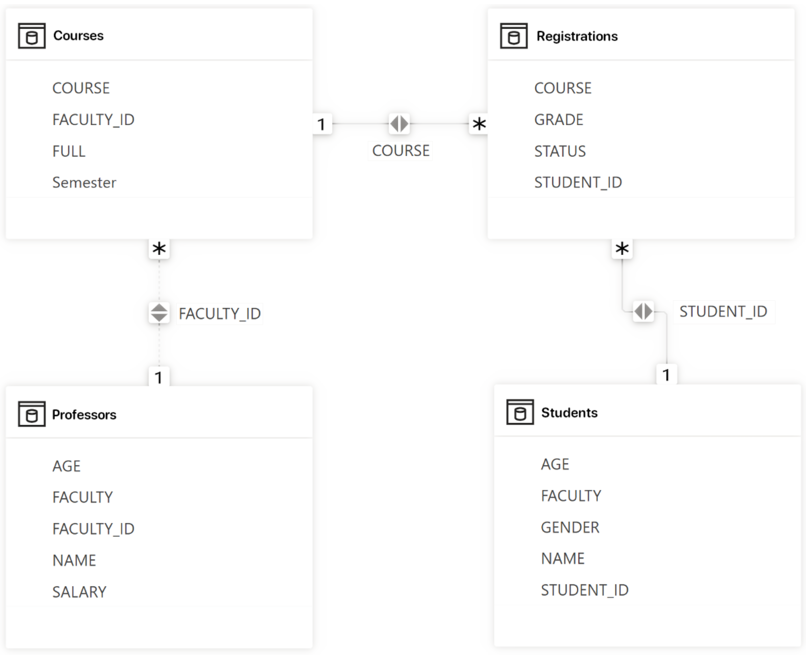


Figure 1.6: Database diagram for the toy example, with 4 tables. Some of the entries for 2 of the tables are shown in the Exercises. The data is also available as an Excel spreadsheet.

Example What would the following toy dataset query return?

A Simple SQL Query

```
SELECT COURSE FROM Courses WHERE FACULTY_ID=1;
```

We break down the query into its command/descriptor structure.

- `SELECT COURSE`: display only the `COURSE` identifier;
- `FROM Courses`: of the observations from the `Courses` table;
- `WHERE FACULTY_ID=4`: for which `FACULTY_ID` is 4.

This query would fetch all courses taught by the instructor #4:

```
  COURSES
1 CGSC101
2 CGSC202
```

We see that this is indeed the case in the `Courses` table:

```
   COURSE FULL Semester FACULTY_ID
1 BUSI202    1   SUMMER          6
2 CGSC101    1   SUMMER          4 <-- *
3 CGSC202    1   WINTER          4 <-- *
4 CHEM404    0   WINTER          8
5 COMP490    1     FALL          9
6 ECON101    1     FALL          1
7 ECON401    0   WINTER          1
8 MUSI101    0   SUMMER         NA
9 PHYS201    0   WINTER          2
```

1.7.3 Key Query Operators

`SELECT/FROM`

The `SELECT` command is nearly always used to interact with data; it is used to **request** data from a table. It is applied to **columns**, which need to be specified, using a **comma-separated list** of columns immediately after the `SELECT` **keyword**.[56] The **wildcard** character (∗) can be used to match **all** columns.

56: Spelling, including the case, matters.

`SELECT` also needs to be told from **which table** to retrieve data; this is accomplished with the `FROM` keyword, after columns have been specified in the query. `FROM` cannot be used without an **argument**, but only one table can be used as input.

The simplest form of a `SELECT` query takes the following form, returning all data within a table.[57]

57: In this case, the Courses table. The table is typically clear from the context.

```
SELECT * FROM Courses;
```

(The output was provided at the end of the previous Section).

The `SELECT` command also allows **aggregate functions** (statistics) to be applied to the selected table columns, including `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`; and more. All rows matching the field being modified will be combined into one unless combined with the `GROUP BY` clause.[58]

58: Not unlike in an Excel pivot table.

Multiple fields can be matched with aggregate functions, and multiple aggregate functions can be used in a query. This can be a useful workaround if a SQL server has **quota restrictions** on the number of queries that can be submitted, allowing multiple fields to be returned with one query.

```
SELECT AVG(SALARY), MAX(AGE) FROM Professors;
```

```
  AVG(SALARY) MAX(AGE)
1    210555.6       67
```

Evidently, the oldest professor is 67 years old, and the average salary is \$210,555.60.[59]

59: Whoa! They're making a killing out there...

WHERE

In SQL, some queries contain **modifiers** that narrow the query **scope**. The most prevalent one of these clauses is `WHERE`. This clause is often seen used with the `SELECT` query, but can also be used to **specify targets** for other queries such as `UPDATE` and `DELETE`.

`WHERE` allows users to specify **constraints** to apply to the database **prior** to returning the results of a query. These constraints typically use **comparison operators**, such as: `>`, `<`, `=`, `NOT`, `LIKE`, `IS`, etc...

Constraints based on numerical values behave as expected, but their behaviour might be unexpected however when operating on a **strings**. Consequently, we recommend consulting the appropriate documentation in the specific database software manual.

We can determine whether a value is `NULL` by using the `IS` conditional clause to match for `NULL` type.

```
SELECT NAME FROM Professors WHERE SALARY >= 60000;
```

```
            NAME
1      Adam Smith
2     Paige Ryans
3        Alex Doe
4      Landon Liu
5 Kyra Carmichael
6    Heather Wong
7   Quine Ngyogne
8     Vikram  Das
9    Samuel Koffi
```

AND/OR/NOT

Clauses, such as WHERE, can be **chained** with other constraints in order to conduct complex queries on a database.

We can dive in further within a result when using a WHERE clause by combining conditions using the AND, OR, and NOT clauses, **Boolean operators** linking query conditions:

- AND returns results where **all** conditions are true;
- OR returns results where **at least one** condition is true, and
- NOT returns results where the next condition is false.

These clauses can further be organized into brackets.

```
SELECT * FROM Professors WHERE
    (SALARY>=60000 AND NOT AGE>60) OR FACULTY IS NULL;
```

```
             NAME SALARY FACULTY_ID AGE           FACULTY
1     Paige Ryans 180000          2  48           Physics
2        Alex Doe 190000          3  37              <NA>
3      Landon Liu 120000          4  34 Cognitive Science
4    Marcel Orosz     NA          5  48              <NA>
5 Kyra Carmichael 200000          6  30          Business
6    Heather Wong 200000          7  34         Economics
7   Quine Ngyogne 115000          8  55         Chemistry
8     Vikram  Das 500000          9  60  Computer Science
9    Samuel Koffi 300000         10  40 Political Science
```

EXISTS

The EXISTS keyword is used determine whether a sub-query returns any rows; it returns true if the sub-query returns at least one row; and false otherwise. It is often used in correlated sub-queries.

A **correlated sub-query** is a query that depends on values from the **outer query**; it is executed for each row of the outer query, and the results are used to **filter** the outer query (often based on some condition in the sub-query).

The syntax for a correlated sub-query is similar to a regular sub-query, but it includes a reference to the outer table in the sub-query.

```
SELECT * FROM Professors WHERE EXISTS
    (SELECT * FROM Courses WHERE
        Professors.FACULTY_ID = Courses.FACULTY_ID);
```

```
             NAME SALARY FACULTY_ID AGE           FACULTY
1      Adam Smith  90000          1  67         Economics
2     Paige Ryans 180000          2  48           Physics
3      Landon Liu 120000          4  34 Cognitive Science
4 Kyra Carmichael 200000          6  30          Business
5   Quine Ngyogne 115000          8  55         Chemistry
6     Vikram  Das 500000          9  60  Computer Science
```

The correlated sub-query identifies professors currently assigned to courses; the outer query returns the list of details for those professors.

HAVING/GROUP BY

The GROUP BY clause is used to aggregate data across multiple rows based on one or more fields.[60] It is used to group data and perform calculations on these groups. The aggregate functions include COUNT, SUM, AVG, MIN, MAX, etc...

60: Again, quite reminiscent of Excel pivot tables.

We can also use the HAVING clause to narrow grouped data further, allowing for the selection only of those results matching a supplementary set of criteria.

```
SELECT AGE, AVG(SALARY) AS AVG_SALARY FROM Professors
    GROUP BY AGE HAVING AVG(SALARY)>90000;
```

```
  AGE AVG_SALARY
1  48     180000
2  37     190000
3  34     160000
4  30     200000
5  55     115000
6  60     500000
7  40     300000
```

IN/BETWEEN

In addition to the use of **Boolean conditionals**, SQL has the ability to match **multiple distinct cases**, either by constraining results to a narrow value of cases specified by a **list**, or by matching within a **continuous range**.

IN allows a set of possible matching values to be specified – any condition contained **within this set** evaluates to true. We can also use the result of another query to specify the contents of this set *via* a SELECT query when specifying the set against which to match.

BETWEEN evaluates to true when a compared value falls strictly **within the bounds** specified by the query. This comparison is performed **inclusively**; it can also be used to match to an **alphabetically** sorted list of strings.

```
SELECT * FROM Professors WHERE FACULTY_ID IN (1, 2)
    AND NAME BETWEEN "Adam Smith" AND "Alex Doe";
```

```
        NAME SALARY FACULTY_ID AGE   FACULTY
1 Adam Smith  90000          1  67 Economics
```

LIMIT/ORDER BY

Some tables store a **large** number of rows, and can overwhelm a receiver; in these cases restricting the number of returned results can be **crucial**. This can be accomplished by using the LIMIT command, which when followed by a numerical value n, returns only the first n results from the query.[61]

61: This command can vary according to the SQL server in use – in some systems, the command is instead TOP.

ORDER BY is another powerful clause, especially when used in conjunction with the LIMIT/TOP clause – it sorts the result set returned by the query, allowing users to specify **sorting columns** (and **directions**: ASC and DESC).[62]

62: This works on numerical values and strings.

```
SELECT * FROM Professors ORDER BY NAME ASC LIMIT 4;
```

```
             NAME SALARY FACULTY_ID AGE   FACULTY
1      Adam Smith  90000          1  67 Economics
2        Alex Doe 190000          3  37      <NA>
3    Heather Wong 200000          7  34 Economics
4 Kyra Carmichael 200000          6  30  Business
```

DISTINCT

When one of the fields being used to return results contains a large number of **duplicate** values, the DISTINCT clause can help narrow the returned data; multiple fields can be marked as **distinct**, which can be useful when searching for **unique** matches after performing a JOIN operation.

```
SELECT DISTINCT NAME From Professors;
```

```
              NAME
1       Adam Smith
2      Paige Ryans
3         Alex Doe
4       Landon Liu
5     Marcel Orosz
6  Kyra Carmichael
7     Heather Wong
8    Quine Ngyogne
9      Vikram  Das
10    Samuel Koffi
```

LIKE

The LIKE keyword is used in a WHERE clause to **search** for a specified pattern in a string column. It is used with the % and _ wildcard characters, to match any **string** or any **single character**, respectively. The pattern provided for matching must be enclosed within **quotes**.

```
SELECT * FROM Courses WHERE COURSE LIKE 'ECON%';
```

```
   COURSE FULL Semester FACULTY_ID
1 ECON101    1     FALL          1
2 ECON401    0   WINTER          1
```

UNION

A **union** in SQL is a set operation which combines the result sets of two or more SELECT statements into a single result set.

The UNION command will combine the output of multiple SELECT queries, with a few restrictions:

- the same number of columns must be selected in all queries;
- the same data type must be used for all selections;
- the result must have the same order.

To include all rows, including duplicates, the UNION ALL operator can be used instead of UNION.

A union can be used for a wide range of purposes, such as combining data from multiple tables, aggregating data from different sources, and generating reports that require data from multiple queries.

```
SELECT NAME AS RESULTS FROM Professors WHERE FACULTY_ID=1
    UNION SELECT COURSE FROM Courses WHERE FACULTY_ID=1;
```

```
     RESULTS
1 Adam Smith
2    ECON101
3    ECON401
```

Note that a UNION will combine all matching results into the same column. This may require careful formatting of the selection ordering when matching multiple columns.

JOIN

A crucial concept of SQL is that of **combining tables** virtually in order to match related data between tables. One approach to doing so is using the JOIN command, which allows users to combine multiple tables into a **single virtual table** by matching like data between the two.

Multiple types of JOIN can be performed:

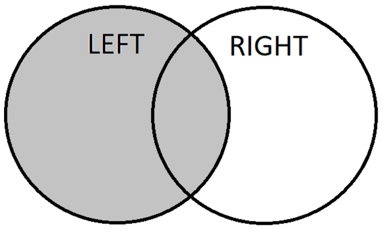
- LEFT JOIN
- RIGHT JOIN
- INNER JOIN
- FULL JOIN
- EXCLUSIVE JOIN

These different forms of JOIN allow data selection to be narrowed to various ranges, based on the **order** in which the tables are joined and the **type** of join used.

LEFT JOIN

LEFT JOIN is a type of join operation that combines rows from two tables based on the chosen **matching condition(s)**, as well as any **unmatched** rows from the **left table**; i.e., the **first** specified table after the FROM clause.[63]

63: The LEFT JOIN is illustrated below:



The resulting table will contain all of the rows from the left table, along with any matching rows from the right table. If a row does not have a match in the **right** table, it contains only NULL values.

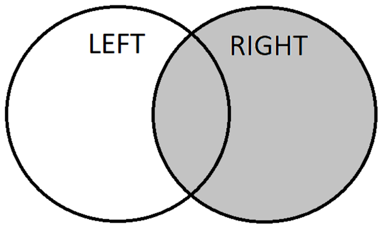
```
SELECT * FROM Professors LEFT JOIN Courses
    ON Courses.FACULTY_ID=Professors.FACULTY_ID;
```

This query will create a list of all professor-assigned-to-course matches, while also listing professors that do not teach any courses.

```
              NAME SALARY FACULTY_ID AGE           FACULTY  COURSE FULL Semester FACULTY_ID
1       Adam Smith  90000          1  67         Economics ECON101    1     FALL          1
2       Adam Smith  90000          1  67         Economics ECON401    0   WINTER          1
3      Paige Ryans 180000          2  48           Physics PHYS201    0   WINTER          2
4         Alex Doe 190000          3  37              <NA>    <NA>   NA     <NA>         NA
5       Landon Liu 120000          4  34 Cognitive Science CGSC101    1   SUMMER          4
6       Landon Liu 120000          4  34 Cognitive Science CGSC202    1   WINTER          4
7     Marcel Orosz     NA          5  48              <NA>    <NA>   NA     <NA>         NA
8  Kyra Carmichael 200000          6  30          Business BUSI202    1   SUMMER          6
9     Heather Wong 200000          7  34         Economics    <NA>   NA     <NA>         NA
10   Quine Ngyogne 115000          8  55         Chemistry CHEM404    0   WINTER          8
11      Vikram  Das 500000          9  60  Computer Science COMP490    1     FALL          9
12    Samuel Koffi 300000         10  40 Political Science    <NA>   NA     <NA>         NA
```

RIGHT JOIN

64: The RIGHT JOIN is illustrated below:



RIGHT JOIN is identical to LEFT JOIN, except that the primary table in this case is the **second** ("right") table appearing after the FROM clause.[64] Generally, a RIGHT JOIN and a LEFT JOIN can be used **interchangeably** by altering the order in which tables are selected.

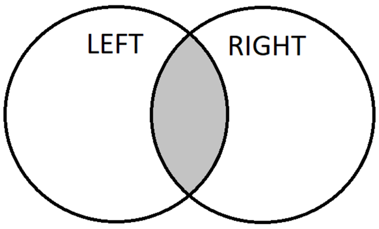
```
SELECT * FROM Professors RIGHT JOIN Courses
    ON Courses.FACULTY_ID=Professors.FACULTY_ID;
```

```
             NAME SALARY FACULTY_ID AGE           FACULTY  COURSE FULL Semester FACULTY_ID
1 Kyra Carmichael 200000          6  30          Business BUSI202    1   SUMMER          6
2      Landon Liu 120000          4  34 Cognitive Science CGSC101    1   SUMMER          4
3      Landon Liu 120000          4  34 Cognitive Science CGSC202    1   WINTER          4
4   Quine Ngyogne 115000          8  55         Chemistry CHEM404    0   WINTER          8
5     Vikram  Das 500000          9  60  Computer Science COMP490    1     FALL          9
6      Adam Smith  90000          1  67         Economics ECON101    1     FALL          1
7      Adam Smith  90000          1  67         Economics ECON401    0   WINTER          1
8            <NA>     NA         NA  NA              <NA> MUSI101    0   SUMMER         NA
9     Paige Ryans 180000          2  48           Physics PHYS201    0   WINTER          2
```

INNER JOIN

`INNER JOIN` is a type of join operation that combines rows from two tables based on the chosen matching condition(s), **omitting any unmatched rows**; the resulting table will contain only rows where both left and right tables meet the match criteria, all unmatched rows will be dropped.[65]

65: The `INNER JOIN` is illustrated below:



```
SELECT * FROM Professors INNER JOIN Courses
    ON Courses.FACULTY_ID=Professors.FACULTY_ID;
```

This query will provide a list of only those records for which there ia professor and a course match.

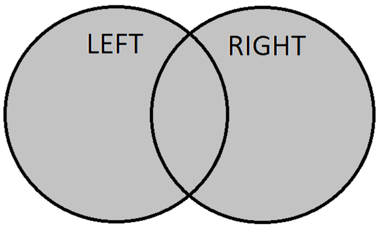
```
             NAME SALARY FACULTY_ID AGE           FACULTY  COURSE FULL Semester FACULTY_ID
1 Kyra Carmichael 200000          6  30          Business BUSI202    1   SUMMER          6
2      Landon Liu 120000          4  34 Cognitive Science CGSC101    1   SUMMER          4
3      Landon Liu 120000          4  34 Cognitive Science CGSC202    1   WINTER          4
4   Quine Ngyogne 115000          8  55         Chemistry CHEM404    0   WINTER          8
5     Vikram  Das 500000          9  60  Computer Science COMP490    1     FALL          9
6      Adam Smith  90000          1  67         Economics ECON101    1     FALL          1
7      Adam Smith  90000          1  67         Economics ECON401    0   WINTER          1
8     Paige Ryans 180000          2  48           Physics PHYS201    0   WINTER          2
```

FULL JOIN

`FULL JOIN` returns **all** rows based on the matching condition(s), including the **rows from both right and left tables**, replacing missing values with `NULL`; the input rows of both tables will be present in the output.

MySQL does not inherently support the `FULL JOIN` as this function is largely "syntactic sugar"; we can emulate it using `UNION` in conjunction with a `LEFT JOIN` and `RIGHT JOIN`.[66]

66: The `FULL JOIN` is illustrated below:



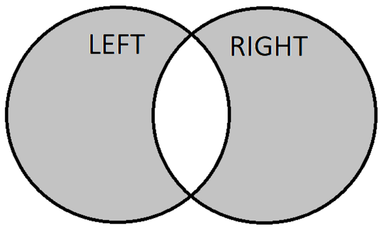
```
SELECT * FROM Courses LEFT JOIN Professors
    ON Courses.FACULTY_ID=Professors.FACULTY_ID
    UNION SELECT * FROM Courses RIGHT JOIN Professors
    ON Courses.FACULTY_ID=Professors.FACULTY_ID;
```

```
    COURSE FULL Semester FACULTY_ID            NAME SALARY FACULTY_ID AGE           FACULTY
1  BUSI202    1   SUMMER          6 Kyra Carmichael 200000          6  30          Business
2  CGSC101    1   SUMMER          4      Landon Liu 120000          4  34 Cognitive Science
3  CGSC202    1   WINTER          4      Landon Liu 120000          4  34 Cognitive Science
4  CHEM404    0   WINTER          8   Quine Ngyogne 115000          8  55         Chemistry
5  COMP490    1     FALL          9     Vikram  Das 500000          9  60  Computer Science
6  ECON101    1     FALL          1      Adam Smith  90000          1  67         Economics
7  ECON401    0   WINTER          1      Adam Smith  90000          1  67         Economics
8  MUSI101    0   SUMMER         NA            <NA>     NA         NA  NA              <NA>
9  PHYS201    0   WINTER          2     Paige Ryans 180000          2  48           Physics
10    <NA>   NA     <NA>         NA        Alex Doe 190000          3  37              <NA>
11    <NA>   NA     <NA>         NA    Marcel Orosz     NA          5  48              <NA>
12    <NA>   NA     <NA>         NA    Heather Wong 200000          7  34         Economics
13    <NA>   NA     <NA>         NA    Samuel Koffi 300000         10  40 Political Science
```

EXCLUSIVE JOIN

An EXCLUSIVE JOIN is a **syntactic concept**; the WHERE clause is appended to a JOIN command specifying to only return rows from the left table if **no matching data exists** in the right table. This modification effectively only return results that are **unique to each table**, but otherwise operate exactly as before.[67]

67: The EXCLUSIVE JOIN is illustrated below:



```
SELECT * FROM Courses RIGHT JOIN Professors
    ON Courses.FACULTY_ID=Professors.FACULTY_ID
    WHERE Courses.FACULTY_ID IS NULL;
```

This query will return a list of all professors not teaching a course.

```
  COURSE FULL Semester FACULTY_ID         NAME SALARY FACULTY_ID AGE           FACULTY
1   <NA>   NA     <NA>         NA     Alex Doe 190000          3  37              <NA>
2   <NA>   NA     <NA>         NA Marcel Orosz     NA          5  48              <NA>
3   <NA>   NA     <NA>         NA Heather Wong 200000          7  34         Economics
4   <NA>   NA     <NA>         NA Samuel Koffi 300000         10  40 Political Science
```

1.7.4 Examples

A Representative SQL Query Typical SQL queries tend to be more complicated than the few examples we have seen so far. The following example can be seen as representative of the level of sophistication/complexity we might encounter.[68]

68: We display the SQL keywords in lower case to make it easier to parse the query; the table and variable names have to be spelled exactly as they appear in the database, however. In practice, it might be a better idea to store the database variables and table names in lower case or camel case, and retain ALL CAPS for the SQL keywords. But you do you.

```
select NAME from
    (Professors left join Courses
    on Professors.FACULTY_ID=Courses.FACULTY_ID)
inner join
    (select COURSE, sum(STATUS in ('DNF', 'FAILED'))
        as Failing_Students
    from Registrations
    where STATUS in ('DNF', 'FAILED')
    group by COURSE order by Failing_Students desc limit 2)
as T on Courses.COURSE=T.COURSE;
```

```
             NAME
1      Adam Smith
2 Kyra Carmichael
```

It can be easier to understand a query if it is broken down from the innermost sub table.

1. We start by noting that we work on the Registrations table, and select only rows that contain a STATUS value of DNF or FAILED.

```
from Registrations
    where STATUS in ('DNF', 'FAILED')
```

2. In the sub-query, we select two fields: the `COURSE` field is returned as it appears in the data, and the count of instances where `STATUS` is `DNF` or `FAILED` (using the aggregation function `SUM`), which was saved as `Failing_Students`, now available to the outer query.

   ```
   select COURSE, sum(STATUS in ('DNF', 'FAILED'))
       as Failing_Students
   ```

3. The sub-query groups the output by the `COURSE` field, ordered by the count of `Failing_Students` in each course, but limited to the two largest instances.

   ```
   group by COURSE order by Failing_Students
       desc limit 2
   ```

4. We can now go to the primary query, in which the Professors table is joined to the Courses table to create a mapping of professors to the courses they teach.

   ```
   (Professors left join Courses
       on Professors.FACULTY_ID=Courses.FACULTY_ID)
   ```

5. The sub-query is assigned the table identifier `T`, which is inner joined with the primary query table, returning a table with the information of the two professors with the most "failing" students.

   ```
   inner join
   ...
   as T on Courses.COURSE=T.COURSE;
   ```

6. Finally, the resultant rows are isolated and only the `NAME` field is outputted, ultimately returning the names of the two professor with the most failing students.

   ```
   select NAME from
   ...
   ```

SQL in R It will not come as a surprise, especially after the reticulate detour of Section 1.6.4, that we can write SQL queries in R, with the appropriate library.[69]

69: The `dbname`, `host`, `port`, `user`, and `password` arguments are those of a test server where the toy example database can be accessed. For obvious reasons, this is a read-only situation. Just as obviously, the arguments would be different when working with a real database; contact your DBA (database admin) and consult the video linked to at the start of this section for more information and troubleshooting.

SQL in R

```
# install required library
library(RMySQL)

# connect to the database
mysqlcon = dbConnect(RMySQL::MySQL(),
    dbname='school', host='ayyws.com', port=3000,
    user='Ruser', password='Ruser')
```

```
[1] "Courses"       "Professors"    "Registrations" "Students"
```

```
# test the connection by listing all tables
dbListTables(mysqlcon)

# submit a query to the database
x = dbSendQuery(mysqlcon, "select * from Registrations;")

# convert the result to an R data frame, and display
data.frame = fetch(x)
print(data.frame)
```

```
  STUDENT_ID  COURSE GRADE     STATUS
1        100 ECON401    NA Registered
2        100 ECON101 10.00     Passed
3        101 ECON101  2.45     Failed
4        102 BUSI202    NA Registered
5        102 ECON101    NA        DNF
6        104 CHEM404    NA Registered
7        104 COMP490  9.80     Passed
8        101 BUSI202  3.52     Failed
```

1.8 Exercises

1. Write pseudo-code that will sort a list of numbers. Identify the inputs and the outputs, and solve the problem "procedurally" on a definite example before generalizing to a general list. You may need to "black box" the manipulation of individual numbers and group of numbers within the list.
2. Write pseudo-code that will enumerate all strings of up to `n` characters taken from the set `A-Z`, with no repeated character. Identify the inputs and the outputs, and solve the problem "procedurally" on a definite example before generalizing. Use "black boxes" as needed.
3. Use `R` to calculate the following quantities:
   - The sum of 1.001, 22.9, and -73.78
   - The square root of 64
   - Calculate the base 10 logarithm of 90, and multiply the result with the cosine of π. Hint: see `?log` and `?pi` for information about how to use .
4. Type the following `R` code, which assigns numbers to objects `x`, `y`.

   ```
   x<-252
   y<-5.5
   ```

   - Calculate the product of `x` and `y`
   - Store the result in a new object called `z`
   - Inspect your workspace by typing `ls()`, and by clicking the Environment tab in RStudio, and find the three objects you created
   - Make a vector of the objects 'x', 'y', and 'z'.
5. You have measured seven cylinders. Their lengths are: 2.1, 10.8, 5.5, 6.6, 9.7, 8.2, 8.1, and the diameters are: 0.4, 0.3, 1.2, 0.9, 0.3,

0.2, 0.1. Read these data points into two vectors (give the vectors appropriate names). Use R to calculate the volume of each cylinder ($V = \text{length} \times \pi \times (\text{diameter}/2)^2$).

6. Input the following data, related to space shuttle launch damage prior to the Challenger explosion. The set covers 6 launches out of 24 that were included in the pre-launch charts used to decide whether to proceed with the launch or not

Temp	Erosion	Blowby	Total
53	3	2	5
57	1	0	1
63	1	0	1
70	1	0	1
70	1	0	1
75	0	2	1

Enter these data into a R data frame, with column names `temperature`, `erosion`, `blowby`, and `total`.

7. Read the following data into R (number of honeyeaters seen at a site in a week). Give the resulting data frame a reasonable name. Type it into Excel or text file and save it as a CSV file or txt.

Day	nbirds	Day	nbirds
Sunday	3	Thursday	8
Monday	2	Friday	1
Tuesday	5	Saturday	2
Wednesday	0		

Enter the following data as new observations of a different week starting on Sunday: 4, 3, 6, 1, 9, 2, 0.

8. Read the data from the space shuttle launch (from the previous section) data into R.
9. Read the following data set (various Australian populations since 1917) into an R object. Write the object into a text file, from R.

Year	NSW	Vic.	Qld	SA	WA	Tas.	NT	ACT	Aust.
1917	1904	1409	683	440	306	193	5	3	4941
1927	2402	1727	873	565	392	211	4	8	6182
1937	2693	1853	993	589	457	233	6	11	6836
1947	2985	2055	1106	646	502	257	11	17	7579
1957	3625	2656	1413	873	688	326	21	38	9640
1967	4295	3274	1700	1110	879	375	62	103	11799
1977	5002	3837	2130	1286	1204	415	104	214	14192
1987	5617	4210	2675	1393	1496	449	158	265	16264
1997	6274	4605	3401	1480	1798	474	187	310	18532

10. What do you think the following R calls do?

```
swiss$var1 <- swiss[,1]>median(swiss[,1])
swiss$var4 <- swiss[,4]>median(swiss[,4])
table(swiss$var1)
table(swiss$var4)
table(swiss$var1,swiss$var4)
```

11. What do you think the following R calls do?

```
median(test, na.rm=TRUE)
min(test, na.rm=TRUE)
max(test, na.rm=TRUE)
quantile(test, na.rm=TRUE)
```

12. In Python:
    a) evaluate $\lfloor 10001/4 \rfloor$ and $\arcsin(\pi/4)$;
    b) obtain the value of s in the following: $a = \pi(1+\ln 5)$, $b = \frac{1}{3+\sqrt{4}}$ and $s = a + b$;
    c) obtain a formatted string of $\sin(\pi^2)$ of width 8, with 5 decimal places;
    d) turn the value of $\sqrt{3}$ into a fixed decimal with 8 decimal places.
13. In Python:
    a) create a list of integers from -10 to 5;
    b) use list comprehension to create a list `(x,y)` so that `x+y > 8` where `x` can be any nonnegative integer at most 10 and `y` can be any positive integer at most 7;
    c) use list comprehension to create a list `(x,y)` so that `y` is the square of `x` and `x` is from 1 to 10;
    d) write one line of code that returns a list obtained from

    ```
    x = ['one', 2, 3, 'four', 5, 6, 'seven', 8, 9, 10,
              'eleven', 12, 13, 'fourteen']
    ```

    by moving all the elements of type `str` to the end of the list. (Hint: Use list comprehension and concatenation. To check if `a` is of type `str`, use `type(a) is str`. To check if `a` is not of type `str`, use `type(a) is not str`.)
14. Write an if statement in Python that prints "`odd`" if `x` is odd and prints "`even`" if `x` is even where `x` is a random integer between -100 and 100, inclusive.

```
  import random
  x = random.randint(-100,100)
```

    (Hint: `x % n` returns the remainder of `x` divided by `n`).
15. Use a single while loop in Python to print all pairs `(x,y)` such that `x+y=100` and `x` ranges from 0 to 50.
16. Write a Python function `myFunc()` that returns the square of `x` if `x` is of type `int` and returns `None` otherwise (hint: `type(x) is int` is the syntax for testing if `x` is of type `int`).

```
def myFunc(x):
    res = None
    ## Your code here

    return res
```

Verify that the function behaves as expected:

```
assert(myFunc(5) == 25)
assert(myFunc('five') is None)
```

17. Write a function `mySoS()` that accepts a list of floats as the only argument and returns the sum of squares of the numbers (assume that the argument is indeed a list of floats – no need to test if the condition is met).

```
def mySoS(ns):
    res = 0
    ## Your code here

    return res
```

Verify that the function behaves as expected:

```
assert(mySoS([1.0,2.0,3.0]) == 14.0)
assert(mySoS([-2.5,1.3,13.4]) == 187.5)
```

18. What is the result of the following code?

```
def mystery(func, n):
    return [func(i) for i in range(n)]

print(mystery(lambda x: (2*x+1)**2, 5))
```

Rewrite the function using an anonymous function (single line).

19. Complete the definition of the `Python` function `myRep()` with arguments `x`, `y`, and `n` (where `x` and `y` can be assumed to be strings and `n` can be assumed to be a nonnegative integer) that returns the string `x+y` repeated `n` times.

```
def myRep(x, y, n):
    res = ''
    # Your code here

    return res
```

Verify that the function behaves as expected:

```
assert(myRep('a','b',3) == 'ababab')
assert(myRep('Python','C',0) == '')
```

20. Complete the definition of the `Python` function `posOfi()` with argument `s` and returns a list of indices at which `s` contains the letter ‘i’ (hint: use the enumerate function).

```
def posOfi(s):
    # Your code here
    return None
```

Verify that the function behaves as expected:

```
print(posOfi("Mississipi"))
print(posOfi("Harry Potter"))
```

21. Complete the following `Python` function which takes a string consisting of a paragraph of sentences ending with a period and returns a list of all the sentences, with leading and trailing spaces stripped. You may assume that every period ends a proper sentence and there are no sentences not ending in a period.

```
def sentences(p):
    # Your code here
    return None
```

Verify that the function behaves as expected:

```
p = 'The essence of Python.  One can sense. But not learn. '
print(sentences(p))
```

22. What effect do the methods `upper()`, `lower()`, and `title()` have on non-alphabetical characters?
23. Complete the following function which takes a list of full names as argument an returns a list of names that are not properly capitalized. For example, for the argument `['John Doe', 'JANE Kelly', 'nicole dunn', 'David Huang']`, the function returns `['JANE Kelly, 'nicole Dunn']`.

```
def badNames(names):
    # Your code here
    return None
```

24. Complete the following function which takes a list `l` of strings as argument and returns a list consisting of the strings in `l` not containing the symbol `-`. For example, given the argument `['Hi', 'Good-bye', 'Ciao', 'Twenty-one']`, the function should return `['Hi', 'Ciao']`.

```
def filterList(l):
    # Your code here
    return None
```

25. Complete the following function which takes a list of pairs as argument and returns a dictionary with the first components as keys and the second components as the corresponding values. For example, given the argument `[(1,'a'),(2,'b')]`, the function returns `{1: 'a', 2: 'b'}`.

```
def pairListToDict(pairs):
    # Your code here
    return None
```

26. Complete the following function which takes a dictionary as argument and removes all the key-value pairs that do not have values of type `str`. For example, calling the function with the dictionary `{'one': 1, 'two': 'Two', 'three': 3}` will change the dictionary to `{'two': 'Two'}`.

```
def filter(d):
    # Your code here
    return
```

27. Complete the following code so that `sq` is a 1D `numpy` array of the squares of the first 100 positive integers. Use list comprehension.

```
sq = np.array([...])
```

28. Obtain a `NumPy` array from the array `sq` in the section by applying the function $\sqrt{x} + 1$ to each entry `x` in `sq` (hint: use broadcasting and `np.sqrt()`).
29. Complete the following definition of `myFunc()` which takes a positive integer argument `n` and a positive real number `d` and generates an array of `n` random values drawn from the standard normal distribution and returns the number of values whose absolute values are less than or equal to `d`. You may assume that `n` is a positive integer and `d` is a non-negative float when `myFunc()` is called (hint: use `numpy.random.randn()` for generating the random array).

```
def myFunc(n, d):
    # Your code here

    return 0
```

Verify that the function behaves as expected:

```
np.random.seed(5900)
assert(myFunc(10000,1) == 6848)
assert(myFunc(100000,2) == 95490)
```

30. Obtain the `iris` data set through `seaborn` and generate some summary statistics.
31. Write code to change the labels in the data frame `crashes` from `Cnnn` to `Incident nnn` and turn that column into an index column. Commit these changes to `crashes`.
32. Extract a data frame from `df` consisting only of the columns `speeding` and `alcohol` for which the `speeding` values are at least 3.0 and the `alcohol` values are at most 4.5.
33. There is a powerful way to filter rows involving complex boolean expressions via the `query()` method. For instance,

```
df.query("ins_losses > 160 & ins_premium < 900 & abbr == 'CA'")
```

```
   case  speeding  alcohol  ...  ins_premium  ins_losses  abbr
4  C005       4.2     3.36  ...       878.41      165.63    CA
```

```
[1 rows x 8 columns]
```

Extract a data frame from `df` *via* `query()` consisting of records for which `alcohol` is at most 4.0 and `abbr` is neither `CA` nor `LA`.

34. Obtain a data frame `df4` by changing the column name of `Student ID` in the data frame `gpa` to `ID`. Then create `df5` by merging `df4` and `df` using `pd.merge(df4, df, on='ID')` and summarize the resulting data frame.
35. Perform an outer join with `df4` from the previous exercise and `dfB`.
36. Drop the observations in the original `gpa` data frame for which the only `NaN` values are found in the `GPA` column.
37. Replace the `NaN` in the original `gpa`'s `Year` column with the string `Unknown`.
38. Modify `markToGrade` so that a mark between 80 to 100 (incusive) is converted to an `A`, a mark at least 70 but less than 80 is converted to a `B`, a mark at least 60 but less than 70 is converted to a `C`, a mark at least 50 but less than 60 is converted to a `D`, and a mark below 50 is converted to an `F`.

```
def markToGrade(x):
    res = ’F’
    # Your code here
    return res
```

Add a Grade column to `df3` containing the converted grades.

39. Obtain the mean for each of the `Year` groups in the `calc` data frame.
40. Obtain the mean, standard deviation, and median for each of the `Year` groups in the `calc` data frame, using `agg()`.
41. Produce a summary of the `calc` data frame giving the `Grade` mean and standard deviation, and the `GPA` median, grouped by `Years`.
42. Complete the definition of a function that returns `Satisfactory` if the average of the array `x` is at least 65.0 and `Unsatisfactory` otherwise.

```
def groupStatus(arr):
    res = ’’
    # Your code here

    return res
```

Determine the group status in the `calc` dataset by both `Sex` and `Year`, for the `Grade` variable.

43. Write a function that produces the pivot table displaying the number of students with a passing grade by `Sex` and `Year` (hint: if `arr` is a `NumPy` array, then `arr >= 50.0` gives an array of the same length such that element `i` is `True` if and only if `a[i] >= 50.0`).
44. Carry out the remaining exercises in both `R` and `Python`. There is no need to do the exercises in any particular order. Take the time to design pseudo-code and think about what the code does before jumping directly into the programming. You may choose to carry out each of the exercises separately, or to write a single program that carries out all of the individual exercises. You will find much

of the base code you need in the chapter's examples, but you may need to tweak and add to this code to carry out the exercises. Do not hesitate to look for information and inspiration on the Internet and in the documentation.

a) Create three variables and assign numerical values to each of these variables. Then write one or more statements that carry out the following types of operations using these variables: addition, subtraction, multiplication, division, raising to a power.
b) Create three variables and assign string values to each of these variables. Write a statement that joins the three strings into a single string. Write some code that prints the string. Write some code that tests to see if a substring of your choice is contained within the larger string.
c) Create three variables and assign lists to each of these variables. Join the three lists into a new list containing three distinct sub-lists (a list of three lists). Create a list from this list without sub-lists (all original list elements are part of a single larger list). Create a fourth list by splitting this resulting list in half and assigning the second half of the list to a new variable. Extract the last item of this list (it can either stay in the original list or be removed from it) and assign this element to a variable.
d) Write a statement that contains at least three nested blocks. Use at least three of the following control flow options: if, if else, while, for, break, continue (`Python` only), next, switch.
e) Write a function that takes three arguments as input and returns one value. Call the function with arguments of your choosing.
f) Execute the relevant command that shows a list of the packages (for `R`) or modules (for `Python`) that are currently installed in your environment. Use the available documentation to determine what some of these do. Write some code that uses functions and objects supplied by these packages.
g) Print to the standard output three sentences of your choosing, on three separate lines, using a single statement of code.
h) Locate a comma separated values (`.csv`) file stored on your computer or online. Read this file into the notebook and store the results in one or more variables.
i) Create a new file and write four lines in `.csv` format to this file. In a separate statement, write four more lines to this existing file, without overwriting the original file.
j) Write enough code to generate at least five different error messages. Copy these error messages into a text document, and write a short note under each explaining the meaning of the error message, and how the code was fixed.
k) Using a language of your choice, write a function that, when passed a dataset, reports 5 interesting pieces of information about the dataset. Load a dataset and run the function on this dataset.
l) Using a language of your choice, write two functions. The output of the first function should work as the input to the second function. The first function should read in a dataset and generate a subset of the dataset based on some chosen

criteria. The second function should read in a dataset and provide summary data of some type for each column in the dataset. Load a dataset and run both functions on the dataset.

m) Write a program that sorts a list of numbers, without using the in-built sorting functions.

n) Write a program that sorts a list of character strings, without using the in-built sorting functions.

45. Consider a database consisting of two tables, as shown below.

Professors

NAME	SALARY	FACULTY_ID	AGE	FACULTY
Adam Smith	60000	1	67	Economics
Paige Ryans	70000	2	48	Physics
Alex Doe	55000	3	37	

Courses

COURSE	FULL	SEMESTER	FACULTY_ID
ECON101	True	Fall	1
PHYS201	False	Winter	2
ECON401	False	Winter	1
...	...	...	...

a) What is the primary key for each table?
b) What are the foreign keys for each table?
c) What are the `NULL` values?
d) What is the relation between these tables?
e) What type of data does each field support?
f) What constraints might we expect each field to have?
g) What would happen if we tried to mix datatypes without enforcing constraints?