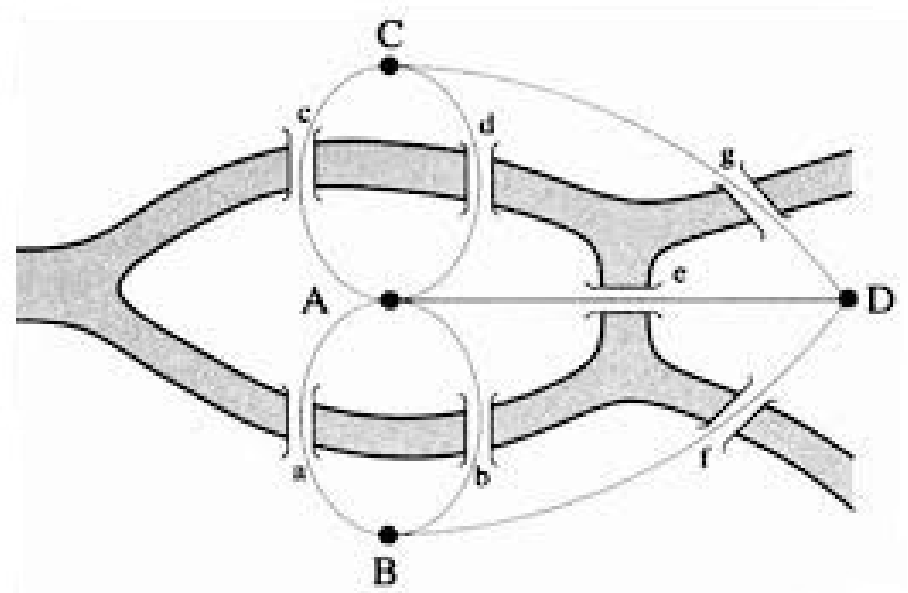


Performance Engineering in Python

Antoine B. Rauzy

PUBLISHED BY THE ALTARICA ASSOCIATION
ALTARICA-ASSOCIATION.ORG

First printing, December 2020
ISBN: 978-82-692273-1-4

The Universal Declaration of Human Rights

Proclaimed by the United Nations General Assembly in Paris on 10 December 1948
(excerpt)

Article 1. All human beings are born free and equal in dignity and rights. They are endowed with reason and conscience and should act towards one another in a spirit of brotherhood.

Article 2. Everyone is entitled to all the rights and freedoms set forth in this Declaration, without distinction of any kind, such as race, colour, sex, language, religion, political or other opinion, national or social origin, property, birth or other status. Furthermore, no distinction shall be made on the basis of the political, jurisdictional or international status of the country or territory to which a person belongs, whether it be independent, trust, non-self-governing or under any other limitation of sovereignty.

⋮

Article 18. Everyone has the right to freedom of thought, conscience and religion; this right includes freedom to change his religion or belief, and freedom, either alone or in community with others and in public or private, to manifest his religion or belief in teaching, practice, worship and observance.

Article 19. Everyone has the right to freedom of opinion and expression; this right includes freedom to hold opinions without interference and to seek, receive and impart information and ideas through any media and regardless of frontiers.

⋮

Article 26.

1. Everyone has the right to education. Education shall be free, at least in the elementary and fundamental stages. Elementary education shall be compulsory. Technical and professional education shall be made generally available and higher education shall be equally accessible to all on the basis of merit.
2. Education shall be directed to the full development of the human personality and to the strengthening of respect for human rights and fundamental freedoms. It shall promote understanding, tolerance and friendship among all nations, racial or religious groups, and shall further the activities of the United Nations for the maintenance of peace.
3. Parents have a prior right to choose the kind of education that shall be given to their children.

Article 27.

1. Everyone has the right freely to participate in the cultural life of the community, to enjoy the arts and to share in scientific advancement and its benefits.
2. Everyone has the right to the protection of the moral and material interests resulting from any scientific, literary or artistic production of which he is the author.

⋮

Contents

Case Studies

Chapter 1

Introduction

1.1 Objectives of the Book

The objective of this book is to present fundamental concepts, methods and tools to assess and to optimize the performance of complex technical and socio-technical systems using the Python programming language. *De facto*, it is also an example-driven introduction to Python itself. Its intended audience is students, engineers and scientists working in all engineering disciplines.

Undoubtedly, computer programs play an increasingly important role in all engineering processes. Consequently, engineers and scientists must be familiar with their use and, at least to some extent, with their development. I do believe however that developing large, professional software is the job of professional software developers. Training a professional software developer takes years. It remains that scientists and engineers from all disciplines can take huge benefits of mastering a programming language such as Python. Python can actually help them to achieve a high productivity in many of their daily tasks, including data processing and analysis, chaining modeling and simulation tools, performing specific calculations and simulations, writing reports and web-pages... These tasks can be at least partly automated by writing scripts, i.e. small programs, quickly and easily developed. Writing scripts does not require high competences in mathematics, computer science or software engineering, just to know the basics and to be ready to give it a try. More than that: it proves to be fun.

My objective, when I created the course "TPK4186 - Advanced Tools for Performance Engineering" at the Norwegian University of Science and Technology (NTNU), was to provide students in engineering curricula with fundamental knowledge and know-how about Python scripting. I wanted to show them that they can, using Python, improve their productivity right now, without having to swallow tons of difficult concepts on algorithms, computational complexity and the like. This book results from this teaching experience. At the time I am writing these lines, I am using it as the compendium of the course.

The reader should not misinterpret my point: I am not saying here that programming is easy, does not require any specific knowledge and that anybody can improvise himself or herself a Python programmer. Learning programming cannot be “left as an exercise”. Only inexperienced programmers pretend that programming is easy. What I am saying and willing to show through these pages is that all scientists and engineers can and should be trained with the fundamental knowledge and know-how about how to write efficient and useful scripts. Moreover, this training needs to be specific to non-professional software developers. It includes, for instance, the ability to design appropriate data structures, to use recursive algorithms and to program in an object-oriented style. This is the reason why this book does not compromise on these topics.

With that respect, Python provides an excellent trade-off between the ability to develop full-fledged programs and to avoid tricky algorithmic details. Of course, there is a price to pay for that: Python scripts are not as efficient as, for instance, C++ programs. But they are efficient enough for most of daily engineering tasks. Moreover, Python comes with many toolboxes. There are toolboxes for virtually all engineering tasks that can be automated. New ones are created and made available on internet everyday.

Using these toolboxes makes your scripts much more efficient than if you would have written them by yourself. Even more importantly, it makes their design much easier, faster and safer: rather than to implement complex algorithms, you just have to write the few lines of code that calls these already implemented algorithms. At the end of the day, when you want to hang a painting at home, you do not build first your own drilling machine. The same idea applies for toolboxes.

Another key feature of Python and its toolboxes is that they are freely available on Windows, Linux and Mac. To use them, you just have to download them.

Last but not least, literally millions of people are using Python daily. The Python community is very active on internet. Not only the full documentation of the language is there, but thousands of forums provide you with answers to nearly all the questions you may have. As a rule of thumb, if you have a problem, you can be sure that someone had it before you and posted valuable information somewhere.

In a word, mastering a scripting language such as Python and being able to use its associated toolboxes is one of expected skills of new generations of engineers and scientists. This book will hopefully helps the reader to acquire this ability.

1.2 How to Read this Book

Programming is like swimming or playing the piano: you learn by practicing. It is good to have some background knowledge, to understand what is going on, but the only way to learn how to write Python scripts is to write Python scripts. There is no such a thing as "theoretical scripting". You have to try, to fail, to try again, to fail again, and so on until at some point, you'll get things done. This book is thus to be read with a Python programming environment open, going forth and back from the text and the programs.

This book gathers many exercises and problems that will take the reader, step by step, towards Python expertise. Most of these exercises and problems answer concrete questions one has to solve when assessing and optimizing the performance of a technical and socio-technical system. With that respect, this book can be seen as a cookbook, full of recipes that you can reproduce at home, and, more importantly, from which you can take inspiration to prepare gala dinners. I provide solutions to all exercises on my web-page `http://www.altarica-association.org/members/arauzy/Teaching/teaching.html`.

Cooking is a good metaphor. As for cooking, you never walk alone when programming: the Python community is there on internet to help you. Aside the official documentation `https://docs.python.org`, hundreds of tutorials, on-line exercises and reference books are available not only in English, but also in many other languages. To write this book, I used in particular references (Lutz, 2011), (Guttag, 2016), (Grayson, 2000) and (Downey, 2018) (there are pdf free versions of the most part of them) as well as many websites, including the one of the W3C consortium `https://www.w3schools.com/python/default.asp`. Pick-up tutorials, books and websites that suit the most to you. Do not hesitate to use them aside this book.

To solve these exercises and problems proposed along these pages, you will need internet. Learning to find your way to the information you need is fully part of the development of your Pythonic skills. Even professional Python developers are making a constant use of internet: why remembering all technical details while you can retrieve them easily?

A last advice: do not panic if you do not fully understand the code provided as the solution to a problem. Especially when doing your first steps in programming, it is often the case that you have to reproduce a code without getting the meaning of every details. Do not worry, you will progressively acquire a better understanding. Run the script, possibly for several sets of input data, possibly modify it slightly, observe the results. Trial and error is the only way to learn programming.

1.3 Organization of the Book

This document proposes two types of exercises: exercises, which can normally be done quickly and easily, and problems, which require more time and thinking.

A number of case studies will be used throughout the document. These case studies are presented only once, then simply referred to. It is strongly advised to do look at them carefully.

This document is organized into two parts, ten chapters (in addition to this introduction) and an appendix.

The objective of the first part is twofold: first, introducing the main constructs of Python, second showing how to use Python to manage engineering data. This part is made of six chapters:

- Chapter 2 introduces the most basic constructs of Python, namely variables, instructions and functions.
- Chapter 3 introduces Python containers, namely tuples, lists and dictionaries. These three data structures are provided with the language. They make it possible to store data of various types, they contain these data, hence their name.
- Chapter 4 introduces first text files. Text files are the simplest way to save data onto a permanent support, e.g. the hard disk of your computer. The remainder of the chapter is dedicated to regular expressions. Regular expressions are a fantastic tool to process texts.
- Chapter 5 deals with modules and packages, i.e. constructs to split large programs into independent, manageable units. Packages are the real richness of Python: they implement the above mentioned toolboxes. As an illustration, this chapter introduces briefly two very useful packages: `NumPy` and `Matplotlib`.
- Chapter 6 presents object-oriented constructs of Python. These constructs make it possible to create data structures to store and perform operations on complex data sets. All subsequent chapters rely on the object-oriented paradigm.
- Finally, Chapter 7 discusses how to save data into files and conversely to read data from these files.

The second part of the document aims at illustrating how to use Python to perform experiments *in silico*. It goes indeed far beyond the scope of this document (and of any single book) to cover all possible experiments. Consequently, we give here only a snapshot of some methods and tools, in four chapters:

- One of the key advantage of computer-based experiments is that their cost is low, at least compared to the one of physical experiments. It is thus in general possible to perform many, possibly by making input parameters vary. Chapter 8 deals with the why and wherefore of randomness in computer-based experiments.
- Chapter 9 shows how to implement domain specific languages, i.e. (mini) computer languages dedicated to a particular engineering purpose. To illustrate the presentation, this chapter builds step by step a calculator for continuous time Markov chains with rewards.
- Chapter 10 discusses, by means of various case studies, how to implement discrete event simulations in Python.
- Finally, Chapter 11 sacrifices to the current trend by giving a flavor of machine learning techniques that can be easily implemented in Python.

Appendix A provides a description of the main case studies used throughout the book.

Appendix B recalls some elements of probability theory.

1.4 Required Configuration

Python was conceived in the late 1980s by the dutch developer Guido van Rossum (van Rossum, 1995). It now developed by many people, acting under the direction of a steering committee. There are two major versions of Python: Python 2 and Python 3. Despite of the efforts of the developers of the language, these two versions are not fully compatible. This course is based on Python 3.

You have thus download and install the Anaconda environment for Python 3.7 or later.

https://www.anaconda.com/download/

On Windows, install Anaconda only for you (not for all users). This will avoid problems when installing packages.

In addition to Anaconda, it is often convenient to have a good text editor to edit data files and programs. If you have a PC under Windows, you should download and install the notepad++ text editor.

https://notepad-plus-plus.org/download/

Part I

Managing Engineering Data

Chapter 2

Basic Constructs

Key Concepts
- Basic types: Boolean, integers, floating point numbers, strings
- Expressions
- Variables
- Assignment, conditional instructions, loops
- Functions

2.1 Introduction

This chapter presents basic constructs of Python: constants, expressions, variables, instructions, functions and (text) files. These constructs can be found under different forms in virtually all programming languages. To introduce them, this chapter proposes a series of small exercises, mostly based on elementary mathematics.

But before starting, it is worth recalling how a computer works. Figure 2.1 gives a schematic representation of the main component of a computer. This view is oversimplified, but sufficient for our purpose.

Strictly speaking, a computer is a made of the two components inside the dashed rectangle, namely a *central processing unit* (CPU) and a *random access memory* (RAM). The central processing unit performs calculations on data, like adding two numbers or changing upper case letters into lower case letters in a word. It cannot store much data however. This is the reason why most of the data are stored in the random access memory. The term random should not be mistaken here. It does not mean that the memory is accessed at random, but rather that any part of the memory can be accessed at any time. The random

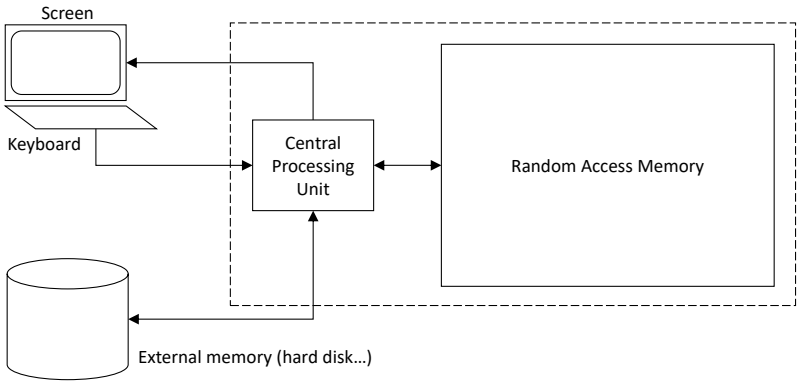


Figure 2.1: Schematic representation of a computer

access memory is split in a number of machine words. Each word is itself made of a number of binary digits (bit), 64 in most of the modern personal computers. All data, whatever their type (numbers, texts...), are stored eventually encoded and stored into machine words. When the computer is turn on, its random access memory is empty. When it is turn off, it is flushed out.

A *computer program* is thus essentially a series of *instructions* that order the computer to:

– Load data from the random access memory into the central processing unit;
– Process in some way these data;
– Put back data, possibly modified, possibly newly created, in the random access memory.

This is not enough however: the computer needs to communicate with external devices such as the keyboard, the screen, the hard disk and many other not represented on the figure (microphones, loud speakers, printers, internet...). Therefore, there are instructions to do so.

The keyboard and the screen are the two main devices to communicate with the user. The keyboard is the *standard input* of the computer, while the screen is its *standard output*. The hard disk makes it possible to store data permanently. It keeps its content even when the computer is turned off. On the hard disk, data are stored into separate units called *files*. Some of these files are readable with a text editor, they are called *text files*. We shall make an extensive use of this type of file. In particular, *Python programs*, also called *scripts* are stored into text files.

Python scripts are not directly executed by the computer. Rather, they are read and executed by another program, called a *Python interpreter*. This program is itself not directly executed by the computer. Rather, it read and executed by another program, called the *operating system*. The operating system, e.g. Windows®, MacOs® or Linux, is in charge of managing the interactions between the different parts of the computer and of providing services to other programs.

These preliminary remarks made, we can start our presentation of basic constructs of Python.

2.2 Variables and Instructions

2.2.1 Variables, constants, expressions and assignments

Variables: As said above, data are stored in the random access memory. Each machine word in the random access memory has an "address", just like houses in a city. This address is however simply a number ranging from 0 to the number of machine words (minus one) available in the memory. It would be especially inconvenient and error prone to have to specify "by hand" addresses. This is the reason why all programming languages introduce the notion of variable.

A *variable* is a name for an address in random access memory where a datum is stored. In Python as in most of the programming languages, variable names , so-called *identifiers*, start with a letter or an underscore '_' which is followed by any number (including zero) of letters, digits, and underscores. The following sequences of characters are valid variables names.

– `x`,
– `_`,
– `CurrentInterestRate`,
– `We_know_where_42_comes_from`.

As a rule of thumb, one tries to give names to variables that make it easy to recall what information the variable is storing. However, too long names make programs difficult to read.

Note that identifiers are case sensitive: `x` and `X` are not the same identifiers.

Constants: Aside variables, Python, again like all programming languages, provides an (infinite) number of *constants*. Constants are of one of the following basic types:

– *Boolean constants*, i.e. `True` and `False`.
– *Integers*, e.g. `0`, `1`, `42`, `-12253811811182...`
– *Floating point numbers*, e.g. `1.2`, `-.056`, `1.23e23`, `5.13e-6...`

– *Strings*, i.e. sequences of characters surrounded either by two '' or two "", e.g.
  – `'This is a string.'`,
  – `"This is also a string"`.

Two important remarks here:
– Conversely to many programming languages, Python accepts integers and floating point numbers as big (and as small) as you want.
– Strings can embed special characters, such as '`\t`' (tabulation), '`\n`'. Moreover, they must fit on one line (we shall see that this not a limitation).

Assignments: The most basic Python instruction is the *assignment*, e.g.

```
1 text = "Hello, world!"
2 anotherText = text
3 text = "I changed my mind."
```

The first above assignment does actually two things:
– It creates the variable `text` (assuming this variable does not already exist).
– It gives it the value `"Hello, world!"`.

The second assignment does also two things:
– It creates the variable `anotherText` (assuming this variable does not already exist).
– It gives it the value of the variable `text`, i.e. `"Hello, world!"`.

The third assignment gives the value `"I changed my mind."` to the variable `text`. The value of the variable `anotherText` remains unchanged, i.e. its value is still `"Hello, world!"`.

It is very important to understand that, in the second assignment, the left hand side member of the assignment, i.e. `anotherText`, denotes the variable to be assigned a value, while in the right hand side member of the assignment. i.e. `text`, denotes the value of the variable `text` and not the variable itself. This makes it possible to use, as the right hand side member of an assignment, not only constants and variables, but also *expressions*, built with constants, variables and operators applying on Boolean, numerical and string arguments. E.g.

```
1 x = 1
2 y = x + 2
3 x = 2 * (x + y)
```

The first assignment gives the value 1 to `x`. The second one gives the value $1+2=3$ to `y`. The third one gives the value $2\times(1+3)=8$ to `x`.

There are hundred of operators available in Python, see the on-line documentation for a review.

Comments: It is often convenient to insert comments in a script. In Python, all what follows a character `#` till the end of the line is a comment. E.g.

```
1 # This is a comment
2 x = 1 # this is also a comment after the instruction
```

It is however advised not to abuse of comments. Using significant identifiers is a much better approach.

2.2.2 Conditional instructions and loops

Aside assignments, there are two other fundamental categories of instructions: conditional instructions and loops.

Conditional instructions: *Conditional instructions* make a choice depending on the value of a *condition*, i.e. a Boolean expression. E.g.

```
z = 0.95
if z<=0.5:
  level = 0
elif z<=0.9:
  level = 1
else:
  level = 2
```

The above code does the following. First, the variable `z` is created and given a value (in this case `0.95`). Then a conditional instruction is applied: if the value of `z` is less or equal to 0.5, the variable `level` is created and assigned the value 0, otherwise if the value of variable `z` is less or equal to 0.9, the variable `level` is created and assigned the value 1, otherwise the variable `level` is created and assigned the value 2.

In this example, the variable `level` is created and assigned a value in all branches of the alternative. This is by no means mandatory. Moreover, there can be any number of `elif` branches (including none) and the `else` branch is optional. `elif` is a contracted form of `else if`.

Indentation: In the above code, instructions to be executed in each branch of the alternative are shifted to the left by two space characters. One says that the text of these instructions are is *indented* (of two space characters).

Conversely to many programming languages where space, new line and tabulation characters are not significant, indentations have a special meaning in Python: they indicate that indented instructions belong to the same *block* of instructions. E.g.

```
if z<=0.5:
  level = 1
  density = 30.4
```

In the above code, if the value of `z` is less than 0.5, the variable `level` is assigned the value 1 and the variable `density` is assigned the value 30.4 (and these variables are created if they did not already exist). This means that the values of `level` and `density` are not modified if the value of `z` is greater than 0.5. Actually, these variables are not even created if they did not exist prior the execution of the conditional instruction.

In contrast, consider the following code.

```
if z<=0.5:
  level = 1
density = 30.4
```

In the above code, if the value of `z` is less than 0.5, the variable `level` is assigned the value 1, but the variable `density` is assigned the value 30.4, whatever the value of `z`. Actually, the assignment of `density` does not belong to the conditional instruction. It comes after.

Beginners should be especially careful about indentation. It is a great source of coding mistakes and errors. Note that the indentation can be made of any number of space and tabulation characters, but it must be the same for all instructions in a group and for the different branches of a conditional instruction. We strongly advise to follow our convention: indentations are always made of two spaces.

Note also that `if`, `elif` and `else` are *keywords*, i.e. reserved identifiers that one is not allowed to use for variables. This is more generally the case for all words introducing instructions.

Loops: *Loops* are instructions that iterate an instruction or a block of instructions until a certain condition is realized. There are several type of loops, but for now, we shall consider only `while` loop and use such a loop to write our first complete script. It is given in Figure 2.2.

```
1 # Code to test the Syracuse conjecture
2 n = 10816191671 # Well, you can put any positive integer here
3 while n!=1:
4   print("n = " + str(n))
5   if n%2==0:
6     n = n//2
7   else:
8     n = 3*n + 1
9 print("Yes! n = " + str(n))
```

Figure 2.2: Python code to test the Syracuse conjuncture

This script can be used to test the Syracuse conjecture, which is probably one of the simplest, however yet unsolved, mathematical problem one may imagine. It works as follows.

Start with any positive integer. Then each term is obtained from the previous term as follows: if the previous term is even, the next term is one half the previous term. If the previous term is odd, the next term is 3 times the previous term plus 1. The conjecture asserts that you will eventually reach 1, whatever number you start with.

The above code starts thus with assigning a value to the variable `n`.

Then, we have the loop. `!=` stands for $\neq$.

The instruction `print`, line 4, prints out a string on the standard output file, i.e. normally the screen. This string is obtained by *concatenating*, i.e. by gluing, two strings: the string `"n = "` and the string obtained by transforming the value of `n` into a string by means of the operator `str`. The operator `+` works a the concatenation when applied on strings, i.e. `"bla" + ", " + "bla"` equals `"bla, bla"`.

The operators `%` (modulo) and `//` (quotient) are respectively the remainder and the quotient of two integers. Note that the division `p/q` of two integers `p` and `q` gives always a floating point number, even though `p` is a multiple of `q`.

Not only you can "add" strings, but you can also access to each character of a string. Characters are of a string are indexed from 0. The *i*th character of a string `s` is accessed by the expression `s[i]`. The length of a string is obtained by the expression `len(s)`. To print one by one the characters of a list, you can thus use the following loop.

```
text = "Hello, world!"
i = 0
while i<len(text):
  print(str(i) + "\t" + text[i])
  i += 1
```

Tests
When you design a program, it is of primary importance to test it extensively. A good methodology consists in designing the tests even before designing the program.
Don't hesitate to insert `print` instructions here and there in your script so to follow what's going on, step by step.

2.2.3 Let's go!

The following exercises illustrate the use of variables, conditional instructions and `while` loops. They illustrate also how to group instructions by means of indentation.

Exercise 2.1 Count Parentheses. In the text of an arithmetic expression, the number of parentheses must be balanced, i.e.

i) There must be as many left parentheses as there are right parentheses;
ii) In any prefix of the text, the number of left parentheses greater or equal to the number of right parentheses.

Question 1. Write a loop that counts the number of left and right parentheses in a string.

Question 2. Add to your loop, instructions to count the number of unbalanced prefixes and that detects the first index at which the string gest unbalanced.

Question 3. Apply your code on different strings.

Exercise 2.2 Syracuse Conjecture. We are not completely finished with the Syracuse conjecture.

Question 1. Modify the code of Figure 2.2 so to count the number of steps it takes to go down to 1.

Exercise 2.3 Second Degree Equations. The simplest thing Python can do is to perform calculations.

Question 1. Write a Python script that solves second degree equations.

Hint: You will need a function that calculates the square root of a number. In Python this function provided by the module `math`. To be able to use it, you need to insert the line `import math` at the beginning of your script. Then, you can obtain the square root of any number n by writing `math.sqrt(`n`)`.

Question 2. Apply your script to solve, if possible, the following equations.

$$\begin{aligned} x^2 - 4x - 5 &= 0 \\ 3x^2 - 2\sqrt{6}x + 3 &= 0 \\ -\frac{1}{2}x^2 - \frac{11}{3}x - \frac{7}{6} &= 0 \end{aligned}$$

Exercise 2.4 Greatest Common Divisor. The Euclid's algorithm to find the greatest common divisor

(gcd) of two natural numbers a and b is defined by the following recursive equations.

$$\begin{aligned}
\gcd(a,a) &= a \\
\gcd(a,b) &= a \qquad \text{if } a = b \\
\gcd(a,b) &= \gcd(a,b-a) \qquad \text{if } b \geq a \\
\gcd(a,b) &= \gcd(b,a-b) \qquad \text{if } b < a
\end{aligned}$$

Question 1. Write a Python script that implements Euclid's algorithm.

Question 2. Apply your script to find the greatest common divisor of the following pairs of natural numbers.

$a = 12345 \quad b = 67890$
$a = 54321 \quad b = 9876$

Exercise 2.5 Secant Method. The secant method is a method to find a root of a function $f(x)$ of a real variable x, i.e. of value of x such that $f(x) = 0$.

The method is based on the following recurrence.

$$x_n \stackrel{def}{=} x_{n-1} - f(x_{n-1}) \times \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

Starting from two arbitrary values x_0 and x_1, it converges (slowly) to the solution.

Question 1. Write a Python script that implements the secant method.

Hint: As the method is purely numerically, it can only find approximate solutions. The method looks thus for an x such that $|f(x)| \leq \varepsilon$ for some small positive value ε. You will need a function that calculates the absolute value of a number. You can do it "by hand" or call a function of the module `math`. To be able to use it, you need to insert the line `import math` at the beginning of your script. Then, you can obtain the absolute value of any number n by writing `math.fabs(`n`)`.

Question 2. Apply your script to find the roots of the following functions.

$$\begin{aligned}
f(x) &= x^2 - 4x - 5 \\
f(x) &= \cos(x) + 2\sin(x) + x^2
\end{aligned}$$

Hint: Functions sine and cosine are defined in the module `math`. They are called as `math.sin(`n`)` and `math.cos(`n`)`.

Exercise 2.6 Simple Pendulum. A simple gravity pendulum is an idealized mathematical model of a real pendulum. It consists in a weight on the end of a mass-less cord suspended from a pivot, without friction. Since in this model there is no frictional energy loss, when given an initial displacement it will swing back and forth at a constant amplitude. The model is based on the following assumptions.

- The cord is mass-less, in-extensible and always remains tensed;
- The weight is a point mass;
- The motion is in two dimensions, i.e. the weight does not trace an ellipse but an arc;
- The motion does not lose energy to friction or air resistance;

– The gravitational field is uniform.
– The support does not move.

The differential equation representing the motion of a simple pendulum is as follows.

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0 \tag{2.1}$$

where g stands for the gravity, L for the length of the cord, and θ for the angular displacement.

We can simulate the movement of the pendulum by iterating on small dt's. Each step consists in performing the following calculations.

1. Calculate the acceleration as a function of the angle, i.e.
$$\ddot{\theta} = -\frac{g}{L}\sin(\theta)$$
2. Calculate the velocity, i.e.
$$\dot{\theta} = \ddot{\theta} \times dt$$
3. Update the angle, i.e.
$$\theta = \theta + \dot{\theta} \times dt$$
4. Update x and y coordinates of the weight, i.e.
$$x = L * \sin(\theta)$$
$$y = -L * \cos(\theta)$$

Question 1. Write a Python script to simulate the motion of a simple pendulum.

Hint: To get the value of π (that you will use to describe the initial angle of the pendulum), as well as the trigonometric functions sine and cosine, you need to use the package `math`. We shall discuss modules and packages in details in Chapter 5. For now, just consider a package as a set of constants and functions. To use the package `math`, we have to insert the following instruction at the beginning of your script.

```
1 import math
```

Then, the above constant and functions can be used as follows.

```
1 theta = (2/3)*math.pi    # initial angle
2 x = L*math.sin(theta)    # calculation of x
```

2.3 Functions

2.3.1 Description

Functions play a central role in nearly all programming languages. They are sequences of instructions that are given a name and that can be executed at different places in the program. Functions take in general one or more *arguments* and return a value. A function is *recursive* if it calls itself. We have already see some predefined functions, e.g. `print`, `str` and the functions of the module `math` such as `sqrt` (square root), `fabs` (absolute value), `sin` (sine) and `cos` (cosine). In this section, we shall see how to define our own functions.

As an illustration, assume we want, for some reason, to capitalize identifiers at different places in a larger program, i.e. to transform an identifier like `number_of_transactions` into `NumberOfTransactions`. Rather than to duplicate the code that does that, we can define a function that we shall call everywhere it is needed.

The code for this function could be as shown in Figure 2.3.

```
def CapitalizeIdentifier(sourceIdentifier):
  targetIdentifier = ""
  firstCharacterInWord = True
  for character in sourceIdentifier:
    if firstCharacterInWord:
      targetIdentifier += character.upper()
      firstCharacterInWord = False
    elif character=="_":
      firstCharacterInWord = True
    else:
      targetIdentifier += character
  return targetIdentifier
```

Figure 2.3: Python code that defines a function to capitalize identifiers

Function definitions start with the keyword `def`. The body of the function, i.e. its sequence of instructions, must be indented.

The code of the function `CapitalizeIdentifier` is rather easy to understand. It introduces however several novelties:

- The `for` loop that iterates (from left to right) on characters of the source string. As we shall see, such `for` loops are pervasive in Python.
- The += assignment. `x += y` is a shorthand way of writing `x = x + y`, here applied to strings.
- The construct `s.upper()`. If `s` is a character or a string (Python makes no difference), this returns the string `s` in which all lower case letters have been replaced by the corresponding upper case letter.

Once defined, we can call the function `CapitalizeIdentifier` by simply passing the string we want to capitalize as argument. E.g.

```
oldId = "number_of_transactions"
newId = CapitalizeIdentifier(oldId)
```

The above code sets the value of the variable `newId` to `"NumberOfTransactions"`.

2.3.2 Let's go!

The following series of exercises are about defining small functions, recursive or not.

Exercise 2.7 Palindrome. A palindrome is a phrase, i.e. a sequence of characters which reads the same backward as forward. No difference is made between upper and lower case letters Punctuation characters and spaces are ignored. E.g.

Madam
A man, a plan, a canal, Panama!
Was it a car or a cat I saw?

Question 1. Write a function that tests whether a given string is a palindrome.

Exercise 2.8 Thue-Morse Sequence. In mathematics, the Thue-Morse sequence is the binary sequence obtained by starting with 0 (or 1) and successively appending the Boolean complement of the sequence obtained thus far: `0`, `01`, `0110`, `01101001`...Note that we can apply the Thue-Morse transformation on any string made of 0's and 1's.

Question 1. Write a function that given a string of 0's and 1's, creates the next element of that string in a Thue-Morse sequence.

Question 2. Using the previous function, write another function that iterates k times the Thue-Morse transformation.

Question 3. Using these functions, calculate the 10th element of the Thue–Morse sequence, starting from "0".

Exercise 2.9 Grid Walk. Assume a robot moves on a two dimensional grid. At each step, it can go north, east, south or west.

Question 1. Design a function to make the robot visit each node of the grid once and only once.

Exercise 2.10 Factorial. For a given natural number n, factorial of n, denoted $n!$, is the product of all numbers from 1 to n. $n!$ can also be defined recursively as follows.

$$\begin{aligned} 0! &\stackrel{def}{=} 1 \\ 1! &\stackrel{def}{=} 1 \\ n! &\stackrel{def}{=} n \times (n-1)! \end{aligned}$$

Question 1. Write a recursive function that calculates $n!$ for any n.

Question 2. Write a function that uses a `while` loop to calculate $n!$ for any n.

Question 3. Write a function that uses a `for` loop to calculate $n!$ for any n.

Question 4. Using these functions to calculate 20!.

Exercise 2.11 Fibonacci. The Fibonacci series is a function from natural numbers to natural numbers defined by the following recurrence.

$$\begin{aligned} \text{Fibonacci}(0) &\stackrel{def}{=} 1 \\ \text{Fibonacci}(1) &\stackrel{def}{=} 1 \\ \text{Fibonacci}(n) &\stackrel{def}{=} \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) \qquad \text{for } n > 1 \end{aligned}$$

Question 1. Write a recursive function that calculates Fibonacci(n) for any n.

Question 2. Using this function, calculate Fibonacci(30), Fibonacci(32), Fibonacci(34) and Fibonacci(36). What do you observe?

Question 3. Write a function that uses a `while` loop to calculate Fibonacci(n) for any n.

Question 4. Using this second function, calculate Fibonacci(30), Fibonacci(32), Fibonacci(34) and Fibonacci(36). What do you observe?

Exercise 2.12 Binomial. Recall that the binomial of n and p, denoted $\binom{n}{p}$, is the number of way one can pick-up p elements in a set of n elements. $\binom{n}{p}$ is defined as follows.

$$\binom{n}{p} \stackrel{def}{=} \frac{n!}{p!(n-p)!}$$

Question 1. Write a function that calculates $\binom{n}{p}$ for any n and p using one of the functions to calculate $n!$ implemented for exercise 2.10.

Question 2. Use this function to calculate $\binom{10}{4}$.

$\binom{n}{p}$ can also be defined recursively as follows.

$$\binom{n}{p} \stackrel{def}{=} \begin{cases} 1 & \text{if } p \leq 0 \text{ or } p > n \\ \binom{n-1}{p-1} + \binom{n-1}{p} & \text{otherwise} \end{cases}$$

Question 3. Write a recursive function that calculates $\binom{n}{p}$ for any n and p.

Question 4. Use this second function to calculate $\binom{10}{4}$.

Exercise 2.13 Tower of Hanoi. The Tower of Hanoi is a puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape, see Figure 2.4.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.

Question 1. Write a function that describes a strategy to move a stack of n disks from rod `A` to rod `B`.
Hint: Think recursively!

Question 2. Apply your algorithm to the case where $n = 6$.

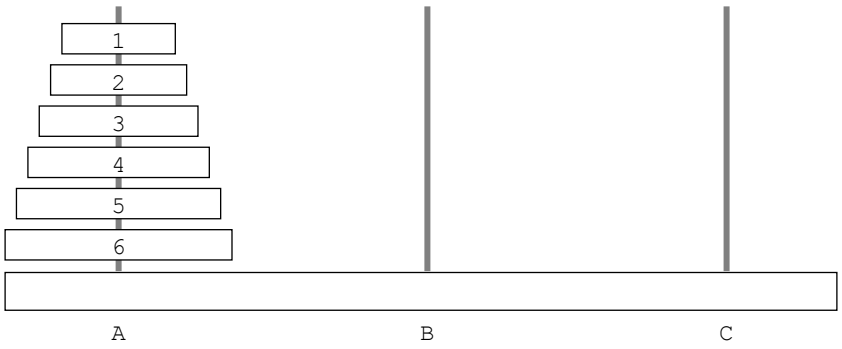


Figure 2.4: Tower of Hanoi

Problem 2.14 Logistic Map. The logistic map is one of the most fascinating mathematical functions ever. It is described by the following recursive equation.

$$x_{n+1} = r \times x_n \times (1 - x_n) \tag{2.2}$$

where $x_0 \in [0,1]$ and $r \in [0,4]$.

Once given the values of r and x_0, it defines the series $x_{(i)}$. As $x_n \times (1 - x_n)$ is always less than 0.25 if $0 \leq x_n \leq 1$, the condition on r ensures that $x_{n+1} \in [0,1]$.

The logistic map was popularized in a 1976 paper by the biologist Robert May as a very simplified model of the evolution of the number of individuals in a population. x_n represents the proportion at the nth generation of the number of individuals with respect to a maximum population determined by the environmental conditions. The logistic map is often cited as an archetypal example of how complex, chaotic behavior can arise from very simple non-linear dynamical equations.

The objective of this exercise is to study experimentally its behavior.

Question 1. Design a Python function that calculates the first n terms of the series for given values of r and x_0.

Question 2. Modify your function such that it prints out the last k terms.

Question 3. Using your function, design an experiment to test the hypothesis that, at least for values of r less than 2.8, the series converges towards the same limit value x_∞, no matter the value of x_0.

Question 4. Using your function, design an experiment to test what happens for values of r greater than 2.8.

Chapter 3

Containers

Key Concepts
- Tuples
- Lists
- Dictionaries
- Iterators

In the previous chapter, we saw how individual data such as numbers or strings can be stored into variables. In this chapter, we shall present containers. Containers are Python constructs in which an arbitrary large number of data can be stored. All programming languages provide some means to create containers. Python is specific in that it provides three ready-to-use containers: tuples, lists and dictionaries. We shall look at them in turn.

3.1 Tuples

3.1.1 Description

A *tuple* is an ordered and unchangeable collection of data. Tuples are themselves data, i.e. they can be stored into variables. Data of a tuple are surrounded with parentheses and separated with commas. E.g.

```
1 cardSuits = ("Spade", "Heart", "Diamond", "Club")
```

Once a tuple created, it is possible to access to its elements via their indices. Elements of a tuple are indexed from 0 upward. In the above example, the string `"Space"` is thus at index 0, the string `"Heart"` at index 1 and so on. To access the ith element of a tuple, it suffices to write the tuple, or more often the name of the variable in which the tuple is stored, followed by the index surrounded with square brackets. E.g. the following script prints the third elements of the tuple stored in the variable `cardSuits`.

```
1 print("The third suit: " + cardSuits[2])
```

It is possible to access elements of a tuple starting from the last index. In our example, the string `"Club"` is at index -1, the string "Diamond" at index -2 and so on. Attempting to access an element outside the scope of a tuple raises an exception, i.e. an error, which stops the execution of the script abruptly (at least if the exception is not conveniently caught).

It is possible to repeat data in a tuple. Moreover, a tuple can contain other tuples (and more generally any type of data).

A few operations (other than creation and access to individual elements) are available on tuples, see reference documentation for more details.

Because they are not modifiable once created, tuples are seldom used in concrete scripts. They are however sometimes useful, typically when writing a function that returns more than one result.

3.1.2 Let's go!

Exercise 3.1 Complex Numbers. Recall that a complex number is a pair of real numbers (a,b), written $a+bi$. a is called the real part and b is imaginary part. All arithmetic operations on real numbers can be lifted up to complex numbers, taking into account that, by definition, $i^2 = -1$.

In this exercise you shall implement functions to deal with complex numbers. Complex numbers will be encoded by means of Python tuples (in that case pairs).

Question 1. Write the following functions:

- `ComplexNew(a, b)` that creates and returns a complex number from two reals a and b, which are respectively its real and imaginary parts.
- `ComplexRealPart(c)` that returns the real part of a complex number.
- `ComplexImaginaryPart(c)` that returns the imaginary part of a complex number.

Question 2. Using the functions written in the previous question, write the following functions:

- `ComplexConjugate(c)` that returns the conjugate of the complex number given as argument (recall that the conjugate of $a+bi$, denoted as $\overline{a+bi}$, is $a-bi$).
- `ComplexAdd(c1, c2)` that returns the sum of the two complex numbers given as arguments.
- `ComplexSub(c1, c2)` that returns the difference of the two complex numbers given as arguments.
- `ComplexMul(c1, c2)` that returns the product of the two complex numbers given as arguments.
- `ComplexDiv(c1, c2)` that returns the quotient of the two complex numbers given as arguments.
- `ComplexPow(c, n)` that returns the complex number given as first argument raised to the power of the positive integer given as second argument.

Question 3. Write a function `ComplexPrint(c)` that prints out the complex number given as argument.

Question 4. Using the functions defined in the previous questions, calculate the values of the following expressions.

$$
\begin{aligned}
c_1 &= \frac{i-4}{2i-3} \\
c_2 &= \frac{1+i}{1-i} - (1+2i)(2+2i) + \frac{3-i}{1+i} \\
c_3 &= 2i(i-1) + \left(\overline{\sqrt{3}+i}\right)^3 + (1+i)\overline{(1+i)}
\end{aligned}
$$

3.2 Lists

3.2.1 Description

A *list* is an ordered and modifiable collection of data. Lists are the most basic and widely used containers. There are used in virtually all scripts.

Data stored in the list do not need to be of the same type. It is however strongly recommended to always store data of the same type in a list.

There are several ways to create lists. The simplest one consists in declaring all members of the list. It works as follows.

```
fruits = ["Apple", "Banana", "Cherry", "Durian"]
```

It is then possible to append elements to the list using the `append` method:

```
fruits.append("Elderberry")
```

The empty list is denoted `[]`.

`mylist[i]` returns ith element of the list, starting from 0. `fruits[0]` returns `"Apple"`, `fruits[1]` returns `"Banana"` and so on. It is also possible to use negative indices to address elements of a list. `mylist[-1]` returns the last element of the list, `mylist[-2]` returns the element before the last element and so on. `fruits[-1]` returns thus `"Elderberry"`, `fruits[-2]` returns `"Durian"` and so on.

Be careful: using an index outside the range of the list raises an exception. If this exception is not caught, it aborts the execution of the script.

The number of elements of a list is obtained with the function `len(myList)`.

Hence `len(fruits)` returns 5.

The function `del(myList[i])` removes the ith element of the list, starting from 0. Hence, `del(fruits[2])` removes `"Cherry"` from the list, which becomes `["Apple", "Banana", "Durian", "Elderberry"]`.

The `myObject in myList` construct tests whether the object `myObject` is a member of the list `myList`. It returns a Boolean value. Hence `"Durian" in fruits` returns `True` while `"Cherry" in fruits` returns `False` (as we just removed `"Cherry"` from the list).

The key operation on lists consists in applying some procedure successively on each member of the list. This can be done in several ways. The first two ways consist in iterating on indices, e.g.

```
index = 0
while index<len(fruits):
  print(fruits[index])
  index += 1
```

Or simpler:

```
index = 0
for index in range(0, len(fruits)):
  print(fruits[index])
```

Python provides however the general concept of *iterator* which makes it possible to iterate on each member of a collection (in that case a list) without explicitly referring to the way items of this collection can be individually accessed. It works simply as follows.

```
for fruit in fruits:
  print(fruit)
```

Python provides a lot functionalities to perform operations on lists and their elements. In what follows, we shall only review a few.

list.**extend**(otherList) This appends the elements of `otherList` to the list `list`.

list[i:j] This returns a list made of elements from index i to index j (non included) of the list `list`. E.g. `fruits[1:3]` creates a list `["Banana", "Durian"]`. `fruits[1:3]` is called a *slice* of the list `fruit` in the Python jargon.
If the first index is omitted, the copy starts from the first element. If the second one is omitted, the copy goes til the last element. Therefore, `fruits[:]` makes a full copy of the list.

list.**`sort`**(*key=None*, *reverse=False*) This sorts the elements of the list `list` in place (the list itself is modified). The argument are optional. They are used to customize the sort.
E.g. `fruits.sort(reverse=True)` transforms the list `fruits` into:
`["Elderberry", "Durian", "Banana", "Apple"]`.

3.2.2 Let's go!

Exercise 3.2 List Aggregators. In this exercise, we shall define functions to aggregate the values of stored in a list of floating point numbers.

Question 1. Write a Python function that calculate the sum of the elements of a list.

Question 2. Write a Python function that calculate the product of the elements of a list.

Question 3. Write a Python function that calculate the minimum of the elements of a list.

Question 4. Write a Python function that calculate the maximum of the elements of a list.

Question 5. Write a Python function that calculate the mean of the elements of a list.

Question 6. Test the above functions with the list made of the 10 first integers shuffled at random.
Hint: To shuffle the elements of a list at random, use the function `shuffle` of the package `random`.

Exercise 3.3 List Selectors. In this exercise, we shall define functions to select some of the values of stored in a list of integers.

Question 1. Write a Python function that takes a list of integers as argument and returns a tuple made of the two lists of respectively its odd and even members.

Question 2. Write a Python function that takes a list of integers and a number k as arguments and returns a tuple made of the two lists such that:
- The first list contains the elements of indices k, $2k$, $3k\ldots$
- The second list contains the other elements.

Question 3. Write a Python function that takes a list of integers and a number k as arguments and returns the list of the k highest values found in the list given as argument.

Question 4. Test the above functions with the list made of the 10 first integers shuffled at random.
Hint: To shuffle the elements of a list at random, use the function `shuffle` of the package `random`.

Exercise 3.4 Sieve of Eratosthenes. The sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a given limit n. The idea is simple:

- First, one creates the list of all integers from 2 to n.
- Then, one considers one by one the elements of this list. Considering the integer i consists in removing from the list all multiples of i, $2 \times i$, $3 \times i$ and so on until $k \times i > n$.

Question 1. Write a Python script that implements the sieve of Eratosthenes.

Question 2. Apply your script for $n = 100$.

Exercise 3.5 Removing Duplicated Elements. It is often the case that we want to get rid of duplicated elements in a list.

Question 1. The function `randint` of the module `random` makes it possible to generate uniformly at random an integer comprised between two bounds l and h. Use that function to create a list of $n = 50$ integers comprised between $l = 1$ and $h = 20$.

Question 2. Complete your script with the instructions to remove duplicated elements from the list.
Hint: The simplest solution may be to create a second list with non-duplicated elements.

Exercise 3.6 Quicksort. Python provides the built-in function `sort` that sorts the elements of a list. We shall however implement a sort function by ourselves, for the sake of the pedagogy. The algorithm we shall implement is called "quicksort".

Question 1. Write a function that given a list of numbers L splits L into two sublists: the sublist of elements that are smaller than the first element of the list and the sublist of elements that are bigger than the first element of the list (the first element of the list is included in neither of the two sublists).

Question 2. Use the function you wrote in the previous question to create a recursive function that sorts a list.

Question 3. Test your function on randomly created lists.

Exercise 3.7 Quantiles. Assume we are given a list of n values taken by an indicator while observing some phenomenon. Each value is a floating point number. We want to calculate the quantiles for the indicator, i.e. to split the list in k buckets of approximately equal sizes such that the first bucket contains the n/k lowest values, the second bucket contains the n/k next values, and so on until the kth bucket that contains the n/k highest values. Then, we can calculate, the lowest, the highest and the mean value of each bucket.

Question 1. The function `triangular` of the module `random` makes it possible to generate floating points at random according to a triangular distribution. It takes three parameters: the lower bound l, the upper bound h and a mode $l \leq m \leq h$. Use that function to create a list of $n = 1000$ floating

point numbers with $l = 0$, $h = 1$ and $m = 0.6$.

Question 2. Split the list into $k = 20$ buckets of approximately equal size.
Hint: Encode the buckets by means of list and the maintain a list of buckets.
Hint: Take n/k as the maximum size of the bucket. The ith bucket should then contain all elements of the list whose index is comprised between $(i-1) \times n/k$ and $i \times n/k$.

Question 3. For each bucket, calculate the lowest, the highest and the mean value. Print these three values together with index of the bucket and the number of elements in the bucket. Use one line per bucket and separate the value with tabs (`"\t"`) characters.

Exercise 3.8 Reliability of k-out-of-n Systems. We consider a system with n elements $e_1, e_2, \ldots, e_n$ working in parallel. Each e_i may fail independently with a probability p_i. The system as a whole is failed if at least k elements are failed. We are given the list of the p_i's and we want to determine the probability that the system is failed.

As there are n elements and each of them can be in two states (working or failed), there are in total 2^n possible configurations of the system. Enumerating all these configurations to seek for those in which the system is failed would take much to much time as n gets big.

Fortunately, a much better algorithm exists.

The first step in designing this algorithm is to establish the following recurrence, which is very close to what we used to calculate binomials (exercise 2.12).

$$p(k,[p_1,\ldots,p_n]) = \begin{cases} 1 & \text{if } k \leq 0 \\ 0 & \text{if } k > n \\ p_n \times p(k-1,[p_1,\ldots,p_{n-1}]) + (1-p_n) \times p(k,[p_1,\ldots,p_{n-1}]) & \text{otherwise} \end{cases}$$

Question 1. Write a function `CalculateProbability` that calculates the probability of failure of a k-out-of-n system. Your function should take three arguments: the list of n probabilities, n and k. It should implements the above recurrence.

Question 2. Write a function that creates a list of n probabilities generated uniformly at random between two bounds l and h (use the function `uniform` of the package `random` to do so).
Use this function to generate three lists of probabilities drawn in $[0.1, 0.2]$, with respectively 20, 24 and 28 elements.

Question 3. Apply the function `CalculateProbability` to each of the three lists generated in the previous question for $k = 10$.
What do you observe?

The increasing calculation time is due to the fact that each call of the function, but in terminal case, generates two recursive calls. Consequently, even if the algorithm is better than a brute force enumeration of all configurations, it is still very costly.

Now, we can observe that calling the function with parameters $\langle n,k \rangle$ generates one call with parameters $\langle n-2,k-2 \rangle$, two calls with parameters $\langle n-2,k-1 \rangle$, and one call with parameters $\langle n-2,k \rangle$.

An idea is thus to memorize the result of the call with parameters $\langle n-2,k-1 \rangle$ the first time we calculate it. Then, we just have to pick-up the result the second time we need it. This technique is called *caching*.

Question 4. Write a function `CalculateProbabilityWithCache` that calculates the probability of failure of a k-out-of-n system. Your function should take four arguments: the list of n probabilities, n, k and a cache. The cache should be implemented by means of a dictionary. Keys of that dictionary are pairs (n,k) and values are the calculated probabilities.

Question 5. Apply the function `CalculateProbabilityWithCache` to each of the three lists generated question Question 2. for $k = 10$.
What do you observe?

Problem 3.9 Logistic Map, take 2. We shall now go back to Problem 2.14 so to redo experiments at a larger scale, thanks to some additional tooling.

Recall that the logistic map is a series of real numbers in $[0,1]$ defined by the following recursive equation.

$$x_{n+1} = r \times x_n \times (1 - x_n)$$

where $x_0 \in [0,1]$ and $r \in [0,4]$.

For each of the following questions, design the functions that are required and test them on different sets of values.

Question 1. Design a function that returns the list of pairs (i,x_i) for the first n elements of the series, for given values of r, x_0 and n.
Design a function that prints out the pairs of the list (one pair per line).

Question 2. Modify your function so that returns the list of pairs $(n-k+1,x_{n-k+1})$, $(n-k+2,x_{n-k+2})$, $\ldots(n,x_n)$, for given values of r, x_0, n and k. Use for that a sliding window of k pairs.

Question 3. Design a function that given a list of pairs (i_1,x_1), $\ldots(i_k,x_k)$ and a value x tests if one of the x_i's is such that $|x_i - x| < \varepsilon$, for a certain value of ε.
Modify the function designed in the previous section so that it stops iterating if your test function returns `True`.

Question 4. Design a function that prints out the values i, x_i, and $|x_i - x|$ for each pair (i,x_i) of the list, where x is the last value of the list.

Question 5. Use the above functions to study the behavior of the logistic map.

3.3 Dictionaries

3.3.1 Description

Dictionaries are like lists, except that they are indexed by objects, typically strings, and not by integers.

Assume, for instance, that we want to manage the numbers of kilos of each type of fruits we have in our grocery store. We can use dictionary can use a dictionary to do so, e.g.

```
1 fruitCounts = dict()
2 fruitCounts["Apple"] = 12
3 fruitCounts["Banana"] = 33
4 fruitCounts["Cherry"] = 0
```

The function `dict()` returns an empty dictionary.

`fruits["Apple"] = 12` adds an entry for the *key* `"Apple"` to the dictionary and associates this key with `value` 12. It the dictionary had already an entry for the key `"Apple"`, the previous value of this entry is forgotten.

As for lists, dictionaries can contain values of different types, although it is highly recommended to that all values have the same type.

`myDict[myKey]` returns the value of the entry of key `myKey` of the dictionary `myDict`. If the dictionary does not contain an entry for this key, an exception is raised, just as when one attempts to refer an element whose index is outside the range of a list. This is the reason why it is much preferable to use the method `myDict.get(myKey, defaultValue)` to access the value associated with `myKey`. Usually, the special constant `None` is used as the default value. E.g. `fruitCounts.get("Banana", None)` returns 33, while `fruitCounts.get("Durian", None)` returns `None`. In this case, it would be also reasonable to take 0 as a default value, making no distinction between the cases where the dictionary contains an entry associated with the value 0 (as for `"Cherry"`) and the case it contains no entry for the key (as for `"Durian"`).

As for lists, the function `del(myDict[myKey])` remove the entry `myKey` from the dictionary `myDict`. If the dictionary does not contain an entry for the given key, an exception is raised.

It is possible to iterate over the entries (the keys) of a dictionary in a similar way as for lists. E.g.

```
1 for fruitName in fruitCounts:
2   numberOfKilos = fruitCounts[fruitName]
3   print(fruitName + "\t" + str(numberOfKilos))
```

Note that entries are stored in the order they have been created (and not, for instance, in lexicographic order). To get entries sorted, one must use the `sorted` function that applies on iterators. E.g.

```
1 for fruitName in sorted(fruitCounts):
2   numberOfKilos = fruitCounts[fruitName]
3   print(fruitName + "\t" + str(numberOfKilos))
```

3.3.2 Let's go!

Exercise 3.10 Roll a Dice. In this exercise, we shall roll a virtual dice a number of times and count how many times we get each face.

Question 1. Write a Python function that selects n times at random a face, i.e. an integer comprised between 1 and 6 and that stores in a dictionary the numbers of occurrences of each face.
Hint: Use the function `randrange` of the module `random` to draw at random the faces.

Question 2. Write a function that prints out the frequencies of each face.

Question 3. Perform experiments with $n = 100, 1000, 10000, 1000000$. What do you observe?

Exercise 3.11 Character Frequencies. In this exercise, we shall define a set of functions to calculate the frequencies of characters into a file.

Question 1. Write a Python function that reads a file line by line and then each character of a line. Use a dictionary to store, for each character, the number of occurrences of this character.
Hint: Use the function `rstrip` of strings to remove the end-of-line characters.

Hint: Use the function `lower` of strings to ensure that lower case and upper case occurrences of characters are counted together.

Question 2. Write a Python function that prints out the characters and their number of occurrences stored in the dictionary in decreasing order (one character and its number of occurrences per line).

Question 3. Apply these two functions on the files `Lorem Ipsum.txt` and `Universal Declaration of Human Rights.txt`.

Problem 3.12 Train Regularities. This problem relies on the case study "Train Regularity" described in Appendix A.5. The file `regularite.csv` contains data about regularities of trains circulating on the French railway network at different dates. The objective of this problem is to calculate the mean regularity of the trains on the dataset.

We shall need first two auxiliary functions.

Question 1. Write a Python function `CalculateMean(values)` that calculates the mean value of the elements of a list `values` of floating point numbers.

Question 2. We will need to manage a dictionary whose keys are the trains and whose values are the lists of observed regularities at different dates. Write a Python function `AddValueToDictionary( dictionary, key, value)` that:

- Creates a list containing only the value `value` and insert this list in the dictionary `dictionary` (with the key `key`) if the dictionary does not contain an entry for the key `key`.
- Appends the value `value` to the list `dictionary[key]` otherwise.

We shall now parse the file `regularite.csv` and load a dictionary with the observed regularities.

Question 3. Write a Python function `ImportFile(fileName, trains)` that reads the file `fileName` (which is assumed to be at the same format as `regularite.csv`) and loads regularities into the dictionary `trains`.
Hint: To simplify your task, you can adjust the parse function `ImportFile(fileName, trains)` provided in the file `TrainRegularityParser.py`.
Hint: What characterizes a train is not only its line, but also its category. Keys of your dictionary should thus be pairs (tuples) `(category, line)`.

Question 4. Write a Python function `ExportRegularities(trains, fileName)` that exports in the file `fileName` the list of observed regularities of each train.

Question 5. Write a Python function `ExportMeanRegularities(trains, fileName)` that exports in the file `fileName` the mean regularity of each train.

Chapter 4

Files and Regular Expressions

Key Concepts
- Text files
- String format
- Regular expressions
- Matching
- Substitutions

4.1 Files

4.1.1 Description

So far, we considered only data stored in the computer memory. These data are created while executing the program... and lost once the execution is terminated. In many practical applications however, we want to be able to save and retrieve data, to make them *persistent*.

A very common way to achiever this goal is to save the data into *text files*. Text files are typically stored onto external devices such as hard disks or USB sticks. Python can read and write all sorts of files, but the simplest are text files. Python scripts are stored into text files.

Text files can be opened essentially in three modes: read, write and append.

- In the read mode, the file can only be read. The reading is sequential, i.e. is it perform character after character (or line by line), from the very first character of the file to the very last, without the possibility to go backward. To be opened in read mode, a file must already exist.
- In the write mode, the file can only be written. The writing is sequential, i.e. that characters are added at the end of file one by one, without the possibility to go backward. Opening a file in write mode creates it if it did not exist, and erases its content if it did exist.
- The append mode is essentially the same as the write mode except that if the previous content of the file is kept if the file was already existing (new characters are appended after this previous content).

The function to open a text file is `open(`*`fileName`*`,`*`mode`*`)` where *`fileName`* is the name of the file on the hard disk and *`mode`* is either `"r"` for read, `"w"` for write or `"a"` for append.

Opening a file may fail, e.g. if the file does not exist when opening it in read mode or if it already opened by another program in write and append modes. When a file is attempted to be opened and the opening fails, Python raises an *exception*. We shall present exceptions latter in this document. For now, the only thing to know is that if an exception is raised and is not caught, the execution of the program is aborted abruptly (with an error message), something we want to avoid at all costs. The instruction `try...except...` makes it possible to catch exceptions.

To illustrate all that, assume that we want to create a function `FileAppend` that appends the content of a source file to a target file (and creates this target file if it did not exist). Assume moreover that we

want that this function takes the two file names as arguments and returns 0 if the operation is successful and 1 otherwise. The code for this function could be as shown in Figure 4.1

```
def FileAppend(sourceFileName, targetFileName):
  try:
    sourceFile = open(sourceFileName, "r")
  except:
    print("Unable to open file " + sourceFileName)
    return 1
  try:
    targetFile = open(targetFileName, "a")
  except:
    print("Unable to open file " + targetFileName)
    return 1
  for line in sourceFile:
    targetFile.write(line)
  targetFile.flush()
  targetFile.close()
  sourceFile.close()
  return 0
```

Figure 4.1: Python code that defines a function to append the content of a source file to a target file

The `try...except...` instruction works as follows: the instructions under the `try` branch are executed. In the above code, the source file is thus attempted to open in read mode. If no exception is raised when executing these instructions, Python just ignores the `except` branch. Otherwise, it executes instructions of this branch.

The core of the function `FileAppend` is the `for` loop that reads line by line the source file and write these lines, one by one, onto the target file.

The instruction *`file`*`.close()` closes a file that has been previously opened. It is very important to close all opened files. Forgetting to do so can mess up things.

The instruction `targetFile.flush()` ensures that all lines are actually written into the target file before closing it.

The code below calls twice the function `FileAppend` onto the file `MyFile.txt`.

```
FileAppend("MyFile.txt", "MyFileTwice.txt")
FileAppend("MyFile.txt", "MyFileTwice.txt")
```

The first call copies the file `MyFile.txt` onto the file `MyFileTwice.txt` (assuming this file did not already exist). The second one appends a second copy of content the file `MyFile.txt` to the file `MyFileTwice.txt`, which now exists.

It is not worth to enter into more technical details for now. But any script reading and writing information onto external files should work more or less like our function `FileAppend`:

- Files must be opened either in read, write or append modes using the `try...except...` instruction.
- Files opened in read mode are preferably read line by line using a `for` loop.
- Files opened in write or append modes must be flushed before being closed.
- All opened files must be closed.

4.1.2 Let's go!

The following series of exercises are about dealing with text files. We shall use text files `"Lorem Ipsum.txt"` and `"Universal Declaration of Human Rights.txt"`, i.e. case studies "1"

and "2" presented in Appendix A.1.

Exercise 4.1 File Reformatting. This exercise aims at implementing some basic file formatting tools.

Question 1. Write a function that opens a source file in read mode, reads in line by line and copies it into a target file opened in write mode. The function shall take the names of the source and target file as parameters.
Test your function on the text files "`Lorem Ipsum.txt`" and "`Universal Declaration of Human Rights.txt`".

Question 2. Same question, but with the additional constraint that in the target file, there must be at most 80 characters on each line. If a line in the source file exceeds 80 characters, it must be broken into several lines of at most 80 characters each.
Test your function on the text files "`Lorem Ipsum.txt`" and "`Universal Declaration of Human Rights.txt`".
Hint: Use two nested loops: the outer one to read the file line by line, the inner one to read each line character by character.

Question 3. Same question, but with the additional constraint that words should not be split. Moreover, a line should not start with spaces.
Hint: You can use two indices to traverse the current line: one that indicates the starting position of the line you shall print, and one to indicate the current position. If the current index is 80 characters ahead the start index, the characters between the two indices are printed out into the target file. However, if the character pointed by the current index is not a space, the current index is moved back until a space is found.
Test your function on the text files "`Lorem Ipsum.txt`" and "`Universal Declaration of Human Rights.txt`".

Exercise 4.2 File Counts. This exercise consists in counting the number of characters, words, sentences and paragraphs in a text file.

Question 1. Write a function that takes the name of a file as argument and prints out the number of characters, words, sentences and paragraphs in that file. End of line characters ('`\n`') should not be counted.
Hint: Use two nested loops: the outer one to read the file line by line, the inner one to read each line character by character. Determine the situations in which a new word, a new sentence and a new paragraph are started or completed. To determine whether a character is a letter, you can use the method `chr.isalpha()`.

Question 2. Apply your function to the text files "`Lorem Ipsum.txt`" and "`Universal Declaration of Human Rights.txt`".

Exercise 4.3 Capitalization of Identifiers. This exercise consists in applying the function `CapitalizeIdentifier` defined Section 4.1.1 to all identifiers of a Python program.

Question 1. Write a function that takes the name of a source file and a target file as arguments and

copies the source file into the target file capitalizing identifiers.
Hint: Use two nested loops: the outer one to read the file line by line, the inner one to read each line character by character. Determine the situations in which a new identifier is started or completed. To determine whether a character is a letter, you can use the method `chr.isalpha()`, to determine whether it is a letter or a digit you can use the method `chr.isalnum()`.
Do not forget comments (that should not be modified) and more importantly Python keywords (that must be copied verbatim).

4.1.3 Formatting outputs

So far, we used the functions `print`, *`file`*`.write`, `str` and the string concatenation + to print out results. Python offers a wide range of more powerful and more convenient methods to format outputs.

The basic idea is build the actual output string from a template string using the `format` method. Let us see how it works on some examples.

```
>>> print("The capital of {} is {}.".format("Norway", "Oslo"))
The capital of Norway (*@\texttt{is}@*) Oslo.
```

Applying `format` substitutes `Norway` for the first `{}` and `Oslo` for the second one.

Up to now, nothing particularly fancy. It gets more interesting if we have to use several times an argument. This can be done either by referring to arguments via their indices. In addition, it is possible to format numbers in many different ways, as illustrated by the code below.

```
>>> template = """The capital of {0} is {1}.
The metropolis of {1} counts {2:1.2f} millions inhabitants."""
>>> print(template.format("Norway", "Oslo", 1588457/10**6))
The capital of Norway (*@\texttt{is}@*) Oslo.
The metropolis of Olso counts 1.59 millions inhabitants.
>>> print(template.format("Suede", "Stockholm", 2352549/10**6))
The capital of Suede (*@\texttt{is}@*) Oslo.
The metropolis of Stockholm counts 2.35 millions inhabitants.
```

Arguments can be also given names. E.g.

```
>>> template = """The capital of {country} is {capital}.
The metropolis of {capital} counts {inhabitants:1.2f} millions inhabitants."""
>>> print(template.format(country="Denmark", capital="Copenhagen", \\
inhabitants=2057737/10**6))
The capital of Denmark (*@\texttt{is}@*) Copenhagen.
The metropolis of Copenhagen counts 2.06 millions inhabitants.
```

Exercise 4.4 Component Unreliability. A system involves components of several types, e.g. pumps, valves, pipes... The probability of failure at time t $U_C(t)$ of each of a component C is described by a negative exponential distribution as follows.

$$U_C(t) = 1 - e^{-\lambda_C \times t} \tag{4.1}$$

where λ_C is the hourly failure rate of the component.

We write into a file the value of $U_C(t)$ for different components and different mission times. Each

line of the file shall describe one component:
- First, its name encoded onto 6 characters. The name should be left justified.
- Then, its hourly failure rate written in scientific notation, with two digits after the decimal point, e.g. $1.23e-6$.
- Then, the values of $U_C(t)$, for $t = 1000$, 2000 and 10000, which should be written as the failure rate. Numerical values must be separated by one space.

Question 1. Write a function that given the code of a component and its failure rate, returns the corresponding line (a string) of the file.
Hint: To calculate the exponential of a number, you need the predefined function `exp` of the module `math`. The why and wherefore of Python modules and packages will be explained Chapter 5. For now, you just have to do two things:
First, insert the following line, at the beginning of your script.

```
import math
```

Second, call the function `math.exp` to calculate the exponential, e.g.

```
z = 1 - math.exp(-failureRate*t)
```

Hint: Do not try to remember all format commands. Just look at Python documentation when you need them.

Question 2. Apply you template to print out into a file `Unreliabilities.txt` the probabilities of failures, for the following components.
- Shutdown valves, code: `SDV`, hourly failure rate 0.00000123.
- Solenoid valves, code: `SV`, hourly failure rate 0.00000345.
- Electric pumps, code: `EPMP`, hourly failure rate 0.0000789.

4.2 What are regular expressions?

A regular expression, regex or regexp, is a sequence of characters that define a search pattern. Usually such patterns are used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. Regular expressions have been developed by theoretical computer scientists in the context of formal language theory.

Python has a built-in package called `re`, which can be used to work with regular expressions. Another, third party, package called `regex` is also available. `regex` is compatible with `re` and offers a few additional functionalities. We shall work however with `re`, which is sufficient for most of the applications. We shall see more about modules and packages Chapter 5. For now, just consider a package as a set of functions. To use the functions defined in the package `re`, you have to import it, i.e. to insert the following instruction at the beginning of your script.

```
1 import re
```

Most of the functions implemented by `re` take two arguments: the pattern to be search for and the string in which this pattern is searched.

4.2.1 Patterns

Patterns are strings that are interpreted in a special way. In Python, it is highly recommended to prefix these patterns with the character '`r`'. The reason is that a string prefixed with '`r`' is interpreted as a *raw*,

i.e. it passes through backslashes without change. This is very useful as patterns make an extensive use of backslashes.

Assume that we want to look for the sequence of three letters `bla` into a text. Then the corresponding pattern is simply the string `r"bla"`. Up to now, nothing fancy.

So-called meta-characters can however be used to describe much more interesting patterns. They are as follows.

`[...]` A character in a set, e.g. `[bla]` matches either `b`, `l` or `a`. The set of alphabetic letters is matched by `[a-z]` for lower case letters and `[A-Z]` for upper case letters. The set of alphabetic digits is matched by `[0-9]`.

`[^...]` The complementary of character in a set, e.g. `[^"]` matches all characters but the double quote '"'.

`.` A dot matches any character, the newline character.

`^` The string must start with the pattern that follows the caret.

`$` The string must end with the pattern that follows the dollar sign.

`*` Any number of occurrences of the pattern that precedes the star sign. E.g. the pattern `[0-9]*` matches any number of digits, including 0.

`+` Any number of occurrences of the pattern that precedes the plus sign. E.g. the pattern `[0-9]+` matches any positive number of digits.

`?` 0 or 1 occurrence of the pattern that precedes the question mark. E.g. the pattern `[0-9]?` matches 0 or digits.

`{n}` *n* occurrences of the pattern that precedes the braces. E.g. the pattern `[0-9]{3}` matches any sequence of 3 digits.

`{m,n}` between *m* and *n* occurrences of the pattern that precedes the braces. E.g. the pattern `[0-9]{3,6}` matches any sequence of 3 to 6 digits.

`|` `|` denotes an alternative, e.g. the pattern `bla|bli` matches either the text `bla` or the text `bli`.

`(.)` The parentheses are used to memorize the string that matches the sub-pattern given between the parentheses.

The `re` package of Python provides more patterns, but those listed above are in general fully sufficient for common operations on texts.

4.2.2 Special Sequences

The character '\' makes it possible to escape metacharacters, e.g. the pattern `\(` matches the character '('.

Beyond meta-characters, the character '\' makes it possible to define shortcuts. Here follows a non exhaustive list (see the documentation for a complete list.).

`\d` Matches any digit. This special sequence is thus equivalent to `[0-9]`.

`\s` Matches whitespace characters, which includes `[ \t\n\r\f\v]`.

`\w` Matches characters considered alphanumeric in the ASCII character set. This special sequence is thus equivalent to `[a-zA-Z0-9_]`.

`\n` Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, `(.+)-\1` matches `'bla-ba'` or `'bicho-bicho'`.

`\b` Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. '`\bfoo\b`' matches '`foo`', '`foo.`', '`(foo)`', '`bar foo baz`' but not '`foobar`' or '`foo3`'.

Here follows a sample of patterns.

– Integers:

```
r"(+|-)?[0-9]+".
```

– Python identifiers:

```
r"[a-zA-Z_][a-zA-Z0-9_]*".
```

– Strings surrounded by double quotes:

```
r'"([^\"\\]|\\.)*"'.
```

– Email addresses:

```
r"[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]2,".
```

– HTML tags:

```
r"<([a-zA-Z][a-xA-Z0-9]*)\b[^>]*>(.*?)</\1>".
```

– IP addresses:

```
r"\b\d{1,3}\.d{1,3}\.d{1,3}\.d{1,3}\b"
```

– Visa card numbers:

```
r"[0-9]{4}([ ]?[0-9]{4}){3}".
```

– Valid dates in the dd-mm-yyyy format:

```
r"(0[1-9]|[12][0-9]|3[01])[- /.](0[1-9]|1[012])[- /.](19|20)\d\d"
```

4.3 The `re` Package

4.3.1 Main Functions

`re` implements, among others, the following functions.

`search`(*`pattern, string`*)
Scan through *`string`* looking for the first location where the regular expression *`pattern`* produces a match, and return a corresponding match object (see next section). Return `None` if no position in the string matches the pattern. E.g.

```
import re

sentence = """When beetles battle beetles in a puddle paddle battle and
the beetle battle puddle is a puddle in a bottle, they call this
a tweetle beetle bottle puddle paddle battle muddle."""

result = re.search(r"\w*u\w*", sentence)
if result:
  print("I found " + result.group() + " at " + str(result.start()))
else:
  print("I cannot find that")

>>> I found puddle at 33
```

`match`(*`pattern, string`*)
If zero or more characters at the beginning of *`string`* match the regular expression *`pattern`*, return a corresponding match object. Return `None` if the string does not match the pattern; note that this is different from a zero-length match. E.g.

```
result = re.match(r"W\w*", sentence)
if result:
  print("Yep, the sentence starts with " + result.group())
else:
  print("Nope")

>>> Yep, the sentence starts with When
```

split(*pattern*, *string*, *maxsplit*=0)

Split *string* by the occurrences of *pattern*, i.e. return the list of sequences of characters separated by the occurrences of *pattern*. If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits occur, and the remainder of the string is returned as the final element of the list. E.g.

```
words = re.split(r"[\s.]+", sentence)
print(words)

>>> ['When', 'beetles', 'battle', 'beetles', 'in', 'a', 'puddle', 'paddle',
'battle', 'and', 'the', 'beetle', 'battle', 'puddle', 'is', 'a', 'puddle',
'in', 'a', 'bottle,', 'they', 'call', 'this', 'a', 'tweetle', 'beetle',
'bottle', 'puddle', 'paddle', 'battle', 'muddle', '']
```

findall(*pattern*, *string*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result. E.g.

```
words = re.findall(r"b\w+", sentence)
print(words)

>>> ['beetles', 'battle', 'beetles', 'battle', 'beetle', 'battle', 'bottle',
'beetle', 'bottle', 'battle']
```

sub(*pattern*, *replacement*, *string*, *count*=0)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by *replacement*. If the pattern is not found, *string* is returned unchanged. *replacement* can be a string or a function; if it is a string, any backslash escapes in it are processed, e.g. `\n` is converted to a single newline character. E.g.

```
newSentence = re.sub(r"bottle", "big bottle", sentence)
print(newSentence)

>>> When beetles battle beetles in a puddle paddle battle and
the beetle battle puddle is a puddle in a big bottle, they call this
a tweetle beetle big bottle puddle paddle battle muddle.
```

4.3.2 Match Objects

Match objects always have a Boolean value of `True`. Since functions `match` and `search` return None when there is no match, you can test whether there was a match with a simple if statement:

```
result = re.search(pattern, string)
if result:
    # do something with result
```

Match objects support the following methods:
The above list is non-exhaustive.

`Match.`**`group`**`(`*`number1, ...`*`)`
Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, *number1* is set by default to zero and the whole match is returned. If a group argument is zero, the corresponding return value is the entire matching string; If a group is contained in a part of the pattern that did not match, the corresponding result is `None`. If a group is contained in a part of the pattern that matched multiple times, the last match is returned. E.g.

```
import re

sentence = """When beetles battle beetles in a puddle paddle battle and
the beetle battle puddle is a puddle in a bottle, they call this
a tweetle beetle bottle puddle paddle battle muddle."""

result = re.search("r(\w*u\w*)|(\w*tt\w*)", sentence)
if result:
  if result.group(1):
    print("I found " + result.group(1))
  if result.group(2):
    print("I found " + result.group(2))
else:
  print("I found none of them")

result = re.search(r"[^ dt]*([dt])\1[^ dt]* [^ dt]*([dt])\2[^ dt]*", sentence)
if result:
  print("I found '" + result.group(0) + "' with:")
  for n in range(1, 3):
    if result.group(n):
      print("group " + str(n) + " = " + result.group(n))
else:
  print("I found none of them")

>>> I found battle
>>> I found 'puddle paddle' with:
group 1 = d
group 2 = d
```

`Match.`**`start`**`(`*`number`*`=0)`
Return the index of the start of the sub-string matched by the group *number*.
`Match.`**`end`**`(`*`number`*`=0)`
Return the index of the end of the sub-string matched by the group *number*.

4.4 Checking and Counting

4.4.1 Rational

Regular expressions are useful in many situations. A first category of applications is when one has to check that a file is a the correct format or to count the number of occurrences of some items in a file.

The idea is then to scan line by line the file (we saw how to do that Section 4.1) and to apply the suitable operation(s) and pattern(s) onto each line.

Recall that it is almost always a good idea to trims line, i.e. to remove the end-of-line character at the end of it, as this character may depend on the operating system.

The general pattern of a function checking that a file is at the correct format (if this check can be done on a line by line basis) is as follows.

```
import sys
import re

def CheckFile(fileName):
  # returns 0 if and only if the file exists and matches the format
  try:
    file = open(fileName, "r")
  except:
    sys.stderr.write("Unable to open file " + fileName + "\n")
    sys.stderr.flush()
    return 1
  error = 0
  lineNumber = 0
  for line in file:
    lineNumber += 1
    line = line.rstrip() # trims the line
    if not re.match(r"here your pattern", line):
      error += 1
      sys.stderr.write("Error at line " + str(lineNumber) + "\n")
      sys.stderr.flush()
  return error
```

The check is general more complex. It may require for instance to split the line into pieces. But the general idea is there.

4.4.2 Let's go!

Exercise 4.5 Typical Patterns. This exercise aims at making you familiar with writing patterns.

Question 1. Write a pattern that matches capitalized words (words that starts with an upper case letter, e.g. `Mary`). Apply the `findall` function to extract the list of all capitalized words of a sentence.

Question 2. Write a pattern that matches Python paths, i.e. sequences of identifiers separated with dots, e.g. `block2.pump1._state`. Apply the `findall` function to extract the list of all paths in a string.

Exercise 4.6 Counting Words and Sentences. Another simple exercise.

Question 1. Write a Python script that counts the numbers of words and sentences of a text file.

Question 2. Apply your script on the text file "`Lorem Ipsum.txt`".

Exercise 4.7 Patterns in Files. Write a script (one for each of the following items) that prints out every line from the file that matches the requirement. Apply your script to the file "`I Have a Dream.txt`".

Question 1. The line has a '`I`'.

Question 2. The line starts with a '`I`'.

Question 3. The line contains at least six times the character '`e`'.

Question 4. The line has at least two consecutive vowels (`a`, `e`, `i`, `o`, `u`) like in the word "`bear`". Print out also what are these two vowels and their position in the sentence.

Question 5. The line has a double character, e.g. `oo`. Print out also what is this character and its position in the sentence.

Question 6. The line contains the same word twice. Print out also which is this word and its position in the sentence.

Problem 4.8 Regularity Checking. The file `regularite.csv` contains data about the regularity of trains circulating on the French network at different dates (see case study presented in Appendix A.5). Each line of this file (but the first one, which is the header) is organized as follows.

- The first cell (column) contains the date at which the regularity has been measured. The format of the date is yyyy-mm-dd.
- The second cell contains the family of trains. It should be either `transilien`, `TER`, `intercites` or `TGV`.
- The third cell contains the name of the line.
- The fourth cell contains the regularity of the trains circulating on this line this day. This should be a floating point number comprised between 0 and 100 (both inclusive).
- The fifth cell contains possibly an explanation for the lack of regularity.

Question 1. Write a Python script that verifies that at file `regularite.csv` is at the right format. You have to check:

(a) That each line contains the right number of items.
(b) That the dates are at the right format.
(c) That the categories of trains are one of the categories listed above.
(d) That the regularities are percentages.

In some cases, the regularity is missing (for some technical reason, like the corresponding trains did not circulate this day). This is acceptable, but you should emit a warning. You have to count the numbers of errors and warnings and print out messages for each faultly lines, with their number.

Question 2. Apply your script on the text file `regularite.csv`.

Question 3. Copy the file `regularite.csv` into a file `buggy.csv`, introduce some errors into this file, and apply your script on it so to verify that your script detects the errors.

Problem 4.9 DNA Sequencing. A nucleic acid sequence is a succession of letters that indicate the

order of nucleotides forming alleles within a DNA. The possible letters are A, C, G, and T, representing the four nucleotide bases of a DNA strand — adenine, cytosine, guanine, thymine.

You are given the text file "`DNASequence.txt`" that contains a DNA sequence.

Question 1. Write a Python script that test whether the pattern 'GAATTC' occurs in the sequence.

Question 2. Write a Python script that counts the number of times the pattern 'GAATTC' occurs in the sequence.

Question 3. Write a Python script that counts the number of times the patterns 'GGACC' and 'GGTCC' occur in the sequence.

Question 4. Write a Python script that counts the number of times the pattern consisting of 3 'A' followed by any letter followed by a 'C' occurs in the sequence.

Question 5. Write a Python script that counts the number of times the pattern consisting of a 'A' or a 'C' followed by any letter followed by a 'T' occurs in the sequence.

Question 6. Write a Python script that counts the number of times the pattern consisting of a 'C' followed by any letter but a 'G' followed by 3 to 5 'A' occurs in the sequence.

Question 7. Write a Python script that counts the number of times the pattern consisting of 3 'G' followed by any number of 'A' (but at least one) followed by 3 'T' occurs in the sequence.

Question 8. Write a Python script that counts the number of times the pattern consisting of at least 5 times either the letter 'A' or the letter 'T' occurs in the sequence. Sub-patterns of such a pattern should not be counted, e.g. in the sequence "CAATTAATTG", the count should be 1.

Question 9. Write a Python script that counts the number of times the pattern consisting of 3 letters followed by same three letters, e.g. 'CCTCCT', occurs in the sequence.

Question 10. Write a Python script that counts the number of times the pattern consisting of 4 letters followed by same three letters, e.g. 'CCTACCTA', occurs in the sequence. For each pattern found, print the position in the sequence as well as the group of 8 letters.

Exercise 4.10 License Plates. All Norwegian license plates for civilian vehicles have a prefix of two letters, followed by a sequence of numbers. Military license plates have numbers only. Most vehicles have five-digit registration numbers between 10000 and 99999. Motorcycles, farming equipment and trailers have four-digit registration numbers between 1000 and 9999 is used. Temporary plates have three-digit registration numbers between 100 and 999. Dealer plates have two-digit registration numbers between 10 and 99.

You are given the file "`licensePlates.txt`" that contains a list of license plates, one per line.

Question 1. Write a Python script that extracts license plates of regular vehicles, those of motorcycles, farming equipment and trailers, and finally the others. The result must be written into three separated files.

Question 2. As you should have noticed, some license plates for regular vehicles, motorcycles, farming equipment and trailers contain spaces and some are not. Modify your script so to print out all plates so that letters are separated from digits by one space.

4.5 Rewriting

4.5.1 Rational

It is often the case that we get raw text files that we want to format so to make them easier to read and to understand. We can of course do that using our text editor. However, processing text files "by hand" is both tedious and error prone. It is thus often safer and faster to write Python scripts that do the job for us.

4.5.2 Let's go!

Exercise 4.11 One Sentence per Line. You are given the text file "`Lorem Ispum.txt`". For the sake of clarity, you want to rewrite this text file so that each line of the file contains at most one sentence.

Question 1. Write a Python script that reads a text file and adds an extra newline character after each dot character '`.`'.

Question 2. Modify your script so that there is no extra space at the beginning of lines and that new lines are not added at the end of the last sentences of paragraphs.

Exercise 4.12 Speech Cleaner. You downloaded from internet the text file "`I Have a Dream.txt`". Unfortunately, this is a raw material, which contains not only Martin Luther King speech, but also reactions of the participants to the meeting. For the sake of clarity, you want to rewrite this text file so that:

- Each line of the file contains at most one sentence;
- The reactions of the participants (in parentheses and between square brackets) are removed.
- Extra-space are removed.

Question 1. Write a Python script that reads a text file and performs the above cleaning.

Question 2. Apply your script on the file "`I Have a Dream.txt`".

Problem 4.13 Preparing emails. The file "`Students.csv`" contains information about students you want to communicate with. The file "`EmailTemplate.txt`" contains the template of an email you want to send to each of them. You want thus to create a personalized file, one per student, with the appropriate content.

Question 1. Write a Python function that tests whether the subfolder `Students` exists and creates it if it does not.
Hint: Use to the package `os` to do so.

Question 2. Write a Python function that:

- Takes as arguments the first name, the family name and the email address of a student as well as the list of courses this student is registered to;
- Reads the file "`EmailTemplate.txt`";
- Prints out a file obtained from "`EmailTemplate.txt`" by substituting:
  - the first name of the student for `STUDENT_FIRST_NAME`;
  - the family name of the student for `STUDENT_FAMILY_NAME`,

  - the email address of the student for `STUDENT_EMAIL`, and finally
  - the list of courses the student is registered to for `STUDENT_COURSES`.

  The name of the file should be in the form "`FamilyNameFirstName.txt`", e.g. "`SmithAlice.txt`".

Question 3. Test your function with arguments you give "by hand".

Question 4. Write a function to process the file "`Students.csv`", i.e. to create a file for each student.

Question 5. Test your whole script.

Another example of the need for programs that read and transform is cryptography, the science of secret, which aims at providing way for two partners to exchange messages that cannot be understood by a third party. The idea is that the sender cipher messages and the receiver decipher them. These two operations can indeed performed by hand, but it is much less tedious and error prone to let the machine do them for you.

Exercise 4.14 Substitution Ciphers. A substitution cipher is a method of encrypting by which units of plaintext are replaced with ciphertext, according to a fixed system; the "units" may be single letters (the most common), pairs of letters, triplets of letters, mixtures of the above, and so forth. The receiver deciphers the text by performing the inverse substitution.

We shall consider the simplest substitution cipher which consists in substituting one letter for another letter, e.g.

- '`a`' is replaced by '`c`'.
- '`b`' is replaced by '`E`'.
- '`c`' is replaced by ' ' (space).
- and so on.

Question 1. Write a Python script that reads a text file and ciphers it.

Question 2. Write a Python script that reads a text file and deciphers it.

Question 3. Test your ciphering and deciphering scripts on messages of various lengths.

Note that substitution ciphers are much too easy to break to be used in practice. Their key weakness stands in that they do not modify the relative frequencies of letters in the text. For instance, occurrences of letter '`e`' represent statistically 12.702% of plain English texts, while those of letter '`z`' represent only 0.074%. Using relative frequencies of letters is thus easy to guess what the substitution scheme.

Exercise 4.15 Morse Code. The Morse code is a character encoding scheme used in telecommunication that encodes text characters as standardized sequences of two different signal durations called dots and dashes or dits and dahs. The Morse code is named for Samuel F. B. Morse, an inventor of the telegraph.

The International Morse Code encodes the 26 English letters A through Z, some non-English letters, the Arabic numerals and a small set of punctuation and procedural signals (prosigns). There is no distinction between upper and lower case letters. Each Morse code symbol is formed by a sequence of dots and dashes. The dot duration is the basic unit of time measurement in Morse code transmission. The duration of a dash is three times the duration of a dot. Each dot or dash within a character is followed by period of signal absence, called a space, equal to the dot duration. The letters of a word are

separated by a space of duration equal to three dots, and the words are separated by a space equal to seven dots.

The Morse code is given in Table 4.1.

Question 1. Write a Python function that takes a plain message (a string) in input and returns a string containing the message encoded with the Morse code.
Hint: Declare a global dictionary containing the Morse code.

Question 2. Write a Python function that takes a message coded with Morse code in input and returns a string containing the original (decoded) message.
Hint: Reverse first the global dictionary containing the Morse code.

Question 3. Write a Python function that plays a Morse message.
Hint: This works only if you have a PC under Windows. Use the package `time`, to sleep between two sounds with the function `sleep` and the package `winsound`, to play sounds with the function `Beep`.

Question 4. Apply the above functions on the following messages:

(a) `"SOS"`

(b) `"I love you!"`

(c) `"What hath God wrought?"`

(d) `"CQD THIS IS TITANIC. WE HAVE STRUCK AN ICEBERG. SINKING. REQUEST IMMEDIATE ASSISTANCE. WOMEN AND CHILDREN ONBOARD. COME AT ONCE. POSITION 41.44N 50.24W"`

Table 4.1: Morse code

A	.-	B	-...	C	-.-.	D	-..
E	.	F	..-.	G	-.	H	....
I	..	J	.-	K	-.-	L	.-..
M	-	N	-.	O	---	P	.-.
Q	--.-	R	.-.	S	...	T	-
U	..-	V	...-	W	.-		
X	-..-	Y	-.-	Z	--..		

0	-----	1	.----	2	..---	3	...--
4	....-	5	.....	6	-....	7	--...
8	---..	9	----.				

.	.-.-.-	,	--.--	?	..--..	!	-.-.--
'	.----.	/	-..-.	(	-.--.	)	-.-.--
:	---...	;	-.-.-.	+	.-.-.	-	-....-
=	-...-	&	.-...	_	..--.-	\	.-..-.
$	...-..-	@	.--.-.	\n	' '	' '	' '

Chapter 5

Modules and Packages

Key Concepts
- Modules
- Packages
- `NumPy`
- `Matplotlib`

Knowing how to work with modules and packages is an important skill for any Python developer. It is not too difficult to do, at least if staying at simple use level. This chapter introduces both notions (modules and packages), starting with modules. It presents also briefly two very interesting packages for scientific computing and data analysis: `NumPy`, `Matplotlib`.

5.1 Modules

5.1.1 Presentation

Modular programs refers to the idea of breaking larges program into separate, smaller, more manageable subprograms called modules. Individual modules can then be assembled together to create a larger application. There are several advantages to modularizing programs:

- This makes the development easier and less error-prone. A module typically focus on a relatively small and independent portion of the problem at stake.
- This improves the maintainability of programs. As modules are written in a way that minimizes interdependence, the modifications done on to a single module will hopefully have a limited impact on other parts of the program.
- This favors re-usability from programs to programs. If well designed, a module designed for a first program can be reused in a second one. Moreover, evolutions made for the second program can benefit retrospectively to the first one.
- More technically, modules define separate name spaces, which helps avoiding collisions between function name.

A *module* is thus a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Modules are used to gather functions and classes (which we shall look at later) related to a given domain. We have already seen two modules `sys`, `math` and `random`. Here is a non-exhaustive list of standard modules, which are frequently used:

- `sys` which provides functions and variables used to manipulate different parts of the Python run-time environment.
- `os` which provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, ...

– `math` which provides some of the most popular mathematical functions.
– `string` which provides operations on strings.
– `re` and `regexp` which provide operations on strings using regular expressions (see Chapter 4).
– `random` which provides functions to draw numbers at pseudo-random.

And indeed, you can define your own modules, which is the only way to structure your code in entities of reasonable size when your program gets big.

Assume you developed functions `Fct1`, `Fct2` and `Fct3` and that you put them into the module `mdl.py` located in the subdirectory `dom1` of the directory `dev` in which you put all your programs. Assume moreover that your are now developing a script `scr.py` located in the subdirectory `dom2` of `dev` and that you want to use functions `Fct1`, `Fct2` and `Fct3` in this script.

There are two way to proceed. Both starts by telling Python where to look for modules (in addition to the directories it already knows).

```
import sys
sys.path.insert(0, '../dom1')
```

`../dom1` is the relative path from the current directory to the target directory.

Note that if both the imported module and the importing script are in the same directory, it is not necessary to add anything to `sys.path`.

Then, the first one consists in importing the module `mdl.py` itself, by means of the following directive.

```
import mdl
```

Now, to use functions `Fct1`, `Fct2` and `Fct3` it suffices ro call them, by their name prefixed with "`mld.`". E.g.

```
x = mdl.Fct1(a, b, c)
```

The second way consists in importing the functions directly:

```
from mdl import Fct1, Fct2, Fct3
```

Now, the functions can be called as if they were defined in the current script. E.g.

```
x = Fct1(a, b, c)
```

If you want to important all definitions of a module, you can use the $\star$ character:

```
from mdl import *
```

5.1.2 Let's go!

Exercise 5.1 Grid Walker. To test user defined modules, you shall implement a program that walk on a two dimensional grid. This program should be separated into two parts: a module `Grid` that implements movements along the grid and the main script that implement the user interface of the program.

Question 1. Design the module `Grid`. This implies implementing functions `GoNorth`, `GoSouth`, `GoEast`, `GoWest`, `MakeRandomMove`, `PrintPosition` and `Reset`, with their obvious

meaning.
Hint: To describe the position, you can use two variables, e.g. `x` and `y` that are initialized to 0 (and take back this value when reset). To modify these variables inside functions, you need to declare them global inside each function by means of the instruction `global x` (respectively `global y`).
To draw a move at random, you can use the function `randint(1, 4)` of the module `random`, which draws 1, 2, 3 or 4 with equal probabilities, and chose the direction from that.

Question 2. Design the main script `Walker` that implements the user interface. This interface should consists in a simple loop reading a character on the input and choosing the action accordingly, e.g. `n` goes north, `s` goes south, ... and `q` quits the program.
Hint: Use the predefined function `input` to read the character on the input, e.g. `chr = input()`.

Problem 5.2 Matrices. List can contain any type of objects, including lists. The objective of this problem is to design a module that provides a set of functions to create and to perform operation on matrices of floating point numbers. Matrices will be implemented as lists of lists of floating point numbers.

Question 1. Write a function `MatrixNew(m, n)` that creates a matrix with *m* rows and *n* columns and fills it with 0's.

Question 2. Write a function `MatrixGetNumberOfRows(matrix)` and a function `MatrixGetNumberOfColumns(matrix)` that return respectively the number of rows and the number of columns of a matrix.

Question 3. Write a function `MatrixGet(matrix, rowIndex, columnIndex)` that returns the value stored in the given cell of the matrix. This function should return `None` in case the row index or the column index are out of the range of the matrix.
Write a function `MatrixSet(matrix, rowIndex, columnIndex, value)` that sets the given value into the cell of the matrix of the given row and column indices. This function should do nothing in case the row index or the column index are out of the range of the matrix.

Question 4. Write a function `MatrixPrint(matrix)` that prints the content of the matrix. Each row must be printed on a separated line. Values in a row must be separated by tabulation (character '`\t`'.

Question 5. Write:
- A function `MatrixNewUnit(matrix, numberOfRows)` that creates a unit square matrix with the given of rows.
- A function `MatrixCopy(matrix)` that creates a copy of the given matrix.
- A function `MatrixTranspose(matrix)` that creates the transpose of the given matrix.

Question 6. Write a function `MatrixMultiplyByScalar(matrix, scalar)` that returns a matrix equal to the matrix given in argument multiplied by the given scalar.

Question 7. Write a function `MatrixSum(matrix1, matrix2)` that returns the sum of the two matrices given as arguments. This function should return `None` in case the two matrices do not have the same dimensions.

Question 8. Write a function `MatrixProduct(matrix1, matrix2)` that returns the product of the two matrices given as arguments. This function should return `None` in case the number of columns of the first matrix differs from the number of rows of the second one.

Question 9. Write a function `MatrixPower(matrix, n)` that returns a matrix equals to the matrix given as argument raised to the power of n. This function should return `None` in case the matrix is not a square matrix.
Hint: This function can be implemented by multiplying $n-1$ times the matrix. However, there is a much more elegant (and efficient) algorithm based on the following recurrence.

$$\begin{aligned} M^0 &= I \\ M^1 &= M \\ M^{2k} &= M^k \times M^k \\ M^{2k+1} &= M^k \times M^k \times M \end{aligned}$$

Question 10. Write a function `MatrixInverse(matrix)` that returns the inverse of the matrix given as argument, if it exists.
Hint: Use Gaussian elimination and define intermediate functions:
- `MatrixSwapRows(matrix, rowIndex1, rowIndex2)`.
- `MatrixMultiplyRowByScalar(matrix, rowIndex, scalar)`.
- `MatrixSubtractRow(matrix, rowIndex, subtractedRowIndex, subtractedRowScalar)`.

Exercise 5.3 Matrix Equations. Using the module you have designed problem 5.2, solve the following equations.

Question 1.

$$X = 3A^3 - 5(BC)^2$$

where A, B and C are the following matrices.

$$A = \begin{bmatrix} 1 & 0 & -1 & 2 \\ 0 & 3 & 1 & -1 \\ 2 & 4 & 0 & 3 \\ -3 & 1 & -1 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 2 \\ 3 & -1 \\ 0 & -2 \\ 4 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 3 & -2 & 0 & 5 \\ 1 & 0 & -3 & 4 \end{bmatrix}$$

Question 2.

$$X = A^T A - 2A$$

where A is the following matrix.

$$A = \begin{bmatrix} 3 & -1 \\ 0 & 2 \end{bmatrix}$$

Question 3.

$$A^T X B = C$$

where A and B are the following matrices.

$$A = \begin{bmatrix} 2 & 4 \\ 3 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \quad C = \begin{bmatrix} 7 & 5 \\ 6 & 1 \end{bmatrix}$$

The next problem relies on the case study "Student Cohort" presented in Appendix A.4.

Problem 5.4 Student Cohort. The objective of this problem is to create a Python package to manage a cohort of students such a the one described in case study "Student Cohort".

We want to be able to perform various queries on the database such as retrieving a student from his first name and family name, finding all students who are born in July or finding all students who registered to both TPK4186 and TPK5120.

The specification of the package are as follows.

- The personal data of each student are recorded into a dictionary. Keys of this dictionary are strings giving the name of the datum, e.g. `"FAMILY-NAME"`, `"DATE-OF-BIRTH"`, ... Values are the data themselves.
- Dates of birth are encoded by means of triples (day, month, year).
- Courses to which the student is registered are encoded by means of a list.
- The database itself is encoded as a dictionary. The keys of this dictionary keys student identification numbers. Its values are students records.

Question 1. Write a function `Student_New(identificationNumber)` that creates and returns a new student record. Write accessors for student records, i.e. the following functions (with their obvious meanings).

- `Student_GetIdentificationNumber(student).`
- `Student_GetFamilyName(student).`
- `Student_SetFamilyName(student, familyName).`
- `Student_GetFirstName(student),`
- `Student_SetFirstName(student, firstName).`
- `Student_GetDayOfBirth(student).`
- `Student_GetMonthOfBirth(student).`
- `Student_GetYearOfBirth(student).`
- `Student_SetDateOfBirth(student, day, month, year).`
- `Student_IsRegistered(student, course).`
- `Student_Register(student, course).`
- `Student_CancelRegistration(student, course).`
- `Student_GetCourses(student).`

Question 2. Write a function `Cohort_New()` that creates and returns a new cohort. Write accessors for the cohort, i.e. the following functions (with their obvious meanings).

- `Cohort_LookForStudent(cohort, identificationNumber).`
- `Cohort_GetStudents(cohort)` (should return the list of student records).
- `Cohort_NewStudent(cohort, identificationNumber)` (should return the student record). item `Cohort_RemoveStudent(cohort, student).`

Question 3. Write a function `PrintStudent(student)` that prints the data of a student and a function `PrintStudentList(studentList)` that prints the data of each student of a list.

Question 4. Using the functions defined in the previous questions, creates the database given in

Table A.3 and print out all of the student data.

We shall now implement functions to query the database. These functions will enter into two categories:

- Tests, i.e. function that take a student record and a datum as parameters and return either `True` or `False`. E.g. the function `Student_IsIdentificationNumber(student, identificationNumber)` returns `True` if and only if the student has the given identification number.
- Queries, i.e. functions that take one or several lists of student records as parameters and return a new list of students records. E.g.

Question 5. Implement the following tests (with their obvious meaning):

- `Student_IsIdentificationNumber(student, identificationNumber)`.
- `Student_IsFamilyName(student, familyName)`.
- `Student_IsFirstName(student, firstName)`.
- `Student_IsDayOfBirth(student, dayOfBirth)`.
- `Student_IsMonthOfBirth(student, monthOfBirth)`.
- `Student_IsYearOfBirth(student, yearOfBirth)`.
- `Student_IsRegistered(student, course)`.

Question 6. Implement the following functions.

- `Query_Select(studentList, test, datum)` that returns the student records `student` of the given list that `test(student, datum)` return `True`.
- `Query_Union(studentList1, studentList2)` that returns the list of student records that are either in `studentList1` or in `studentList2`.
- `Query_Intersection(studentList1, studentList2)` that returns the list of student records that are both in `studentList1` or in `studentList2`.
- `Query_Difference(studentList1, studentList2)` that returns the list of student records that are in `studentList1` but not in `studentList2`.

Question 7. Use the functions implemented in the previous questions to get the following lists of students.

(a) Students whose family name is Johnson.
(b) Students born in 1995.
(c) Students registered to PYS4160.
(d) Students registered to both PYS4160 and PYS4265.
(e) Students born in March or in April.
(f) Students born in 1996 or in 1997 who are not taking PYS4265.

5.2 Packages

5.2.1 Presentation

Large applications may include many modules. As the number of modules grows, it becomes difficult to manage of them all. *Packages* provide a mean to group and organize them.

Packages allow for a hierarchical structuring of the module name-space using dot notation. In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names.

Creating a package is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. As a developer of relatively simple Python scripts:

- You seldom have to create packages.

– You will for sure use massively packages.

Packages are what makes the richness of Python. There are packages available for virtually all computational tasks in engineering.

Conversely to modules/packages like `sys`, `re` or `random`, which come with the default Python implementation, you need to install these packages before using them (one of the interest of the Anaconda environment is that it comes with many pre-installed packages).

In the standard Python environment, this is achieved by means of the command `pip` (pip is a recursive acronym for "Pip Installs Packages"). In the Anaconda environment, this is achieved by means of the command `conda`. Both commands are going onto internet to seek (in predefined repositories) for the packages to be installed, download and install them.

Assume we want to install the `myPackage` package which provides some functionalities we need for our application. To do so, we first have to open an Anaconda Prompt (i.e. a command interpreter). Then simply enter the following command to install the `myPackage` package.

```
conda install myPackage
```

This starts an interactive script that installs the package.

The package `myPackage` may contain several modules, like `myModule1`, `myModule2`... These modules can be imported as before, by prefixing them with `myPackage`. E.g.

```
import myPackage
import myPackage.myModule1
myModule1.Function1()
myPackage.myModule2.Function2()
```

5.2.2 Let's go!

Problem 5.5 MP3 Player. Assume we want to develop an MP3 music player. The package `pygame`, which as its name suggests is intended to develop games, provides functionalities to do so. Unfortunately, this package is not part of the Anaconda distribution. We have thus to install it ourselves.

Question 1. Install the `pygame` package.

Hint: Not all versions of pygame are compatible with Python 3.7. You have to find one which is. At the time I wrote these lines, one was available on the `delichon` repository. It could be installed with the following command.

```
conda install -c delichon pygame
```

Question 2. The implementation of a MP3 player using pygame is a bit tricky. The script `MP3Player-OneSong.py` proposes one that you can try to run and understand.

Question 3. Modify the script `MP3PlayerOneSong.py` so to play a list of songs in loop.

Hint: You may need to introduce a global variable `stopped` that is tested and possibly modified in each functions. To use a global variable `stopped` in the function `PlayFile`, you have to declare it as follows.

```
def PlayFile(fileName):
  global stopped
  if stopped:
    return
  # rest of the function
```

Using the modules `glob` and `random` create the list of `mp3` files in a folder (directory), sort it at random and play it in loop.

5.3 The `NumPy` Package

5.3.1 Presentation

NumPy is the core library for scientific computing in Python (van der Walt et al., 2011). NumPy is often used along with packages like SciPy (Scientific Python) and Matplotlib (plotting library). Compared to MatLab, this combination is more modern and more complete.

NumPy's main object is the homogeneous multidimensional array. They are tables of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. NumPy provides a collection of functions to process these arrays. Compared to similar functions that would be implemented on standard Python lists, they are much more efficient. The interest of NumPy is thus (at least) twofold:

- First, it provides an efficient implementation of multidimensional arrays.
- Second, it provides a variety of ready-to-use functions to process these arrays.

NumPy's arrays are called `ndarray`. In NumPy, dimensions are called *axes*. The number of axes is the *rank* of the array. The *shape* of an array is a tuple of integers giving the size of the array along each dimension.

Array Creation

`ndarray`'s can be initialized from nested Python lists. As for lists, their elements are accessed using the square brackets notation. The following code illustrates these points.

```
import numpy as np

b = np.array([[1,2,3],[4,5,6]])    # Creates a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

`NumPy` provides actually a many functions to create arrays, e.g.

```
import numpy as np

a = np.zeros((3, 3, 2))       # Creates a 3x3x2 array filled with 0's
b = np.ones(10)               # Creates a vector of 1'
c = np.full((3,4), 0.1)       # Creates a 3x4 matrix filled with 0.1's
d = np.eye(3)                 # Creates a 3x3 identity matrix
e = np.random.random((2,10)) # Creates a 2x10 matrix filled with random
                              # numbers drawn uniformly at random in [0, 1[
```

Basic Operations

Arithmetic operators on arrays apply element-wise. A new array is created and filled with the result. E.g.

```
import numpy as np

a = np.array([10, 20, 30, 40])
b = np.arange(4) # creates the array [0, 1, 2, 3]
b = b + 1
c = a - b
d = c**2

print(c)
print(d)

>>>
[ 9 18 27 36]
[  81  324  729 1296]
```

Unlike in many matrix languages, the product operator $\star$ operates element-wise in NumPy arrays. The matrix product can be performed using the `@` operator (in Python >=3.5) or the `dot` function or method. E.g.

```
import numpy as np

A = np.array([[1, 1], [0, 1]])
B = np.array([[2, 0], [3, 4]])
C = A * B    # elementwise product
D = A @ B    # matrix product
E = A.dot(B) # matrix product

print(C)
print(D)
print(E)

>>>
[[2 0]
 [0 4]]
[[5 4]
 [3 4]]
[[5 4]
 [3 4]]
```

Some operations, like += and $\star$=, act in place and modify their argument rather than create a new array. E.g.

```
import numpy as np

np.random.seed(12345)
A = np.ones((2,3))
B = np.random.random((2,3))
A *= 3
B += A

print(A)
print(B)

>>>
[[3. 3. 3.]
 [3. 3. 3.]]
[[3.92961609 3.31637555 3.18391881]
 [3.20456028 3.56772503 3.5955447 ]]
```

Many unary operations, such as computing the sum, minimum or the maximum of all the elements in the array, are implemented as methods of the `ndarray` class (we shall see what are methods in Chapter 6. For now, it suffices to consider them as functions with a particular syntax).

```
import numpy as np

np.random.seed(12345)
A = np.random.random((3,3))

print(A)
print("Sum = " + str(A.sum()))
print("Min = " + str(A.min()))
print("Max = " + str(A.max()))
print("Mean = " + str(A.mean()))

>>>
[[0.92961609 0.31637555 0.18391881]
 [0.20456028 0.56772503 0.5955447 ]
 [0.96451452 0.6531771  0.74890664]]
Sum = 5.1643387238311105
Min = 0.18391881167709445
Max = 0.9645145197356216
Mean = 0.5738154137590122
```

Reshaping

Fundamentally, a `nparray` is a list of values. The shape of the array, i.e. the ways it is organized is an information that comes in addition to this list. This makes it possible to "reshape" arrays at low cost. E.g.

```
import numpy as np

A = np.arange(12)
print(A)

>>>
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

```
B = A.reshape((3, 4))
print(B)

>>>
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
C = B.reshape((6, 2))
print(C)

>>>
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
```

```
D = C.reshape((3, 2, 2))
print(D)

>>>
[[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]]
```

```
E = D.reshape(12)
print(E)

>>>
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Indexing, Slicing and Iterating

One-dimensional arrays can be indexed, sliced and iterated over, like Python lists. E.g.

```
import sys
import numpy as np

A = np.arange(12)
B = A[2:5]
print(B)

for value in B:
  sys.stdout.write(str(value) + " ")
sys.stdout.write("\n")

>>>
[2 3 4]
2 3 4
```

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas. E.g.

```
import numpy as np

A = np.arange(12)
B = A.reshape((4, 3))

print(B)
print(B[3, 1])
print(B[:, 1])

>>>
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
10
[ 1  4  7 10]
```

Slicing does not create new arrays, but rather view on an array. This is illustrated by the following code.

```
import numpy as np

A = np.arange(9).reshape((3,3))
V = A[1,:] # V is a view the second row of A
C = np.copy(A[1,:]) # C is a copy the second row of A
print(V)
print(C)
A[1,:] = np.zeros(3)
print(V)
print(C)

>>>
[3 4 5]
[3 4 5]
[0 0 0] # The value of V has been modified by the operation on A
[3 4 5] # The value of C has not been modified by the operation on A
```

Numpy provides a special operator `::` that makes it possible to select elements of an array (or of an axis of a multidimensional array). It works as follows `A[i::k]` selects one elements out of `k`, starting from index `i`. E.g.

```
import numpy as np

A = np.arange(15)
B = A[2::3]

print(B)

>>>
[ 2  5  8 11 14]
```

In case of a multidimensional array:

```
import numpy as np

A = np.arange(15).reshape((3, 5))
B = A[:, 1::2]

print(B)

>>>
[[ 1  3]
 [ 6  8]
 [11 13]]
```

Even though the resulting code is compact and efficient, maintaining it may be difficult.

Stacking

It is possible to stack arrays together. E.g.

```
A = np.arange(6)
B = A.reshape((2, 3))
C = B + 6
D = np.hstack((B, C))
E = np.vstack((B, C))

print(D)
print(E)

>>>
[[ 0  1  2  6  7  8]
 [ 3  4  5  9 10 11]]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Other Operations

In addition to the operations presented in this section, many other are available, some of general purpose, some other dedicated to a particular domain, e.g. Fast Fourier Transform or financial analyses.

A complete list of available operations can be found here:

`https://numpy.org/devdocs/reference/routines.html#routines`

Do not try to memorize these operations. Just go to the documentation each time you need.

5.3.2 Let's go!

Exercise 5.6 Numpy Basics. Write a Numpy script to perform the following operations.

Question 1. Create a 3×3 matrix with values ranging from 2 to 10.

Question 2. Create a 3×3 matrix with values ranging from 0 to 9. Reverse its columns, the reverse its rows.

Question 3. Create a 8×8 matrix and fill it with a checkerboard pattern, i.e. an alternation of 0 and 1 on each row, the off rows starting with a 0 and the even rows with a 1.

Question 4. Convert the values of Centigrade degrees stored in an array into Fahrenheit degrees, and vice-versa. The conversion equation is as follows.

$$\frac{C}{5} = \frac{F-32}{9}$$

Question 5. Write a function that swaps column i and column j of a matrix $m \times n$ and function that swaps it row i and column j (assuming $0 \leq i < m$ and $0 \leq j < n$).

Exercise 5.7 Matrix Equations with NumPy. Same questions as Exercise 5.3, using NumPy's arrays.

5.4 The `Matplotlib` Package

5.4.1 Presentation

One of the most popular uses for Python is data analysis. Data analysts need indeed a way to visualize their data, either that they want to get a better grasp on them, or that they want to convey their results to someone else. `Matplotlib`, originally developed by John D. Hunter (Hunter, 2007), is the most popular package to do so. It provides a very large number of functionalities, and makes things quite easy.

Figure 5.1 shows a first example of what it is possible to do.

The code given in Figure 5.1 does essentially three things: first, it creates the chart pictured in Figure 5.2; second, it prints out this chart in the pdf file `Sales.pdf`; third, it displays the chart on your computer screen.

This code is fairly simple.

- First, the module `pyplot` of `Matplolib` is imported and given the name `plt`.
- Second, the lists of x- and y-values for the two products `BVR724` and `BVR068` are created. Indeed, the list of x-values and y-values of a quantity must contain the same number of elements. In our case, as the x-axis represents months, the two curves can share their x-values.

```
import matplotlib.pyplot as plt

months = [m for m in range(1, 13)]
BVR724 = [4939, 4521, 4767, 4992, 5151, 5378, 4578,
          4634, 5210, 5376, 4470, 4764]
BVR068 = [7366, 7214, 7329, 7634, 7138, 7563, 7282,
          7145, 7190, 7425, 7357, 7626]

plt.plot(months, BVR724, 'go', label='BVR724')
plt.plot(months, BVR068, 'r^', label='BVR068')

plt.title('Monthly sales of BVR724 and BVR068')
plt.ylabel('Sales')
plt.xlabel('Months')
plt.legend()
plt.savefig('Sales.pdf')
plt.show()
plt.close()
```

Figure 5.1: A small script using Matplotlib

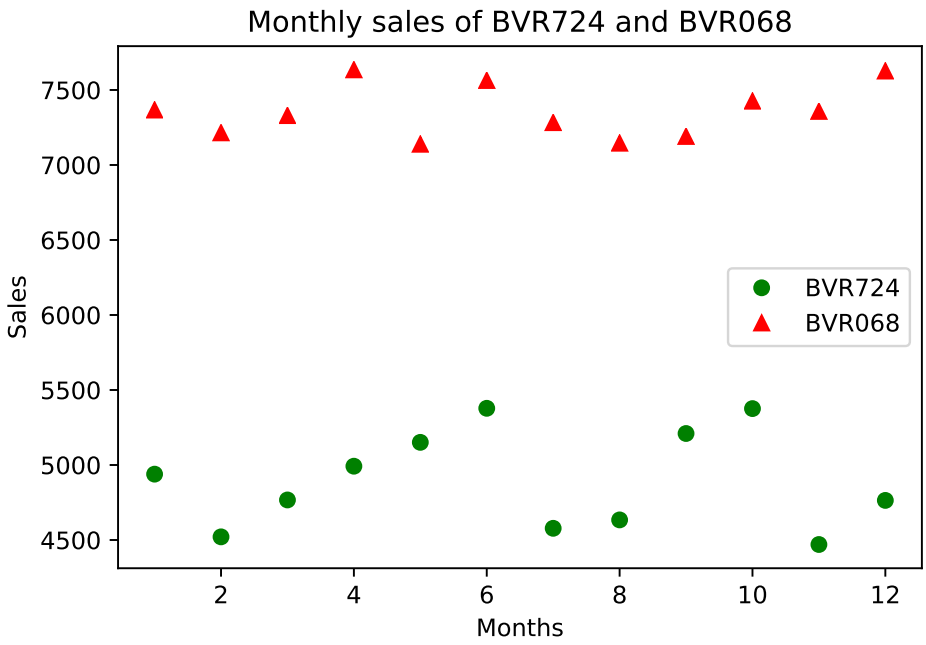


Figure 5.2: The pdf file produced by the script given in Figure 5.1

- Third, the two curves are plotted. The argument `go` requires the points of the curve to be displayed as green circles, while the argument `r^` requires them to be display as red triangles.
- Fourth, the legend is created, i.e. the title of the figure, the labels of the two curves and those of x- and y-axes.
- Fifth, the figure is exported as a pdf file.
- Sixth, the figure is displayed on the computer screen, then closed.

Many options are available to display curves as line, bars, pies... Styles can be used for texts. Figures can be saved in the most widely formats like jpeg, png, pdf... Images can be inserted.

The reader is strongly encouraged to visit the Matplotlib website to discover all these functionalities:

`https://matplotlib.org/index.html`

5.4.2 Let's go!

For all the exercises below, a pdf file is provided (in addition, as usual, to the solution) so that you can have an idea of what the result should look like.

Exercise 5.8 Financial Data Alphabet Inc.. Write a Python program to draw line charts of the financial data of Alphabet Inc. between October 3, 2016 to October 7, 2016, given in the following table.

Date	Open	High	Low	Close
10-03-16	774.25	776.065002	769.5	772.559998
10-04-16	776.030029	778.710022	772.890015	776.429993
10-05-16	779.309998	782.070007	775.650024	776.469971
10-06-16	779	780.47998	775.539978	776.859985
10-07-16	779.659973	779.659973	770.75	775.080017

Exercise 5.9 Weibull Distribution. In probability theory and statistics, the Weibull distributions a continuous probability distribution. It is characterized by two parameters: a scale parameter, usually denoted by α, and a shape parameter, usually denoted β. Its cumulative probability distribution is as follows.

$$Weibull(t,\alpha,\beta) \stackrel{def}{=} 1-\exp\left(-\left(\frac{t}{\alpha}\right)^{\beta}\right)$$

When it represnts a "time-to-failure", the Weibull distribution gives a distribution for which the failure rate is proportional to a power of the time. The shape parameter, β, is that power plus one, and so this parameter can be interpreted directly as follows.

- A value of $\beta < 1$ indicates that the failure rate decreases over time. This happens if there is significant "infant mortality" of the component under study.
- A value of $\beta = 1$ indicates that the failure rate is constant over time, in which case the Weibull distribution reduces to an exponential distribution.
- A value of $\beta > 1$ indicates that the failure rate increases over time. This happens when the component under study is "aging" or "wearing-out"

Write a Python script to plot the values of the Weilbull distribution from time 0 to time 8760 (one year in hours), for $\alpha = 10^3$ and $\beta = 0.5, 0.75, 1.0, 1.5$ and 2.0.

Exercise 5.10 Indefinite Integral. We want to estimate the value I of the following integral.

$$I \stackrel{def}{=} \int_0^{+\infty} \frac{e^{-x}}{1+(x-1)^2}\,dx$$

An idea to do so is to calculate the average value of the function $f(x) = \frac{e^{-x}}{1+(x-1)^2}$ for a suitable sample of x's. We have however a problem here: the integral is indefinite. We shall therefore proceed by steps.

Question 1. Design a Python function that calculates $f(x)$. Using `NumPy` and `matplotlib`, plot this function from 0 to n. Adjust the value of n so visualize what's going on.

From the above question, it appears clearly that f can be separated into two zones: a zone from 0 to a certain value z in which it decreases sharply and a zone from z to ∞ in which it is almost constant.

Question 2. Estimate the value of the integral on the interval $[0,z]$ applying the suggested method.

Question 3. Estimate the value of the integral on the interval $[z,\infty)$ applying the suggested method.

Question 4. Combining the results of the two above questions, estimate the value of the integral.

Exercise 5.11 Retailer Sales. Write a Python program to draw a bar plot by month and by product of the retailer sales (see the script given in Figure 5.1 for the data).

Exercise 5.12 Language Popularity. In 2019, the 10 most widely used programming languages were the following (the popularity is given in percentages of use).

Language	Java	C	Python	C++	C#
Popularity	16.028	15.154	10.020	6.057	3.842
Language	Visual Basic	JavaScrip	PHP	Objective-C	SQL
Popularity	3.695	2.258	2.075	1.690	1.625

Write a Python program to draw a pie chart of the popularity of programming languages. Do not forget the other languages.

Exercise 5.13 Shelling Map. This exercise prepares an experiment we shall perform in the second part of this book.

Question 1. Use `numpy` to build an array T with m rows and n columns and fill with numbers drawn at random uniformly in $[0,1]$.

Question 2. Use `imshow`, color tables and normalization functions to display an image made of $m \times n$ cells such that:

- The cell (i,j) is colored in blue if $T[i][j] < 0.45$.
- The cell (i,j) is colored in white if $0.45 \leq T[i][j] < 0.55$.
- The cell (i,j) is colored in red if $0.55 \leq T[i][j] < 1.0$.

Question 3. Test your functions on a 100×100 array.

Chapter 6

Object-Oriented Programming

Key Concepts
- Object-oriented programming
- Classes, objects, methods
- Data encapsulation, interface
- Inheritance, overloading

6.1 Introduction

Python, like most of the modern programming languages, is object-oriented. Object-oriented programming is the dominant paradigm in software development since decades, for good reasons. In the framework of this book, we shall present only the basic principles. Even occasional programmers are strongly advised to write all their Python scripts in an object-oriented way. This programming methodology eases a lot the design and even more importantly the maintenance of programs.

Thousands of books have been published on the object-oriented paradigm. The one by Bertrand Meyer (Meyer, 1988) is widely considered a foundational text and is still, more than twenty years after its publication, an invaluable source of knowledge. For advanced programmers, the book on design patterns (Gamma et al., 1994) is also worth to study.

Most of the computer programs are dealing with two types of "things": data and operations to be performed on these data. A typical Python script reads or creates input data, performs some operations on these data, and eventually prints out the results of these operations (which are also data). Except for simplistic scripts, storing data into variables is not sufficient. Containers like lists and dictionaries make it possible to store collections of data, which is already much better. If only collections of simple data such as numbers or strings are to be considered, they do the job. It is however often the case that data to be handled are complex. As an illustration, we shall consider the case study "Student Cohort" presented in Appendix A.4.

We want to create a Python script to manage a student cohort. The specifications of this tool may be the following.

- The cohort involves an arbitrary number of students.
- A number of information must be stored for each student: first and family names, date of birth, courses to which the student is registered and so on.
- Each student must be uniquely identified.

Here follows a (non exhaustive) list of operations that can be performed on the cohort.

- Add a new student to the cohort.
- Remove a student from the cohort.
- Register a student to a course.
- Remove a student from a course.

- Check whether a student of a given name belongs to the cohort.
- Get the list of students registered to a given course.
- Get the list of courses a student is registered to.

⋮

-

To store data corresponding to a cohort as specified in the above use case, we could indeed use dictionaries and lists. E.g. the cohort could be managed by means of two lists, one for the students and one for the courses; each student and each course would be represented by a dictionary containing an entry for each information related to the student or the course, e.g. the first and family names of the student, his or her gender and birth date.... Then functions would be written to implement operations described by the specification. This is basically the way we proceeded Exercise 5.4.

However, this way of proceeding is not sustainable: even a minor change in the specification may have dramatic effect on the whole program. Moreover, we would have to remember where is stored what with no help from the program. For our case study, it may be still manageable because the number of types of objects is small. But imagine the nightmare of a new software developer taking over a program that manages dozens of different types of objects!

It could be indeed possible to document the program, i.e. to write a text that describes how the program is organized. The experience shows that this is by no means a good solution: documentations tend to be lengthy, incomplete, ambiguous and worst than all incoherent with the program. It is actually often the case that one modifies in a hurry the program, because the client urges to solve a problem, and has no time to update the documentation, then forgets to update it when the storm is gone, leaving the puzzle to the poor guy who takes over the maintenance of the tool (this poor guy may be the developer himself, six months later).

The solution consists in defining your own containers, dedicated to the storage of specific data. The set of such dedicated containers used in a program is called the *data structure* of this program. A program defines thus essentially two things:

- The data structure in which data specific to each execution of the program are stored.
- Operations to be performed on this data structure.

Hence the title of a famous book by Niklaus Wirth: *Algorithms + Data Structures = Programs* (Wirth, 1976). This book introduced the programming language Pascal and was a manifesto for the so-called *structured programming*, i.e. a methodology for software development based on the definition of data-structures. It is a milestone in software engineering history.

6.2 Classes and Objects

6.2.1 Presentation

In Python, data structures are implemented by means of classes. A *classClass* is a container for data. A datum stored in a class is called an *attribute*. Attributes are also called *fields* or *properties*. An attribute has a name (an identifier) and a value that can be any kind of data.

Once declared, a class can be *instantiated*, i.e. a new container of the type described by the class can be created. This container is an *object*. It is an *instance* of the class. A program can create as many instances of the class as needed.

Coming back to our case study, we can observe that our problem involves essentially entities of two types: cohorts and students. To be complete, we should probably add courses, professors teaching the courses and rooms in which the lectures are given, but we keep our model simple at least for now.

In a first version, we can consider that data associated with a student are the following.

- A unique number that identifies this student in the cohort. Two students with the same name (it happens) and *a fortiori* two students with different names will have different references.
- A first name and a family name.

– A gender.
– A date of birth.
– A list of courses to which he or she is registered.

In Python, a class is declared together with a *creator*, i.e. a function that instantiate the class. The creator has two roles: first, declaring the fields of the class, second instantiating object.

Class declaration works as follows.

```
class Student:
  UNDEFINED = 0
  MALE = 1
  FEMALE = 2

  def __init__(self, identificationNumber):
    self.identificationNumber = identificationNumber
    self.familyName = None
    self.firstName = None
    self.gender = Student.UNDEFINED
    self.dayOfBirth = None
    self.monthOfBirth = None
    self.yearOfBirth = None
    self.courses = []
```

The declaration of a class is introduced by the keyword `class` followed by the identifier of the class, followed by a colon (there may be other information between the identifier and the colon, but we come back on that later).

Once the class declared, three symbolic constants are declared within the class to characterize the gender of students: `UNDEFINED`, `MALE` and `FEMALE`. It is convenient to use such constants rather than either directly values such as 0, 1, 2, or texts such as `"Undefined"`, `"Male"` and `"Female"`: The former are not explicit and thus hard to understand. The latter are error prone, e.g. is `"undefined"` equivalent to `"Undefined"`?.

Then, comes a special function called the creator of the class. In Python, the creator of a class is always called `__init__` (i.e. the word `init` is surrounded by two underscore characters on each side). The creator may take any number of arguments, but at least one usually called `self`. `self`, which is always the first argument, represents the object itself (hence its name). In the above code, the creator of `Student` takes only on argument, i.e. in addition to `self`: the unique identification number `identificationNumber` of the student.

Height attributes are declared for the class `Student`:
– Her/his unique identification number.
– Her/his family name.
– Her/his first name.
– Her/his gender.
– Her/his day of birth.
– Her/his day of month.
– Her/his day of year.
– A list of courses to which she/he is registered.

As said above, once declared, a class can be instantiated. In Python, this works simply as follows.

```
student1 = Student(2961201030)
student2 = Student(1970706032)
```

Note that Python is a managed language, conversely for instance to C++, i.e. that if an object becomes useless, there is no need to destroy it. The Python interpreted does that for you.

Once an object created, its attributes can be accessed using the dot notation, e.g.

```
student1.familyName = "Smith"
student1.firstName = "Alice"
student1.gender = Student.FEMALE
```

Indeed, instances of `Student`, like `student1`, can be stored in lists, dictionaries and other objects.

6.2.2 Let's go!

In the following exercises, you are just asked to create classes to represent the different entities involved in an application. For now, you do not have to implement any treatment. These treatments will be implemented in exercises proposed in the following sections.

Exercise 6.1 Complex Numbers (Classes). In this exercise, we shall revisit exercise 3.1 (about complex numbers) in an object-oriented style.

Question 1. Design a class to handle complex numbers.

Question 2. Instantiate this class for the complex numbers 1, $-i$ and $-1+i$.

Problem 6.2 Retailer Sales (Classes). This problem relies on the case study "Retailer Sales" presented in Appendix A.2. To handle this case study, we need two classes:
- A class `Retailer` to represent the retailer;
- A class `Product` to represent each product (under scrutiny) sold by the retailer.

You are asked to implement these two classes in a module `Retailer.py`.

Question 1. Implement the class `Product`. This class should have (at least) three attributes:
- the (unique) name/reference of the product,
- the profit per unit realized on the product,
- the monthly sales for each month of the year.

Question 2. Implement the class `Retailer`. This class should manage a list of products.

Question 3. Instantiate, in a module "`Main.py`", these classes to represent a few rows of the file "`Sales.tsv`".

Problem 6.3 Warehouse Construction (Classes). This problem relies on the case study "Warehouse Construction" presented in Appendix A.3. To handle this case study, we need two classes:
- A class `Project` to represent the project;
- A class `Activity` to represent each activity of the project.

You are asked to implement these two classes in a module `PERT.py`.

Question 1. Implement the class `Activity`. This class should have nine attributes:
- the name of the activity,
- its description,
- its duration,

- its precedessors, i.e. the list of activities that precede it.
- its successors, i.e. the list of activities that it precedes.
- its early start and completion dates and its late start and completion dates,

Question 2. Implement the class `Project`. This class should manage a list of activities as well as a completion date.

Question 3. Instantiate, in a module "`Main.py`", these classes to represent a few rows of the file "`Warehouse.tsv`".

Problem 6.4 Student Cohort (Classes). This problem relies on the case study "Student Cohort" that is used to illustrate object-oriented construct along this chapter. This case study is presented in Appendix A.4. It is also treated in problem 5.4 using only functions.

To handle this case study, we need two classes:

- A class `Cohort` to represent the cohort of students;
- A class `Student` to represent each student of the cohort.

You are asked to implement these two classes in a module `Cohort.py`.

Question 1. Implement the class `Student`. This class should have (at least) the following attributes.

- The (unique) identification number of the student;
- Her/his first name and family name;
- Her/his gender;
- Her/his day, month and year of birth;
- Her/his email address;
- The list of courses to which she/he is registered.

Question 2. Implement the class `Cohort`. This class should manage a list of students,

Question 3. Instantiate, in a module "`Main.py`", these classes to represent a few rows of the file "`Students.csv`".

6.3 Methods

6.3.1 Presentation

To perform treatments on objects, it is possible to declare and use functions, as we have done so far. However, many treatments are specific to objects of a given class. They are in some sense associated with this class.

For instance, in our case study, we probably want to register students to courses by means of a function rather than to operate directly on the corresponding attribute. Doing so helps to maintain the program: if, for some reason, we shall have to change the internal way courses are registered, e.g. use a dictionary rather than a list, then "clients" of the class `Students` will not be impacted by this change.

This technique is called *data encapsulation*. It consists in hiding, as much as possible, implementation details. The class is declared together with methods that constitute its *interface*, i.e. what is publicly visible. Implementation details on the contrary stay private.

When a function is declared together with a class, it is called a *method*. As creators, methods take at least one argument, usually called `self` that refers to the object calling it.

For instance, the method that registers a student to a course could be as follows.

```
class Student:
  def __init__(self, identificationNumber):
    ...

  def Register(self, course):
    if self.IsRegistered(course):
      return
    self.courses.append(course)
...
student.RegisterCourse("TPK4186")
```

The method `Register` registers the student to the course. Note the use of the dot notation to call methods.

A full application of the data-encapsulation principle consists in defining methods to get and to set each attribute. Such methods are called *accessors*. E.g.

```
class Student:
  def __init__(self, identificationNumber):
    ...

  def GetIdentificationNumber(self):
    return self.identificationNumber

  def IsIdentificationNumber(self, identificationNumber):
    return self.identificationNumber==identificationNumber

...
```

6.3.2 Let's go!

The following exercises and problems continue those of Section 6.2.2

Exercise 6.5 Complex numbers (Methods). We can go on our exercise on complex number (Exercise 6.1).

Question 1. Write accessors to get the real and imaginary parts of a complex number.

Question 2. Design methods to calculate the sum, the difference, the product and the quotient of two complex numbers as well as to calculate the power of a complex number.

Question 3. Write a method that prints out a complex number.

Question 4. Using the methods defined in the previous questions, calculate the values of the following expressions.

$$\begin{aligned} c_1 &= \frac{i-4}{2i-3} \\ c_2 &= \frac{1+i}{1-i} - (1+2i)(2+2i) + \frac{3-i}{1+i} \\ c_3 &= 2i(i-1) + \left(\overline{\sqrt{3}+i}\right)^3 + (1+i)\overline{(1+i)} \end{aligned}$$

Problem 6.6 Retailer Sales (Methods). We shall now go on working on the implementation of the case study "Retailer Sales" (Problem 6.2).

Question 1. Implement accessors for the classes `Retailer` and `Product`.

Question 2. Instantiate these classes and set up the values of their attributes to represent data of the file "`Sales.tsv`".

Question 3. Implement methods to:
- Compute the yearly sales of a product;
- Compute the monthly profit realized on a product;
- Compute the yearly profit realized on a product;
- Compute the monthly profit realized by the retailer;
- Compute the yearly profit realized by the retailer.

Question 4. Test the methods developed in the previous question by printing the monthly and yearly profits realized by the retailer.

Problem 6.7 Warehouse Construction (Methods). We shall now go on working on the implementation of the case study "Warehouse Construction" (Problem 6.3).

Question 1. Implement accessors for the classes `Project` and `Activity`.

Question 2. Instantiate these classes and set up the values of their attributes to represent data of the file "`Warehouse.tsv`".

Question 3. Implement methods to:
- Compute the early starting and completion dates of each activity and the early completion date of the project.
- Compute the late starting and completion dates of each activity.
- Compute the list of critical activities.

Question 4. Test the methods developed in the previous question by printing for each activity its name, its four dates and its criticality. Print out also the early completion date of the project.

Problem 6.8 Student Cohort (Methods). We shall now go on working on the implementation of the case study "Student Cohort". (Problem 6.4).

Question 1. Implement accessors for the classes `Cohort` and `Student`.

Question 2. Instantiate these classes and set up the values of their attributes to represent data of the file "`Students.tsv`".

Question 3. Implement the following methods of the class `Student` (with their obvious meaning):
- `IsIdentificationNumber(student, identificationNumber).`
- `IsFamilyName(student, familyName).`
- `IsFirstName(student, firstName).`
- `IsDayOfBirth(student, dayOfBirth).`

- `MonthOfBirth(student, monthOfBirth)`.
- `IsYearOfBirth(student, yearOfBirth)`.
- `IsRegistered(student, course)`.

Question 4. Implement a class `Query` and the following methods:

- `PrintStudent(student)` that prints the data of a student;
- `PrintStudentList(studentList)` that prints the data of each student of a list;
- `Select(studentList, test, datum)` that returns the student records `student` of the given list that `test(student, datum)` return `True`;
- `Union(studentList1, studentList2)` that returns the list of student records that are either in `studentList1` or in `studentList2`;
- `Intersection(studentList1, studentList2)` that returns the list of student records that are both in `studentList1` or in `studentList2`;
- `Difference(studentList1, studentList2)` that returns the list of student records that are in `studentList1` but not in `studentList2`.

Question 5. Use the class `Query` implemented in the previous questions to get the following lists of students.

(a) Students whose family name is Johnson.
(b) Students born in 1995.
(c) Students registered to PYS4160.
(d) Students registered to both PYS4160 and PYS4265.
(e) Students born in March or in April.
(f) Students born in 1996 or in 1997 who are not taking PYS4265.

6.4 Inheritance and Overloading

6.4.1 Presentation

A full application managing cohorts of students would probably define not only classes for students and cohorts, but also for courses, professors, rooms... Now, students and professors share many characteristics like having a unique identification number, a first and family names, a gender, a date of birth.

Object-oriented languages make it possible to describe such situations by means of *inheritance*. When a class `A` *inherits* from a class `B`, all fields and methods of `B` become automatically fields and methods of `A`. In other words, `A` *is a* `B` with possibly some additional properties.

In Python, inheritance works as follows.

```
class Person:
  UNDEFINED = 0
  MALE = 1
  FEMALE = 2

  def __init__(self, identificationNumber):
    self.identificationNumber = identificationNumber
    self.firstName = None
    self.familyName = None
    self.gender = Person.UNDEFINED

class Student(Person):
  def __init__(self, identificationNumber):
    Person.__init__(identificationNumber)
    # Initialization of attributes specific to students

class Professor(Person):
  def __init__(self, identificationNumber):
    Person.__init__(familyName, firstNames)
    # Initialization of attributes specific to professors
```

The above code declares first the base class `Person`, with all fields and methods associated with a person (we declared no method to make the code short, but there may be indeed an arbitrary number of). Then, it declares the two derived classes `Student` and `Professor` which both inherit from `Person`.

Note that the constructor of `Person` is called in the constructors of `Student` and `Professor`, via the construct `Person.__init__`.

Note also that we could have added a list attribute `courses` to the class `Person`. It this case, this attribute has different meanings in the derived classes. In the class `Students` it would be used to register the students to courses, while in the class `Professor` it would used to define the courses given by professors.

Now assume that the class `Person` implements a method `RegisterCourse` that adds a course to the list. This method is simply defined as follows.

```
  def RegisterCourse(self, course):
    if self.IsRegistered(course):
      return
    self.courses.append(course)
```

It can be applied on instances of both `Student` and `Professor`. However, we may want to change it slightly its behavior in both cases:

- In the case of instances of `Student`, we may want that to append the student to the students of the class `Course` (assuming we have defined this class).
- In the case of `Professor`, we may want to set the attribute `professor` of the class `Course`.

A first solution consists in writing a new method (different in each case). E.g. to declare the following method with the class `Student`.

```
  def RegisterToCourse(self, course):
    self.RegisterCourse(book)
    course.AddStudent(self)
```

Another solution consists in redefining the method `RegisterCourse`. E.g. the redefinition for the class `Student` could be as follows.

```
  def RegisterCourse(self, course):
    Person.RegisterCourse(self, course)
    book.RegisterCourse(self)
```

Now calling the method `RegisterCourse` on an instance of the class `Student` will call the above method (which itself calls the method `RegisterCourse` of the superclass `Person`).

This technique is called *overloading*. It proves to be extremely useful in many practical situations.

6.4.2 Let's go!

Exercise 6.9 Inheritance and Overloading. This exercise aims at testing your understanding of inheritance and overloading.

Question 1. Assume you created the following class.

```
class Address:
  def __init__(self, street, number):
    self.street= street
    self.number = number
```

Create another class `WorkAddress` deriving from `Address` that:

(a) Has an additional attribute `officeNumber`;

(b) Whose attribute `street` has always the value `"S.P. Andersen vegen"`.

Question 2. What will be the output of the following script.

```
class A:
  def __init__(self, value):
    self.value = value
    self.Next()

  def Next(self):
    self.i *= 2;

class B(A):
  def __init__(self, value):
    A.__init__(self, value)

  def Next(self):
    self.value *= 3;

b = B(1)
print("b.value = " + str(b.value))
```

Exercise 6.10 Shapes. This exercise is a classic of object oriented programming.

Assume we want to handle in a uniform way geometric shapes such as squares, rectangles, circles... More specifically, we want to perform the following operations on these shapes.

– Create a shape and give it a name.

- Calculate the perimeter of a shape.
- Calculate the area of a shape.

Question 1. Design a object-oriented solution for shapes.

Question 2. Create a class `ShapeManager` that manages a set of shapes. A `ShapeManager` should provide methods to:

- Create a shape with a given name and add it to the set.
- Remove the shape with a given name from the set.
- Extract the set of perimeters of the shapes.
- Extract the set of areas of the shapes.

Chapter 7

Formatting Data

Key Concepts
- Spreadsheets
- CSV, TSV formats
- `openpyxl` package
- Markup languages
- XML
- The `xml.dom.minidom` package.
- HTML and CSS

This chapter introduces different ways of managing structured data. First, it considers spreadsheets, then XML structures, finally HTML pages.

7.1 Illustrative Case Study

Throughout this chapter, we shall use the case study 4 presented in Appendix A.2 to illustrate the problems at stake and the solutions provided by Python modules that deal with spreadsheets and XML and HTML files.

The main question we shall discuss is how to store the data of Table A.1 so to make it easy to:
- Make data persistent, i.e. to be able to save them into a file on the hard disk of our computer (or any other convenient device) and later load them this file;
- To modify these data by means of some functions as well as to perform on them various statistics;
- To display nicely onto the computer screen.

We shall design a Python program that implements these functionalities.

Each of the above points may call a different answer. For this reason, it is always a good idea to distinguish the internal representation of data from the way they are made persistent and displayed onto the computer screen.

The first step of the design of our program consists thus in defining a data structure to store internally the data. This has been the subject of Problem 6.2 that spreads over the first two sections of Chapter 6. We can thus now focus on interfaces.

7.2 Spreadsheets

7.2.1 Description

Spreadsheets are interactive computer application for organization, analysis and storage of data in tabular form. The most popular spreadsheet program is Microsoft Excel®. Spreadsheets have been developed as

computerized analogs of paper accounting worksheets. The program operates on data entered in cells of a table. Each cell may contain either numeric or text data, or the results of formulas that automatically calculate and display a value based on the contents of other cells.

Spreadsheets are widely used in all engineering disciplines, not to speak indeed about accounting, marketing, sales...

Spreadsheets store data into files at their own format. There exists several Python packages to read and write spreadsheet documents. In this chapter, we shall use the popular `openpyxl` package that works for Microsoft Excel®.

7.2.2 CSV and TSV formats

Most of the spreadsheets make also possible to store data in a simplified textual format called *CSV* or *TSV*. These simplified format store only data and loose all what regarding formatting like fonts and sizes of characters, colors of foreground and background and the like. They are nevertheless convenient as they make it possible to transfer data without have to bother about tool licensing issues.

CSV and TSV files are text files, which means that they can be read and write not only by spreadsheets but also by any text editor. CSV stands for comma separated values and TSV for tabulation separated values. Each row of the table is stored as a line of the CSV/TSV file. Values stored in the cells of arrow are separated with commas/tabulation characters in the file. This implies indeed that values do not contain the character used to separate values.

Exercise 7.1 TSV Interface for the Retailer Sales. We can now to complete the data structure developed in Problem 6.2 (and subsequent problems) with a TSV interface.

A TSV that encodes Table A.1 is very similar to the table itself: each row of the table is encoded by a line of the file and columns are delimited by tabulations. The file `sales.tsv` contains thus the same information as Table A.1.

Question 1. Design a class `TSVParser` that implements methods to import retailer sales from a text file at TSV format.

Question 2. Design a class `TSVPrinter` that implements methods to export retailer sales in a text file at TSV format.

Question 3. Test the classes `TSVParser` by importing the file `sales.tsv` into an instance of `Retailer` and exporting the content of this instance into a file `sales-copy.tsv`.

Hint: It is strongly suggested to split you program into three modules:
- A module `Retailer` (from Problem 6.2) that implements the internal data structure.
- A module `TSVInterface` that implements the TSV interface.
- A module `RetailerSales` that implements the actions to be realized (creation of an instance of the class `Retailer`, import and export of TSV files...

Exercise 7.2 Calculating the Profit of the Retailer. We shall go now a step further, by calculating the profit realized each month on each product.

Question 1. Add a field `profit` to the classes `MonthlySale` and the `YearlySale`. as well as methods `UpdateProfit` to the four classes to update these fields, i.e. to calculate respectively the monthly profit and the yearly profit realized on a product, for each product.

Question 2. Create a method in the class `TSVPrinter` that prints out the monthly profit realized on each product. This method should print the profit in the same way as the quantities of sales are printed out. In addition, this method should print an extra column on the right to print out the yearly profit of each product, and a line at the bottom of the file to print out the profit realized each month by the retailer.

Exercise 7.3 TSV Interface for the Student Cohort. We can come back our case study "Student Cohort" presented in Appendix A.4 and already treated in problem 5.4 and Chapter 6.

Question 1. Create a module `TSVInterface` that implements the TSV interface for the "Student Cohort" case study.

7.2.3 The `openpyxl` Module

Working with TSV or CSV files works fine, but requires quite a "manual" operations, like saving and loading spreadsheets into and from these formats. Consequently, we would like to work directly with spreadsheets.

`openpyxl` is a Python library to read/write Excel 2010 xlsx/xlsm/xltx/xltm files. It was born from lack of existing library to read/write natively from Python the Office Open XML format. A full description of this library is available at `https://openpyxl.readthedocs.io/en/stable/`.

Installation

To install the `openpyxl`, you have to use the `pip` installation program. To do so, open a command window and go to directory in which Python and `pip` are installed, then enter the following command.

```
pip install openpyxl
```

Workbooks and Worksheets

Once installed, handling spreadsheets via `openpyxl` is rather simple. It relies on the class `Workbook`, which is the internal representation of an Excel document. There is no need to create a file on the file system to get started with `openpyxl`. Just instantiate the `Workbook` class and start work:

```
1 workbook = openpyxl.Workbook()
```

A workbook manages worksheets. It is always created with at least one worksheet. You can get it by using the `Workbook.active` property, create new ones and give them titles:

```
1 worksheet1 = workbook.active
2 worksheet2 = workbook.create_sheet("MySecondWorksheet")
3 worksheet1.title = "MyFirstWorksheet"
```

Once you gave a worksheet a name, you can get it as a key of the workbook. For instance, our first worksheet can be accessed from the workbook as `workbook["MyFirstWorksheet"]`, i.e. the workbook works as a dictionary of worksheets. You can therefore iterate on its elements. Other methods are available to manage worksheets, see `openpyxl` documentation for details.

Cells

Cells of a worksheet can be accessed in different ways. The first one consists in using a (string) key, e.g.

```
profitPerSale = worksheet1['B3']
worksheet1['C4'] = 2345
```

There is also the `Worksheet.cell()` method. This provides access to cells using row and column notation:

```
salesForFebruary = ws.cell(row=3, column=4, value=2345)
```

The above code creates the cell (3, 4) and fills it with value 2345.

When a worksheet is created in memory, it contains no cells. They are created when first accessed. Because of that, a code like the following creates 100×100 cells in memory, for nothing.

```
for x in range(1, 101):
  for y in range(1, 101):
    ws.cell(row=x, column=y)
```

Saving and Loading Workbooks into Files

Saving workbooks into files and loading them from files is pretty easy. It works as in the following example.

```
workbook = openpyxl.load_workbook("sales.xlsx")
workbook.active['C4'] = 2345
workbook.save("sales-modified.xlsx")
```

Formulas

Cells can contain operations applying to other cells (like sum, product, mean...), just like in usual spreadsheets. The syntax is the same: the content of the cell is a string with an equal ("=") sign followed by the formula describing the operation.

Assume for instance that we want to sum the contents of cells of the row `C` from column `2` to column `13` and put result in the column `14` (of the command `C`). This can be done as follows.

```
workbook.active['C14'] = "=SUM(C2,C13)"
```

The `openpyxl` module provides quite a few other features. Look to the documentation for more on this topic.

Working with Styles

Styles are used to change the look of data when displayed on screen (or printed out onto paper). They are also used to format numbers. Styles can be applied to the following aspects (among others):

- Font, size, color, underlining...;
- Background colors, borders...;
- Alignment.

Using styles requires to import the module `openpyxl.styles`. Here follows an example of use.

```
1 c2 = workbook.active['c2']
2 c2.font = openpyxl.styles.Font(color=openpyxl.styles.colors.RED,
3   italic=True)
```

7.2.4 Let's Go

The following series of exercises aims at designing a spreadsheet interface to our `RetailerSales` program.

Exercise 7.4 XLS Interface for the Retailer Sales. We shall create a `XLSInterface` module to implement this interface. This module echoes the `TSVInterface` module we have already created.

Question 1. Design a class `XLSParser` that implements methods to import retailer sales from an Excel spreadsheet.

Question 2. Design a class `XLSPrinter` that implements methods to export retailer sales in an Excel spreadsheet.

Question 3. Cross check your `TSVInterface` and `XSLInterface` by importing a file at the TSV format and saving it as an Excel spreadsheet and vice-versa.

Exercise 7.5 Calculating the Profit of the Retailer. We shall go now a step further, by calculating the profit realized each month on each product.

Question 1. Design a new method of your `XLSPrinter` that exports sales in one worksheet and profits in another one. Profits must be organized as in Exercise 7.2. They must be calculated by means of formulas from the worksheet storing sales.

7.3 XML

7.3.1 Description

Spreadsheets are very useful to deal with relatively simple data. When the problem at stake gets more complex, with structured data, other means must be found. One of them consists in the *eXtended Markup Language*, *XML*, specified by the World Wide Web Consortium in form of free open standards.

XML defines a set of rules for encoding documents in a textual data format that is supposedly both human-readable and machine-readable. In reality, it is mainly machine-readable. Moreover, although the design of XML focuses on documents, the language is widely used for the representation of arbitrary data structures such as those used in web services.

Several schema systems exist to facilitate the definition of XML-based languages, while programmers have developed many application programming interfaces (APIs) to aid the processing of XML data.

Figure 7.1 shows an example of XML file.

Line 1 is a header that starts all XML files. The real things start thus line 2. A XML description can be seen as as tree. In our example, the unique root of this tree is the node `RetailerSales`. The node `RetailerSales` has a unique child, the node `product` (defined line 3). The node `product` has in

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <retailerSales>
3    <product reference="BVR724" >
4      <profitPerSale value="1.2" />
5      <sales >
6        <month trigram="Jan" quantity="4939" />
7        <month trigram="Feb" quantity="4521" />
8        <month trigram="Mar" quantity="4767" />
9        <month trigram="Apr" quantity="4992" />
10       <month trigram="May" quantity="5151" />
11       <month trigram="Jun" quantity="5378" />
12       <month trigram="Jul" quantity="4578" />
13       <month trigram="Aug" quantity="4634" />
14       <month trigram="Sep" quantity="5210" />
15       <month trigram="Oct" quantity="4470" />
16       <month trigram="Nov" quantity="5376" />
17       <month trigram="Dec" quantity="4764" />
18     </sales >
19   </product >
20 </retailerSales >
```

Figure 7.1: A XML file

turn two children: the nodes `profitPerSale` and `sales`. And so on. In the XML jargon, nodes are called *elements*.

The characters making up an XML document are divided into markup and content. They are distinguished by the application of simple syntactic rules. Generally, strings that constitute markup either begin with the character "<" and end with a ">".

Elements are delimited by *tags*. Tags come in three flavors:

- Tags that start the description of an element such as `<product>` (line 3);
- Tags that end the description of an element such as `</product>` (line 19);
- Tags that both start and end the description of an element (with no child), such as `<month ... />` (line 6).

Elements have names, like `RetailerSales`, `product`, `month`.... The syntax for names is quite flexible. The are made of any sequence of characters, avoiding characters that are used as delimiters, like "<", ">", "", "/"... It is however recommended to use the same type of identifiers as, for instance, in Python.

Attributes, i.e. pairs (name, value), can be associated with elements. For instance, the element `product` line 3, has one attribute whose name is `reference` and whose value is `BVR724`. Attribute values are delimited with quotes. Attributes can be any text, including text representing floating point numbers (line in Line 4), integers (line in line 6 or simply any text. A node may have any number of attributes, however two attributes cannot have the same name.

Note that a node may have several children with the same name, like in lines 6 and 7.

Note also that terminal nodes can be also free text, data and comments. We shall not use them in this chapter. The reader interested in more details should consult a tutorial on XML, like those provided by the W3C at `https://www.w3schools.com/xml/`.

7.3.2 Parsers and Printers

The code given in Figure 7.1 shows that XML is well suited to store hierarchical data structures. But the main interest of XML stands in the availability of ready-made parsers and printers for many programming languages, including of course Python. These parsers can be given additional grammatical rules that the

XML description should obey. We shall not use this functionality here, but it is highly recommended to have a look on XML schemas, for instance at https://www.w3schools.com/xml/schema_intro.asp/.

There are actually two types of parsers for XML:

- *DOM parsers* which read a XML file and store its content into an internal data structure called a *DOM tree*. DOM stands for Document Object Model.
- *SAX parsers* which read a XML file and perform user defined actions at the beginning and the end of tags. SAX stands for Simple API for XML, where API stands for Application Programming Interface.

The former are much easier to use than the latter. Moreover, DOM trees can be created, modified and printed out into file. Consequently, we shall use them in the framework of this chapter (and this book). SAX parsers are of interest to read very large descriptions that cannot be stored into intermediate data structures such as DOM trees, for efficiency reasons.

Several packages are available to handle DOM trees in Python. In the sequel, we shall use the package `xml.dom.minidom`, which is probably the simplest. `minidom` stands actually for Minimal DOM implementation. This package is described at https://docs.python.org/3.8/library/xml.dom.minidom.html.

Parsing

To read a XML document stored into a text file `fileName.xml`, it suffices to call the function `parse`:

```
1 xmlDocument = xml.dom.minidom.parse("fileName.xml")
```

This function returns an instance of the class `Document` if the file contains a correct XML description and raises an exception otherwise (we shall come back on this point).

A document composes a root node which is stored in the field `documentElement`.

All XML items, i.e. elements, attributes, comments, data, texts... are encoded by means of instances of the class `Node`. The field `nodeType` contains the type of the node, i.e.

- `xml.dom.minidom.Node.ELEMENT_NODE` if the node is an instance of the class `Element`;
- `xml.dom.minidom.Node.ATTRIBUTE_NODE` if the node is an instance of the class `Attr` (attribute);
- `xml.dom.minidom.Node.TEXT_NODE` if the node is an instance of the class `Text`;
- and so on.

The classes `Element`, `Attr`, `Text`... derive from the class `Node`.

Links between nodes are described in Figure 7.2. Fields `parentNode`, `firstChild`, `lastChild`, `nextSibling` and `previousChild` are used to navigate into the DOM tree.

The name of an element is stored in the field `tagName`.

The simplest way to access and to modify attributes of an element is by using the following methods.

- `hasAttribute(name)`: this method returns `True` if the element has an attribute named by `name` and `False` otherwise.
- `getAttribute(name)`: this method returns the value of the attribute named by `name` as a string. If no such attribute exists, an empty string is returned, as if the attribute had no value.
- `setAttribute(name, value)` this method sets the attribute value from a string.

Parsing a XML file consists thus into two steps: first, the file is loaded into a document. Then, this document is navigated so to create the suitable data structures.

There is one pitfall you should be aware of: when parsing the XML document, the minidom parse creates a text node each time it encounters a text, including a sequence of white space, tabulation or end of line characters. This is quite unfortunate that no option is available to prevent this behavior, but that is what the XML standard prescribes. These text nodes have to be ignored when navigating in the tree.

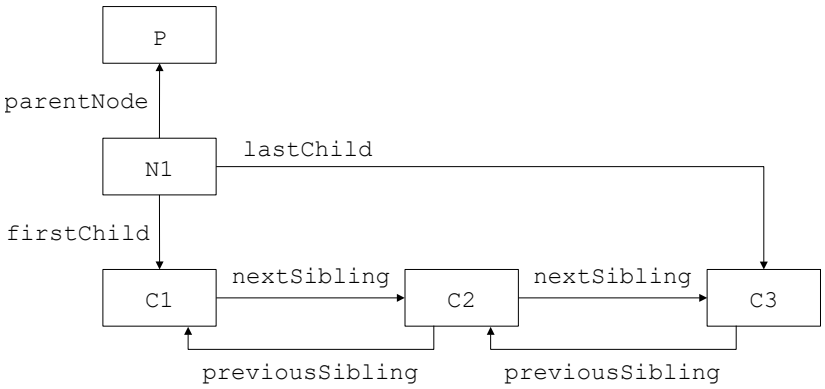


Figure 7.2: Links between XML nodes

Printing

Printing out an internal data structure as a XML file works the reverse way, i.e. first, a XML document is created from the data structure, and then the XML document is printed out into a string using the function `toprettyxml`:

```
1 outputFile.write(xmlDocument.toprettyxml("\t"))
```

`toprettyxml` takes the string used to indent the XML code as argument. In the above code, the indentation is thus a tabulation.

For technical reasons, creating a document is performed into two steps: first, an `implementation` is created (no matter what this is), then the document itself is created:

```
1 xmlImplementation = xml.dom.minidom.getDOMImplementation()
2 xmlDocument = xmlImplementation.createDocument(None, "retailerSales",
3   None)
```

We shall not describe here the three arguments taken by the method `createDocument`. The important point here is that the second argument is the tag name of the root element of the description.

7.3.3 Let's Go

The following series of exercises aims at designing a XML interface to our `RetailerSales` program.

Exercise 7.6 XML Interface for the Retailer Sales. We shall create a `XMLInterface` module to implement this interface. This module echoes the `TSVInterface` and `XLSInterface` modules we have already created.

Question 1. Design a class `XMLParser` that implements methods to import retailer sales from a XML text file.

Question 2. Design a class `XMLPrinter` that implements methods to export retailer sales in a XML text file.

Question 3. Cross check your modules `TSVInterface`, `XSLInterface` and `XMLInterface` by importing a file at one format and saving it at another format.

7.4 HTML

7.4.1 Description

It is often the case that one needs to make data and texts available to a wide audience. One of the solution to do that consists in creating a dedicated web-site, or at least some dedicated pages in a web-site.

HTML is the standard markup language used to create web-pages. HTML stands for Hyper Text Markup Language. HTML files are just text files with a special syntax. Python can be used to generate such pages out of some contents.

The W3C provides a very good tutorial on HTML: `https://www.w3schools.com/html/`. Therefore, we shall give here only basic elements.

HTML provides a number of constructs, called *tags* to structure an hyper-text:

– Tags to create paragraphs and titles at different hierarchical levels.
– Tags to display various types of lists.
– Tags to display special elements such as images, tables, pieces of computer code...
– Tags to create hyper-links within the page or referring to elements in external pages.
– Tags to create forms to be filled by the reader.
– ...

There exist several variants of HTML, for historical reasons. We shall present below only constructs that can be displayed in all web browsers. We shall not present all of them, just enough to make the exercises.

Sections and Paragraphs

Figure 7.3 shows a small HTML code.

Each HTML element is surrounded by an opening tag in the form `<xxx ...>` and a closing tag in the form `</xxx>`, where `xxx` is the type of the element.

The code of Figure 7.3 starts with a header that includes the title of the page. Then comes the body of the page.

In the example, the body is decomposed into several sections. The titles of sections are delimited with the tag `h1` while those of subsections are delimited with the tag `h2`. There can be up to six levels of sectioning.

To format the text of a web page in paragraphs, one uses the tags `<p>` and `</p>`. This means *a contrario* that sequences of space, tabulation and end of line characters are just ignored, i.e. considered as a single space character.

Note that the opening tags of sections contain an attribute `id`. We shall explain its use soon.

Entities

Reserved characters in HTML (like < and >) must be replaced with *character entities*, entities for short. Characters that are not directly available can also be replaced by entities.

The general form of entities is `&name;`.

Here follows a list of frequently used entities:

– Less than <: `&lt;`.
– Greater than >: `&gt;`.
– Ampersand &: `&amp;`.
– Apostrophe ': `&apos;`.
– Quote ": `&quot;`.
– Non-breaking space: ` `.

Accents and letters from other alphabets are also encoded by means of entities, e.g.

– Character é: `&eacute;`.

```
<!DOCTYPE html>
<html>
<head>
  <title>Retailer Sales</title>
</head>
<body>

<h1 id="Introduction">Introduction</h1>
<p>This web-page presents the annual performance of
our favorite retailer.</p>
<p>For now, nothing fancy, but it will get better soon</p>

<h1 id="SalesAndProfits">Sales and Profits</h1>
<p>We shall first present sales, then profits.</p>

<h2 id="Sales">Sales</h2>
<p>To be precised</p>

<h2 id="Profits">Profits</h2>
<p>Hopefully big</p>

<h1 id="Conclusion">Conclusion</h1>
<p>Well, it seems that everything went fine</p>

</body>
</html>
```

Figure 7.3: A small HTML code to illustrate paragraphs and sections

- Character è: `&egrave;`.
- Character ê: `&ecirc;`.
- Character ä: `&auml;`.
- Character å: `&acirc;`.
- Character æ: `&aelig;`.
- Character ø: `&oslash;`.
- Character α: `&alpha;`.
- Character π: `&pi;`.

Lists

HTML provides constructs to create *unordered, ordered and description lists*. For instance, the code to create an unordered list is as follows.

```
<ul>
  <li> First item of the list.</li>
  <li> Second item of the list.</li>
  <li> ....</li>
</ul>
```

Ordered lists are obtained by using the tags `<ol>` and `</ol>` instead of `<ul>` and `</ul>`.

It is possible to change the list item marker (in unordered list) and the number (in ordered list), see documentation.

Tables

In HTML, tables are defined with the tag `<table>`.

Each row of the table is defined with the tag `<tr>`. A table header is defined with the tag `<th>`. Finally, table data/cells are defined with the tag `<td>`.

By default, table headings are bold and centered.

Figure 7.4 shows an example of HTML code for a table.

```
<table>
  <tr>
    <th>First name</th>
    <th>Last name</th>
    <th>Age</th>
  </tr>
  <tr>
    <td>Alice</td>
    <td>Spring</td>
    <td>97</td>
  </tr>
  <tr>
    <td>John</td>
    <td>Doe</td>
    <td>45</td>
  </tr>
</table>
```

Figure 7.4: A small HTML code to illustrate paragraphs and sections

Hyperlinks

Hyperlinks are found in virtually all web page. They allow readers to navigate within pages and from page to page.

Hyperlinks are defined with the tag `<a>`:

```
You can find some documentation on HTML hyperlinks
<a href="https://www.w3schools.com/html/html_links.asp">here</a>.
```

By clicking on `here`, the reader goes to the web page specified with the attribute `href`.

The example above used an absolute URL (URL stands for Uniform Resource Locator, i.e. a full web address). It it also possible to refer to a file located on your computer. To do so, it suffices to give its path, starting from the folder where the web page is located, e.g.

```
To find some more about our retailer case study click
<a href="../FormattingData/RetailerSales/description.html">here</a>.
```

It it finally possible to refer a location within the current web page (and more generally within any web page). This is where the `id` attributes of the code given in Figure 7.3 come into the play. It works as follows.

```
Annual sales are detailed <a href="#Sales">here</a>.
```

Clicking on `here` makes the browser go to the element whose attribute `id` is `Sales`.

Images

In HTML, images are introduced with the tag `<img>`. This tag contains attributes only, and does not have a closing tag.

The attribute `src` specifies the URL (web address) of the image. The attribute `alt` provides an alternate text for an image, if the reader cannot view it for some reason. This attribute is optional.

```
<img src="Pictures/Gardenia.jpg" alt="Gardenia flower" >
```

It also possible to specify the width and the height of the image in pixels via the corresponding attributes:

```
<img src="Pictures/Gardenia.jpg" alt="Gardenia flower"
    width="400" height="600" >
```

Cascading style sheets we shall review now provide more stylistic possibilities.

7.4.2 Cascading Style Sheets

So far, we focused on the content of web pages, not on their look and feel. The latter is however very important to ensure a good communication.

HTML provides special constructs to change the appearance and the organization of the text. For instance, the tag `bold` makes it possible to write the corresponding text in bold font. Mixing up the content with instructions regarding its presentation is however not recommended: it is prone to stylistic inconsistencies and is very hard to maintain.

Cascading style sheets (CSS) solve this issue. They are used to specify, in a separate file, how the various elements of a page should be displayed: size, font and style of characters, text and background colors, space before and after the element...

Separating the logical structure of the text from the way it should be displayed makes the creation and the maintenance of web-pages much easier as if everything was intertwined. Moreover, the same style can be used for different web pages.

To use an external style sheet `myStyle.css`, it suffices to add a link to it in the `<head>` section of the HTML page:

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="myStyle.css">
</head>
<body>
   ⋮
</body>
</html>
```

A cascading style sheet is simply a text file with the extension `.css`. It may look as follows.

```
body {
  background-color: Cyan;
}
h1 {
  background-color: DarkBlue;
  color: White;
}
h2 {
  background-color: DarkBlue;
  color: Yellow;
}
p {
  color: black;
}
```

Text Appearance

CSS properties make it possible to modify the appearance of a textual element. E.g.

- `color`: color of the text;
- `font-family`: font of the text;
- `font-size`: size of the text, which can be given in percentage or in pixels;
- `font-style`: style of the text, which is either `normal`, `italic` or `oblique`;
- `font-weight`: weight of the text, which is either `normal`, `bold` or number;
- `font-variant`: is either `normal` or `small-caps`;
- `text-decoration`: is either `none`, `underline`, `overline`, `line-through` or several of those.
- `text-align`: is either `left`, `right`, `center` or `justify`.
- ...

For instance:

```
h1 {
  color: White;
  font-family: verdana;
  font-size: 300%;
  }
```

Advanced Features

CSS makes it possible to do much more than just highlighting textual elements, including positioning elements into the page, defining independent parts of text and so on.

By combining HTML and CSS, you can define really nice web sites.

We shall not say more here. The interested reader should refer to CSS.

7.4.3 Let's go!

Problem 7.7 HTML Interface for the Retailer Sales. We shall create a `HTMLInterface` module to implement this interface. To do so, we shall proceed in two steps: first, we shall design a `HTMLSerializer` that creates HTML strings for elements of interest such as the list of products and the table of sales. Second, we shall design a `HTMLCreator` that reads a template file and produces a HTML target file by inserting HTML strings produces by our `HTMLSerializer` at the suitable

places.

Let us start with the serializer.

Question 1. Design a class `HTMLSerializer` that implements a method `SerializeListOfProducts` that returns a string containing the HTML list of the products under scrutiny. The profit realized on each sale should be given together with the name of the product.

Question 2. Add a method `SerializeTableOfSales` to the `HTMLSerializer` that returns a HTML string containing the table of monthly sales of products.

Question 3. Add a method `SerializeTableOfProfits` to the `HTMLSerializer` that returns a HTML string containing the table of profits organized as in Exercise 7.2.

We can now design the `HTMLCreator`.

Question 4. Design a class `HTMLCreator` with a method that:
- reads line by line the HTML template file `sales-template.html`;
- substitutes:
  - The list of products obtained by the method `SerializeListOfProducts` for the text `__PRODUCTS__`;
  - The table of sales obtained by the method `SerializeTableOfSales` for the text `__SALES__`;
  - The table of profits obtained by the method `SerializeTableOfProfits` for the text `__PROFITS__`;
- prints out the result into a target HTML file.

Question 5. Define a cascading style sheet so to present nicely your web page.

7.5 Additional Problems and Exercises

Problem 7.8 Universal Declaration of Human Rights. For this problem, we shall come back to the Universal Declaration of Human Rights. The objective is to create a nicely presented web-page for this declaration.

Question 1. Design a Python script that reads the text file `Universal Declaration of Human Rights.txt` and stores its content into a data structure. This script must work for source text file similar to the given example.
Hint: A declaration is made of a preamble and a series of articles. Each article has a number and is made of a series of paragraph.

Question 2. Design a printer that displays the content of the data structure as a HTML page.

Question 3. Design a cascading style sheet (CSS) to obtain a nice display or your web-page.

Problem 7.9 Gamebook. A gamebook is a fiction that allows the reader to participate in the story by making choices. The narrative branches along various paths, typically through the use of numbered paragraphs or pages. For this problem, we shall consider a very simple scenario, found on the web. The objective is to create a nicely presented web-page for this story.

Question 1. Design a Python script that reads the text file `gamebook-example.txt` and stores its content into a data structure. This script must work for source text file similar to the given example.
Hint: A scenario is made of scenes Each scenario has a number, starts with a description, and ends with a possibly empty list of choices. Each choice has a description and sends to a scene.

Question 2. Design a printer that displays the story as a HTML page, with hyperlinks to make choices.

Question 3. Design a cascading style sheet (CSS) to obtain a nice display or your web-page.

Question 4. Same question as above, but each scene should be encoded in a different page.

Part II

Performing Experiments

Chapter 8

Randomized Experiments

Key Concepts
- Management of uncertainties
- Sensitivity analyses
- Randomness, randomized Experiments
- Pseudo-random number generators
- Moments, mean, standard-deviation, confidence interval
- Distribution, histogram, quantiles
- Parametric distribution, Kaplan-Meier indicator
- Stochastic simulation, Monte-Carlo simulation
- Trial, Bernouilli trial

8.1 Introduction

As pointed out in the introduction of this document, one of the key advantage of computer-based experiments is their low cost, at least compared to the one of physical experiments. It is thus in general possible to perform many experiments, within reasonable amount of computational resources, typically by making input parameters varying a bit around their nominal value.

Consider a program P which depends on a set of input parameters $\overrightarrow{i}$ and produces a set of output parameters $\overrightarrow{o}$. Assume moreover that we know, for each input parameter i of $\overrightarrow{i}$ a probability distribution $d(i)$. Then, we can:

1. Draw at random a sample of values of $\overrightarrow{i}$, according to the distributions $d(i)$'s.
2. Calculate the values of output parameters $\overrightarrow{o}$ for each set of values of $\overrightarrow{i}$ in the sample.
3. Calculate statistics on values $\overrightarrow{o}$.

The general principle is called *stochastic simulation*, or *Monte-Carlo simulation*. Each step consisting in drawing a set of values for input parameters and calculating the output parameters is called a *trial*.

This raises however immediately a question: why performing many experiments if the phenomenon at stake is essentially deterministic and consequently one experiment *a priori* suffices? There are actually good reasons to do so, which are worth to review.

First, many phenomena at stake in engineering are in essence non-deterministic. Take, for instance, the failure of a physical component such as a pump, an engine or a valve. We cannot know with precision when this failure will happen, even though we can monitor the condition of the component. Safety and reliability assessments rely thus by definition on stochastic models. Note that there are two types of uncertainties in this case. First, the so-called epistemic uncertainty: the reason for which the component fails may be uncertain in this sense that the physical phenomena at stake may be only partially understood. Second, the so-called aleatory uncertainty: one simply does not know when the component will fail, even though one can make robust statistics on this event, based on experience feedback on operation of fleet of

similar components working in similar conditions. These two types of uncertainties often combine: one often associates probability distributions to the failures of components, but these probability distributions are themselves known only up to some uncertainty.

Second, in some cases, phenomena at stake, although fully deterministic, are sensitive to initial conditions. This has been pointed out by Poincaré in his famous answer to Laplace:

> A very small cause, that we cannot see, may determine an effect that we cannot not to see. In this case, we say that this effect is at random. True that if we knew exactly all laws of the Nature and the exact state of the Universe, we could predict its evolution. But, even assuming that laws of Nature would have no more secret for us, we could only know approximately the state of the Universe. If this principle makes it possible for us to predict the evolution of a phenomenon with the same approximation as we know its initial condition, we use to say that this evolution obeys laws. But not all phenomena are such. It may be the case that small differences in initial conditions produce very big differences in the evolution. A small approximation on the initial conditions would thus induce an enormous mistake in prediction. The prediction is then impossible and we face a random phenomenon.

We shall see examples of such situations later in the chapter and this document. In any case, it is often a good idea, from a methodological standpoint, to test how robust are the conclusions we can draw from the experiments we are performing. By making initial conditions vary slightly at random, we can assess this robustness.

Third, there are many calculations for which no analytical (close-form) solution is available, either because such a solution does not exist, or because it would awfully complex to obtain and to verify. Typically, most of the systems of differential equations enter into either of these categories. Similarly, many the resolution of many discrete problems is so computationally complex, that finding solutions is out of the reach of any current or future technology. In these cases, stochastic simulation is often the only tool at hand.

These are the reason why randomness and computer-based experiments are intrinsically linked. It remains two important issues: first, what does randomness mean in computer-based experiments? second, how to make conclusions out of randomized experiments?

In the remainder of this chapter, we shall look in turn these two questions, starting with the former and the notion of pseudo-random number generators.

8.2 Pseudo-Random Number Generators

8.2.1 Rational

As pointed out in the previous section, in many operational situations, we face an uncertain environment and nevertheless we have to make decisions. *A priori*, this seems impossible: how to make a decision if we do not know what will happen and what will be the consequences of our decisions?

In many cases however, we have at hand some information about the system under study and its environment, in form of statistical data resulting of operation feedback. We can use these data to design stochastic models, i.e. models in which the system evolves at random.

The randomness in stochastic models is controlled in two ways: first, evolutions these models describe are not purely random, but governed by the statistical data; second, they are reproducible. In a word, they are only pseudo-random evolutions. Or to put things differently, the models are purely deterministic but the part that provides series of random numbers, if the same series is provided to the program twice, it will provide exactly the same results.

8.2.2 Description

A *random-number generator* is a device that generates a sequence of numbers that cannot be reasonably predicted better than by a random chance. This definition is somehow circular as it relies on the concept of random chance, but the intuitive idea is there. A full mathematical treatment of the subject goes much beyond the scope of this book.

Random number generators can be hardware random-number generators, which generate genuinely random numbers, or pseudo-random number generators, i.e. algorithms which generate series of numbers which look random, but are actually deterministic. The former are not very convenient, at least in the context of computer-based experiments, for they require to connect computers with such devices. The latter present not only the advantage of being cheaper, but also that series of numbers generated by the algorithm can be reproduced at will.

It remains to find "good" generation algorithms, i.e. algorithms that mimic as much as possible "true" randomness, while being not too expensive from a computational view point. Algorithmic generators can actually suffer from several defects:

- Lack of uniformity of distribution for large quantities of generated numbers;
- Correlation of successive values;
- Poor dimensional distribution of the output sequence;
- The distances between where certain values occur may be distributed differently from those in a "true" random sequence distribution;

Congruential pseudo-random number generators (attempt to) fulfill the needs. A *congruential pseudo-random number generators* is a function f that takes an integer as input (coded onto a finite number of bits, e.g. 64 bits on modern computers), called the *seed*, and returns an integer. Given the initial seed z_0, it is thus possible to generate an arbitrary long sequence of numbers: $z_1 = f(z_0)$, $z_2 = f(z_1)$, ...

In the second half of the 20th century, the standard pseudo-random number generators were linear congruential generators, i.e. generators based on functions of the form:

$$f(z) \stackrel{def}{=} (a \times z + c) \mod m$$

Note that, because congruential generators work with only a finite number of integers (those coded by the machine), there necessary exist two indices i and j, $i < j$, such as $z_i = z_j$. Consequently, at some step, the generator loops. The distance between i and j is called the *period* of the generator. For some seeds, the period may be shorter than for some others.

The periodicity of congruential pseudo-random number generators was really an issue for computer-based experiments when integers were coded on 32 bits. For large number of tries (e.g. 1 million), they were good chances that the same executions were reproduced (as they started with the same seed). The problem was known to be inadequate, but better methods were unavailable.

Fortunately, this problem is now solved, due to the generalization of 64 bits machines, and more importantly to the introduction of techniques based on linear recurrences on the two-element field. With that respect, the invention of the Mersenne-Twister generator in 1997 (Matsumoto and Nishimura, 1998) was a major step forward. This generator (or a successor of thereof) is nowadays implemented in random number generation libraries of programming languages, including of course Python.

8.2.3 Let's go!

Exercise 8.1 Random Choices using the `random` module. The objective of this exercise is to make you use the different functions of the `random` module of the standard Python library.

Question 1. Write a Python script to output a random even number between 0 and 100 inclusive.

Question 2. Write a Python script to output a random number, which is divisible by 5 and 7, between

10 and 150 inclusive.
Hint: You may first generate the list of numbers verifying the property.

Question 3. Write a Python script to output a to generate a list with 10 random numbers between 10 and 50 inclusive, first without replacement, then with replacement.

Question 4. Recall that the cumulative distribution function of the Weibull distribution is defined as follows.

$$Weibull(t,\alpha,\beta) \quad = \quad 1-\exp\left(-\left(\frac{t}{\alpha}^{\beta}\right)\right)$$

where α and β are respectively the scale and the shape parameters (see Exercise 5.9 for more details).
Write a Python script to output a to generate a list with 10 random delays (floating point numbers) generated according the Weibull distribution of parameters $\alpha = 1000$ and $\beta = 3$.

Exercise 8.2 Random Choices using the `numpy` package. The objective of this exercise is to make you use the different functions of the `random` module of the `numpy` package.

Question 1. Write a NumPy script to generate 10 random numbers from the normal distribution.

Question 2. Write a NumPy script to generate 10 random integers between 0 and 2 inclusive.

Question 3. Write a NumPy program to create a 10x10 matrix with random values between 0 and 1. Find the minimum, maximum and mean values.

Exercise 8.3 Approximation of π. This exercise is a classic. The idea is to approximate the value of π by doing a series of randomized experiments. Here is the idea. The surface of a circle of radius r is $\pi \times r^2$. In case $r = 1$, it gives simply π. Now, if we pick up two numbers x and y between -1 and 1, the corresponding point (x,y) is in the circle if $x^2 + y^2 \leq 1$, i.e. if its distance to the origin is less than 1. Consequently if we choose x and y at uniformly at random in the interval $[-1,1]$, the probability that (x,y) is in the circle is π. If we draw in this way many points, we can approximate the value of π by the observed frequency at which the drawn points fell in the circle.

Remark that we can simplify the experiment by drawing our two points uniformly at random in $[0,1]$. Doing so, we can approximate the value of $\pi/4$.

Question 1. Design a Python script to assess the value of π according to the above method.

Question 2. Perform a series of experiments with different numbers of points. What do you observe?

Question 3. We can extend our remark that it suffices to draw values at random in $[0,1]$ as follows. Consider a point (x,y) drawn at uniformly random in $[0,1]^2$.

- With a probability $1/4$, both x and y will be less than $1/2$. In this case, we know, without calculating $x^2 + y^2$ that the point fells within the circle.
- With a probability $1/4$, $x \leq 1/2$ and $y > 1/2$.
- With a probability $1/4$, $x > 1/2$ and $y \leq 1/2$.
- With a probability $1/4$, both x and y will be bigger than $1/2$.

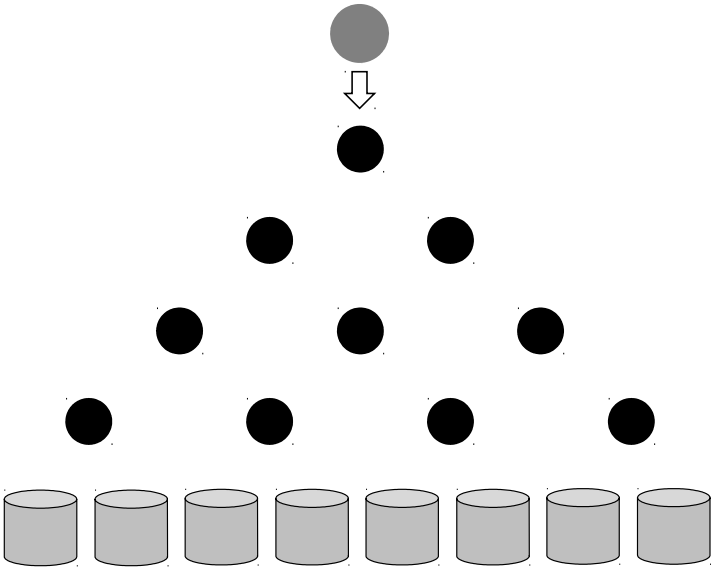


Figure 8.1: Physical device that calculates 2^{-3}.

What does this remark suggest? Design another Python script to assess the value of π more efficiently than the brute force simulation of Question 1..

Exercise 8.4 Falling Ball. At the "Cité des Sciences et de l'Industrie de la Vilette" (Paris, France), which is a science museum, the physical device pictured in Figure 8.1 is presented in the section dedicated to mathematics. It consists in a vertical board in which a number of plots (represented by black circles on the figure) are organized in a triangle. Balls are dropped vertically from above the top plot. A ball hitting a plot has (in theory at least) one chance over two to go to the right and one chance over two to go to the left. In both cases, it hits the plot of the rank below and eventually falls in one the buckets located below the board.

Question 1. Design a Python script to mimic this physical experiment (with an arbitrary depth).

Question 2. Perform experiments with your script. What do you observe?

8.3 Basic Statistics

Essentially two types of indicators can be observed when performing computer-based experiments:

- Numerical indicators that are assimilated to real-valued variables.
- Discrete indicators such as Boolean variables or variables taking their values into a small set of symbolic constants.

The statistics made on these two types of indicators are different. For numerical indicators, one is mostly interested in how the values of the indicator are distributed. For discrete indicators, one is mostly interested in the frequency of each value of the indicator.

We shall consider in turn both cases. But before doing so, an important practical remark must be made: To get significant results, one often needs to consider very large sample, i.e. number of experiments. But in such case, storing all individual values taken by the indicators would lead to a memory overflow. It

would be indeed possible to store values into an external memory (e.g. hard disk), but accesses to external memories are very slow. Doing so would therefore slow down programs. Consequently, statistics must be made using a reasonable amount of memory.

8.3.1 Moments

In statistics, a *moment* is a specific quantitative measure of the shape of a function. The mathematical concept is closely related to the concept of moment in physics.

The n-th moment μ_n of a real-valued continuous function f of a real variable about a value c is defined as follows.

$$\mu_n \stackrel{def}{=} \int_{-\infty}^{\infty} (x-c)^n f(x) dx \tag{8.1}$$

The moment of a function, without further details, refers usually to the above expression with $c = 0$.

Note that the n-moment does not necessarily exist (because the above integral may be infinite).

If f is a probability density function, then the value of the integral above is called the n-th moment of the probability distribution. The first moment is called the *mean*, the second moment is called the *variance*, the third moment is called the *skewness*, and the fourth moment is called the *kurtosis*.

More generally, if F is a cumulative probability distribution function of any probability distribution, which may not have a density function, then the n-th moment of the probability distribution is given by the following integral.

$$\mu'_n \stackrel{def}{=} E[X^n] = \int_{-\infty}^{\infty} x^n dF(x) \tag{8.2}$$

where X is a random variable that has this cumulative distribution F, and E is the expectation operator or mean.

Mean The *mean* of a probability distribution is thus the long-run arithmetic average value of a random variable having that distribution. In this context, it is also known as the *expected value*.

For a data set $S = \{x_1, x_2, \ldots, x_n\}$, the *arithmetic mean*, also called the *mathematical expectation* or *average*, typically denoted by $\bar{x}$ (pronounced "x bar"), is the sum of the values divided by the number of values in the data set.

$$\bar{x} \stackrel{def}{=} \frac{\sum_{i=1}^{n} x_i}{n} \tag{8.3}$$

$\bar{x}$ must be distinguished from the mean μ of the underlying distribution. However, the law of large numbers ensures that the larger the size of the sample, the more likely it is that its mean is close to the actual population mean.

As pointed out above, in practice, it may not be possible to store all of the x_i's of the sample. It is however always possible to calculate their sum "on-the-fly", i.e. each time a new value is obtained, it is added to the current sum. Then, when the mean is to be calculated, it suffices to divide the accumulated sum by the number of values in the sample.

Exercise 8.5 Wealth Distribution in Norway. The file `Norway2018.csv` contains the estimated net wealth in NOK per family of a sample of 100,000 Norwegian families[a] Some of the estimated net wealth are negative, meaning that the corresponding family has debts.

Question 1. Design a Python script that reads this file, stores the values in a `NumPy` array, and calculates the mean net wealth of families of the sample (using the suitable `NumPy` function).

Question 2. Design a Python script that reads this file and calculates on-the-fly the mean net wealth of families of the sample.

[a] These data have been randomly generated from actual statistics made by the *Statistik Sentralbyrå(SBD)*, in 2018.

Variance and Standard-Deviation As for the mean, we need to estimate the variance and the standard-deviation from the sample.

Recall that the *variance* of a random variable X, denoted $\mathrm{Var}(X)$, is the expectation of the squared deviation of X from its mean μ:

$$\mathrm{Var}(X) \overset{def}{=} E\left[(X-\mu)^2\right]$$

Intuitively, $\mathrm{Var}(X)$ measures how far a set of (random) numbers are spread out from their average value.

The *standard-deviation* of a random variable X, denoted $\sigma(X)$, is the square root of its variance.

$$\sigma(X) \overset{def}{=} \sqrt{\mathrm{Var}(X)}$$

The expression defining the variance can be expanded:

$$\begin{aligned}\mathrm{Var}(X) &\overset{def}{=} E\left[(X-E[X])^2\right]\\ &= E\left[X^2-2XE[X]+E[X]^2\right]\\ &= E[X^2]-2E[X]E[X]+E[X]^2\\ &= E[X^2]-E[X]^2\end{aligned}$$

The naïve algorithm to estimate the empirical variance $\overline{\mathrm{Var}}(X)$ (and the empirical standard deviation $\overline{\sigma}(X)$) from a sample consists simply in applying the above formula, i.e. in calculating the mean of the squares and the square of the mean of the values in the sample, and in dividing their difference by the number of values. To do so, it suffices to accumulate the sum of the squares of the values and the sum of the values, as for the mean.

$$\overline{\mathrm{Var}}(X) \overset{def}{=} \frac{\sum_{i=1}^{n} x_i^2}{n} - \left(\frac{\sum_{i=1}^{n} x_i}{n}\right)^2 \tag{8.4}$$

$$\overline{\sigma}(X) \overset{def}{=} \sqrt{\overline{\mathrm{Var}}(X)} \tag{8.5}$$

In case the two components of the right hand side of equation 8.4 are similar in magnitude, this algorithm may suffer from biases and numerical instability. Fortunately, there exist better algorithms to handle these cases. A first correction is provided by the Bessel's formula, which consists in multiplying the variance obtained by the naïve algorithm by a factor $\frac{n}{n-1}$. This does not solve however numerical instability. The Welford's online algorithm makes it possible to get a good estimate of the variance "on-the-fly", see e.g. (Knuth, 1998).

In most of practical cases however, the naïve algorithm is good enough.

Exercise 8.6 Wealth Distribution in Norway (Standard Deviation). We shall use the scripts designed for exercise 8.5 to calculate the standard-deviation of the net wealth distribution (per family) in Norway.

Question 1. Add the calculation of the standard-deviation to the Python script designed for exercise 8.5, question Question 1. (use the suitable `NumPy` function).

Question 2. Add the calculation of the standard-deviation to the Python script designed for exercise 8.5, question Question 2..

Error Factors and Confidence Intervals A *point estimate* is a single value given as the estimate of a population parameter of interest, for example the mean. In contrast, an *interval estimate* specifies a range within which the parameter is estimated to lie. *Confidence intervals*, also called *confidence ranges*, are commonly reported along with point estimates of the same parameters, to show the reliability of the estimates. A confidence interval comes with a confidence level, which specifies the probability that the actual value of the parameter lies in the given interval. For a same sample, the smaller the confidence range, the smaller the confidence level, and vice-versa, the larger the confidence level the larger the confidence range. Most commonly, the 95% confidence level is used. However, other confidence levels are also used, for example, the 90% and the 99% confidence levels.

The *margin of error* or *error factor* is usually defined as the "radius" (or half the width) of a confidence interval.

Let $\overline{x}$ and $\overline{\sigma}(x)$ be respectively the observed mean and standard-deviation on a sample of size n. Then, the error factor $\mathrm{EF}_\alpha(x)$ corresponding to a confidence level α is defined as follows.

$$\mathrm{EF}_\alpha(x) \stackrel{def}{=} t_\alpha \times \frac{\overline{\sigma}(x)}{\sqrt{n}} \tag{8.6}$$

The confidence interval $\mathrm{CI}_\alpha(x)$ is then defined as follows.

$$\mathrm{CI}_\alpha(x) \stackrel{def}{=} [\overline{x} - \mathrm{EF}_\alpha(x), \overline{x} + \mathrm{EF}_\alpha(x)] \tag{8.7}$$

The factor t_α is obtained, assuming a normal distribution of the values, by looking at the table defining the normal law. Typical values chosen for t_α are $t_{90\%} = 1.64$, $t_{95\%} = 1.96$ and $t_{99\%} = 2.58$.

Exercise 8.7 Wealth Distribution in Norway (Confidence Ranges). We shall use the scripts designed for exercise 8.5 to calculate confidence ranges on the net wealth distribution (per family) in Norway.

Question 1. Add the calculation of the 90%, 95% and 99% confidence ranges to the Python script designed for exercise 8.5, question Question 1. (use the suitable `NumPy` functions).

Question 2. Add the calculation of the 90%, 95% and 99% confidence ranges to the Python script designed for exercise 8.5, question Question 2..

The intervals calculated in exercise 8.7 can be interpreted as follows. There are 90 chances out of 100 that the average net wealth lies somewhere between 5378794 and 5633700 NOK, 95 chances out of 100 that it lies between 5353926 and 5658569 NOK, and finally 99 chances out of 100 that it lies between 5305742 and 5706752 NOK.

Note, and this is very important to understand, that the parameter that is estimated is the average net wealth of Norwegian families, not the net wealth itself.

Note also that these figures assume a normal distribution for the net wealth of Norwegian families. However, this is not verified in practice. The reason is that, although Norway is very egalitarian, compared to nearly all other countries in the world, there are still very rich people.

8.3.2 Histograms and Quantiles

For symbolic values, the calculation of the moments is sufficient. For numerical values, one may be interested in addition to the distribution of values. This can be done in two ways: either via the extraction of histograms or via the calculation of quantiles. We shall look at both in turn.

Histograms Consider a sample of values $X = \{x_1, x_2 \dots x_n\}$. A *histogram* H for X is built over a series of *bins*, i.e. contiguous intervals: $H = \{[h_1, h_2], [h_2, h_3] \dots [h_{k-1}, h_k]$ such that $h_j < h_{j+1}$ and all $x_i's$ are comprised between h_1 and h_k. Extracting the histogram consists then in counting the number of $x_i's$ lying in each bin. Although it is not strictly necessary, it is in general assumed that bins are of equal size.

Histograms can be very interesting as they summarize how the x_i's are distributed. They are a convenient abstraction of the sample.

In principle, extracting a histogram from a sample is thus rather simple. The extraction of the histogram is done in three steps:

1. The minimum and maximum values of the sample are determined (by scanning all x_i's).
2. The interval between the minimum value and the maximum value is split into k sub-intervals of equal size.
3. By scanning again all x_i's, one determines how many values lie in each interval.

A approximate discrete cumulative distribution can be obtained from the histogram by summing the number of values in intervals preceding the considered intervals.

Note that it is not necessary to store the values of the sample in computer memory to extract a sample. It suffices to scan them twice: once to determine the bins (by extracting minimum and maximum values of the sample), and once to count the number of values lying in each bin. This assumes however that values are stored somewhere, which is not always possible: in some experiments, values are just calculated but not stored. Histograms have thus to be extracted on-the-fly . A solution to this problem consists in choosing *a priori* the minimum and maximum values (and therefore the intervals) in such way that all values lie between them and that they are not too far from the actual minimum and maximum.

The main problem of extracting histograms stand however elsewhere, namely in the fact that values of the sample may be very unevenly distributed between the minimum and maximum values. In such cases, considering bins of equal size does not provide meaningful results, as illustrated by our wealth distribution example.

Exercise 8.8 Wealth Distribution in Norway (Histogram). We shall use the scripts designed for exercise 8.5 to extract histograms of the net wealth distribution (per family) in Norway.

Question 1. Using the Python script designed for exercise 8.5 (question Question 1.) to extract the histogram of the net wealth of Norwegian families first for 10, then for 20 bins (use the suitable `NumPy` functions).

Question 2. Same question but using the Python script designed for exercise 8.5 and extracting histograms on-the-fly.

Quantiles An alternative approach consists in calculating quantiles.

Quantiles are cut points dividing the range of a probability distribution into successive intervals with equal probabilities, or dividing the observations in a sample in the same way to estimate their values. Common quantiles have special names: *quartiles* (when the probability distribution is divided in 4), *deciles* (when it is divided in 10), *centiles* (when it is divided in 100). The *median* is the point such that half of the values in the sample are below and half are above, i.e. the sample is divided into two sub-intervals.

In principle, estimating quantiles is rather simple. Let $x_1, \dots, x_n$ the n numerical values in the sample. The extraction of quantiles is done in three steps:

1. The x_i's are sorted in ascendant order.
2. The sorted list of the x_i's is split into q sub-list of equal size (where q depends on which quantiles one wants to obtain).
3. The i-th q-quantile is the highest value in the i-th sorted sub-list.

Making the above principle to work on-the-fly, i.e. without recording all the x_i's is rather tricky: only approximated values can be obtained. A full presentation of algorithms that perform such on-the-fly calculation goes beyond the scope of this book. The main idea is to maintain successive bins of equal probabilities. Values accumulated in each bin are considered as random variables, for which a mean, a standard-deviation as well as extremum values can be calculated on-the-fly.

Exercise 8.9 Wealth Distribution in Norway (Quantiles). We shall use the scripts designed for exercise 8.5 to calculate quantiles of the net wealth distribution (per family) in Norway.

Question 1. Using the Python script designed for exercise 8.5 (question Question 1.) to extract quantiles 0.01, 0.05, 0.10, 0.33, 0.5, 0.66, 0.9, 0.95 and 0.99 of the net wealth of Norwegian families (use the suitable `NumPy` function).

Question 2. Draw the distribution of net wealth using quantiles obtained in the previous question and `Matplotlib`.

Drawing distributions as in question Question 2. is a perfect illustration of a quote sometimes attributed to Napoléon Bonaparte:

> A good sketch is better than a long speech.

It shows how unevenly the net wealth is distributed, most of the richness being concentrated in the hand of some happy fews. Again, the figure would even worse if most, if not all, of other countries in the world.

Note that this can be seen also by comparing the mean net wealth which is about 5.5 millions NOK, with the median net wealth, which is just above 1 million NOK. Such a discrepancy indicates in general a heavy tail situation, where a few individual in the sample have a value significantly higher than the average.

The above distribution is characteristic of a power law . In statistics, a *power law* is a functional relationship between two quantities x and y, where a relative change in one quantity, say x, results in a proportional relative change in the other quantity y, such that one quantity varies as a power of another, i.e. $y \simeq x^k$. Power laws have attracted recently a lot of attention as the appear in many physical and social-technical phenomena, such as the number of one's "friends" in a social network.

8.3.3 Let's Go!

Exercise 8.10 Elliptic Integral. Assume we want to estimate the value of the following integral.

$$F(x) = \int_0^1 \sqrt{1-x^4}$$

One way of doing so consists in taking a large number of values of x (between 0 and 1), to calculate the value of the function $f(x) = \sqrt{1-x^4}$ for each value of x, then to calculate the mean of these values.

Question 1. Design a Python function to apply the above idea, with n values equally distributed, i.e. 0, $1/n$, $2/n$...

Question 2. Design a Python function to apply the above idea, with n values drawn at (pseudo-)random.

Question 3. Perform the experiment with both functions for $n = 10^4$, 10^5 and 10^6.

Exercise 8.11 Volume of hyperspheres. In geometry, a hypersphere, or n-sphere is a hypersurface of dimension n in the Euclidian space $\mathbb{R}^{n+1}$. The hypersphere of radius R is the set of points $(x_1, x_2, \ldots, x_{n+1})$ such that:

$$\sqrt{x_1^2 + \cdots + x_{n+1}^2} \quad \leq \quad R$$

Question 1. Design a Python script to estimate the volume of a n-sphere of radius R, by means of a randomized method.

Question 2. Check the estimates you obtain with existing analytical formulas (look on Wikipedia).

Exercise 8.12 Box-Muller Method. The Box–Muller method (Box and Muller, 1958) is used to generate numbers according to the normal distribution (see Section 8.4.1). It uses two independent random numbers U and V distributed uniformly on $[0,1]$. Then the two random variables X and Y defined as follows both have the standard normal distribution, and are independent.

$$X \quad \stackrel{def}{=} \quad \sqrt{-2\ln U}\cos(2\pi V)$$
$$Y \quad \stackrel{def}{=} \quad \sqrt{-2\ln U}\sin(2\pi V)$$

Question 1. Design a Python function to apply the above idea, i.e. to generate two series of numbers $X_1, X_2 \ldots$ and $Y_1, Y_2 \ldots$ according to the Box-Muller method.

Question 2. Calculate the mean, the standard-deviation and the histograms of both samples, for the bins -3, -2, -1, 0, 1, 2 and 3.

Question 3. Use `matplotlib` to draw the histograms of the two samples, with 20 bins each.

Problem 8.13 Height Distribution. The file `Heights.csv` contains a sample of individuals (one row per individual). For each individual, four data are provided:

- A unique number (column 1).
- The age (in year) of the individual (column 2).
- The gender of the individual (column 3): 1 for men, 2 for women.
- The height in centimeters of the individual (column 4).

Your objective is to make some observations on the distribution of heights of individual per gender and per age category.

Question 1. Design a Python function that reads the file `Heights.csv` and calculates the proportion of men and women, so to check that, as one can expect, there are about as many men than women in the sample.

Question 2. As we expect the average height of men to be slightly bigger than the one of women, we have to test this first hypothesis. To do so, separate the sample in two subsamples, one for men, one for women. Calculate the minimum, maximum and mean heights in each subsample.

Question 3. Using `matplotlib`, display the distributions of heights for men and women (as histograms).

Table 8.1: Characteristics of the uniform distribution

Probability density function	$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{if } x > b \end{cases}$
Cumulative probability function	$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{b-a} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases}$
Mean	$\frac{1}{2}(b-a)$
Median	$\frac{1}{2}(b-a)$
Variance	$\frac{1}{12}(b-a)^2$

Question 4. Another hypothesis we want to check is that the height of people increased in the last decades, due to the better nutrition and the lighter work conditions. Test this hypothesis for both men and women, by splitting the corresponding samples into three categories of ages: 18-29 years, 30-54 years and 55-74 years. Using `matplotlib`, draw the cumulative distributions of heights, for each category.

8.4 Some Important Probability Distributions

Indeed, not all of the phenomena are uniformly distributed. Think for instance to the size of people in a given population or the distribution of the wealth in the same population. Therefore, if we want to model and simulate them, we need to mimic their actual distributions, at least to a reasonably good extent.

In practice, there are two ways of defining probability distributions:

– The first one consists in using parametric distributions.
– The second one consists in using so-called empirical distributions, i.e. distributions defined by a set of points between which the value of the function is interpolated.

We shall review them in turn.

8.4.1 Parametric distributions

The following list of parametric distributions gathers only those that are frequently used in the framework of engineering. There exists indeed many others, used in different contexts.

Uniform distribution The *continuous uniform distribution*, or simply *uniform distribution*, is such that for each member of the family, all intervals of the same length on the distribution's support are equally probable. The support is defined by the two parameters, a and b, $a \leq b$, which are its minimum and maximum values.

Table 8.1 gives the characteristics of the uniform distribution.

Figure 8.2 shows the shape the cumulative distribution function of the uniform distribution.

Normal distribution The *normal distribution*, also called (or *Gaussian* or *Gauss* or *Gauss-Laplace distribution*) is one of the most convenient to represent phenomena issued from several random sources. It is defined by means of two parameters: its mean, usually denoted as μ, and its standard-deviation, usually denoted as σ, (or equivalently its variance σ^2). It is denoted $\mathcal{N}(\mu, \sigma)$.

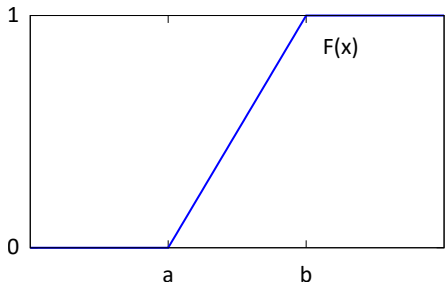


Figure 8.2: Cumulative distribution function of the uniform distribution

Table 8.2: Characteristics of the normal distribution

Probability density function	$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
Cumulative probability function	$F(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right)$
Mean	μ
Median	μ
Variance	σ^2

Table 8.2 gives the characteristics of the normal distribution. In this table, the function $\text{erf}(x)$ is the error function defined as follows.

$$\text{erf}(x) \stackrel{def}{=} \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{8.8}$$

It gives the probability for a random variable with normal distribution of mean 0 and variance $1/2$ to fall in the interval $[-x, x]$.

Figure 8.3 shows the shape the cumulative distribution function of the normal distribution.

Lognormal distribution A *lognormal distribution* is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is normally distributed, then $Y = \ln X$ has a lognormal distribution. A lognormal process is the statistical realization of the multiplicative product of many independent random variables, each of which is positive. As for the normal distribution, it is characterized by the mean μ and the standard-deviation σ of X.

Table 8.3 gives the characteristics of the lognormal distribution.

Figure 8.4 shows the shape the cumulative distribution function of the lognormal distribution.

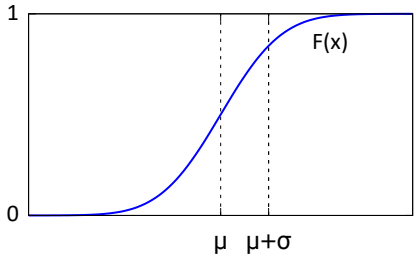


Figure 8.3: Cumulative distribution function of the normal distribution

Table 8.3: Characteristics of the lognormal distribution

Probability density function	$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x-\mu)^2}{2\sigma^2}\right)$
Cumulative probability function	$F(x) = \frac{1}{2}\left(1+\operatorname{erf}\left(\frac{\ln x-\mu}{\sigma\sqrt{2}}\right)\right)$
Mean	$\exp\left(\mu+\frac{\sigma^2}{2}\right)$
Median	$\exp(\mu)$
Variance	$\left(\exp\left(\sigma^2\right)-1\right)\exp\left(2\mu+\sigma^2\right)$

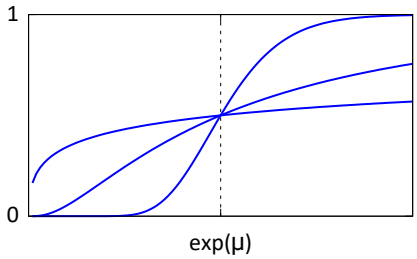


Figure 8.4: Cumulative distribution function of the lognormal distribution

Exponential distribution The *exponential distribution* represents typically the life-span of a component without memory, aging nor wearing (Markovian hypothesis). The probability that the component is working at least $t+d$ hours knowing that it worked already t hours is the same as the probability that it works d hours after its entry into service. In other words, the fact that the component worked correctly for t hours does not change its expected life duration after this delay.

The exponential distribution is defined by means of a single parameter, the transition rate, usually denoted by λ. This transition rate is the inverse of the mean life expectation.

Table 8.4 gives the characteristics of the exponential distribution.

Figure 8.5 shows the shape the cumulative distribution function of the exponential distribution.

Weibull distribution The exponential distribution assumes a constant failure rate over the time. This is not always realistic because of aging effects: at the beginning of its life the component has a decreasing failure rate, corresponding to debug (or infant mortality), then for a long while, its failure rate remains constant, then the wear-out period starts where the failure rate increases. This is the so-called bathtub curve.

This phenomenon is (piece wisely) captured by the *Weibull distribution* which takes two parameters: the shape parameter, usually denoted α, and the scale parameter, usually denoted β.

Table 8.5 gives the characteristics of the Weibull distribution. In this table, the Γ function is defined as

Table 8.4: Characteristics of the exponential distribution

Probability density function	$f(x) = \lambda e^{-\lambda x}$
Cumulative probability function	$F(x) = 1-e^{-\lambda x}$
Mean	λ^{-1}
Median	$\lambda^{-1}\ln 2$
Variance	λ^{-2}

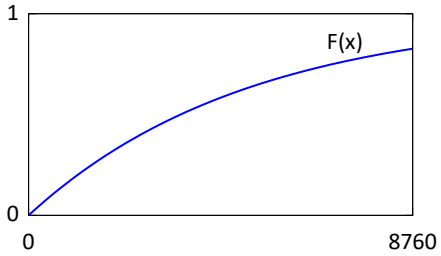


Figure 8.5: Cumulative distribution function of the exponential distribution

Table 8.5: Characteristics of the Weibull distribution

Probability density function	$f(x) = \frac{\beta}{\alpha}\left(\frac{x}{\alpha}\right)^{\beta-1} e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Cumulative probability function	$F(x) = 1 - e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Mean	$\alpha\Gamma\left(1+\frac{1}{\beta}\right)$
Median	$\alpha(\ln 2)^{\frac{1}{\beta}}$
Variance	$\alpha^2\left(\Gamma\left(1+\frac{2}{\beta}\right) - \left(\Gamma\left(1+\frac{1}{\beta}\right)\right)^2\right)$

follows.

$$\Gamma(z) \stackrel{def}{=} \int_0^{\infty} x^{z-1} e^{-x}\, dx \tag{8.9}$$

Figure 8.5 shows the shape the cumulative distribution function of the Weibull distribution.

Triangular distribution The *triangular distribution* is a continuous probability distribution with lower limit a, upper limit b and mode c, where $a < b$ and $a \leq c \leq b$.

The triangular distribution is typically used as a subjective description of a population for which there is only limited sample data. It is therefore often used in business decision making when not much is known about the distribution of an outcome (say, only its smallest, largest and most likely values).

Table 8.6 gives the characteristics of the triangular distribution.

Figure 8.7 shows the shape the cumulative distribution function of the triangular distribution.

Binomial distribution The *binomial distribution* of parameters n and p is a discrete probability distribution. It represents the number of successes in a series of n independent experiments, each asking a

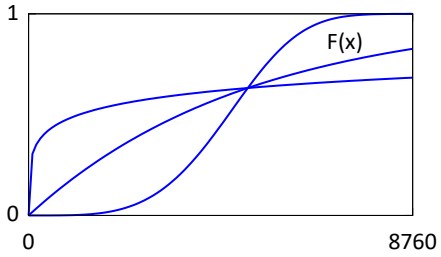


Figure 8.6: Cumulative distribution function of the Weibull distribution

Table 8.6: Characteristics of the triangular distribution

Probability density function	$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{if } a \leq x < c \\ \frac{2}{b-a} & \text{if } x = c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{if } c < x \leq b \\ 0 & \text{if } x > b \end{cases}$
Cumulative probability function	$F(x) = \begin{cases} 0 & \text{if } x \leq a \\ \frac{(x-a)^2}{(b-a)(c-a)} & \text{if } a < x \leq c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{if } c < x < b \\ 1 & \text{if } x \geq b \end{cases}$
Mean	$\frac{a+b+c}{3}$
Median	$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \geq \frac{a+b}{2} \\ b - \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \leq \frac{a+b}{2} \end{cases}$
Variance	$\frac{a^2+b^2+c^2-ab-ac-bc}{18}$

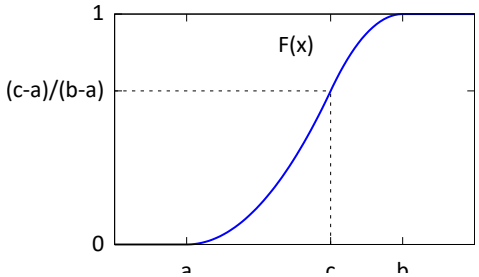


Figure 8.7: Cumulative distribution function of the triangular distribution

Table 8.7: Characteristics of the binomial distribution

Probability mass function	$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$
Cumulative probability function	$F(k) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$
Mean	np
Median	$\lfloor np \rfloor$ or $\lceil np \rceil$
Variance	$np(n-p)$

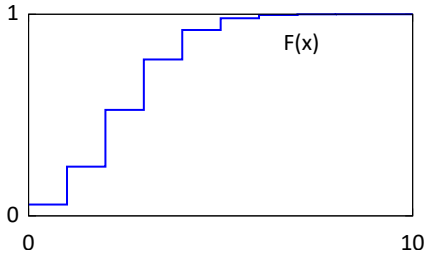


Figure 8.8: Cumulative distribution function of the binomial distribution for $n = 10$ and $p = 0.25$

yes–no question. Each experiment gives the answer yes with the probability p and no with the probability $q = 1 - p$.

The binomial distribution is frequently used to model the number of successes in a sample of size n drawn with replacement from a population of size N. In case the sample is drawn without replacement,so the resulting distribution is a hypergeometric distribution. However, if N much larger than n, the probability to draw twice the same individual is very low, so the binomial distribution is a good approximation of the hypergeometric distribution.

Table 8.7 gives the characteristics of the binomial distribution.

Recall that the expression for the binomial is as follows.

$$\binom{n}{k} \stackrel{def}{=} \frac{n!}{k!(n-k)!} \tag{8.10}$$

Figure 8.8 shows the shape the cumulative distribution function of the binomial distribution for $n = 10$ and $p = 0.25$.

The *Bernoulli distribution* is a special case of the binomial distribution where a single trial is conducted, i.e. $n = 1$.

8.4.2 Empirical distributions

Empirical distributions are given by a list of points $(x_1, y_1), \ldots, (x_n, y_n)$, $n \geq 2$, such that $x_1 < \ldots < x_n$ and, if one considers cumulative distribution functions, $y_1 \leq \ldots \leq y_n$. They are typically obtained via series of observations.

In between points, the value of the function is obtained by interpolation. There are basically two ways to do this interpolation:

– To consider the distribution as a *stepwise distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{8.11}$$

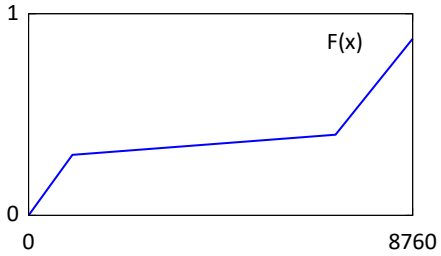


Figure 8.9: Cumulative distribution function of the piecewise uniform distribution

– To consider the distribution as a *piecewise uniform distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i + (y_{i+1} - y_i)\frac{x - x_i}{x_{i+1} - x_i} & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{8.12}$$

Figure 8.9 shows a piecewise uniform distribution defined by 4 points: $(0,0)$, $(1000,0.3)$, $(7000,0.4)$ and $(8760,0.876)$.

Until recently, empirical distributions were mostly used when it was hard to fit them with any parametric distribution. For instance, the *Kaplan–Meier estimator* (Kaplan and Meier, 1958), also known as the product limit estimator, is a non-parametric statistic used to estimate the survival function from lifetime data. In reliability engineering, Kaplan–Meier estimators may be used to measure the time-to-failure of machine parts. In medical research, they are often used to measure the fraction of patients living for a certain amount of time after treatment. This situation may generalize with easy internet communications: there is no more need to store data into a very compact form (the name of the parametric function and its parameters). Therefore, there is less and less reasons to spend time and energy in fitting empirical observations with parametric distributions, not to speak about the arbitrariness of the choice of the parametric distribution.

8.4.3 Let's Go!

Exercise 8.14 Exponential Distribution. The exponential distribution has the quite surprising property to be *memoryless*, which we shall illustrate in this exercise.

Consider this a physical component, e.g. a pump or a valve, whose failures are exponentially distributed with a constant failure rate $\lambda = 10^{-4}h-1$, i.e. that the mean time to failure of the component is 10^4 hours (about one year). The probability $Q(t)$ that the component is failed at time t is thus defined as follows.

$$Q(t) = 1 - \exp(-\lambda \cdot t) \tag{8.13}$$

Assume we want to perform a Monte-Carlo simulation using, among others, the component in question. To do so, we shall use Equation 8.13 in the reverse way, i.e. to draw at random failure times for the component. This process is described by the following equation (obtained just by taking the inverse of the above equation).

$$t = -\log(z) \times \frac{1}{\lambda} \tag{8.14}$$

where z is a random variable uniformly distributed in $[0,1]$.

Assume that we drew a time to failure t_1 from a random value z_1, i.e. $t_1 = -\log(z_1) \times \frac{1}{\lambda}$. Consider the delay $t_0 = -\log(z_0) \times \frac{1}{\lambda}$, for a certain predefined value z_0, typically $z_0 = 0.5$. There are two cases:

- Either the component is already failed at time t_0, i.e. $t_1 \leq t_0$ and $z_1 \leq z_0$.
- Or it is not, i.e. $t_1 > t_0$ and $z_1 > z_0$.

In the second case, we can draw a new time to failure $t_2 = t_0 - \log(z_2) \times \frac{1}{\lambda}$, i.e to consider the component as good as new at time t_0, which is precisely the memoryless property. This property goes against our intuition that says that t_2 must be on average bigger than t_1 because of the wear-out of the component at time t_0. So, the question is how t_1 and t_2 compare?

Question 1. Design a Python script that performs a Monte-Carlo simulation to assess the probabilities:
- p_1 that $t_1 \leq t_0$,
- p_2 that $t_1 > t_0$ and $t_1 \leq t_2$, and finally
- p_3 that $t_1 > t_0$ and $t_2 > t_1$.

Problem 8.15 PERT Diagram. Figure 8.10 shows the PERT diagram of a project made of 14 activities labelled from `A` to `N`. Two additional "fake" activities `S` and `T` have been added to represent the beginning and the termination of the project. It is assumed that the duration of each activity of the project obeys a triangular distribution. The file `Project.csv` contains a full description of the project:
- Each row describes an activity.
- Column 1 gives codes of activities.
- Column 2 gives a description of activities (this column is left empty in the file `Project.csv`).
- Column 3 gives the lists of preceding activities of activities.
- Column 4 gives the minimum durations of activities.
- Column 5 gives the most likely durations of activities.
- Finally, column 6 gives the maximum durations of activities.

Your objective is to implement a program that reads project description files and calculates the distribution of the duration of the project. For the sake of the simplicity, we shall assume that activities are always given in order, i.e. all activities preceding an activity are described before this activity.

Question 1. Design a Python module to manage projects and activities.

Question 2. Design a Python module to read and write project at the csv format, taking the file `Project.csv` as an example.

Question 3. Design functions to calculate early start and completion dates, late start and completion dates and criticality of activities

Question 4. Design functions to study:
- How frequently activities are critical.
- The distribution of project duration.

Question 5. Using the functions designed in the previous questions, perform a study on, for instance, 10,000 executions.
- Print out, for each activities, the proportion of executions in which the activity is critical.
- Print out the minimum, maximum and mean duration of the project as well as its standard deviation.
- Using `matplotlib`, display the histogram of the duration of the project.

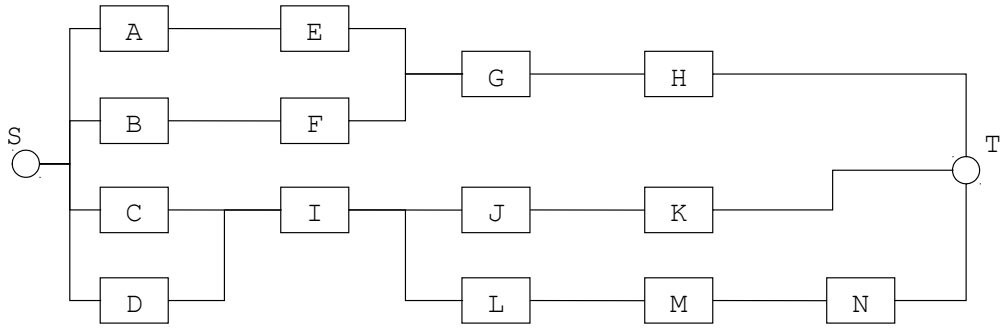


Figure 8.10: A PERT Diagram.

Exercise 8.16 Patient Survival. The oncology service proposes a new treatment to patients with a cancer of type A (a very cancer which is extremely difficult to cure). To compare the efficiency of this new treatment with the regular one, two groups of 23 patients each have been studied: the group 1, which received the regular treatment and the group 2, which received the new treatment.

The file `Deaths.csv` contains the dates at which patients of both groups and of group 2 died.

Question 1. Using `NumPy` and `matplotlib`, draw the Kaplan-Meier indicators of the probability of survival of patients.

8.5 How Large Samples Should Be?

8.5.1 Setting the Problem

Statistics consist eventually in making inferences on properties of a population from a sample of this population. This raises immediately the question of about the size of the sample. How large should it be so to be actually representative of the population? Or, to put it in other terms, what is the risk we take by making inferences on the whole from what we observe on a limited sample of that population?

These questions apply indeed to Monte-Carlo simulation. Larger number of executions lead in general to an increased precision of estimated indicators. The law of large numbers and the central limit theorem are useful mathematical tools to describe this phenomenon. Namely, by the strong law of large numbers, we know that the estimated mean of a distribution $\bar{x}$ tends with a probability 1.0 to its mean μ as n goes to infinity. Moreover, we can calculate an error factor and a confidence interval, as defined by equations 8.6 and 8.7.

These results can be interpreted as follows.

- For a fixed number of executions n, if we want to increase our confidence in the result, i.e. to reduce α, we need to widen the confidence interval, and vice-versa.
- If we want to increase our confidence in the result without widening the confidence interval or if we want to narrow down the confidence interval without decreasing our confidence of the result, we need to increase n.
- However, the improvement is governed by the $\frac{1}{\sqrt{n}}$ factor: to reduce the width of the confidence interval by a factor 2, we need 4 times more executions, to reduce it by a factor 10, we need 100 times more executions, and so on.

Note that, and it is a remarkable fact, the above results do not depend on the size of the population, i.e. in our case of the number of possible executions. This is what makes polls possible. With a relatively limited sample, it is possible to get a good picture of the whole population.

In systems engineering, executions are in general not too computationally costly as models have a

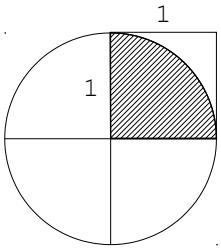


Figure 8.11: Estimation of π by sampling points in the unit square

limited size and systems are observed on limited mission-times. As a rule of thumb, it is thus reasonable not to do less than 10,000 executions. This number is also sufficient in many cases.

In some cases, the increase in precision for larger sample sizes is minimal, as illustrated by the following classical example.

Exercise 8.17 Estimation of π. Assume we want to estimate the value of π. We can design a Monte-Carlo simulation to do so.

The idea is as follows. Each execution consists in drawing two numbers x and y uniformly at random in the range $[0,1]$. The point (x,y) lies thus somewhere is the unit square. Now there are two cases: either the point lies in the (quater of the) unit disk (the region bounded by the unit circle), i.e. $x^2+y^2 \leq 1$, or not, as illustrated Figure 8.11.

By the law of large numbers, when the number of points n in the sample tends to infinity, the proportion k/n of points that lie within the unit tends to the surface of that unit disk, i.e. $\frac{1}{4}(\pi \times r^2) = \pi/4$. To get an estimate $\overline{\pi}$ of π it suffices thus to take the proportion of points (x,y) in the sample such that $x^2+y^2 \leq 1$ and to multiply it by 4.

In theory, this works fine. In practice, ...

Question 1. Design a Python script to estimate the value of π according to the above principle. Study the evolution of the 95% confidence range as the size of the sample increase.

You should observe that the improvement obtained by taking a large sample rather than a small one is not very significant, as the result is coarse anyway. This does not say that the law of large numbers and the central limit theorem do not apply in this case, just that the convergence rate towards the limit is low.

8.5.2 Let's Go!

Problem 8.18 k-out-of-n Systems. Most of the (complex) technical systems work because they are regularly maintained. In some cases however, systems cannot be (easily) maintained: think, for instance, to satellites or to piping systems installed on the sea bed for offshore oil extraction. In such cases, the only possibility to circumvent failures of components is to embed a number of spare parts in the system, and to reconfigure it when a failure occurs.

In the sequel, we shall assume that the system embeds n components and that to work correctly, it requires k out of these n components. We shall assume moreover that components have the same probability of failure whether they are working or in standby. We shall proceed into two steps:

– First, we shall design a Python script that calculates the exact probability that the system is working, given the probabilities of failure of components.

– Second, we shall see how large should be the sample of a Monte-Carlo simulation that attempts to estimate this probability. Indeed, the size of the sample depends on the probability.

To calculate the exact probability of failure of a k-out-of-n system, we shall use a programming technique called *dynamic programming*. The idea is as follows.

We shall compute the probabilities that 0, 1,... $k-1$ and k or more components are failed. To do so, we shall consider each component in turn:

– Initially, i.e. when no component has been considered, the probability that there are 0 component failed is 1.0 while all other probabilities are 0.

– Now, assume that we have already considered the first $i-1$ components, $1 \leq i \leq n-1$, i.e. that we know the probabilities $Q_{i-1,0}$, $Q_{i-1,1}$,... $Q_{i-1,k}$. Then, the probability $Q_{i,j}$, $0 \leq j \leq k$, that there are j components failed can be calculate as follows.

$$Q_{i,j} = \begin{cases} (1-p_i) \times Q_{i-1,j} & \text{if } j = 0 \\ p_i \times Q_{i-1,j-1} + (1-p_i) \times Q_{i-1,j} & \text{if } 0 < j < k \\ p_i \times Q_{i-1,j-1} + Q_{i-1,j} & \text{if } j = k \end{cases} \tag{8.15}$$

The above principle makes it possible to calculate the exact probability of failure of a k-out-of-n system in a number of operations which is proportional to $k \times n$, i.e. very efficiently for all practical purposes. Note moreover that the probabilities of failure of components, i.e. the p_i's, do not need to be the equal.

Question 1. Design a Python function that calculates the exact probability of failure of a k-out-of-n applying the dynamic programming principle.
Hint: You will need two tables: one of size n to store the p_i's and one of size $k+1$ to store the $Q_{i,j}$.
Hint: Perform calculations of equation 8.15 from $j = k$ down to $j = 0$.

Question 2. Taking $k = 4$ and $n = 10$ and a probability of failure p identical for all components, calculate the probability of failure of the system for $p = 0.10, 0.09, \ldots 0.01$.

We shall now try to obtain the same results by means of Monte-Carlo simulations. A trial consists here in drawing the state of each of the n components at random according to the probability of failure of this component, then calculating the number of failed components. Such a trial is called a *Bernouilli trial* as its outcome is a binary variable.

Let N be the number of trials performed during the Monte-Carlo simulation. Let N_0 and N_1 be the number of trials in which the system was respectively working and failed ($N_0 + N_1 = N$). Then, the probability of failure is estimated as N_1/N. Moreover, the confidence interval is determined as follows:

$$\left[\frac{N_1}{N} - \frac{z}{N}\sqrt{\frac{N_1 \cdot N_0}{N}}, \frac{N_1}{N} + \frac{z}{N}\sqrt{\frac{N_1 \cdot N_0}{N}} \right]$$

where z is obtained from the table of the normal distribution as explained in Section 8.3.1, i.e. $z = 1.64$ for a 90% confidence interval, $z = 1.96$ for a 95% confidence interval and $z = 2.58$ for a 99% confidence interval.

Question 3. Design a Python function that performs a Monte-Carlo simulation to assess the probability of failure of a k-out-of-n. Calculate the 90%, 95% and 99% confidence intervals.

Question 4. Taking $k = 4$ and $n = 10$ and a probability of failure p identical for all components, assess the probability of failure of the system for values of p ranging from 0.10 down to 0.01, and different number of trials.

Chapter 9

Domain-Specific Languages

Key Concepts
- Domain-Specific Languages
- Parsers, Lexers, Grammars
- Extended Backus-Naur Form
- Continuous Time Markov Chains

Domain-specific languages are computer languages dedicated to a particular application domain. This contrasts with general-purpose languages, such as Python. Special-purpose computer languages have been designed since the very beginning of computers, but the term "domain-specific language" has become more popular due to the rise of domain-specific modeling language (Fowler, 2010).

The primary interest of domain-specific modeling languages is that they make it possible to design models in simple and intuitive way, without the need of diving into the code of a programming language.

This chapter illustrates this idea by implementing step by step a solver for continuous time Markov chains with rewards.

9.1 Mathematical Framework

Introduced by the Russian mathematician Andrei Markov in the early 20th century, Markov chains are nowadays pervasive in science and engineering. Everywhere a process with some uncertainty is under study, Markov chains are in the background. Markov chains are thus a fundamental modeling tool, if not directly, at least as a underlying conceptual framework.

9.1.1 Continuous-Time Markov Chains

A *continuous-time Markov chain* is a stochastic process, i.e. a random variable $\{X(t), 0 \leq t \leq \infty\}$. The values assumed by $X(t)$ are called states and belong to a finite set. Moreover, $X(t)$ must satisfies the so-called memoryless property or Markovian assumption, i.e. for all integers n and for any sequence $t_0 < t_1 < \ldots < t_n < t$, the following equality holds.

$$p(X(t) = s \mid X(t_0) = s_0, X(t_1) = s_1, \ldots, X(t_n) = s_n) \quad = \quad p(X(t) = s \mid X(t_n) = s_n) \tag{9.1}$$

where $p(E)$ denotes the probability of the event E. The above property is called memoryless because the fact that the system was in state s_0 at t_0, s_1 at t_1 and so on is irrelevant.

When the probabilities of transitions out of the state $X(t)$ do not depend on the time t, the continuous-time Markov chain is said *homogeneous*.

In the sequel, we shall simply say Markov chain for continuous-time homogeneous Markov chain.

Markov chains can be represented as graphs. Nodes represent states of the chain. Edges represent transitions between states. Edges are labeled with transition rates. The rate λ_{ij} of a transition from state s_i to state s_j is defined as the limit when dt tends to 0 of the conditional probability $p(X(t+dt) = s_j \mid X(t) = s_i)$.

If we consider Markov chains as a modeling language, we can even take this representation as the definition of Markov chains: A *Markov chain* is a labeled graph (S,T), where:

- S is a finite set of *states*.
- T is a finite set of *transitions* between states, i.e. of triples (s,t,r), where $s,t \in S$, $s \neq t$ and r is positive real number, called the *rate* of the transition.

As an illustration, consider the pumping system represented Figure 9.1 (left). This system is made of three different pumps `P1`, `P2` and `P3` that may fail and be repaired. `P1` and `P2` are main pumps while `P3` is a spare pump, i.e. when `P1` or `P2` fail, `P3` is started. `P3` cannot fail when not in operation. Moreover, it always starts when demanded. The system works if two pumps are in operation (and working). When the system is failed, it starts immediately to be repaired. Failure rates λ_1 to λ_6 of pumps depends on the configuration of the system. Repair rates depend on which pumps are failed, namely it is μ_{12} when `P1` and `P2` are failed, μ_{13} when `P1` and `P3` are failed and μ_{23} when `P2` and `P3` are failed.

Under the above assumptions, the availability of the system, i.e. the probability that it works at time t, can be assessment by means of the Markov chain pictured Figure 9.1 (right).

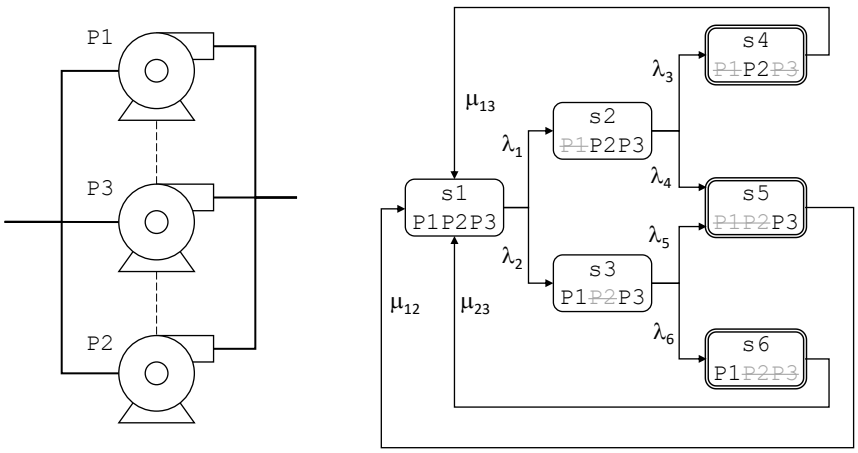


Figure 9.1: A pumping system (left) and its Markov chain model (right)

9.1.2 Sojourn Times and Rewards

Let $\mathscr{M} : (S,T)$ be a Markov chain. The key problem is to determine, given a distribution $\pi(0)$ of probability over the states of S at time 0, what will be the distribution $\pi(t)$ of probability over the states at time t.

Usually, the initial distribution of probabilities takes the value 1 in one state, the initial state, and 0 in all other states. In our example, the initial state is the state in which the three pumps are working.

The integral, over a given mission time t, of the probability $p_s(u)$ to be in the state $s \in S$ at time u defines the *sojourn time* $s_s(t)$ in that state:

$$s_s(t) \stackrel{def}{=} \int_{u=0}^{t} p_s(u)\,du \tag{9.2}$$

Determining the sojourn times in each state makes it possible to calculate interesting indicators defined via rewards. A *reward* r is a function from S to $\mathbb{R}$. that measures some quantity of interest. Once defined a reward can be used into ways:

– By calculating the mathematical expectation $E[r(t)]$ of the reward at time t:

$$E[r(t)] \stackrel{def}{=} \sum_{s \in S} p_s(t) \times r_s \tag{9.3}$$

– By calculating the average value $E_{avg}[r(t)]$, from time 0 to time t of this expectation:

$$E_{avg}[r(t)] \stackrel{def}{=} \frac{\int_{u=0}^{t} E[r(u)]\ du}{t} \tag{9.4}$$

$$= \frac{\sum_{s \in S} s_s(t) \times r_s}{t} \tag{9.5}$$

As an illustration, consider again our pumping system. We can define at least two rewards of interest.

First, each pump `Pi` has a certain capacity κ_i, typically expressed in percentage of the demand (assuming the latter constant through the time). Consequently, it is worth to assess the expectation $E[\nu(t)]$ of the actual quantity ν of fluid pumped by the system at time t, as well as its average expectation $E_{avg}[\nu(t)]$ from time 0 to time t.

Similarly, each pump `Pi` consumes a certain quantity of energy ε_i, that depends on how much fluids it pumps. Consequently, it is worth to assess the expectation $E[\varepsilon(t)]$ of the energy consumption of ε of the circuit at time t, as well as its average expectation $E_{avg}[\varepsilon(t)]$ from time 0 to time t.

9.1.3 Parameters

From what preceedes, it is clear that a Markov model involves a number of parameters: transition rates and rewards. These parameters are often defined in terms of more detailed parameters. For instance, the transition rate λ_1 of the Markov chain pictured in Figure 9.1 is actually the failure rate of the pump `P1`, which depends itself on the flow circulating through `P1` in the state `s1`. Similarly, the energy consumption of the system in state `s2` is the sum of the energy consumption of pumps `P2` and `P3`, which themselves depend on the flow circulating through these pumps in this configuration of the system.

Parameters are thus defined by means of equations, one equation per parameter.

9.1.4 Wrap-Up

A Markov model is thus a quintuple (S,T,R,P,E) where:

– S is a finite set of *states*.
– T is a finite set of *transitions* between states, i.e. of triples (s,t,r), where $s,t \in S$, $s \neq t$ and r is an arithmetic expression defining the *rate* of the transition.
– R is a finite set of *rewards* defined on states of S.
– P is a finite set of *parameters* involved in the definitions of rates and rewards.
– E is a set of equations defining rewards and parameters. These equations are of the form $p = e$, where p is a reward or a parameter and e is an arithmetic expressions (not depending on p).

As an illustration, consider again our pumping system. We shall make the following assumptions:

– The main pumps `P1` and `P2` are identical and operated in the same way.
– In state `s1`, pumps `P1` and `P2` provide 50% of the demand each.
– In state `s2`, the pump `P2` provides 60% of the demand, while the pump auxiliary `P3` provides 20%.
– In state `s3`, the pump `P1` provides 60% of the demand, while the pump auxiliary `P3` provides 20%.
– In states `s4`, `s5`, and `s6`, the production is stopped.
– The failure rate $\lambda_{\texttt{Pmr}}$ of pumps `P1` and `P2` is $1.23 \times 10^{-3} h^{-1}$ when they are operated to provide 50% of the demand. It is doubled when they are operated to provide 60% of the demand.
– The failure rate $\lambda_{\texttt{Pa}}$ of `P3` is $6.78 \times 10^{-3} h^{-1}$.
– The mean time to repair $MTTR_{\texttt{Pm}}$ of pumps `P1` and `P2` is 24 hours, while the mean time to repair $MTTR_{\texttt{Pa}}$ of the pump `P3` is only 4 hours. Repairs are made one after the other.
– The energy consumption of pumps `P1` and `P2` is $3kWh^{-1}$ when they are operated to provide 50% of the demand, and $4kWh^{-1}$ when they are operated to provide 60% of the demand.

– The energy consumption of the pump `P3` is $1kWh^{-1}$.

The model (S,T,R,P,E) of the pumping circuit is thus such that:

– The sets S of states and T of transitions are those given in Figure 9.1.

– Rates are defined as follows.

$$\lambda_1 = \lambda_2 = \lambda_{\texttt{Pmr}}$$
$$\lambda_4 = \lambda_5 = \lambda_{\texttt{Pmd}}$$
$$\lambda_3 = \lambda_6 = \lambda_{\texttt{Pa}}$$
$$\lambda_{\texttt{Pmr}} = 1.23 \times 10^{-3}$$
$$\lambda_{\texttt{Pmd}} = 2 \times \lambda_{\texttt{Pmr}}$$
$$\lambda_{\texttt{Pa}} = 6.78 \times 10^{-3}$$
$$\mu_{12} = 2 \times 1/MTTR_{\texttt{Pm}}$$
$$\mu_{13} = \mu_{23} = 1/MTTR_{\texttt{Pm}} + 1/MTTR_{\texttt{Pa}}$$
$$MTTR_{\texttt{Pm}} = 24$$
$$MTTR_{\texttt{Pa}} = 4$$

– The reward ν is defined by the following equations (undefined values are equal to 0).

$$\nu(\texttt{s1}) = 2 \times \nu_{\texttt{Pmr}}$$
$$\nu(\texttt{s2}) = \nu(\texttt{s3}) = \nu_{\texttt{Pmd}} + \nu_{\texttt{Pa}}$$
$$\nu_{\texttt{Pmr}} = 50$$
$$\nu_{\texttt{Pmd}} = 60$$
$$\nu_{\texttt{Pa}} = 20$$

– Finally the reward ε is defined by the following equations (undefined values are equal to 0).

$$\varepsilon(\texttt{s1}) = 2 \times \varepsilon_{\texttt{Pmr}}$$
$$\varepsilon(\texttt{s2}) = \varepsilon(\texttt{s3}) = \varepsilon_{\texttt{Pmd}} + \varepsilon_{\texttt{Pa}}$$
$$\varepsilon_{\texttt{Pmr}} = 3$$
$$\varepsilon_{\texttt{Pmd}} = 4$$
$$\varepsilon_{\texttt{Pa}} = 1$$

To assess Markov models we have to implement data structures in which will be encoded the elements of the models (states, transitions, rewards, parameters and equations) and then to implement assessment algorithms working on these data structures.

It would be possible to create data structures encoding each model by means of Python functions. However, this would be tedious if we have several models to work on. A better idea consists in designing a domain specific modeling language, i.e. to define a grammar for models. Models can then be described into text files, which are loaded by our Python program, then assessed.

9.2 Data Structures

We need data structures to encode our Markov models. Namely, we need data structures to encode Markov models, states, transitions, parameters, (arithmetic) expressions and rewards. We shall develop a package `MarkovModels` to do so.

9.2.1 Markov models, states and transitions

The class `MarkovModel` is a container for all declarations. The key question is how to encode transitions. For engineers used to work with matrices, a matrix looks probably a quite natural encoding. It has however a fundamental drawback: the size of a matrix in $\mathcal{O}(n^2)$, where n is the number of states. In turns out that, in general, there is much less transitions than that. Transitions matrices are sparse. Consequently, it much

better to manage transitions into a list. We shall see that all operations required for the assessment of Markov chains can be performed efficiently from this list.

`MarkovModel` should thus manage:

– A dictionary of states;
– A list of transitions;
– A dictionary of parameters;
– A dictionary of rewards.

For now, the class `State` has just one attribute: the name of the state.

Still for now, the class `Transition` has just three attributes:

– The source state;
– The target state;
– The rate, which is an expression.

9.2.2 Expressions and parameters

The class `Parameter` has three attributes:

– A name;
– A definition, which is an expression;
– A value, which is a floating point number.

The class `Expression` manages arithmetic expressions. For the sake of simplicity, we shall consider only the following types of arithmetic expressions.

– References to parameters;
– Real values;
– Additions, subtractions, multiplications and divisions;
– Unary minus.

It is worth to use coding techniques we have already seen for multi-type classes:

– Define global variables for types, e.g.

```
class Expression:
  PARAMETER = 1
  REAL = 2
  ...
```

– Define static methods to create instance of each type, e.g.

```
  @staticmethod
  def NewParameterReference(self, parameterName):
    self = Expression(Expression.PARAMETER)
    self.parameterName = parameterName
    return self

  @staticmethod
  def NewReal(self, value):
    self = Expression(Expression.REAL)
    self.value = value
    return self
```

9.2.3 Rewards

The class `Reward` has two attributes:

– A name;

– A dictionary that associates that associates values with states.

Values are expressions.

9.2.4 Let's go!

Problem 9.1 Data structures for Markov models. This problem consists of a unique question:

Question 1. Design the package `MarkovModels` according to the specifications of this section.

9.3 Language

We need now to specify a grammar for our domain specific language as well as the package `TextInterface` that implements classes to read and to write Markov models from and into (text) files.

9.3.1 Grammars

Artificial languages such as Python or the language we shall specify now are described by means of grammars. Words of these grammars are atomic syntactic elements such as variable keywords, identifiers, delimiters, operators,... Sentences of these grammars are well formed expressions, instructions, functions and class definitions,...

Linguists found out that describing the grammars of natural languages such as English, Norwegian or Japanese turns out to be extremely difficult, if even possible. Fortunately, describing the grammars of artificial languages is a much easier task as, conversely to the former, it is possible to design the grammar so to make it easy to be described.

Grammars describe elements of languages by means of so-called production rules. For instance, a transition can be described as the keyword `transition`, followed by the name of the source state, followed by the name of the target state, followed by an arithmetic expression describing the rate and terminated by a semicolon. This gives the following production rule:

$$\text{Transition} \leftarrow \text{'}\texttt{transition}\text{'}\ \text{Identifier}\ \text{Identifier}\ \text{Expression}\ \text{';'}$$

The keyword "`transition`" and the semicolon ";" are terminal symbols. This is why they are surrounded with simple quotes. "Transition", "Identifier" and "Expression" are non-terminal symbols, which means that they must be defined by a production rule.

To be simple, grammars must obey two principles:

– First and foremost, they must be context-free, i.e. that their production rules must be independent of the context in which they are applied. Context dependence is one of the problematic aspects of natural languages.

– Second, they must make it possible to decide by reading as few symbols as possible, hopefully only one, which production rule to apply. This is the reason why we added the keyword `transition` in front in the above production rule. For this reason, the following production rule for transitions would be more difficult to apply.

$$\text{Transition} \leftarrow \text{Identifier}\ \text{'|-'}\ \text{Expression}\ \text{'->'}\ \text{Identifier}\ \text{';'}$$

The reader interested in more details about grammars and more generally computer languages should consult reference books, e.g. (Hopcroft et al., 2006; Aho et al., 2006).

9.3.2 Extended Backus-Naur form

Grammars of computer languages are often described using a meta-notation called *extended Backus-Naur form*, EBNF for short.

A EBNF description for a grammar is a set of production rules in the form:

lhs ::= *rhs*

where *lhs* is a non terminal symbol and *rhs* is a EBNF expression.

EBNF expressions are as follows.

- Terminal symbols are surrounded with simple quotes.
- Non-terminal symbols are capitalized, e.g. `MarkovModel`.
- A sequence of elements is just written by putting these elements in sequence (as in the production rules of the previous section).
- The bar symbol "`|`" denotes an alternative.
- The star symbol "$\star$" put after an expression denotes the repetition any number of times (including zero) of the expression.
- The star symbol "+" put after an expression denotes the repetition at least one time of the expression.
- The question mark symbol "`?`" put after an expression denotes the repetition zero or one time of the expression.
- Parentheses "`(`" and "`)`" are used to group expressions.
- Regular expressions are used to describe terminal symbols.

To illustrate the use of EBNF, we shall design the grammar of our language for Markov models. Figure 9.2 gives this grammar.

The declaration of a Markov model is thus introduced by the keyword `model`, followed by the name of the model, followed by any number of declarations, and terminated with the keyword `end`.

A declaration is either a state declaration, a transition declaration, a parameter declaration, or a reward declaration.

A state declaration is introduced by the keyword `state`, followed by the name of the state, followed possibly by the initial probability of the state, and terminated with a semi-colon. By default, the initial probability of a state is 0.

And so on.

Note that we used a parenthesized prefix notation "à la LISP" for arithmetic expressions. This is a very convenient way to avoid problems of precedence between operators, to the price of a somewhat unconventional way of writing things.

In addition to this grammar, a model must obey a number of other rules that must checked separately, e.g.

- A state must be declared before being used in a transition or a reward declaration.
- A transition cannot have the same source and target states.
- All identifiers used in expressions must refer to declared parameters.
- The expression defining a rate must always have a positive value.
- The initial probabilities of states must be comprised between 0 and 1, and must sum to 1.

9.3.3 Lexer

Reading a model from a file consists of two phases:

- First, the split is read and split into tokens. Tokens are atomic syntactic elements such as identifiers, numbers, delimiters and operators. Once read, they are added one by one to a list.
- Second, one applies grammatical rules on the list of tokens.

The first phase is called *lexical analysis* and is in general implemented in a separated class that we shall call `Lexer`.

```
MarkovModel ::=
    'model' Identifier Declaration* 'end'

Declaration ::=
      StateDeclaration | TransitionDeclaration | ParameterDeclation
    | RewardDeclaration

StateDeclaration ::=
    'state' Identifier ( '(' initial = Number ')' )? ';'

TransitionDeclaration ::=
    'transition' Identifier Identifier Expression ';'

ParameterDeclaration ::=
    'parameter' Identifier '=' Expression ';'

Expression ::=
    Identifier | Number | '(' Operator Expression* ')'

Operator ::=
    '+' | '-' | '*' | '/'

RewardDeclaration ::=
    'reward' Identifier '=' StateValue (',' StateValue)* ';'

StateValue ::=
    Identifier ':' Expression

Identifier ::=
    [a-zA-Z_][a-zA-Z0-9_]*

Number ::=
    ([0-9]+(.[0-9]*)?|.[0-9]+)([eE][+-]?[0-9]+)?
```

Figure 9.2: EBNF grammar of Markov models

A class `Token` is used to store tokens. Each token has a type, a value (the string it consists of), a line and a column numbers. These two numbers are very convenient to locate errors. In our case there are 19 different types:

- Delimiters "`;`", "`:`", "`,`", "=", "`(`", and "`)`";
- Operators "+", "−", "`*`", and "`/`";
- Keywords "`model`", "`end`", "`state`", "`initial`", "`transition`", "`parameter`", and "`reward`";
- And finally, the two generic types number and identifier.

The class `Lexer` implements methods to read tokens from a file and insert them in order into a list. Files are read line by line. Regular expressions are used to read tokens of a line one by one.

If a non recognized character or group or characters is encountered, an error message is emitted and the lexical analysis is aborted.

9.3.4 Parser

The second phase of the analysis consists in applying the rules of the grammar to actually read and create the model. It is called syntax analysis. This phase is performed by a class that we shall call `Parser`.

Rules of the grammar is implemented simply by means of recursive functions. The parser manages an index (a position) in the list of tokens. Initially, this index is set to 0. Then, it is incremented as the parsing progresses. Tokens are thus consumed one by one, in order. Figure 9.3 shows a possible implementation of the two methods of the parser that read respectively a model and a declaration.

```
def ReadModel(self):
  token = self.GetGivenToken(Token.KEYWORD_MODEL)
  token = self.GetGivenToken(Token.IDENTIFIER)
  self.model = MarkovModel(token.string)
  while True:
    token = self.GetCurrentToken()
    if token.type == Token.KEYWORD_END:
      break
    self.ReadDeclaration()
  token = self.GetGivenToken(Token.KEYWORD_END)

def ReadDeclaration(self):
  token = self.GetCurrentToken()
  if token.type == Token.KEYWORD_STATE:
    self.ReadStateDeclaration()
  elif token.type == Token.KEYWORD_TRANSITION:
    self.ReadTransitionDeclaration()
  elif token.type == Token.KEYWORD_PARAMETER:
    self.ReadParameterDeclaration()
  elif token.type == Token.KEYWORD_REWARD:
    self.ReadRewardDeclaration()
  else:
    message = "Line {0:d}, column {1:d}: unexpected token\n".format(
      token.lineNumber, token.columnNumber)
    self.RaiseError(message)
```

Figure 9.3: A possible implementation of the parser (partial)

The function `GetGivenToken` tests the element of the list of tokens at the current position. If this token is of the expected type, it is consumed, i.e. the index is incremented. Otherwise, an error message is emitted and the parsing is aborted.

The method `ReadDeclaration` tests also the current token and, according to what this token is, calls the corresponding method. Our grammar is designed in such way that reading the next token is always sufficient to decide what to do. Tokens are greedily consumed, i.e. choices are never backtracked and the token index is never decremented.

Note finally that the data structure is created while performing the syntax analysis. This is not always possible. For more complex languages, an intermediate data structure is created, called an abstract syntax tree.

9.3.5 Let's go!

Problem 9.2 Parser for Markov models. We can now design a package `TextInterface` for Markov models.

Question 1. Design a printer for Markov models, according to the grammar given in Figure 9.2.

Question 2. Design a lexer for Markov models, according to the grammar given in Figure 9.2.

Question 3. Design a parser for Markov models, according to the grammar given in Figure 9.2.

Question 4. Design a checker for Markov models, according to the grammar given in Figure 9.2.

Question 5. Write the Markov model of the pumping circuit discussed Section 9.1 at according to the grammar defined in this section. Read and write it with your parser and printer.

9.4 Assessment Algorithms

9.4.1 Principle of the calculation of numerical solutions of Markov chains

We can associate to each Markov model (S,T,R,P,E) defines a $|S| \times |S|$ matrix Q where the q_{ij}'s, are given by the transitions of T if $i \neq j$, and

$$q_{ii} = -\sum_{j \in S, j \neq i} q_{ij}$$

The matrix Q is called the *infinitesimal generator* or the transition rate matrix of the Markov chain.

As an illustration, the infinitesimal generator of the Markov model representing the pumping system is as follows.

$$\begin{pmatrix} -\lambda_1-\lambda_2 & \lambda_1 & \lambda_2 & 0 & 0 & 0 \\ 0 & -\lambda_2-\lambda_3 & 0 & \lambda_2 & \lambda_3 & 0 \\ 0 & 0 & -\lambda_1-\lambda_3 & \lambda_1 & 0 & \lambda_3 \\ \mu_{12} & 0 & 0 & -\mu_{12} & 0 & 0 \\ \mu_{13} & 0 & 0 & 0 & -\mu_{13} & 0 \\ \mu_{23} & 0 & 0 & 0 & 0 & -\mu_{23} \end{pmatrix}$$

Let $\pi(t)$ be the vector of probabilities to be in state i at time t. $\pi(t)$ is called the vector of *transient probabilities* (a time t). $\pi(0)$ denotes thus the vector of initial probabilities. It can be shown, applying the so-called Chapman-Kolmogorov equations, that the following equality holds.

$$\pi(t) = \exp(t \times Q) \times \pi(0) \tag{9.6}$$

where $\exp(t \times Q)$ is defined as follows.

$$\exp(t \times Q) \stackrel{def}{=} \lim_{n \to \infty} \sum_{k=0}^{n} \frac{(t \times Q)^k}{k!} \tag{9.7}$$

In the case of Markov chains, this series converges, at least theoretically. In practice, things are a bit more complex. The problem is twofold. First, in order to assess $\pi(t)$ efficiently one should perform only products of matrices by vectors. Computing the product of two matrices is actually computationally expensive. Moreover, powers of a matrix tend to be filled up, even though the original matrix is sparse. Second, because of rounding errors, coefficients of matrices should be kept small (preferably between 0 and 1).

To tackle this problem, two ideas can be applied (Stewart, 1994).

First, the following equality holds.

$$\frac{(t \times Q)^k}{k!} \times \pi = \frac{t \times Q}{k} \times \left(\frac{(t \times Q)^{k-1}}{(k-1)!} \times \pi \right) \tag{9.8}$$

Therefore, the k-th term of the series can be obtained from the $k-1$-th term by multiplying a vector (the $k-1$-th term) by the matrix $t.Q$.

Second, the following equality holds.

$$\exp(t \times Q) \times \pi \quad = \quad \exp(dt \times Q) \times (\exp((t - dt) \times Q) \times \pi) \tag{9.9}$$

Therefore, the computation of $\exp(t \times Q) \times \pi(0)$ can be replaced successive computations of $\pi(t_i) = \exp(dt_i \times Q) \times \pi(t_{i-1})$ such that $t_0 = 0$, $t_i = t_{i-1} + dt_i$, $dt_1 + dt_2 + \ldots = t$ and the dt_i's are small.

Combining these two ideas lead to several algorithms. These algorithms differ on the way they assess $\pi(t + dt) = \exp(d \times Q) \times \pi(t)$. The choice of dt is also an important issue.

Here come three important remarks.

- First, as we shall see in the next section, dt can be kept constant during the whole computation. Consequently, the matrix $dt \times Q$ can be computed once for all.
- Second, steady state probabilities are often reached before the given mission time t. Consequently, it is worth to test whether two successive terms $\pi(t_i)$ and $\pi(t_{i+i})$ are nearly equal, so to stop the computation as early as possible.
- Third, decomposing the calculation of $\exp(t \times Q) \times \pi$ in a series of computations n of $\exp(dt \times Q) \times \pi(t_i)$ has another significant advantage: It makes it possible to assess numerically the integral defining sojourn times as follows.

$$s_s(t) \quad \approx \quad \sum_{i=1}^{n} \frac{p_s(t_{i-1}) + p_s(t_i)}{2}$$

The basic algorithmic scheme is given Figure 9.4.

```
MatrixExponentiation(Q, π(0))
    π' = π(0) // the vector π(0) contains the initial probabilities
    σ = 0 // σ is the vector of sojour times
    dt = SelectDt(Q)
    P = dt × Q
    d = 0
    while d < t:
        d = d + dt
        π = exp(P) × π'
        σ = σ + (π' + π)/2
        if π ≈ π':
            break
        π' = π
    if d < t:
        σ = σ + (t − d) × π
    return (π, σ)
```

Figure 9.4: Basic algorithmic scheme to assess transient probabilities of a Markov chain.

Where t denotes the mission time and Q stands for the infinitesimal generator of the Markov chain.

To make this scheme a concrete algorithm, one has to answer the following questions.

- How to select dt?
- How to assess $\exp(dt \times Q) \times \pi$?
- How to test that $\pi \approx \pi'$?

The answers to these question, especially the second one, determine the actual algorithm.

9.4.2 Choice of dt

The algorithmic scheme pictured Figure 9.4 assumes a new value is selected for dt at each iteration. This value should depend on Q (which is invariant) and possibly on π. If dt depends only on Q, then it remains constant through the whole process. $dt \times Q$ can be computed once for all, which saves a lot of calculations. If dt depends also on π, then its value must be actually computed at each iteration. To be interesting, such a computation must save more time than it costs. In practice, it is very hard to find any heuristics that could justify this overhead. Consequently, we shall work with constant dt's.

The choice of an appropriate dt is a major issue of the implementation of iterative methods. If dt is chosen too large, the coefficients of the matrix are too large and rounding errors make the computation diverge. If dt is chosen too small, the computation is very expensive. Moreover, as noted in (Shampine et al., 1997), the impact of rounding errors may increase as the number of operations increases. If all the coefficients of the matrix $dt.Q$ range between 0 and 1, the expectation that rounding errors cause problems is minored. Hence, the following rule of thumb can be applied: choose dt such the product $\rho = dt \times \max_{i \in S} -q_{ii}$ stays in the range $[0,1]$. The product ρ provides a mean to choose dt in a uniform way:

$$dt \approx \frac{\rho}{\max_{i \in S} -q_{ii}} \tag{9.10}$$

Equality 9.10 depends only on the matrix Q. Therefore dt can be kept constant through the computation. It is worth noticing that, strictly speaking, to make the system solvable, the matrix $I + qt \times Q$ must be stochastic, which in turn means that ρ must be smaller than 1. In practice however, values greater than 1 can sometimes be used, even with much care. Moreover, dt must be a fraction of the considered time period, i.e. $t = dt \times k$ for some integer k.

Once the value of dt (or equivalently of ρ) is selected, the value of $\pi(t)$ can be assessed using the algorithm presented above. However, it is not possible to estimate the error with a single run. Stewart suggests in (Stewart, 1994) to perform a second calculation with a smaller value for dt, say $dt/2$. If the two results are not close enough, the process is reiterated.

9.4.3 Assessment of $\exp(dt \times Q) \times \pi$

As dt is kept constant, the sparse matrix $dt.Q$ can computed once for all, by computing the probability $p_{(s,t,r)}$ of each transition (s,t,r) as follows.

$$p_{(s,t,r)} = dt \times r \tag{9.11}$$

In the sequel, we shall call P the new matrix hence calculated: $P = dt \times Q$.

Then, calculating $\rho = P \times \pi$, for a given vector π of probabilities of states, can be done as shown in Figure 9.5 (recall that P is encoded as Q as a list of transitions).

```
1 MultiplyMatrixByVector(P, π):
2     ρ = 0⃗
3     forall (s, t, p) in P:
4         ρ(s) = ρ(s) − p × π(s)
5         ρ(t) = ρ(t) + p × π(s)
6     return ρ
```

Figure 9.5: Algorithm to calculate the product ρ of the probability matrix P by a probability vector π.

Equation 9.8 can be used to assess $\exp(P) \times \pi$ with a proviso however: it is indeed impossible to compute an infinite series. The iteration is thus stopped when two successive terms are close enough. Consequently, we have here again to test that two vectors are nearly equal. Figure 9.6 shows the algorithm.

```
CalculateSeries(P, π):
    ρ' = π
    τ' = π
    k = 0
    while true:
        k = k+1
        τ = MultiplyMatrixByVector(P, τ')
        τ = τ/k
        ρ = ρ'+τ
        if ρ ≈ ρ':
            break
        ρ' = ρ
        τ' = τ
    return ρ
```

Figure 9.6: Algorithm to calculate $\rho = \exp(P) \times \pi$.

9.4.4 Testing that $\pi \approx \rho$

To test whether two probability vectors π and ρ are nearly equal, we can select a threshold value ε and exit the loops if the following inequality holds.

$$\max_{i \in S} \left(\frac{|\pi_i - \rho_i|}{\pi_i} \right) \quad \leq \quad \varepsilon \tag{9.12}$$

where π_i (respectively ρ_i) denotes the i-th value in the vector π (respectively ρ). Other norms can be defined to test the convergence (Quarteroni et al., 2000). The above one works fine. ε must be chosen quite low (say 10^{-9}) in order not to stop too early.

9.4.5 Let's go!

Problem 9.3 Assessment algorithm for Markov models. We can now design a package `Calculator` to implement assessment algorithms for Markov models.

Question 1. Design a class `Vector` that implements all operations on vectors that are needed by the algorithms: filling with a scalar, multiplication by a scalar, sum....

Question 2. Design a class `Calculator` in order to implement assessment algorithms discussed in this section.pre Start with the preprocessing of the model, i.e. the calculation of the values of all expressions.

Question 3. In the class `Calculator`, implement the matrix exponentiation algorithm described in this section. This algorithm calculates the probabilities and sojourn times in each state for a given mission time.

Question 4. Complete your algorithm with the calculation of values and average values of rewards.

Exercise 9.4 Basic Markov models. We can now test the program we designed with some basic Markov models.

Question 1. Calculate the probabilities and the sojourn times in each state of the model pictured in Figure 9.7 (left) for different values of the parameter λ and different mission times. This model represents a non-repairable component. λ represents the failure rate of the component. The probability to be in state `F` (failed) at time t is given by the following formula.

$$U(t) = 1 - e^{-\lambda \times t} \tag{9.13}$$

Question 2. Calculate the probabilities and the sojourn times in each state of the model pictured in Figure 9.7 (right) for different values of the parameters λ and μ and different mission times. This model represents a repairable component. λ and μ represent respectively the failure rate and the repair rate of the component. The probability to be in state `F` (failed) at time t is given by the following formula.

$$U(t) = \frac{\lambda}{\lambda + \mu} \times \left(1 - e^{-(\lambda+\mu)\times t}\right) \tag{9.14}$$

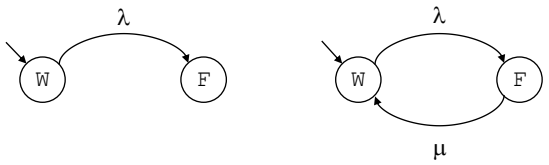


Figure 9.7: Basic Markov models

9.5 Application

This section contains a number of problems and exercises that can be solved with the program we designed in the previous sections.

Markov models are widely used in reliability engineering. Here follow a few exercises on this theme.

Exercise 9.5 2-out-of-3 systems. Consider a 2-out-of-3 system, i.e. a system that is working if at least two of its elements are working. Assume that each component fails according to an exponential distribution and with a failure rate $\lambda = 5.0 \times 10^{-4}$. Assume moreover that the system may experience a common cause failure which is also exponentially distributed with a failure rate $\lambda = 5.0 \times 10^{-5}$. Assume finally that the mean time to repair of system is 10 hours, whether there are 1, 2 or 3 components failed.

Question 1. Design a model in your domain specific language to encode the Markov chain described above.

Question 2. Assess the probabilities and the sojourn times in each state at $t = 1000, 2000, \ldots, 10000$.

Problem 9.6 Oil and gas production facility. This case study is borrowed to reference (Kawauchi and Rausand, 2002).

The system we consider is an oil and gas production facility consisting of height units pictured in Figure 9.8. Gas separated from the well fluid at upstream side is fed to the facility, treated through

separators (HPS-A,B,C) and dehydrators (DEH-A,B), and led to compressors (CMP-A,B). The make-up compressor (MUP) is installed to enable the CMP-A,B to discharge gas with full flow rate even if some of gas treatment units (HPS or DEH) are failed. It is assumed that the MUP is equipped with gas treatment units, which are dedicated to the MUP. The maximum throughput capacity for each unit is shown on the figure.

Reliability data for each component are given in the following table.

Unit	Failure rate (per hour)	Repair rate (per hour)
HPS-A,B,C	8.91×10^{-5}	2.54×10^{-3}
DEH-A,B	3.11×10^{-5}	3.95×10^{-3}
CMP-A,B	3.50×10^{-5}	5.14×10^{-3}
MUP	1.00×10^{-2}	

Question 1. Design a Markov model in your domain specific language to represent the system described above.

Hint: As the model is quite large, designing it by hand is tedious and error prone. It is therefore a good idea to generate it by means of... a Python script.

Question 2. Assess the probabilities and the sojourn times in each state for mission times $1, 2, \ldots 5$ years.

Question 3. Assess the expected production of the facility over $1, 2, \ldots 5$ years periods.

Question 4. Determine the best operation policy for the MUP:

- Start the MUP when either all HPS are failed, or all DEH are failed.
- Start the MUP when either HPS or DEH cannot produce 100% of the expected capacity.

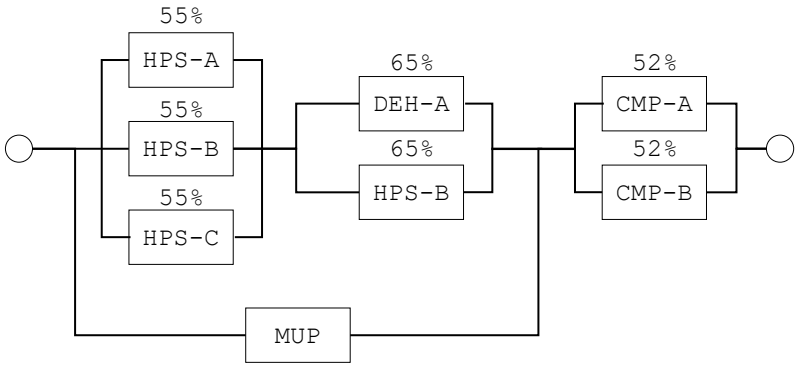


Figure 9.8: An oil and gas production facility

Markov models are often developed in biology and medicine. Here follow a few example.

Exercise 9.7 HIV Infection. This model has been developed to represent the effects of HIV infection. A sample of 467 HIV-positive men were monitored between 1984 and 1993. A 7-state Markov chain has been developed where stages 1–6 represent a range of values of an immunological marker for HIV, and state 7 corresponds to AIDS. Researchers gave the following estimates for the monthly transition rates of the model.

	1	2	3	4	5	6	7
1		0.055					
2	0.008		0.060				
3		0.008		0.039			
4			0.008		0.033		0.006
5				0.009		0.029	0.007
6					0.002		0.042

Question 1. Design a model in your domain specific language to encode the Markov chain described above.

Question 2. Starting from the first stage of HIV infection, estimate the probability of developing AIDS within k years, for $k = 5, 10, 15, 20$.

Question 3. Find the mean time to develop AIDS from each HIV state.
Hint: The mean time to develop AIDS is the first time at which half of the population initially in the stage under study has developed AIDS.

Question 4. HIV care involves a type of medication called antiretroviral therapy (ART) and regular visits with the doctor. Here follows a table with an estimated monthly cost (in dollars) of the treatment at each stage of the infection.

stage	1	2	3	4	5	6	7
cost	1800	1800	2000	2000	3000	3000	4500

Estimate the total cost of the treatment for a patient over k years, for $k = 5, 10, 15, 20$.

Exercise 9.8 Compartmental models. The COVID-19 crisis made the general audience discover compartmental models used by epidemiologists to study the propagation of a disease. Figure 9.9 shows such model.

When the epidemic outbreaks, 100% of the population is susceptible to be infected, but not infected yet. Then people start to be infected at some daily rate, e.g. $\lambda = 5 \times 10^{-5}$. The incubation period δ lasts on average a certain number of days, e.g. $\delta = 10$, which another way to say that an infected person gets either recovered or sick at a rate $1/\delta$. There is a certain probability π that an infected person develops the disease, e.g. 0.1. The disease lasts on average a certain number σ of days, e.g. $\sigma = 20$. Finally, a sick person has a certain probability ϕ to recover, e.g. $\phi = 0.9$.

Question 1. Design a model in your domain specific language to encode the compartmental model described above.

Question 2. Assuming a population of 6 millions inhabitants, study the evolution of the epidemic over a one year period:
- Determine the expected number of people declaring the disease.
- Determine the expected number of healthy carriers, i.e. the number of people who got infected without being sick.
- Determine the lethality of the disease, i.e. the expected number of people killed by the disease.

– Determine the hospital capacity required to treat sick persons.

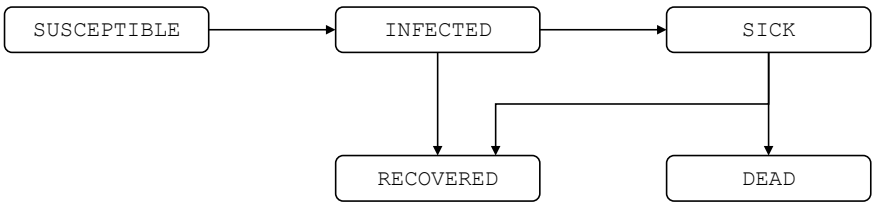


Figure 9.9: A compartmental model used in epidemiology

Chapter 10

Discrete Behavioral Simulations

Key Concepts
- Discrete behavioral models
- Synchronous and asynchronous models
- Cellular automata
- Discrete event simulations
- Agent-based simulations

10.1 Introduction

In the previous chapter, we considered "static" simulations, i.e. simulations that consider the state of the system under study at a given point in time, disregarding how this state has been reached, if this makes any sense. In this chapter, we shall study simulations that mimic its evolution through the time. Physics, chemistry and many other branches of science are all about the description and the prediction of the dynamical behavior of physical systems. Systems of differential equations are the key mathematical framework to do so. However, we shall not consider them here. Rather, we shall concentrate on *discrete behavioral models*, as they are in general the suitable tool in the context of performance engineering. Moreover, solving systems of differential equations requires dedicated tools the study of which goes beyond the scope of this book.

Discrete behavioral models represent the system under study as a set of interacting components. They assume that the system changes of state at discrete points in time, and thus stays unchanged in between these points.

Roughly speaking, there are two main classes of discrete behavioral models:

- *Synchronous models* in which all components of the system perform an action at the considered points of time. In these models the time is seen as a sequence of evenly distributed dates, like the tics of a clock. The time is thus put in one-to-one correspondence with integers: initial state (0), first tic (1), second tic (2)...
- *Asynchronous models* in which only one component, or at least only a small subset of components, is performing an action at a time. In these models, the time is also seen as a sequence of dates, but these dates are unevenly distributed. The time is thus put in one-to-one correspondence with countable subsets of $\mathbb{R}^+$.

In both cases, the time is thus assimilated as a countable set of dates, hence the name discrete models. Discrete models are opposed to continuous models (like systems of differential equations) that consider the time as continuous, i.e. in one-to-one correspondence with $\mathbb{R}^+$, or at least with dense subsets of $\mathbb{R}^+$ like the set $\mathbb{Q}^+$ of non-negative rational numbers.

In this chapter, we shall look in turn at both types of models: first, at synchronous models, with the notion of *cellular automata*, then at asynchronous models, with the notion of *discrete event systems*.

Aside the above classification, another one is extremely important: in some models (synchronous or asynchronous), the number of components as well as the way components interact may evolve through the time. In general, the terms cellular automaton and discrete event system is reserved for the latter, while the former are called *agent-based models*. As of today, there is no unique, widely accepted, generic mathematical framework to describe agent-based models. For this reason, and because they are significantly more difficult to implement than discrete event systems, we shall not consider them in this book. There are however programming environments and languages dedicated to agent-based simulations, such as NetLogo (Wilensky and Rand, 2015).

10.2 Cellular Automata

10.2.1 Presentation

A *cellular automaton* consists of a regular grid of cells. The grid can be in any finite number of dimensions, usually one or two. The grid forms a geography, i.e. each cell has a neighborhood made of the adjacent cells. In general, one considers cells are in one of a finite number of states.

In the initial state ($t = 0$), each cell is thus in one state. Its state at time $t + 1$ is obtained from its state and the states of the cells in its neighborhood as time t, by some fixed rule, which is the same for all cells and does not change over time. The states of all cells are updated simultaneously. The model is thus synchronous.

For instance, the *Game of Life*, invented by British mathematician John Horton Conway in 1970, and popularized by Martin Gardner (Gardner, 1970), is played on a two dimensional rectangular grid. The neighborhood of a cell consists of the eight cells surrounding that cell. Each cell can be either dead or alive. The updating rule is very simple:

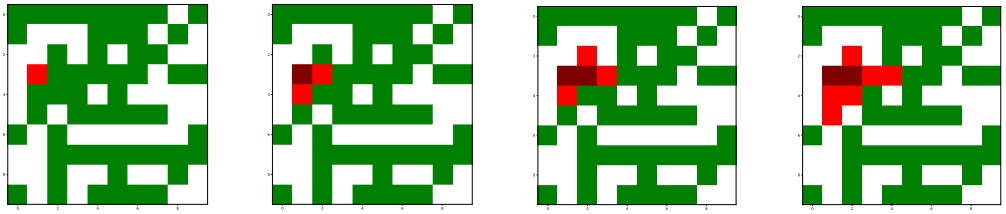
- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Although extremely simple, this mathematical game shows a quite surprising richness, with oscillating configurations, sliding configurations, configurations expanding *ad infinitum* or having other intriguing properties.

In his famous book "A New Kind of Science", Stephen Wolfram (Wolfram, 2002) showed that the even simpler model of one dimension cellular automaton, where the neighborhood of a cell just consists of the two adjacent cells, have the expressive power of Turing machines, i.e. can compute any computable mathematical function.

Cellular automata can be used, to some extent, to model physical and even phenomena. As an illustration, we shall consider a simple model of forest fire propagation. The map consists of a finite rectangular two-dimensional grid. The neighborhood of a cell (i, j) consists of the four adjacent cells $(i-1, j)$, $(i, j-1)$, $(i+1, j)$ and $(i, j+1)$ (cells on the border of the grid have thus less neighbors). Each cell is in one of the four states `empty`, `forest`, `burning` or `burnt`. The initial state of each cell is either `empty` with a certain probability p, or `forest` with the probability $1 - p$. Moreover, a fire is ignited, i.e. that a cell of the grid is chosen at random and its state is set to `burning`. The updating rule is a follows.

- A cell that was in the state `empty` remains in that state.
- A cell that was in the state `forest` remains in that state if none of its neighboring cell is in the state `burning`. Other, it is either ignited, i.e. passes to the state `burning` with a probability q, or remains in the state `forest` with a probability $1 - q$.
- A cell that was in the state `burning` either passes to the state `burnt` with a probability r, or remains in the state `burning` with a probability $1 - r$.
- Finally, a cell that was in the state `burnt` remains in that state.



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

Figure 10.1: The first steps of a simulation on a 10×10 grid.

Figure 10.1 shows the first steps of a simulation on a 10×10 grid. Empty cells are pictured in white, forest cells in green, burning cells in red and finally burnt cells in maroon.

`A priori`, implementing the above rules is quite easy. The only thing one must be careful about is to ensure that all cells are updated simultaneously, i.e. that the states of cells at time $t+1$ are all calculated from their states at time t. To do so, it suffices basically to create two grids and to alternate between these two grids. For instance, for even values of t (starting from 0), the first grid encodes the current state and the second one the next state, while for off values of t that is the converse.

The above principle has however a drawback: it does a lot of useless work. For instance, the states of empty cells is updated at each step, while we know upfront that it will not change. Similarly for burnt cells. Moreover, it consumes quite a lot of memory as it requires to maintain two grids. The solution consists in maintaining the list `burningCells` of the cells in the state `burning`. Then, the updating process is performed into three steps:

1. One builds the list `ignitableCells` of cells in the state `forest` with at least one neighbor in the state `burning`. This is done by iterating on the list `burningCells`. In order not to insert twice a cell in the list `ignitableCells`, these cells can be given the fictitious state `ignitable`. Once `ignitableCells` is fully calculated, the states of its members can be turned back to `forest`.
2. The list `burningCells` is memorized has `previousBurningCells` and a new, empty, list `burningCells` is created. The list `previousBurningCells` is iterated through and the new state of its cells is chosen at random. If a cell remains in the state `burning`, it is appended to the list `burningCells`.
3. The list `ignitableCells` is iterated through and the new state of its cells is chosen at random. If a cell passes to the state `burning`, it is appended to the list `burningCells`.

Not only the above idea saves a lot of memory, but it accelerates significantly the updating process.

It can be applied in many circumstances, beyond the implementation of cellular automata.

Problem 10.1 Forest Fire. The objective of this problem is to implement in Python a forest fire simulator, according the principles described in this section.

Question 1. Design a class `Map` that implements:
- A method `Initialize` that chooses the initial states of cells;
- A method `IgniteFire` that ignites a fire somewhere in the map.
- A method `UpdateState` that updates the current state of each cell of the grid.

Question 2. Using `matplotlib`, design a class `Animator` that shows graphically the fire propagation.
Hint:

  - To visualize grids, you can use the method `imshow` of the module `matplotlib.pyplot`.
  - To animate the simulation, you can use the method `FuncAnimation` of the module `matplotlib.Animation`. To make this method work with `Spyder`, you have to enter the command `%matplotlib qt` in `Spyder`'s console.
  - Stop the simulation when no more cell is burning, you have to define an iterator and passes it as the argument `frames` of `FuncAnimation`. Look at the proposed solution if you cannot make it.

Question 3. Design a class `Simulator` that implements a Monte-Carlo simulation on fire propagations. Two indicators are of interest:
  - The number of steps before the fire extinguishes.
  - The number of cells burnt at the end of the fire propagation.

Question 4. Perform a Monte-Carlo simulation for on a 100×100 grid, with the following probabilities:
  - Initial probability of drawing a forest: 0.55.
  - Probability that an ignitable cell is ignited: 0.70.
  - Probability that a burning cell is extinguished: 0.20.

  What do you observe?

Models such as the above forest fire model show interesting properties.

First, for some values of the parameters, the indicators of interest (e.g. the number of burnt cells at the end of the simulation) may follow a power law (see Section 8.3.2).

Second, still according to some values of the parameters, they may exhibit phase transitions. In physics, the term *phase transition* is most commonly used to describe transitions between solid, liquid, and gaseous states of matter. These states of the matter have different physical properties. During a phase transition of a given medium, certain properties of that medium change discontinuously, in function of some parameter, such as the temperature or the pressure. Phase transitions commonly occur in the nature and are used today in many technologies. From a theoretical point of view, transition phases may be understood as zero–one laws. In probability theory, a *zero–one law* is a result the statement is that the limit of certain probabilities must be either 0 or 1, but no intermediate value. Zero-one laws appear typically in models of statistical physics.

10.2.2 Let's go!

Problem 10.2 Ising Model. The Ising model, named after the German physicist Ernst Ising, is a mathematical model of ferromagnetism in statistical mechanics. Translated in terms of cellular automata, it goes as follow.

It consists of a two-dimensional rectangular grid. The neighborhood of a cell consists in the four adjacent cells. Each cell represents the spin of a particle, which can be either up or down. The initial configuration is chosen at random, with an equiprobability for spins to be up and down.

The updating rule is a follows. One counts the number n of cells in the neighborhood oriented as the considered cell. The energy Δ_E to flip the cell is $2 \times n$. Then, the spin of the cell is flipped with a probability p defined as follows.

$$p \stackrel{def}{=} \exp\left(-\frac{\Delta_E}{T}\right) \tag{10.1}$$

In the above equation T represents the temperature. Intuitively, the higher the temperature, the higher the probability of flipping cells.

The objective of the problem is to implement the Ising model.

Question 1. Design a class `Map` that implements:
- A method `Initialize` that sets the initial states to `dead`;
- A method `UpdateState` that updates the current state of each cell of the grid.
  Hint: In this case, there is not better solution that to implement the dual grids idea to go from the current state to the next one.

Question 2. Using `matplotlib`, design a class `Animator` that shows graphically the state of the grid.
Hint:
- To visualize grids, you can use the method `imshow` of the module `matplotlib.pyplot`.
- To animate the simulation, you can use the method `FuncAnimation` of the module `matplotlib.Animation`. To make this method work with `Spyder`, you have to enter the command `%matplotlib qt` in `Spyder`'s console.

Question 3. Test your program with a 100×100 grid and temperatures ranging from 1.0 to 5.0 (and about 1000 steps). What do you observe?

Problem 10.3 Game of Life. The objective of this problem is to implement in Python Conway's Game of Life according the principles described in this section.

The idea is to maintain the list of living cells. To compute the next state, one considers all the cells in the neighborhood of living cells. Then, one classifies these cells into four categories:
- The cell that were alive and remain alive.
- The cell that were alive and become dead.
- The cell that were dead and remain dead.
- The cell that were dead and become alive.

Question 1. Design a class `Map` that implements:
- A method `Initialize` that sets the initial states to `dead`;
- A method `GetNeighborhoodOfCell` that returns the neighborhood of a cell.
- A method `GetNextStateOfCell` that calculates the next state of a cell.
- A method `UpdateState` that updates the current state of each cell of the grid.

Question 2. Using `matplotlib`, design a class `Animator` that shows graphically the state of the grid.
Hint:
- To visualize grids, you can use the method `imshow` of the module `matplotlib.pyplot`.
- To animate the simulation, you can use the method `FuncAnimation` of the module `matplotlib.Animation`. To make this method work with `Spyder`, you have to enter the command `%matplotlib qt` in `Spyder`'s console.
- Stop the simulation when no more cell is alive or when a fixed number of iterations has been reached, you have to define an iterator and passes it as the argument `frames` of `FuncAnimation`. Look at the proposed solution if you cannot make it.

Question 3. Perform a simulation for on a 17×18 grid, with the initial structure, called "Pulsar" by John Conway, pictured in Figure 10.2. What do you observe?

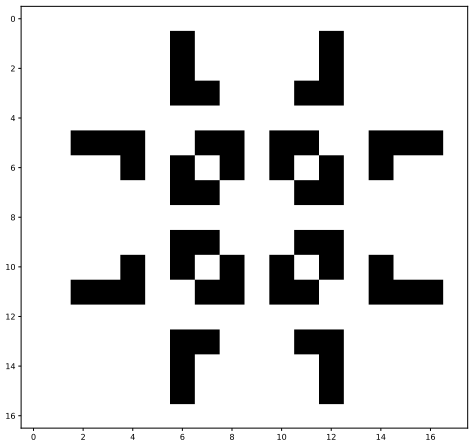


Figure 10.2: The “Pulsar" structure at Game of Life.

Problem 10.4 Schelling Model of Segregation. In 1971, the American professor of political economy, Nobel prize of economy, Thomas Schelling published a simple mathematical model which gives a possible explanation about why American cities are are segregated in white districts and Afro-American districts (Schelling, 1971). Translated in terms of cellular automata, the model works as follows (we give below our own interpretation).

The city is represented by a two-dimensional rectangular grid. Each cell represents a house, occupied by a family. The neighborhood of a cell consists of the 24 surrounding cells, see Figure 10.3. Cells can be in three states: `empty`, `white` or `black`. Initial states are chosen at random, with a small proportion of empty cells, say 10%, and the same proportion of white and black cells, thus 45%. The updating rule is as follows.

- The family occupying a house is "unhappy" if less than a proportion p of its neighbors are of its community. The proportion p is typically taken to be 0.3.
- Unhappy families move to an empty house, picked at random.

The objective of this problem is to implement the Schelling model and to study the effects of the choice of the proportion p.

Question 1. Design a class `City` that implements:
- A method `Initialize` that sets the initial states of houses;
- A method `UpdateState` that updates the current state of each cell of the grid.

The method `UpdateState` should start by creating two lists: the list of unhappy families and the list of empty houses. Then, it must shuffle the list of unhappy families. Finally, for each unhappy family, it should pick up at random an empty house, a move the family to that house.

Question 2. Using `matplotlib`, design a class `Animator` that shows graphically the state of the city.

Question 3. Perform experiments with 100×100 grid and values of p ranging from 0.1 to 0.5 (and about 1000 iterations). What do you observe?

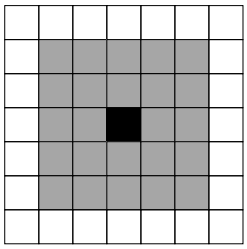


Figure 10.3: Schelling's neighborhood.

10.3 Deterministic Discrete Event Systems

10.4 Stochastic Discrete Event Systems

10.4.1 Presentation

When considered at a high level of abstraction, most of the technical and socio-technical systems show a discrete behavior. Their temporal evolution can be thought as a sequence of events. Initially, the system is in a given state. Then, an event occurs and the system changes instantaneously of state. It remains in this second state until a new event occurs that makes it changes instantaneously of states. And so on.

As an illustration, consider the queue of customers waiting to be served at the counter of a bank. To describe the dynamical behavior of this system, we need to represent:

- The arrival process of clients in the queue, which can be seen as discrete: clients arrive one by one, at unevenly distributed instants.
- The service process at the counter, which can also be seen as discrete: the client at the counter is served after a finite amount time, even though this time may vary.
- The behavior of clients in the queue, who may give up after a while, if their remaining waiting time looks too long. This behavior can again be seen discrete: once his mind is made, the client quits definitely the queue.

The types of questions we are interested in about such system are typically concerned with measures of performance and might include:

- How long do clients wait in the queue before they are served?
- What is the average length of the queue?
- What is the probability that the queue will exceed a certain length?
- What is the proportion of clients who will leave dissatisfied?

These are questions that need to be answered so to evaluate how to improve the situation. Some of the problems that are often investigated in practice are:

- Is it worth to invest money in reducing the service time?
- How many counters should be opened?
- Should priorities for certain types of clients (e.g. disabled people) be introduced?
- Is the waiting area for clients adequate?

Indeed, in reality, a client does not move instantaneously from his home to the bank. He is moving continuously from the former to the latter location. But for the purpose of the analysis, a discrete abstraction is sufficient and convenient. We can think the system as if it was discrete.

In some cases, it is possible to answer the above questions by means of analytical formulas. In most of the cases however, computer-based experiments are the only solution to study the problem (aside indeed *in situ* observations).

In the reminder of this section, we shall now look at how to implement such computer-based experiments. But before entering into programming details, we need to formalize the problem, i.e. to give it a mathematical definition.

10.4.2 Formal Definition

The following formal definition of discrete event systems is strongly inspired from the notion of guarded transition systems (Rauzy, 2008), itself generalizing formalisms like stochastic Petri nets (Ajmone-Marsan et al., 1994).

Definition A *discrete event system* is a pair $\langle \mathscr{S}, T \rangle$ where:

- $\mathscr{S}$ is a set of *states*. $\mathscr{S}$ may be finite or infinite.
- T is a finite set of *transitions*. Each transition t of T is a triple $\langle g, d, a \rangle$ where:
  - g is a Boolean condition, i.e. a function from $\mathscr{S}$ to $\{0,1\}$ (representing respectively false and true). g called the *guard* of the transition. We say that the transition t is *enabled* in the state $S \in \mathscr{S}$ if $g(S) = 1$.
  - d is function from $\mathscr{S}$ to $\mathbb{R}^+$. d is called the *delay* of the transition. In many models, delays do not depend on the current state S. They may however be either deterministic or stochastic.
  - a is a function from $\mathscr{S}$ to $\mathscr{S}$. a is called the *action* of the transition. Assume that at a given step i, the system is in the state S_i and the transition is enabled in that state. Then, *firing* the transition is making the system change from state S_i to state $S_{i+1} = a(S_i)$.

Consider again our bank example and assume for now that no client gives up while waiting his turn. The state of the system can be simply described by a pair (q, s) of integer variables, where:

- q represents the number of clients in the queue, and
- s represents the number of clients currently served.

The set $\mathscr{S}$ of possible states of the system is thus $\mathbb{N} \times \mathbb{N}$, i.e. the set of all possible pairs of integers. This set is infinite. For practical reasons, we may assume that there are never more than 10 clients waiting (there is not enough room in the bank agency) and that only one client is served at a time. In this case, the set $\mathscr{S}$ is reduced to product of integer ranges $[0,10] \times [0,1]$, which is indeed finite.

The evolution of the system can be represented by means of three transitions:

- A first transition $t_a : \langle g_a, d_a, a_a \rangle$ to represent the arrival of a client in the queue.
  - g_a is constantly equal to 1, as nothing prevents the arrival of a new client. We may however limit the size of the queue, for instance to 10 clients. In this case, $g_a = q \leq 10$.
  - d_a is the interarrival delay. d_a obeys typically an exponential distribution.
  - a_a consists in incrementing q by 1.
- A second transition $t_b : \langle g_b, d_b, a_b \rangle$ to represent the beginning of the service of a client.
  - $g_b = q > 0 \wedge s < 1$, i.e. it checks that there is a client in the queue and the counter is available.
  - $d_b = 0$ if we assume that the next client starts to be served immediately after the service of the previous one is completed.
  - a_b consists in decrementing q by 1 and incrementing s by one.
- A third transition $t_c : \langle g_c, d_c, a_c \rangle$ to represent the completion of the service of a client.
  - $g_c = s > 0$, i.e. it checks that there is actually a client currently served.
  - d_c is the duration of the service of client. d_c obeys typically a triangular distribution.
  - a_c consists in decrementing s by one.

In a more elaborated model, we may represent clients individually, i.e. represent the queue by a list of clients. There may be several counters and so on. The three above transitions have then to be modified so to add and remove clients from the queue and serve the clients at the counters.

Abstract Syntax It is convenient to give an abstract syntax to discrete transition systems. In the sequel, we shall denote a transition $t : \langle g, d, a \rangle$ as $t : g \xrightarrow{d} a$, using Boolean formulas to describe guards and instructions (in pseudo-code) to describe actions.

As an illustration, the three transitions of our example can be denoted as follows.

t_a : true $\xrightarrow{d_a} q = q + 1$
$t_b : q > 0 \wedge s < 1 \xrightarrow{0} q = q - 1, s = s + 1$
$t_c : s > 0 \xrightarrow{d_c} s = s - 1$

Semantics A discrete event system encodes implicitly a set of possible executions. To define formally this set of executions, we need to introduce the notion of schedule.

Let $M : \langle \mathscr{S}, T \rangle$ be a discrete event system. A *schedule* of M is a function δ from T in $\mathbb{R}^+ \cup \{\infty\}$, i.e. a function that associates a firing date to each transition of M. If a transition t is enabled in the current state, then $\delta(t) \in \mathbb{R}^+$. Otherwise, $\delta(t) = \infty$.

An *execution* of M is a sequence $(S_0, \delta_0) \xrightarrow{t_1} (S_1, \delta_1) \cdots \xrightarrow{t_n} (S_n, \delta_n)$, $n \geq 0$, where the S_i's are states of $\mathscr{S}$, the δ_i's are schedules of M and the $t_i : g_i \xrightarrow{d_i} a_i$'s are transitions of T.

An execution must verify the following conditions.

1. $g_{i+1}(S_i) = 1$, i.e. that the transition fired at step $i+1$ must be enabled in the state S_i.
2. $S_{i+1} = a_{i+1}(S_i)$ for $1 \leq i \leq n$, i.e. that the state S_{i+1} is obtained from the state S_i by firing the transition t_{i+1}.
3. $\delta_i(t_{i+1}) \leq \delta_i(t)$ for all transitions $t \in T$, $t \neq t_i$ and $0 \leq i < n$, i.e. that the transition that is fired at step $i+1$ must be one of those with the earliest firing dates.
4. δ_{i+1} is obtained from δ_i as follows. For all $t : g \xrightarrow{d} a \in T$:
   - If $g(S_{i+1}) = 0$, then $\delta_{i+1}(t) = \infty$.
   - If $t = t_{i+1}$ and $g(S_{i+1}) = 1$, then $\delta_{i+1}(t) = \delta_i(t) + d$.
   - If $t \neq t_{i+1}$, $g(S_i) = 1$ and $g(S_{i+1}) = 1$, then $\delta_{i+1}(t) = \delta_i(t)$.
   - If $t \neq t_{i+1}$, $g(S_i) = 0$ and $g(S_{i+1}) = 1$, then $\delta_{i+1}(t) = \delta_i(t) + d$.

As an illustration consider again our bank example.

Initially, there is no client in the bank, so $q_0 = 0$ and $s_0 = 0$. In this state, only the transition t_a is enabled. Assume, for instance, that the first client arrives after 3 minutes. Then $\delta_0(t_a) = 3$, $\delta_0(t_b) = \infty$ and $\delta_0(t_c) = \infty$.

As t_a is the only enabled transition at step 0, the first event that occurs in the system is the firing on this transition, at the date $\delta_0(t_a)$, i.e. 3. The state S_1 is thus $(q_1 = 1, s_1 = 0)$.

In this state, both transitions t_a and t_b are enabled. A new delay is thus calculated for the transition t_a, e.g. 4, and the delay 0 is calculated for the transition t_b. The schedule δ_1 is thus $\delta_1(t_a) = 3 + 4 = 7$, $\delta_1(t_b) = 3 + 0 = 3$ and $\delta_1(t_c) = \infty$.

As $\delta_1(t_b) < \delta_1(t_a)$, t_b is fired at the next step, i.e. at date 3 (again). We have thus $q_2 = 0$ and $s_2 = 1$. In this state, both transitions t_a and t_c are enabled. As t_a was already enabled in state S_1, its firing date remains unchanged, i.e. $\delta_2(t_a) = \delta_1(t_a) = 7$. The transition t_c was not enabled in state S_1, so a new delay is calculated for this transition, e.g 2, and $\delta_2(t_c) = 3 + 2 = 5$.

And so on.

10.4.3 Advanced Constructs

In addition to the above formalization, two constructs prove to be useful: weights and memory policies associated with transitions. We shall review them in turn.

Weights of Transitions When there are several candidate transitions, one of them is selected and fired. The question is therefore how this choice is made. In our implementation, it is made at random. Each transition t is associated with a *weight* w_t, i.e. a function that takes the current state as parameter and

returns a non negative number. The probability $p_t(S)$ to select the candidate transition t in the set of C of candidate transitions in the state S is defined as follows.

$$p_t(S) \stackrel{def}{=} \frac{w_t(S)}{\sum_{t \in C} w_t(S)} \tag{10.2}$$

Note that the weight of candidate transition is a particular state may be equal to zero. In this case, the transition cannot be selected.

By default, the weight of transitions is equal to 1 and is independent of the current state. Consequently, all candidate transitions have the same probability to be fired.

Memory Policies of Transitions It may be the case that a transition t, scheduled at the date d_s to be fired at the date d_f, ceases to be enabled at some date d, $d_s \leq d \leq d_f$ because another transition has been fired and the guard of t is no longer satisfied. In this case, t is simply removed from the schedule. The next time t is enabled, a new delay is calculated. This is the default behavior.

In some case, it may be interesting however to consider that the task represented by transition is simply interrupted for a while, until it can be restarted. In this case, the time remaining until the transition will be fired is memorized when it ceases to be enabled, and use as the new delay when it is enabled again.

10.4.4 Stochastic Simulations

Discrete event systems are of special interest to represent evolutions of systems subject to uncertainties. The arrival date and the service time of clients in our queuing example are typical examples of such uncertainties.

A *stochastic discrete event system* is discrete event system in which at least one of the delays associated with transitions is stochastic, i.e. obeys a certain probability distribution. It is worth noticing that, apart for the delays associated with transitions, all mechanisms involved in stochastic discrete event systems are deterministic.

Stochastic discrete event systems are in general assessed by means of Monte-Carlo simulations. The idea is fairly simple: one performs a large number of executions, drawing at random delays of transitions (each time the calculation of a firing date is involved). The executions are expanded until a certain mission time is reached. Along each execution, the value of some indicators (random variables) are calculated. Then, one performs statistics on these values, over the different executions.

The indicators that may be calculated belong to three categories.

First, indicators regarding to transitions. E.g.

– The number of transitions fired during the execution.
– The number of times a given transition has been fired during the execution.
– The first date at which a given transition has been fired during the execution.

Second, indicators relying on predicates over states. A *predicate* over states is a Boolean function over states. E.g.

– The first date at which a state satisfying the predicate has been reached during the execution.
– The number of times a state satisfying the predicate has been reached during the execution.

Third, indicators relying on reward over states. A *reward* over states is a real-valued function over states. E.g.

– The minimum, maximum or mean value the reward takes during the execution.

The above list is indeed non-exhaustive, although is covers most of the needs. It can be extended at will.

In our bank example, we may be interested in:

– The number of times the transition t_a is fired, which gives an indicator on the number of clients who entered in the bank.
– The number of times the transition t_c is fired, which gives an indicator on the number of clients who have been served.

- The maximum and mean values of the reward q, which gives an indicator on the maximum and mean numbers of clients waiting.
- The mean value of the reward that takes the value 1 when $s = 0$ and the value 0 otherwise, which gives an indicator on the time the operator at the counter remains idle.

We can now look at how to implement these ideas.

10.4.5 Implementation

In the above description, most of the notions apply to any stochastic discrete event system. The Python package `Alea` implements them and makes it possible to develop stochastic simulators for a large class of discrete event systems at little programming cost. `Alea` is itself developed in Python, with only pedagogical objectives in mind. It is made of two modules: `DiscreteEventSystems` and `DiscreteEventSimulators`.

The Module `Alea.DiscreteEventSystems` This module implements several classes to describe discrete event systems:

`DiscreteEventSystem`
: This class manages a list of instances of the class `Transition`.

`Transition`
: A transition is made of a name, a guard, a delay and an action.
  - The guard must be a function that takes an instance of `State` (or of a class derived from `State`) as parameter and returns a Boolean.
  - The delay must be an instance of the class `Delay` (or of a class derived from `Delay`).
  - The action must be a function that takes an instance of `State` (or of a class derived from `State`) as parameter and modifies it.

`State`
: This is an abstract class that must be derived to implement concrete discrete event systems.

`Delay`
: This is also an abstract class that implements a method `Calculate` that takes an instance of `State` (or of a class derived from `State`) as parameter and returns a positive floating point number.

The module `Alea.DiscreteEventSystems` provides a number of classes derived from the class `Delay`. These classes implements concrete delays (the calculation of which do not depend on the current state):

`DiracDelay`
: Constant delays.

`UniformDelay`
: Stochastic delays uniformly distributed between two bounds.

`ExponentialDelay`
: Stochastic delays exponentially distributed.

`TriangularDelay`
: Stochastic delays distributed according to a triangular distribution described by two bounds and a mode.

`StepwiseEmpiricalDelay`
: Stochastic delays distributed according to a stepwise distribution described by a list of points.

`LinearEmpiricalDelay`
: Stochastic delays obtained by means of a linear interpolation of a distribution described by a list of points.

```
class BankState(Alea.DiscreteEventSystems.State):
  def __init__(self, bank):
    Alea.DiscreteEventSystems.State.__init__(self, bank)
    self.maximumSize = 10
    self.numberOfClientsWaiting = 0
    self.numberOfClientsAtCounter = 0

  @staticmethod
  def New(bank):
    return BankState(bank)

  def Reset(self):
    self.numberOfClientsWaiting = 0
    self.numberOfClientsAtCounter = 0

  def IsClientArrivalEnabled(self):
    return self.numberOfClientsWaiting<self.maximumSize

  def FireClientArrival(self):
    self.numberOfClientsWaiting += 1
```

Figure 10.4: Implementation of the class `BankState` (partial view)

More delays, depending or not of the current state, can be added by deriving the abstract class `Delay` or one of the above concrete class.

The class `DiscreteEventSystem` is designed according to the pattern "*Factory*" (Gamma et al., 1994). It provides methods to create states and transitions.

Implementation of the Bank Counter Example Figure 10.4 shows a partial view on the class `BankState` that can be used to implement the discrete event system representing our bank example.

The class `BankState` is derived from the class `Alea.DiscreteEventSystems.State`. It represents the concrete state of the queue, i.e. the number of clients waiting to be served and the number of clients currently served.

The static method `New` creates a new bank state. *Static methods*, which are introduced by the directive `@staticmethod`, do not take an instance of the class as first parameter. They can be called using the class, e.g. `BankState.New(myBank)`. Therefore, `BankState.New` can just be seen as a function that takes an instance of `DiscreteEventSystem` as parameter and returns an new instance of `BankState`.

The method `Reset` overrides the one of the base class `State`. It resets the queueing system on its initial state.

Finally, the methods `IsClientArrivalEnabled` and `FireClientArrival` implement the guard and the action of the transition representing the arrival of a client.

Figure 10.5 shows how to use the class `BankState` to implement the queueing system. First, an instance of the class `DiscreteEventSystem`, with the method to create a state as parameter. Second, the transition representing a client arrival is created. The method that creates this transition takes four parameters: the name, the guard and the action defined in the class `BankState` and an exponential delay (of mean 5). Other transitions of this discrete event system are created in the same way.

Exercise 10.5 Bank Counter. We can now implement the discrete event system representing our bank counter example.

```
bank = Alea.DiscreteEventSystems.DiscreteEventSystem(BankState.New)
T1 = bank.NewTransition("ClientArrival",
  BankState.IsClientArrivalEnabled,
  Alea.DiscreteEventSystems.ExponentialDelay(5),
  BankState.FireClientArrival)
```

Figure 10.5: Implementation of discrete event system representing the bank (partial view)

Question 1. Complete the class `BankState` with methods to represent the guards and the actions of the transitions representing the beginning and the completion of the service of a clients. We shall assume that:

- The arrival of clients is exponentially distributed with a mean interarrival time of 5 minutes.
- The next client starts to be served immediately after the service of the previous one is completed.
- The service time of a client is distributed according to a triangular distribution with a lower bound of 3 minutes, an upper bound of 8 minutes and a mode of 5 minutes.

Question 2. Using the methods developed in the previous question, complete the description of the discrete event system that represents the queuing system.

A class `Weight` implements base weights. It is associated by default to all transitions. This class can be derived to create special weights.

The memory policy of transitions is by default `RESTART`, which means that a new delay is calculated each time the transition gets enabled. To memorize the remaining time and use it as the new delay, the attribute `policy` of the transition must be set to `MEMORY`.

The Module `Alea.DiscreteEventSimulators` This module implements several classes to describe stochastic simulators of discrete event systems:

`DiscreteEventSimulator`
: This class encodes stochastic simulators. It manages a list of instances of the class `Indicator` according to the design pattern "Factory".

`Schedule`
: This class implements schedules.

`Event`
: This class implements events.

`Indicator`
: This abstract class represents indicators to be calculated via Monte-Carlo simulations.

The class `Indicator` provides a number of services, including calculating the size, the mean, the standard deviation, various error factors, the minimum and maximum values of the considered sample. It is also possible to record all of the values of the sample, so to be able to calculate quantiles and histograms (e.g. using `matplotlib`, as explained Section 8.3.2).

It is derived into concrete classes that implement the indicators described in Section 10.4.4:

`FiringCount`
: Number of transitions fired during an execution.

`TransitionFiringCount`
: Number of times a given transition is fired during an execution.

`TransitionFirstFiringDate`
: First date at which a given transition is fired during an execution.

```
simulator = Alea.DiscreteEventSimulators.DiscreteEventSimulator()
simulator.SetSystem(bank)
I1 = simulator.NewTransitionFiringCount("I1", T1, False)
...

simulator.MonteCarloSimulation(60, 1000)

print("I1\t{0:.2f}".format(I1.GetMean()))
...
```

Figure 10.6: Implementation of stochastic simulator for the bank counter (partial view)

`PredicateSatisfactionCount`
: Number of times an execution goes through a state that satisfies a given predicate.

`PredicateFirstSatisfactionDate`
: First date at which an execution enters into a state that satisfies a given predicate.

`PredicateSatisfactionTime`
: The time during which the predicate is satisfied during an execution.

`RewardMinimumValue`
: Minimum value of a reward during an execution.

`RewardMaximumValue`
: Maximum value of a reward during an execution.

`RewardMeanValue`
: Mean value of a reward during an execution.

Monte-Carlo Simulation To perform a Monte-Carlo simulation on our bank counter example, one needs to proceed in four steps:

1. Creation of the discrete event system representing the bank counter (as in Exercise 10.5).
2. Creation of an instance of `DiscreteEventSimulator` and of the indicators one wants to assess.
3. Monte-Carlo simulation.
4. Display of the results for each indicator.

Figure 10.6 shows a partial view of the three last steps.

Exercise 10.6 Bank Counter (bis). We can now implement the Monte-Carlo simulator for our bank counter example.

Question 1. Complete the program so to assess:
- The number of clients entering in the bank in one hour.
- The number of clients served in one hour.
- The maximum and mean numbers of clients waiting.
- The time the operator at the counter remains idle.

Question 2. Perform a Monte-Carlo simulation for a 1 hour mission time. What do you observe? Was it expected?

Question 3. One of the problems is that some clients wait until the bank agency closes, while they have no chance to be served. How would you improve the situation?

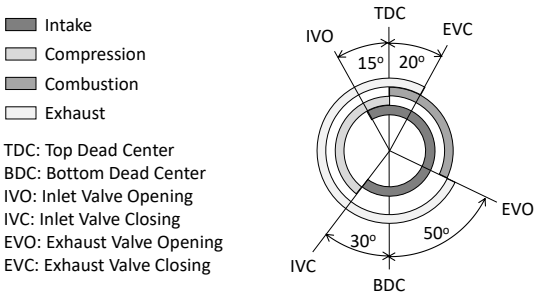


Figure 10.7: Time diagram for a petrol four stroke engine.

10.4.6 Let's go!

We shall start with two purely deterministic models.

Problem 10.7 Four Stroke Engine. The four strokes (cycles) of a four-stroke engine are:

- Intake, in which the piston pulls an air-fuel mixture into the cylinder by producing vacuum pressure through its downward motion.
- Compression, in which the piston compresses the air-fuel mixture.
- Combustion, in which the compressed air-fuel mixture is ignited and the explosion produces a mechanical work via the crankshaft.
- Exhaust, in which the spent air-fuel mixture is expelled.

In practice, these four strokes overlap. They are described by means of time diagram, such as the one pictured in Figure 10.7. TDC and BDC denote respectively the top and down positions of the piston. A full cycle is thus made of two revolutions of the piston. The time spend in each stroke can thus be described in terms of angles. To get the "real" time, one needs to know in addition the number of rotations per minute. The angles given in Figure 10.7 are typical for a petrol engine.

Question 1. Design a discrete event system representing a four stroke engine.

Question 2. Perform a simulation to ensure that the engine works as expected.
Hint: You can add a `print` instruction in the simulator to trace firings of events during the simulation.

Problem 10.8 Horse Gaits. Horse gaits are the various ways in which a horse can move, either naturally or as a result of specialized training by humans. We shall focus here on the three natural gaits, namely the walk, the trot and the gallop.

Question 1. The walk is a four-beat gait that averages about 7 kilometers per hour. When walking, the legs follow this sequence: left hind leg, left front leg, right hind leg, right front leg, in a regular 1-2-3-4 beat.
Design a discrete event system representing this gait.
Perform a simulation to ensure that the horse has at most one leg up at a time and that all legs are up evenly.
Hint: You can add a `print` instruction in the simulator to trace firings of events during the simulation.

Question 2. The trot, which is the working gait for a horse, is a two-beat gait: the horse moves its legs in unison in diagonal pairs. It that has a wide variation in possible speeds, but averages about 13 kilometers per hour.
Adjust the previous discrete event system to represent this gait.
Perform a simulation to ensure that the horse has at most two legs up at a time and that all legs are up evenly.

Question 3. The gallop is a four-beat gait. It is the fastest gait of the horse, averaging about 40 to 48 kilometers per hour. In the wild, it is used when the animal needs to flee from predators.
The horse gallops either on the left or the right. The best way to understand the gallop is to start from the fourth beat, where the horse jumps in the air (his four legs are thus up). The beats are as follows.
(a) The horse puts down his right hind leg.
(b) The horse puts down simultaneously his right front leg and left hind leg.
(c) The horse puts down his left front leg.
(d) The horse jumps.
The above beats are for the a right gallop. The left gallop is symmetrical.
Modify the previous discrete event system to represent this gait.
Perform a simulation.

Let us study now some stochastic models.

Exercise 10.9 PERT Diagram. Altough this has not much technical interest, because there are better way to do it, we can come back to Problem 8.15.

Question 1. Design a discrete event system to represent the PERT diagram given in Figure 8.10.

Question 2. Perform a Monte-Carlo simulation and display the histogram of project durations.

Exercise 10.10 Assembly Line. An automated assembly line is in charge assembling one part of type 1 with two parts of type 2 to get a final product. Parts are produced independently and are stored into a buffer before being assembled. The whole line in pictured in Figure 10.8. Parts of type 1 and weight respectively 5kg and 2kg. The buffer is overweighted is the sum of the weights of the parts it contains exceeds 20kg. Times to produce parts and assemble them may vary:
- The time to produce parts of type 1 is uniformly distributed between 10 and 12 minutes.
- The time to produce parts of type 2 is uniformly distributed between 3 and 4 minutes.
- The time to assemble parts is uniformly distributed between 8 and 11 minutes.

Question 1. Design a discrete event system to represent this assembly line.

Question 2. Perform a Monte-Carlo simulation to assess i) the number of final products produced over a 10 hours period and ii) to ensure that the buffer is never overweighted.

Question 3. Determine what should be the resistance of the buffer so to ensure it is never overweighted (or at least that the probability that it gets overweighted is extremely low).

Question 4. Assuming that you cannot improve the resistance of the buffer that much, how would you slow down the production of parts so to to ensure it is never overweighted (or at least that the

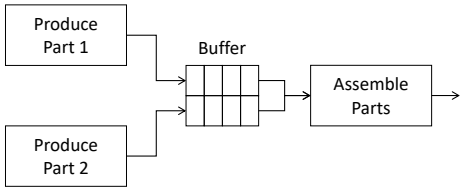


Figure 10.8: An assembly line.

probability that it gets overweighted is extremely low).

Problem 10.11 Helicopter Engine. The engine of an helicopter (motor plus rotor) is maintained to the body of the helicopter by four fasteners arranged in a rectangle: left front, right front, left rear and right rear. These fasteners may break. We shall assume that the probability of failure of a fastener is exponentially distributed. However, the mean time to failure depends on the condition of the other fasteners, due to mechanical constraints and vibrations:

- If the four fasteners are OK, the mean time to failure is 2000 hours (120000 minutes).
- If three out of the four fasteners are OK, the mean time to failure is 10 hours (600 minutes).
- If only two fasteners are still OK and that they are arranged in diagonal, the mean time to failure is 10 minutes.
- Otherwise, all fasteners that are not already broken break immediately.

Question 1. Design a discrete event system to represent the fastening of the helicopter engine. Study the evolution of the probability of a crash with the time. When would you place maintenance operations?

Question 2. Assume now that an alarm is raised as soon as one of the fastener is broken, so that the pilot can perform an emergency landing. Study the evolution of the probability of a crash with the time necessary to perform the emergency landing. Do the alarm make flights safer?

Problem 10.12 Ant Food. An ant is living on a $n \times n$ rectangular grid in which k chunks of food are available. Each chunk is in a cell. There may be several chunks in the same cell. Our ant is looking for food. She starts her exploration right in the center of the grid. Then, she goes at random, first looking for food on her current cell, then moving at random towards either the north, or the south, or the east or the west. She can go on like this too long however: starvation makes her die after a given number l of steps.

Question 1. Design a discrete event system to represent the ant search for food.

Question 2. Perform a Monte-Carlo simulation with $n = 10$, $k = 5$ and $l = 100$ to assess:

- The chance our ant has to find food.
- How long on average does it take her to find food, when she does.

Now, it is well known that ants deposit pheromone trails where they pass. These trail lead members of its own species towards a food source, while representing a territorial mark in the form of an allomone to organisms outside of their species. In our case, we may imagine that our ant uses her own pheromone

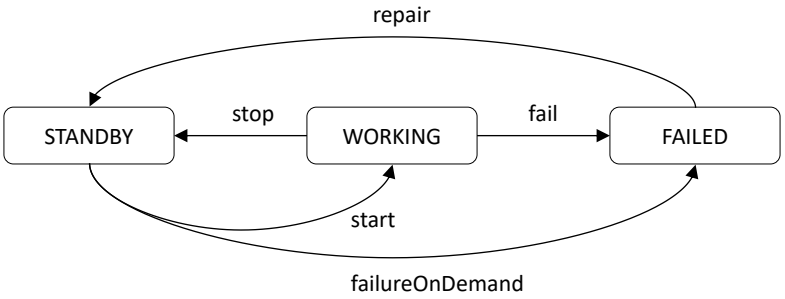


Figure 10.9: State diagram for a standby component

trail just like in the Tom Thumb tale, i.e. she is reluctant to explore a cell she has already explored.

Question 3. Modify your discrete event system so to represent pheromone trail and use it to avoid exploring a cell twice.
Hint: A good occasion to associate weights to transitions!

Question 4. Re-do the same Monte-Carlo simulation as in the question above. Does it help our little friend?

Exercise 10.13 Standby Reliability. In this problem, we shall consider a motor group consisting of two identical motors, a main one and a spare one. We shall make the following assumptions.

- Only one motor is working at a time. The other motor is in standby mode, except if it is failed.
- When working, a motor may fail. The mean time to failure is 8000 hours.
- When attempted to start, a motor may fail on demand, with a probability 0.02.
- When a motor is failed, its repair starts immediately and lasts between 24 and 48 hours (the distribution of repair time is uniform). The motor is as good as new after a repair.
- Initially, both motors are in standby mode.

Figure 10.9 shows a state diagram for a standby component.

Question 1. Design a discrete event system to represent the motor group.
Hint: Use weights to represent the probabilities of successful starts and failures on demand.

Question 2. Perform a Monte-Carlo simulation over a 1 year period (8760 hours) to assess:

- The mean down-time of the group.
- The mean up-time of the each motor.

What do you conclude?

Exercise 10.14 Trolley System. Figure 10.10 shows a system consisting of two trolleys A and B. The trolley A loads raw material at its load station, travels anti-clockwise and offloads the raw material at the common offloading station. The trolley B loads raw material at its load station, travels clockwise and offloads the raw material at the common offloading station.

The problem is indeed not to let a collision happens, i.e. forbid the situation where both trolleys are at the offloading station. For this reason, the circuit of each trolley has been divided in 6 sections (plus the loading and offloading stations). If a trolley A enters in its third section while the trolley B is

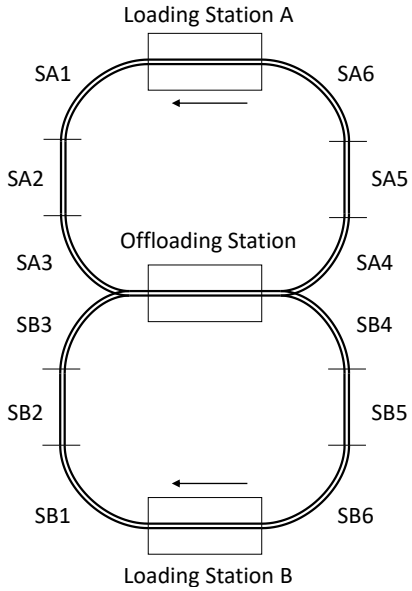


Figure 10.10: A system consisting of two trolleys

offloading, then the trolley A must wait that the trolley B moves to the fourth section before entering itself in the offloading station. Symmetrically for the trolley B. If both trolleys are in third section, then one must be given the priority and the other must wait until it leaves the offload station to enter itself in the station.

We shall make the following additional assumptions.

- The loading time is uniformly distributed between 3 and 5 minutes.
- The travel time on each section is constant and equal to two minutes.
- The offloading time is uniformly distributed between 2 and 3 minutes.

Question 1. Design a discrete event system to represent the trolley system. It would be elegant to find a way of implementing the priority system that alternates situations in which the trolley A is given the priority and those in which it is the trolley B which is given the priority.
Hint: Use the `MEMORY` policy to represent the fact that, in case of conflict, the trolley which has not be given the priority must interrupt its travel to the offloading station to let the other trolley pass.

Question 2. Perform a Monte-Carlo simulation to assess:

- The number of times each trolley offloads it content.
- Whether all dangerous situations have been prevented.

Problem 10.15 Diesel Generators. An hospital installed two diesel generators in case its regular source of electric power (the grid) is lost. These two diesel generators are thus normally stopped. They are started on demand. Still, they may fail even not in operation. Their potential failures are detected via regular maintenance operations. If a failure is detected, the generator is immediately repaired.

During a maintenance operation the diesel generator is out of service. Consequently, maintaining both generators at the same time would lose the protection against a loss of the grid. For this reason, maintenance operations are staggered. A way to achieve that consists in considering two periods of operation:

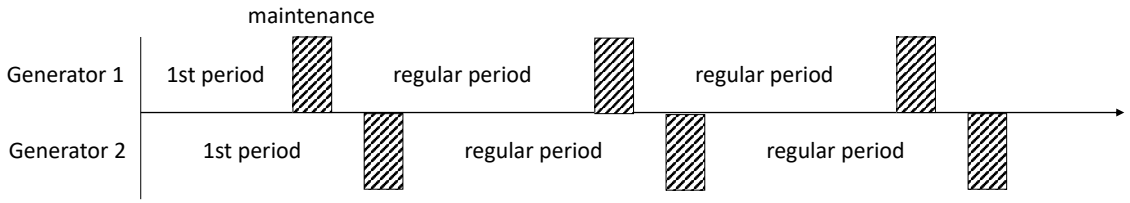


Figure 10.11: Time line for diesel generators

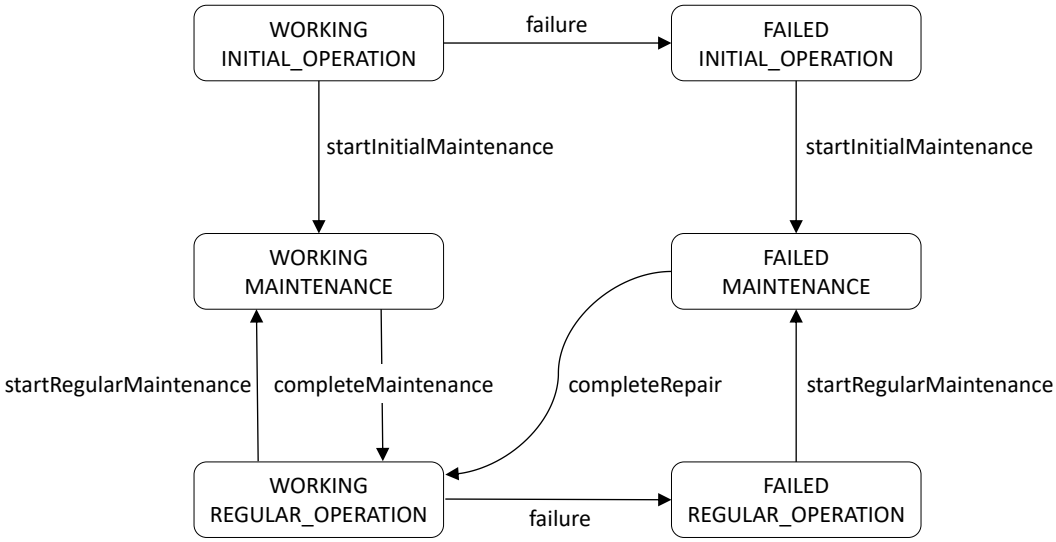


Figure 10.12: State diagram for a diesel generator

– The initial period, which is different for the two generators.
– The regular period, which is the same for the two generators.

Figure 10.11 illustrates this principle. Figure 10.12 shows a state diagram for a diesel generator. We shall make the following additional assumptions.

– One generator is sufficient to provide the required electric power the time it takes for the grid to be available again.
– Each generator fails independently of the other. Its failures are exponentially distributed. The mean failure time is 20 000 hours (about once in two years).
– A maintenance operation takes 4 hours if the generator is working.
– A repair takes 48 hours. The generator is as good as new after a repair.

Question 1. Design a discrete event system to represent the diesel generator group.

Question 2. Perform Monte-Carlo simulations over a 50 000 hours period (about five years of operation) to assess the mean down-time of the diesel generator group.
The objective of these simulations is to tune the duration of the first periods of operation of both diesel generators as well as the duration of the regular operation periods so to minimize down-times.

Chapter 11

Machine Learning

Key Concepts
- Machine Learning
- Training set
- Supervised, unsupervised and reinforcement learning
- Classification and regression
- `Scikit-learn`
- Dynamic programming
- Caching

11.1 Introduction

At the time I am writing these lines, Machine Learning, which is often equated with Artificial Intelligence, is a buzz word. There is not a single project proposal or industry advertisement that does not mention it, in a way or another. Understanding what Machine Learning really consists in, including the limits of this approach, is thus part of the nowadays engineer background. This is the objective of this chapter. It remains that data science is the job of data scientists, who are highly skilled and highly trained specialists. The following presentation aims thus at explaining general principles, not at making the reader a data scientist.

Machine Learning is thus a set of algorithmic techniques that build a statistical model from a sample of data, called the *training set*. This statistical model is then used in order to make predictions or decisions about future data. The specificity of modern machine learning techniques is that the statistical model does not look like what a human expert would have designed. Rather, it is encoded into an internal data structure, which is in general too large to be humanly intelligible. A fixed set of rules is then used to compute results from input data. What is demanded to the whole mechanism is not to help humans to understand the phenomenon under study, but to mimics the statistical properties of this phenomenon. With that respect, it can be seen as a black box. The quality of a machine learning techniques is thus measured in terms of percentage of "good" answers, i.e. of predictions it makes that match the "reality".

This has three immediate consequences:

- First, machine learning assumes that the statistical properties of the phenomenon under study do not change through the time, or at least is sufficiently stable. Conversely to human learning, it has very limited capacities for inferring, from examples, general rules that apply beyond the domain of these examples.
- Second, the quality of the learning depends heavily of the quality of the training set. As in all statistical inference process, the latter must reflect as closely as possible the population under scrutiny. But it must be also sufficiently large to train the data structure and sufficiently focused so that the essential statistical properties can be captured.

The know-how of data scientists consists for the most part in methods to acquire and to prepare data.

– Third, the machine learning technology is interesting to study phenomena that are too complex to be studied by usual statistical means. Because of this complexity, the amount of data necessary to train the data structure and the data structure itself must be very large. In other words, machine learning is in essence extremely data and calculation resource consuming.

The above remarks do not intend to minimize the interest of machine learning. Spectacular results have been obtained with this technology and probably even more spectacular ones will be obtained in the future. They just insist on that that machine learning is not a magic wand, but just a technology.

Another very important remark: when we speak about data, we actually speak about numbers. Machine learning algorithms work with numbers, and numbers only. This means that prior to be machine learnable, datasets (including texts, photos, videos...) must be transformed into numerical datasets.

In general, input datasets must be in the form of two-dimensional matrices where rows represent elements of the sample and columns represent measured features of these elements. Each element of the sample is characterized by the values taken by a predefined set of parameters that are often referred to as *features*. Features are essentially the same as variables in a scientific experiment. They are characteristics of the phenomenon under study that can be measured in some way.

Roughly speaking, machine learning algorithms can be divided into three categories:

Supervised learning: In this approach, the training set consists of pairs (i, r) where i is the input, i.e. values of features, and r is the expected result. Expected results are often referred to as *labels*.
The algorithm learns from these pairs. Then, for any values of features i, it guesses the corresponding label r.
Image and speech recognition, medical diagnoses are typical applications of supervised learning.

Unsupervised learning: In this approach, the training set consists only of input values. The learning phase consists in finding clusters of similar input values. Then, when a new input value is presented, the algorithm guesses which cluster this value belongs to.
Anomaly detection (e.g. to find fraudulent transactions) and customer classification (in view of proposing them products or services) are typical applications of unsupervised learning.

Reinforcement learning: This approach is concerned with how the mechanism should take actions in an environment in order to maximize some reward. Reinforcement learning stands thus somehow in between supervised and unsupervised learning.
The famous program Alpha-Go, that defeated the world best go players, is based on this technology.

Supervised learning is the most widely applied and also the most successful approach. Unsupervised learning, although it proves to be useful in several fields, has strong limitations due to the fact that it barely looks for similarities and does not bring much information. Finally, reinforcement learning is, as of today, limited to a very restricted set of applications.

There are two types of supervised learning depending on whether the expected result is discrete (and small) or continuous:

Classification: In this case, the result consists of classes and the objective is to predict the class of unlabeled data.

Regression: In this case, the result consists of one or more continuous variables and the objective is to predict the value of these variables for unlabeled data.

Recognition of the species of animals on photos, fraud detection on internet transactions, or spam filtering are a typical examples of classification problems. The prediction of the weight of a salmon as a function of its age and the condition in which it is raised is typical example of regression problem.

A special case of classification is the so-called *outlier detection*. In this case, the objective is the identification of rare data which raise suspicions by differing significantly from the majority of the data.

In this chapter, we shall introduce techniques for supervised and unsupervised learning, using the Python package `Scikit-learn` (Pedregosa et al., 2011; Buitinck et al., 2013). `Scikit-learn` implements in an efficient way a cohort of read-to-use machine learning algorithms. For this reason,

```
def Binomial(n, p):
  if p==0 or p==n:
    return 1
  return Binomial(n-1, p-1) + Binomial(n-1, p)
```

Figure 11.1: Recursive function to calculate $\binom{n}{p}$

it is one of the most popular machine learning package. Moreover, it is embedded into the Anaconda distribution.

This chapter contains relatively few exercises and problems. The reason is that the programming part of machine learning analyses is easy thanks to `Scikit-learn`. `Scikit-learn` comes with a number of datasets, both toy examples and "real-world" examples with which the reader can exercise. An additional and almost infinite source of datasets is available on `www.kaggle.com`. Kaggle, now a subsidiary of Alphabet Inc., is an online community of data scientists and machine learning practitioners. It allows users to find and publish data sets, explore and build models, work with other data scientists and machine learning engineers, and participate to competitions to solve data science challenges.

Before diving into machine learning techniques *stricto sensu*, it is worth looking at how machines, i.e. algorithms, can learn.

11.2 How Can Algorithms Learn?

The way algorithms can be summarized by a simple aphorism: trading space for computation time. Algorithms "learn" by storing, in a way or another, results of computations and reusing these results to save time in future computations. This general principle applies not only to machine learning, but to wide range of algorithms. It takes two main forms that we shall review in turn: dynamic programming and caching.

11.2.1 Dynamic Programming

Dynamic programming is an algorithmic method to solve optimization problems. It has been introduced by Richard Bellman in the late 1950's. At that time, programming meant planing in the sense of scheduling the tasks of a project, not computer programming. Bellman used the term "dynamic" to insists on the temporal aspects of the problems at stake, and also because it was impressive.

The key idea is to decompose recursively the problem under study into sub-problems. The decomposition is done upfront, i.e. prior any subproblem is solved. Then, the problem is actually solved bottom-up starting from the basic sub-problems and using already calculated results.

This idea is well illustrated by the calculation of the binomial $\binom{n}{p}$, where n and p are integers such that $0 \leq p \leq n$. The recursive algorithm to calculate `Binomial(n, p)`, is given in Figure 11.1.

This algorithm is very elegant, but suffers from a fundamental drawback: To calculate $\binom{n}{p}$, one needs to calculate $\binom{n-1}{p-1}$ and $\binom{n-1}{p}$. But to calculate $\binom{n-1}{p-1}$, one needs to calculate $\binom{n-2}{p-2}$ and $\binom{n-2}{p-1}$ and to calculate $\binom{n-1}{p}$, one needs to calculate $\binom{n-2}{p-1}$ and $\binom{n-2}{p}$. This means that to calculate $\binom{n}{p}$, one needs to calculate once $\binom{n-2}{p-2}$, twice $\binom{n-2}{p-1}$ and once $\binom{n-2}{p}$. Decomposing further the calculation, one can easily verify that the number of recursive calls of the algorithm grows exponentially with p.

The solution to this problem is fairly simple: learn from already made calculations.

In dynamic programming terms, the first idea is to decompose the calculation of $\binom{n}{p}$ into pieces, just like in the Pascal's triangle (see Figure 11.2).

So, it would be possible to create a $(n+1) \times (p+1)$ matrix to store and reuse intermediate results.

A second idea second idea makes it possible to save computer memory usage. It consists in noticing that to calculate values of $\binom{n}{j}$, $j = 0, 1 \ldots p$, one needs only the values of $\binom{n-1}{j}$, $j = 0, 1 \ldots p$. In other

n/p	0	1	2	3	4	5	$\cdots$
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
$\vdots$							

Figure 11.2: Pascal's Triangle

```
def Binomial(n, p):
  results = [0 for i in range(0, p+1)]
  results[0] = 1
  for i in range(1, n+1):
    for j in range(p, 0, -1):
      results[j] = results[j] + results[j-1]
  return results[p]
```

Figure 11.3: Dynamic programming function to calculate $\binom{n}{p}$

words, if we start from $i = 0$ and we calculate the values of $\binom{i}{j}$ upward till $i = n$, just be looking the values of $\binom{i-1}{j}$.

The corresponding algorithm is given in Figure 11.3. It is easy to verify that the number of steps of this algorithm is proportional to $n \times p$.

Over the years, dynamic programming has found applications in numerous domains, especially in operation research (see e.g. References (Cormen et al., 2001) and (Kleinberg and Tardos, 2006) for in-depth treatments of the subject).

11.2.2 Caching

Dynamic programming is very efficient, as illustrated in the previous section by the calculation of binomials. It requires however to determine upfront what will be the elementary steps of the calculation. One is not always able to do so.

Still, any recursive algorithm can learn and the learning principle is rather simple: use a cache, i.e. a data structure in which results of computations can be stored. When one has to perform a calculation, one first looks up the cache to see whether this has been already made and memorized. If so, one just returns the result. If not, one performs the calculation and stores the result into the cache.

Note that, to apply this principle, it is not necessary to memorize all of the intermediate calculation results into the cache. To put it differently, the algorithm can learn and... forget, in this sense that old values stored into the cache can be replaced by new ones.

The key point is that it insertion and retrieving of results into and from the table should be efficient: if storing and retrieving results is nearly as costly as to perform the calculation, the caching mechanism has no interest. A standard way to achieve such an efficiency is to use a hashtable.

A *hashtable* is a table in which items are stored. To know at which index a given item is stored, one calculates a *hashcode* for this item, i.e. a number ranging from 0 to the size of the table minus one.

As an illustration, we could decide to use a hashtable with 1,000 entries to calculate the binomials. The hashcode would be then calculated from n and p and the size of the table, in this case $1,000$, e.g.

```
def Binomial(n, p):
  cache = [(-1, -1, -1) for i in range(0, 1000)]
  return CachedBinomial(n, p, cache)

def CachedBinomial(n, p, cache):
  if p==0 or p==n:
    return 1
  hashCode = (13*n + 7*p) % len(cache)
  (nc, pc, rc) = cache[hashCode]
  if nc==n and pc==p:
    return rc
  r = CachedBinomial(n-1, p-1, cache) + CachedBinomial(n-1, p, cache)
  cache[hashCode] = (n, p, r)
  return r
```

Figure 11.4: Function to calculate $\binom{n}{p}$ with caching

$(13 \times n + 7 \times p) \mod 1000$.

The corresponding algorithm is given in Figure 11.4.

Note that it is also possible to store lists as entries of the cache. In this case, different results with the same hashcode can be stored into the table and the memorization mechanism can be complete, i.e. remember all of the calculated values.

Caching is used in many applications, including, for instance, by your favorite web-browser to access faster to the web-pages you are visiting often. Literally, your browser learns the pages you like.

11.2.3 Let's Go!

The following exercises are applications dynamic programming and caching mechanisms.

Exercise 11.1 ***k*-out-of-*n* Systems.** Consider a k-out-of-n system, i.e. a system that is working if at least k out of its n components are working. Assume that each component i, $1 \leq i \leq n$ has a probability p_i to fail.

Question 1. Design a recursive algorithm to calculate the probability of failure of the system, given the probabilities of failure of the components.

Question 2. Design a dynamic programming algorithm to calculate the probability of failure of the system, given the probabilities of failure of the components.

Question 3. Design a caching algorithm to calculate the probability of failure of the system, given the probabilities of failure of the components.

Question 4. Test your three algorithms for different values of n, k and the p_i's.

Problem 11.2 Trawler's Route. A trawle fishes on a fishing zone represented by a $r \times c$ grids. Each cell of the grid contains a certain amount of fish that the trawler can capture. The trawler navigates from west to east. After visiting a cell, it can go north-east, east or south-east, in the limits of the fishing zone. It can starts from any cell on the west-most column, i.e. column 0. The problem is to find the best route,

i.e. the route on which the trawler can capture to most fish.

Question 1. Design a recursive algorithm to calculate the best route for the trawler.

Question 2. Design a dynamic programming algorithm to calculate the best route for the trawler.

Question 3. Design a caching algorithm to calculate the best route for the trawler.

Question 4. Test your three algorithms for fishing zones of different sizes and different fish densities.

This little excursion in the magic world of recursive algorithms made, we can come back to our matter, namely machine learning methods *stricto sensu*.

11.3 Classification

11.3.1 Presentation

Classification is probably the most widely used form of machine learning. There exists a wide variety of classification methods:

- Linear models;
- Support vector machines;
- Nearest neighbors;
- Decision trees and forests;
- Neural networks;
- ...

The problem is indeed that none of these methods outperforms the others, nor even performs consistently better than another one. Another way to say that each method has its own advantages and drawbacks and that, except if you have reached the black belt level in machine learning and are able to guess upfront what is the right method to use, you have to try them all.

Presenting these methods in depth goes way beyond the scope on this chapter, this book and probably any book. The involved mathematical developments are far from trivial. Our exposition will thus stay at a quite superficial level. It is neither exhaustive, nor detailed. Developing machine learning algorithm is the job of highly skilled data analysts. Our objective is thus:

i) to give the reader a flavor of the problems at stake,
ii) to show the relative simplicity of using `Scikit-learn`, and
iii) to discuss how an experimental study should be carried out.

11.3.2 Case Study: a Monitored Process

Description To introduce classification techniques, we shall consider throughout this section the following artificial, but pedagogical, case study. Consider thus an industrial process that is monitored by measuring 10 independent parameters: `A1`, `A2`, `B1`, `B1`, `C1`, `C2`, `D1`, `D2`, `D3` and `D4` (see Figure 11.5). For the sake of the simplicity, we shall assume that the values of these 10 parameters are uniformly distributed between 0 and 1 (which can be obtained by normalizing their actual values). Except for these parameters, the process can be considered as a black box.

Assume thus that the process can be in two states. For certain values of the parameters, it is in the state (represented by the value) 0. For some others, it is the state (represented by the value) 1. In other words, the process can be seen as a function L from $[0,1]^{10}$ into $\{0,1\}$. The problem is indeed that we do not know this function.

Our objective is thus to "learn" the function L, by feeding our machine learning algorithm with a sample of observed pairs (parameter values, state).

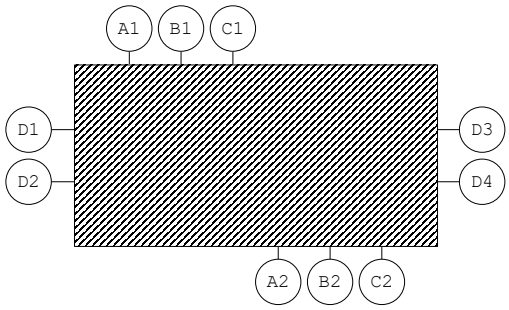


Figure 11.5: A monitored process

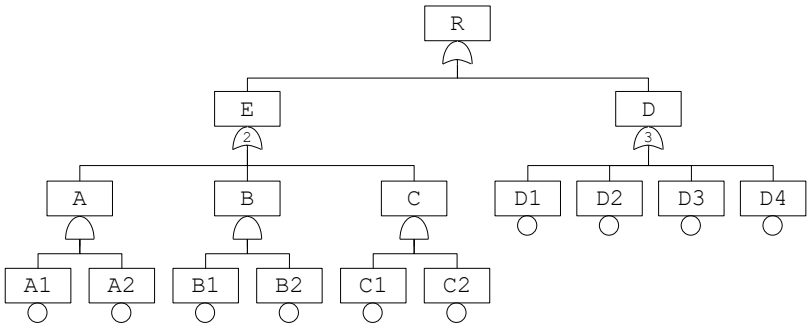


Figure 11.6: The fault tree used to calculate the reward R

In other words, the 10 parameters are the features of our machine learning problem and the two states of the process are the labels. We have thus a typical classification problem.

The hidden function To perform experiments, we shall define arbitrarily the function L in two steps:

- First, a reward $\texttt{R} \in [0,1]$ is calculated from the ten parameters.
- Second, a threshold τ is used to label examples. Examples x such that $\texttt{R}(x) < \tau$ receive the label 0, examples x such that $\texttt{R}(x) \geq \tau$ receive the label 1.

This reward R is actually the probability of the Boolean function defined by the following set of equations.

$$
\begin{aligned}
\texttt{R} &= \texttt{E} \vee \texttt{D} \\
\texttt{E} &= \text{2-out-of-3}(\texttt{A},\texttt{B},\texttt{C}) \\
\texttt{A} &= \texttt{A1} \wedge \texttt{A2} \\
\texttt{B} &= \texttt{B1} \wedge \texttt{B2} \\
\texttt{C} &= \texttt{C1} \wedge \texttt{C2} \\
\texttt{D} &= \text{3-out-of-4}(\texttt{D1},\texttt{D2},\texttt{D3},\texttt{D4})
\end{aligned}
$$

This set function corresponds to the fault tree pictured in Figure 11.6

Given a value for each attribute, i.e. a probability for the basic events A1, A2...D4, it is easy to calculate bottom-up the probability of the top event R. To do so, it suffices to apply the function to calculate the probability of a k-out-of-n system, see Exercise 11.1 (recall that $x_1 \vee \ldots x_n \equiv$ 1-out-of-n$(x_1,\ldots x_n)$ and $x_1 \wedge \ldots x_n \equiv$ n-out-of-n$(x_1,\ldots x_n)$).

To tune the problem, we shall assume moreover that:

- The probabilities of the Ai's, the Bi's and the Ci's are uniformly distributed in $[0.1, 0.2]$;
- The probabilities of the Di's are uniformly distributed in $[0.05, 0.10]$.

Table 11.1: Characteristic of data of the case study "Monitored Process" (classification)

Training set (10,000 examples)

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Label 0	5166	51.7%	8959	89.6%	9890	98.9%
Label 1	4834	48.3%	1041	10.4%	110	1.1%

Test set (10,000 examples)

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Label 0	5107	51.1%	8979	89.8%	9895	99.0%
Label 1	4893	48.9%	1021	10.2%	105	1.1%

With these values, the value of `R` is about 0.003, about half of this probability coming from the subtree `E` and half coming from the subtree `D`.

Finally, we shall consider three values for the threshold τ:

- $\tau_1 = 0.00305$ that corresponds approximately to the median of `R`, i.e. the situation in which half of the examples are labeled with 0 and half with 1.
- $\tau_2 = 0.00395$ that corresponds approximately to the 90% quantile, i.e. the situation in which 90% of the examples are labeled with 0 and 10% with 1.
- $\tau_3 = 0.00480$ that corresponds approximately to the 90% quantile, i.e. the situation in which 99% of the examples are labeled with 0 and 1% with 1.

Table 11.1 gives the characteristics of the training and test sets we used for experiments reported in this chapter.

We are now ready to explore (some of) the classification methods implemented in `Scikit-learn`.

11.3.3 Linear Models

Linear models are used primarily for regression. They aim actually at determining the coefficients of a function in form:

$$\hat{y}(w,x) \quad = \quad w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n \tag{11.1}$$

where the x_i's are the features.

However, *logistic regression*, despite its name, is a linear model intended for classification. It fits data with a logistic function, i.e. a sigmoid ("S" shape) curve. It is used in various fields, including medical and social sciences.

To perform classification with the `Scikit-learn` implementation of logistic regression works as illustrated in Figure 11.7.

In this code, `trainingSample` and `testSample` are two-dimensional arrays, in which each row contains the values of attributes for a particular point, and `trainingLabels` is one-dimensional array containing the corresponding labels.

In our case study, each row of `trainingData` is thus a list of 10 values in $[0,1]$ and each label is either 0 or 1.

Table 11.2 presents results obtained with logistic regression on our case study.

With about 90% of good predictions, the logistic regression performs well on our case study. At a first glance, it seems that the higher the value of τ, the better the results. This is however an illusion. When τ gets large, the proportion of examples labeled with 1 drops. The better scores obtained on high values τ reflect only the fact that the algorithm "bets" everything on the label 0.

```
# Importation of the required modules
from sklearn import linear_model

# Loading of training data and labels
trainingData = numpy.genfromtxt("trainingData.csv")
trainingSample = trainingData[:, :-1]
trainingLabels = trainingData[:, -1]

# Creation and training of the model
model = linear_model.LogisticRegression()
model.fit(trainingSample, trainingLabels)

# Loading of test data and labels
testData = numpy.genfromtxt("testData.csv")
testSample = testData[:, :-1]
testLabels = testData[:, -1]

# Prediction of labels of test data
predictedLabels = model.predict(testSample)
```

Figure 11.7: Skeleton of the Python code using logistic regression

Table 11.2: Results obtained on the case study "Monitored Process" with logistic regression

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Tested 0	5107	51.1%	8979	89.8%	9895	99.0%
Tested 1	4893	48.9%	1021	10.2%	105	1.1%
Predicted 0	5275	52.8%	10000	100%	10000	100%
Predicted 1	4725	47.2%	0	0%	0	0%
Successes 0	4665	91.3%	8979	100%	9895	100%
Successes 1	4283	87.5%	0	0%	0	0%
Total	8948	89.5%	8979	89.8%	9895	99.0%

Table 11.3: Results obtained on the case study "Monitored Process" with support vector machines

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Tested 0	5107	51.1%	8979	89.8%	9895	99.0%
Tested 1	4893	48.9%	1021	10.2%	105	1.1%
Predicted 0	5652	56.5%	10000	100%	10000	100%
Predicted 1	4348	43.5%	0	0%	0	0%
Successes 0	4664	91.3%	8979	100%	9895	100%
Successes 1	3905	79.8%	0	0%	0	0%
Total	8569	85.7%	8979	89.8%	9895	99.0%

11.3.4 Support Vector Machines

A *support vector machine* model consists in a hyper-plane that separates the examples with different labels, considered as points in space, by a gap as wide as possible. In addition to performing this linear classification, support vector machines can efficiently perform a non-linear classification using the so-called the kernel trick. Intuitively, this method consists in mapping examples into high-dimensional feature spaces.

According to the literature, support vector machines are used extensively, with good results, in areas as diverse as text and hypertext categorization, classification of images and classification of proteins in biology.

The `Scikit-learn` code to call the support vector machine algorithm is similar to the one of Figure 11.7:

```
# Importation of the required modules
from sklearn import svm

# Loading of training data and labels
...

# Creation and training of the model
model = svm.SVC()
model.fit(trainingSample, trainingLabels)
```

Table 11.3 presents results obtained with the support vector machine algorithm on our case study.

The performance of this algorithm are similar (a bit worse) than those of logistic regression. Its running times are higher however.

11.3.5 K-Neighborghs Classifier

The principle behind *nearest neighbor methods* consists in finding a predefined number of examples of the training sample which are the closest to the point under study, and in predicting the label of the latter from those of the former. Neighbors-based classification does not attempt to construct a general internal model, but simply stores instances of the training data.

Despite their simplicity, nearest neighbor methods have been successful in a large number of classification and regression problems, including handwritten digits and satellite image scenes.

The `Scikit-learn` code to call the nearest neighbor classification algorithm is similar to the one of Figure 11.7:

Table 11.4: Results obtained on the case study "Monitored Process" with the k-neighborghs classifier

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Tested 0	5107	51.1%	8979	89.8%	9895	99.0%
Tested 1	4893	48.9%	1021	10.2%	105	1.1%
Predicted 0	5096	51.0%	9202	92.0%	9953	99.5%
Predicted 1	4904	49.0%	798	8.0%	47	0.5%
Successes 0	4429	86.7%	8816	98.2%	9879	99.8%
Successes 1	4226	86.4%	635	62.2%	31	29.5%
Total	8655	86.5%	9451	94.5%	9910	99.1%

```
# Importation of the required modules
from sklearn.neighbors import KNeighborsClassifier

# Loading of training data and labels
...

# Creation and training of the model
model = KNeighborsClassifier()
model.fit(trainingSample, trainingLabels)
```

Table 11.4 presents results obtained with the k-neighborghs classifier algorithm (k=5) on our case study.

K-neighborghs classification performs slightly worse than logistic regression on examples with balanced numbers of examples labeled with 1 and labeled with 0. On the other hand, although it degrades less in case of unbalanced data sets (at least on our case study).

11.3.6 Naive Bayes Classifiers

Naive Bayes methods are a set of supervised learning algorithms based on Bayes' theorem with the "naive" assumption of statistical independence of features. Bayes' theorem can be stated as follows.

$$P(y|x_1,\ldots x_n) = \frac{P(y)\times P(x_1,\ldots x_n|y)}{P(x_1,\ldots x_n)} \tag{11.2}$$

The idea is to start from an *a priori* distribution and, using the above equality, to improve this distribution each time a new example is introduced in the training set.

There are different naive Bayes classifiers that differ mainly by the assumptions they make regarding the distribution of $P(x_i|y)$. Naive Bayes classifiers work quite well in many real-world situations, notably document classification and spam filtering. They require a small amount of training data and are very fast to estimate the necessary parameters.

The `Scikit-learn` code to call the naive Bayes classification algorithms is similar to the one of Figure 11.7, e.g.:

Table 11.5: Results obtained on the case study "Monitored Process" with the Gaussian naive Bayes classifier

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Tested 0	5107	51.1%	8979	89.8%	9895	99.0%
Tested 1	4893	48.9%	1021	10.2%	105	1.1%
Predicted 0	5204	52.0%	9588	95.9%	9960	99.6%
Predicted 1	4796	48.0%	412	4.1%	40	0.4%
Successes 0	4947	96.9%	8979	100%	9895	100%
Successes 1	4636	94.7%	412	40.4%	40	38.1%
Total	9583	95.8%	9391	93.9%	9935	99.3%

```
# Importation of the required modules
from sklearn.naive_bayes import GaussianNB

# Loading of training data and labels
...

# Creation and training of the model
model = GaussianNB()
model.fit(trainingSample, trainingLabels)
```

Table 11.5 presents results obtained with the Gaussian naive Bayes classifier algorithm on our case study.

On our case study, this classifier performs very well. We have however other cases in which it performs poorly.

11.3.7 Decision Trees

Decision Trees are a non-parametric supervised learning method used for classification and regression. The idea is to build a binary decision tree where at each node of the tree, a decision is made depending on the value of a feature. The leaves of the tree are the labels. Decision Trees algorithms try to maintain the binary trees they build balanced, so to ensure a fast decision process.

The `Scikit-learn` code to call the default decision tree classification algorithm is similar to the one of Figure 11.7:

```
# Importation of the required modules
from sklearn.tree import DecisionTreeClassifier

# Loading of training data and labels
...

# Creation and training of the model
model = DecisionTreeClassifier()
model.fit(trainingSample, trainingLabels)
```

Table 11.6 presents results obtained with the decision tree classifier algorithm on our case study.

On this case study, the decision tree classifier does not perform very well.

The *random forest classifiers* extend decision trees by creating several trees instead of a single one and by making these different trees vote on the result. This requires introducing some randomness in the construction of the trees, typically regarding the selection of the feature to be considered first. Random

Table 11.6: Results obtained on the case study "Monitored Process" with the decision tree classifier

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Tested 0	5107	51.1%	8979	89.8%	9895	99.0%
Tested 1	4893	48.9%	1021	10.2%	105	1.1%
Predicted 0	5061	50.6%	8983	89.8%	9880	98.8%
Predicted 1	3918	49.4%	1017	10.2%	120	1.2%
Successes 0	4086	80.0%	8495	94.6%	9790	98.9%
Successes 1	3933	80.1%	533	52.2%	15	14.3%
Total	8004	80.4%	9028	90.3%	9805	98.0%

forest classifiers are indeed more resource consuming than simple decision trees, but they may give also better results.

11.3.8 Multi-Layer Perceptron

Last, but certainly not least, come neural networks. *Neural networks* are are learning algorithms vaguely inspired by the biological neural networks that constitute the brains of animals.

A neural network is a collection of connected nodes called neurons. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. Neurons are organized into successive layers. A neuron of the layer n receives signals from some of the neurons of the layer $n-1$, processes these signals and sends the calculated result to some of the neurons of the layer $n+1$. Signals are real numbers. The output of each neuron is computed by some non-linear function of the sum of its inputs. The learning phase consists in adjusting the parameters if these functions (one per neuron). Starting with initial default parameters, back-propagation is typically used the connection weights so to compensate errors found in the assessment.

Neural networks and machine learning are tightly linked. This became even more true with the fantastic successes of *deep learning* and its may applications including vehicle control, trajectory prediction, process control, natural resource management, quantum chemistry, game playing, pattern recognition in radar systems, face identification, signal classification, 3D reconstruction, gesture, speech, handwritten and printed text recognition, medical diagnosis, automated trading, data mining, automated translation, social network, e-mail spam filtering and many other. In deep learning, neural networks may have a highly complex of organization, with many hidden layers and sub-networks in charge of specific tasks. Training such networks is extremely resource consuming. This is the reason why dedicated hardware platforms are used, which used thousands of dedicated processors.

Indeed, a generic tool like `Scikit-learn` cannot provide such capacities. Rather it implements the so-called *multi-layer perceptron* classifier. In this classifier, the leftmost layer, known as the input layer, consists of a set of neurons representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $s = w_1 \cdot x_1 + \ldots + w_n \cdot x_n$ followed by a non-linear activation function $g(s)$, like the hyperbolic tan function. Intermediate layers are called hidden layers The output layer receives the values from the last hidden layer and transforms them into output values, i.e. the predicted labels. By default, `Scikit-learn` implements a multi-layer perceptron with one hidden layer made of 100 neurons.

The `Scikit-learn` code to call the multi-layer perceptron classification algorithm is similar to the one of Figure 11.7:

Table 11.7: Results obtained on the case study "Monitored Process" with the multi-layer perceptron classifier

	$\tau_1 = 0.00305$		$\tau_2 = 0.00395$		$\tau_3 = 0.00480$	
Tested 0	5107	51.1%	8979	89.8%	9895	99.0%
Tested 1	4893	48.9%	1021	10.2%	105	1.1%
Predicted 0	5086	50.9%	9017	90.2%	10000	100%
Predicted 1	4914	49.1%	983	9.8%	0	0%
Successes 0	4879	95.5%	8920	99.3%	9895	100%
Successes 1	4686	95.8%	924	90.5%	0	0%
Total	9565	95.7%	9844	98.4%	9895	99.0%

```
# Importation of the required modules
from sklearn.neural_network import MLPClassifier

# Loading of training data and labels
...

# Creation and training of the model
model = MLPClassifier(max_iter=1000)
model.fit(trainingSample, trainingLabels)
```

Table 11.7 presents results obtained with the multi-layer perceptron classifier algorithm on our case study (we set up the parameter `max_iter` to 1000).

The performance of the multi-layer perceptron classifier are impressive, but when the number of examples of each class are very unbalanced. It running times tend to be higher than those of other methods, even though they remain reasonable (on our case study).

11.3.9 Discussion

In this section, we explored different classification methods. There exist many others. Moreover, each of the methods we look can has multiple variants and parameters that can be tuned for each particular purpose. Making an exhaustive study is thus impossible.

Still, a number of questions remain open.

First, we can remark that most of the classifiers perform better on instances for which the state is 0 than on those for which it is 1, even if the numbers of examples of each class are balanced. The author has no explanation for this asymmetry. Note that the results remain the same if we switch labels (i.e. that state 1 instances are better detected than state 0 ones).

Second, in the above experiments, we used a training set made of $10,000$ examples. It can be argue that this is too much or too few. Taking a training set with $100,000$ examples instead increases very significantly the running times, especially for support vector machines and the multi-layer perceptron. The success rate is improved as well, as shown in the following table (for τ_1).

sample size/method	LR	SVM	KNN	NB	DT	MLP
10,000	89.5%	85.7%	86.5%	95.8%	80.0%	95.7%
100,000	95.0%	94.0%	89.8%	95.9%	84.1%	95.8%

These are significant improvements, but the result is much less spectacular on larger values of τ. There is almost not improvement for τ_3.

What is the "good" size of the training set results of a tradeoff between the quality of the results and the amount of resources required to get them.

Third, preprocessing of data is a key step of any machine learning process. In our case study, we could for instance remark, by means of an analysis of the process under study, that features `A1` and `A2`, `B1` and `B2`, and `C1` and `C2` are tightly linked. We could thus replace these six features by three more accurate ones, namely $\texttt{A} = \texttt{A1} \wedge \texttt{A2}|$, $\texttt{B} = \texttt{B1} \wedge \texttt{B2}$ and $\texttt{C} = \texttt{C1} \wedge \texttt{C2}$. The reduced function L' to be discovered is thus now a function from $[0,1]^7$ into $\{0,1\}$. Surprisingly enough, this does not necessarily improves the scores of methods. It even degrades the performance for most of them (all but decision trees and the multi-layer perceptron).

Surprising facts like this one are not uncommon in the machine learning world!

11.3.10 Let's Go!

Problem 11.3 Monitored Process (classification). This problem consists in re-doing the experiments described along this section.

Question 1. Design a Python script to generate training and test data according the description of Section 11.3.2.

Question 2. Apply to the generated data the various classification algorithms reviewed in this section (and possibly some other, like random forests):

- Logistic regression;
- Support vector machines;
- K-neighbors classifier;
- Naive Bayes classifier;
- Decision trees;
- Multi-layer perceptron.

As already said, `Scikit-learn` comes with a number of datasets, both toy examples and "real-world" examples. The most famous is probably the iris dataset involved in the following problem.

Problem 11.4 Iris Dataset. The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper (Fisher, 1936). The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other. This dataset is perhaps the best known database to be found in the pattern recognition literature.

To load this dataset, you have to import `sklearn.datasets` and to use the method `load_iris()`:

```
1 from sklearn import datasets
2
3 dataSet = datasets.load_iris()
4 sample = dataSet['data']
5 labels = dataSet['target']
```

This sample (a `NumPy` matrix) contains 150 instances. Attributes are the following.

1. Sepal length in cm;

2. Sepal width in cm;
3. Petal length in cm;
4. Petal width in cm;

`labels` is a `NumPy` vector. Labels are numbers from 0 to 2: 0 for `Iris-setosa`, 1 for `Iris-versicolour`, and 2 for `Iris-virginica`.

Question 1. Design a script using `NumPy` functions to calculate:
- The minimum, maximum and mean sepal lengths.
- The minimum, maximum and mean sepal widths.
- The minimum, maximum and mean petal lengths.
- The minimum, maximum and mean petal widths.

Question 2. Split the dataset into a training set and a test set of respectively 120 and 30 instances.
Hint: : This requires to shuffle at random the dataset first.

Question 3. Apply to the generated data the various classification algorithms reviewed in this section (and possibly some other, like random forests):
- Logistic regression;
- Support vector machines;
- K-neighbors classifier;
- Naive Bayes classifier;
- Decision trees;
- Multi-layer perceptron.

11.4 Regression

11.4.1 Presentation

As classification, regression is a supervised learning problem. Conversely to classification, it involved labels that are continuous variables. The objective of regression is thus to calculate a reward by combining, in some way, the values of attributes.

There exist even more machine learning methods for regression than for classification: basically, all of the methods we looked at in the previous section have their regression counterpart and many more are implemented in `Scikit-learn`. We shall review here only very few of them.

Beyond the choice of the method to use, there is an additional issue regarding regression: how to assess the performance of a method on a given dataset? For classification, the problem is rather easy: statistics on number of instances of each class and of successes for each class are sufficient. For regression, one has to measure the distance between the actual reward $y(x)$ and the predicted reward $\hat{y}(x)$ (where x is the value of attributes), i.e. eventually to define an appropriate notion of distance between $y(x)$ and $\hat{y}(x)$.

A first idea is to calculate the sum or the mean of the residual $r_i = y(x_i) - \hat{y}(x_i)$ for all values x_i of attributes in the test set. The problem is that the residual can be negative and thus a positive residual can cancel a negative one. A way to tackle this problem is to take absolute values of residual, leading to the our first indicator, the *mean absolute error* (MAE):

$$\text{MEA} \stackrel{def}{=} \frac{1}{n}\sum_{i=1}^{n} |y(x_i) - \hat{y}(x_i)| \tag{11.3}$$

Another classical way to tackle the problem is to consider the *root mean squared error* (RMSE):

$$\text{RMSE} \stackrel{def}{=} \sqrt{\frac{\sum_{i=1}^{n} \left(y(x_i) - \hat{y}(x_i)\right)^2}{n}} \tag{11.4}$$

Table 11.8: Characteristic of data of the case study "Monitored Process" (regression)

Training set (10,000 points)

Version	1	2
Minimum reward	1.192×10^{-3}	6.860×10^{-4}
Maximum reward	6.099×10^{-3}	4.627×10^{-3}
Mean reward	3.078×10^{-3}	1.969×10^{-3}

Test set (10,000 points)

Version	1	2
Minimum reward	1.320×10^{-3}	6.820×10^{-4}
Maximum reward	5.822×10^{-3}	4.441×10^{-3}
Mean reward	3.082×10^{-3}	1.974×10^{-3}

For both the MAE and the RMSE, the smaller the result, the better the method is performing. The RMSE is always larger than the MAE. It is also is more sensitive to outliers (big differences). Hence, if the outliers are undesirable, the RMSE better evaluates how well the method is performing. We could make them even more undesirable by taking the power $1/k$, $k = 3,4\ldots$, of the mean of powers k of residuals.

The R-Squared indicator, also called coefficient of correlation, aims at testing whether the method reflects the variance of the reward. It is defined as follows.

$$R^2 \stackrel{def}{=} 1 - \frac{\sum_{i=1}^{n} (y(x_i) - \hat{y}(x_i))^2}{\sum_{i=1}^{n} (y(x_i) - \bar{y})^2} \tag{11.5}$$

where $\bar{y}$ stands for the mean of actual rewards.

This indicator must be handled with care as it is often difficult to interpret.

`Scikit-learn` provides ready made functions to calculate these indicators. They work as follows.

```
import math
from sklearn import metrics

MAE = metrics.mean_absolute_error(actualRewards, predictedRewards)
RMSE = math.sqrt(metrics.mean_squared_error(actualRewards,
  predictedRewards))
R2 = metrics.r2_score(actualRewards, predictedRewards)
```

11.4.2 Case Study: a Monitored Process (bis)

To introduce regression techniques, we shall just use our previous case study (described Section 11.3.2). Rather than to calculate a class, we shall take the reward `R` calculated from the 10 attributes.

We shall consider two versions:

– Version 1, in which the probabilities of the `Ai`'s, the `Bi`'s and the `Ci`'s are uniformly distributed in $[0.1, 0.2]$ while the probabilities of the `Di`'s are uniformly distributed in $[0.05, 0.10]$ (as previously).

– Version 2, in which the probabilities of the `Ai`'s, the `Bi`'s and the `Ci`'s are uniformly distributed in $[0.1, 0.2]$ while the probabilities of the `Di`'s are uniformly distributed in $[0.03, 0.07]$.

Table 11.8 gives the characteristics of the training and test sets we have used for experiments reported in this section.

11.4.3 Experiments and Discussion

The `Scikit-learn` code to call the regression algorithms is similar to the one of Figure 11.7. For instance, the code to apply linear regression is as follows.

Table 11.9: Experimental results obtained on the "Monitored Process" case study with different regression methods

Version	1		1	
Mean	3.082×10^{-3}		1.974×10^{-3}	
Method	MAE	RMSE	MAE	RMSE
Linear Regression	8.588×10^{-5}	1.129×10^{-4}	6.972×10^{-5}	9.493×10^{-5}
Support Vector Machines	7.311×10^{-4}	8.696×10^{-4}	7.455×10^{-4}	8.505×10^{-4}
K-Neighbors	1.764×10^{-4}	2.258×10^{-4}	9.834×10^{-5}	1.251×10^{-4}
Decision Trees	3.330×10^{-4}	4.219×10^{-4}	2.528×10^{-4}	3.166×10^{-4}
Multi-Layer Perceptron	1.908×10^{-3}	2.368×10^{-3}	6.729×10^{-3}	8.287×10^{-3}

```
# Importation of the required modules
from sklearn import linear_model

# Loading of training data and labels
...

# Creation and training of the model
model = linear_model.LinearRegression()
model.fit(trainingSample, trainingRewards)

# Loading of test data and labels
...

# Predicting rewards
predictedRewards = model.predict(testSample)
```

Table 11.9 presents the results we obtained with the two versions of our case study, using some of the algorithms implemented in `Scikit-learn`.

A few remarks here:

- Linear regression outperforms all other methods.
- Multi-layer perceptron gives very, very poor results. Adding another layer improves the situation, but still the results are not good.
- The other methods perform not very good, given that the mean error represents a significant portion of the mean value.

There is of course room for improvement here. But the above experiment show significant differences in the performance of algorithms.

11.4.4 Let's Go!

Problem 11.5 Monitored Process (regression). This problem consists in re-doing the experiments described along this section.

Question 1. Design a Python script to generate training and test data according the description of Section 11.4.2.

Question 2. Apply to the generated data some of the regression algorithms implemented in `Scikit-learn`.

Problem 11.6 Boston Housing Dataset. The Boston Housing Dataset, available in `Scikit-learn`, is a derived from information collected by the U.S. Census Service concerning housing in the area of Boston MA. Attributes are as follows.

CRIM: per capita crime rate by town;
ZN: proportion of residential land zoned for lots over 25,000 sq. ft.;
INDUS: proportion of non-retail business acres per town;
CHAS: Charles River dummy variable (1 if tract bounds river; 0 otherwise);
NOX: nitric oxides concentration (parts per 10 million);
RM: average number of rooms per dwelling;
AGE: proportion of owner-occupied units built prior to 1940;
DIS: weighted distances to five Boston employment centers;
RAD: index of accessibility to radial highways;
TAX: full-value property-tax rate per $10,000;
PTRATIO: pupil-teacher ratio by town;
B: $1000(Bk-0.63)^2$ where Bk is the proportion of blacks by town;
LSTAT: Percentage of lower status of the population;
MEDV: Median value of owner-occupied homes in $1000's.

The last attribute is the target reward.

Question 1. Design a script using `NumPy` functions to calculate the minimum, maximum and mean values of each attribute.

Question 2. Split the dataset into a training set and a test set of respectively 400 and 106 instances.

Question 3. Apply to the generated data various regression algorithms

11.5 Clustering

11.5.1 Presentation

Clustering belongs to unsupervised machine learning techniques. Given a data set, clustering algorithms classify each data point of the set into a specific group called *cluster*. The assumptions behind these methods are:

i) Elements with similar features in the real life are represented by data points with similar attributes.
ii) Reciprocally, data points with similar attributes represent elements with similar features.

In other words, elements with dissimilar features in real life are represented by data points with sufficiently dissimilar attributes so that they can be distinguished.

This raises immediately a number of questions:

- How to make effective this notion of similarity? How to measure how much two data points are similar?
- If each pair of data points in the data set have to be compared, the complexity of the clustering algorithm is at least quadratic, which may be prohibitive for large data sets. So, how to make the clustering efficient?
- How many clusters do we want to obtain at the end of the process? Intuitively, the fewer the better, but at least two. Is it always feasible?
- Do we have to set in advance the number of clusters we want to obtain or is it up to the algorithm to decide? If so, according to which criteria?
- How to measure the efficiency of the algorithm? Again, according to which criteria?

`Scikit-learn` implements several clustering algorithms. We shall present here the results obtained with only one of them (KMEANS). The others give actually quite disapointing results on our case study.

Studying these questions goes far beyond the objective of this book. We shall just show in the sequel the results obtained with some of the clustering algorithms implemented in `Scikit-learn` on a variation of our case study.

11.5.2 Case Study: a Monitored Process (ter)

In order to test clustering algorithms, we shall modify slightly our case study. The idea is, given two numbers $0 \leq \theta_1 \leq \theta_2$, to create artificially two clusters:

- Examples x for which $R(x) < \theta_1$;
- Examples x for which $R(x) > \theta_2$.

Examples x such that $\theta_1 \leq R(x) \leq \theta_2$ are simply discarded.

For the purpose of our example, we can choose $\theta_1 = 0.00275$ and $\theta_2 = 0.00335$ that correspond approximately to the quantiles $1/3$ and $2/3$. In this way, a third of the examples belong to the first cluster, a third to the second one and a third are rejected.

11.5.3 KMeans

The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia. This algorithm requires the number of clusters to be specified.

The `Scikit-learn` code to call the clustering algorithms is again similar to the one of Figure 11.7. For instance, the code to apply the KMeans algorithm is as follows.

```
# Importation of the required modules
from sklearn.cluster import KMeans

# Loading of training data and labels
...

# Creation and training of the model
model = KMeans(n_clusters=2)
model.fit(trainingSample, trainingRewards)

# Loading of test data and labels
...

# Predicting labels
predictedLabels = model.predict(testSample)
```

Note that there is a small difficulty with the above code: the clusters are not necessarily numbered as the labels used to test the algorithm.

On our case study, the result can be presented in the following matrix.

label/cluster	0	1
0	486	4502
1	4199	813

The algorithm classifies thus correctly $4502 + 4199 = 8701$ examples out of 10,000, i.e. its success rate is 87.0%.

11.5.4 Let's Go!

Problem 11.7 Monitored Process (clustering). This problem consists in re-doing the experiments described along this section.

Question 1. Design a Python script to generate training and test data according the description of Section 11.5.2.

Question 2. Apply to the generated data some of the clustering algorithms implemented in `Scikit-learn`.

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

Marco Ajmone-Marsan, Gianfranco Balbo, Giuseppe Conte, Susanna Donatelli, and Giuliana Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons, New York, NY, USA, 1994. ISBN 978-0471930594.

George E. P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 29(1):610–611, 1958. doi: 10.1214/aoms/1177706645.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *European Conference on Machine Learning and Principles and Practices of Knowledge Discovery in Databases (2013). Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Second ed.)*. The MIT Press, Cambridge, MA, USA, 2001. ISBN 0-262-03293-7.

Allen B. Downey. *Think Complexity: Complexity Science and Computational Modeling (2nd edition)*. O'Reilly Media, Inc, Sebastopol, CA 95472, USA, 2018. ISBN 978-1492040200.

R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2): 179–188, 1936. doi: 10.1111/j.1469-1809.1936.tb02137.x.

Martin Fowler. *Domain Specific Languages*. Addison-Wesley, Boston, MA 02116, USA, October 2010. ISBN 978-0321712943.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, Boston, MA 02116, USA, October 1994. ISBN 978-0201633610.

Martin Gardner. Mathematical games – the fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, October 1970. doi: 10.1038/scientificamerican1070-120.

John E. Grayson. *Python and Tkinter Programming*. Manning, Shelter Island, NY, USA, 2000. ISBN 9781884777813.

John V. Guttag. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. MIT Press, Cambridge, MA 02142-1315, USA, 2016. ISBN 978-0262529624.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (third edition)*. Pearson Education International, New York City, New York, USA, 2006. ISBN 978-0321455369.

John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3): 90–95, 2007. doi: 10.1109/MCSE.2007.55.

Edward Lynn Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of American Statistics Association*, 53(282):457–481, 1958. doi: 10.2307/2281868.

Yoshio Kawauchi and Marvin Rausand. A new approach to production regularity assessment in the oil and chemical industries. *Reliability Engineering and System Safety*, 75:379–388, 2002.

Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, London, England, 2006. ISBN 978-0321349422.

Donald E. Knuth. *The Art of Computer Programming, volume 2: Seminumerical Algorithms (3rd edn)*. Addison-Wesley, Boston, MA, USA, 1998. ISBN 978-0201896848.

Andrei N. Kolmogorov. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer, Berlin, Germany, 1933. ISBN 978-3540061106.

Mark A. Lutz. *Programming Python: Powerful Object-Oriented Programming (4th edition)*. O'Reilly Media, Inc., Sebastopol, CA 95472, USA, January 2011. ISBN 978-0596158101.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equi-distributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998. doi: 10.1145/272991.272995.

Bertrand Meyer. *Object-Oriented Software Construction*. MIT Electrical Engineering and Computer Science. Prentice Hall, Upper Saddle River, New Jersey, USA, 1988. ISBN 978-0136290490.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfor, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, (12):2825–2830, 2011.

Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Méthodes numériques pour le calcul scientifique*. Springer Verlag, Wiesbaden, Germany, 2000. ISBN 978-2-287-597015.

Antoine Rauzy. Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability*, 222(4):495–505, 2008. doi: 10.1243/1748006XJRR177.

Thomas Crombie Schelling. Dynamic models of segregation. *Journal of Mathematical Sociology*, 1: 143–186, 1971. doi: 10.1080/0022250X.1971.9989794.

Lawrence F. Shampine, Rebecca Chan Allen, and Steven Pruess. *Fundamentals of Numerical Computing*. John Wiley and Sons, Hoboken, NJ, USA, 1997. ISBN 978-0-471-16363-3.

William J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, New Jersey, USA, 1994. ISBN 978-0691036991.

Stefan van der Walt, Stefan Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011. doi: 10.1109/MCSE.2011.37.

Guido van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Neederland, May 1995.

Uri Wilensky and William Rand. *An Introduction to Agent-Based Modeling - Modeling Natural, Social, and Engineered Complex Systems with NetLogo*. The MIT Press, Cambridge, MA, USA, 2015. ISBN 978-0262731898.

Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, Upper Saddle River, New Jersey 07458, USA, 1976. ISBN 978-0130224187.

Stephen Wolfram. *A New Kind of Science*. Wolfram Media Inc, Champaign, IL, USA, June 2002. ISBN 978-1579550080.

Index

Part III

Appendix

Appendix A

Case Studies

Throughout this book, we use a number of case studies to illustrate various concepts at stake. This chapter presents these case studies, independently of the concrete treatments implemented to tackle them.

A.1 Texts

Case Study 1 Lorem Ipsum. In publishing and graphic design, Lorem ipsum is a placeholder text commonly used to demonstrate the visual form of a document without relying on meaningful content. Replacing the actual content with placeholder text allows designers to design the form of the content before the content itself has been produced.

The lorem ipsum text is typically a scrambled section of "*De finibus bonorum et malorum*, a first-century fefore Christ Latin text by Cicero, with words altered, added, and removed to make it nonsensical, improper Latin.

We shall use this piece of text (stored in the file `LoremIpsum.txt` to test various algorithms.

Case Study 2 Universal Declaration of Human Rights. The Universal Declaration of Human Rights is a milestone document in the history of human rights. Drafted by representatives with different legal and cultural backgrounds from all regions of the world, the Declaration was proclaimed by the United Nations General Assembly in Paris on 10 December 1948 (General Assembly resolution 217 A) as a common standard of achievements for all peoples and all nations. It sets out, for the first time, fundamental human rights to be universally protected and it has been translated into over 500 languages.

We shall use here the English version, stored in the file `Universal Declaration of Human Rights.txt`.

Case Study 3 I Have a Dream. "I Have a Dream" is a public speech that was delivered by American civil rights activist Martin Luther King Jr. during the March on Washington for Jobs and Freedom on August 28, 1963, in which he called for civil and economic rights and an end to racism in the United States. Delivered to over 250,000 civil rights supporters from the steps of the Lincoln Memorial in Washington, D.C., the speech was a defining moment of the civil rights movement and among the most iconic speeches in American history.

A.2 Retailer Sales

Case Study 4 Retailer Sales. A food retailer wants to perform some analysis on its annual sales. The analysis focuses on 10 products. Each product is identified by a reference and generates a profit per unit sale. Monthly sales of each product have been recorded. All these data are presented Table A.1.

Table A.1: Monthly sales for each product of the retailer sales case study

Product	Profit	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
BVR721	1.20	5869	5286	5175	4829	4903	5170	5297	4414	4537	4196	4492	5436
BVR446	0.80	7445	7225	7298	7195	7267	7261	7552	7169	7508	7257	7165	7528
BVR889	0.85	6562	6390	6544	7423	7161	6929	7090	7020	6521	7006	6120	6251
BKR802	1.10	3336	3360	3497	3309	3556	3393	3183	3674	3807	3241	3399	3595
BKR013	0.85	7066	5310	6911	5433	5616	5326	6373	5708	6608	6922	6033	7447
DRY572	0.50	7251	7665	7430	7047	7540	7512	7438	7357	7364	7545	7258	7706
DRY633	0.90	6645	6470	6338	6671	6932	6287	6678	6546	6835	6616	6577	6111
PRD265	0.30	8391	8861	8440	8499	8650	8540	8644	8455	8622	8471	8489	8734
PRD911	0.40	8081	7504	7639	8715	7302	7839	8692	7498	7632	7532	8600	7395
PRD920	0.55	5587	3594	6815	5946	5120	6594	5289	4095	4855	4678	4712	4851

A.3 Warehouse Construction

Case Study 5 Warehouse Construction. Table A.2 shows the activities of a warehouse construction project. Each activity has a code (first column), a description (second column), a duration in weeks (third column) and a list of activities that must be completed before the activity starts. Eventually, activities of the project form the network pictured in Figure A.1.

For the sake of the simplicity, we shall assume that an activity is always declared in the table after its preceding activities.

In project management, such a table is the input data for the so-called PERT analysis (program evaluation and review technique). This method consists in calculating for each activity:

- Its early starting date, i.e. the earliest date at which the activity can start.
- Its early completion date, i.e. the earliest date at which the activity can be completed.
- Its late starting date, i.e. the latest date at which the activity can start without delaying the whole project.
- Its late completion date, i.e. the latest date at which the activity can be completed without delaying the whole project.

The method to calculate these dates is rather simple. It is based on the following remarks.

- The early completion date of a activity is always the sum of its early starting date and its duration.
- The early starting date of an activity is the latest early completion dates of its preceeding activities.
- Symmetrically, the late starting date of an activity is its late completion date minus its duration.
- Finally, the late completion date of an activity is the earliest late starting dates of the activities its precedes.

For instance, the early staring date of the activity D is the determined by early completion dates of activities A and B, since these two activities precede D. In the other way round, the late completion date of the activity A is determined by the late starting dates of activities C, D and E since A precedes these three activities.

The idea is thus to traverse the network of activities first in the direction of the arrows, i.e. from left to right on Figure A.1. This first traversal makes it possible to calculate the early starting and

completion dates of each activity as well as the early completion date of the project as a whole.

A second traversal that goes into the reverse direction make it possible to calculate the late starting and completion dates of each activity.

Activities whose early and late starting dates (and early and late completion dates) are the same are critical in this sense that any delay in these activities delays the whole project.

Table A.2: Activities of the warehouse construction project

Code of activity	Description	Duration (weeks)	Preceding activities
A	Agreement of the owner	4	
B	Preparation of the ground	2	
C	Order of materials	1	A
D	Excavation	1	A, B
E	Order of doors and windows	2	A
F	Delivery of materials	2	C
G	Concrete foundations	2	D, F
H	Delivery of doors and windows	10	E
J	Frame, roofs, walls	4	G
K	Installation of doors and windows	1	H, J

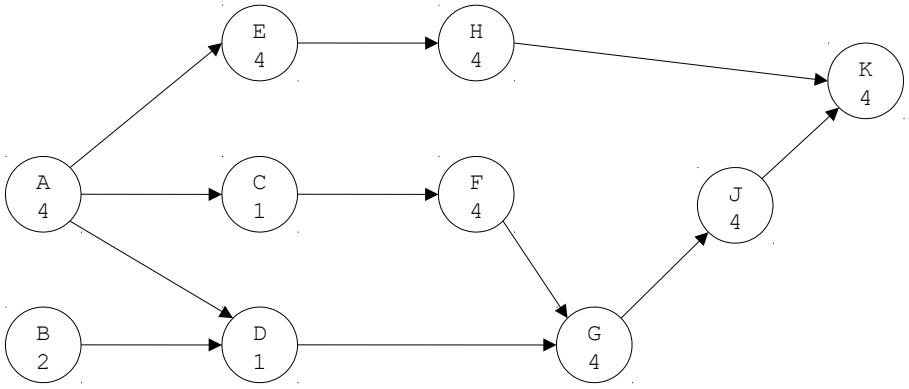


Figure A.1: The network of activities of the warehouse construction project

A.4 Student Cohort

Case Study 6 Student Cohort. This case study consists of a small database of (virtual) cohort of students. Information stored in the database for each student consists of following fields.

- A unique identification number, e.g. 1950224042.
- A family name and a first name, e.g. Doe, John.
- A gender (male or female).
- A date of birth, e.g. 24/02/1995.
- An electronic address.
- Moreover, each student can register to courses identified by their code, e.g. TPK4186, TPK5120...

The list of students and courses they are registred to are given Table A.3.

The unique identification number makes possible to retrieve the gender and the date of birth of the student:

– The first number encoded the gender of the student: it is equal to 1 for male students and to 2 for female students.
– The next two numbers encode the year of birth (e.g. 95 means 1995 and 04 means 2004).
– The next two numbers encode the month of birth.
– The next two numbers encode the day of birth.
– The last three numbers are attributed to students in the order they register at their university.

The unique identification number of Alice Smith is 2961201002, therefore we now that she is born December the 12th 1996.

The electronic address of students are in the form `firstName.familyName@ntnu.no`.

The file `Students.csv` contains personal data of students, one line per students. Data are separated with tabulations. `.csv` files are text files that can be loaded into spreadsheet editors such as Excel®, see Section 7.2.2.

A.5 Train Regularities

Case Study 7 Train Regularities. European train companies are now publishing a number of data related to their operations in open source. The French SNCF (Société Nationale des Chemins de Fer) designed a website to make these data available to companies, public institutions and individuals who want to use them:

`https://ressources.data.sncf.com/explore/?sort=modified`

The regularity of the trains is measured as the percentage of travellers who arrive at their final destination with a delay less than a certain threshold (in minutes). The threshold depends on the category of trains.

The SNCF operates four categories of trains:
– Transilien, which are local trains of the Paris region;
– TER (Transport Express Régional), which are regional trains;
– Intercités, which are trains circulating between cities;
– TGV (Train à Grande Vitesse), which are high speed trains.

The regularity threshold of Transiliens, TER and intercités is 5 minutes.

The regularity of TGV takes into account the duration of the journey. A TGV is considered as "on time" if its delay is:
– less than 5 minutes for a journey of 1:30 hours or less.
– less than 10 minutes for a journey between 1:30 and 3 hours.
– less than 150 minutes for a journey over 3 hours.

The CSV file `regularite.csv` contains data about the regularity of trains circulating on the French network, at different dates. The CSV format is a textual format for spreadsheets, see Section 7.2.2 for detailed explanations.

This spreadsheet contains a number of rows. In each row,
– The first cell (column) contains the date at which the regularity has been measured.
– The second cell contains the family of trains.
– The third cell contains the name of the line.
– The fourth cell contains the regularity of the trains circulating on this line this day.
– The fifth cell contains possibly an explanation for the lack of regularity.

Table A.3: Students in the data base

Number	Family name	First name	Courses
2961201002	Smith	Alice	MAT4105, MEC4131, INF4172, INF4223, MAT4260, MNG4255
1970706004	Johnson	Bob	MAT4105, PYS4160, INF4172, INF4223, PHI4203, MNG4255
2960921005	Williams	Carol	MAT4105, PYS4160, INF4172, INF4223, ECO4215, PHI4203
1970217008	Jones	David	MAT4105, MEC4131, INF4172, INF4223, PYS4265, INF4261
2950307011	Brown	Eve	MAT4105, PYS4160, INF4172, INF4223, MAT4260, MNG4255
1960115013	Davis	Frank	MAT4105, PYS4160, INF4172, INF4223, PYS4265, PHI4203
2970701014	Miller	Grace	MAT4105, PYS4160, INF4172, INF4223, PHI4203, PYS4265
2950504017	Wilson	Heidi	MAT4105, MEC4131, INF4172, INF4223, MAT4240, PHI4203
1970324019	Hall	Ivan	MAT4105, PYS4160, INF4172, INF4223, PYS4265, INF4261
2970626020	Allen	Joan	MAT4105, PYS4160, INF4172, INF4223, MAT4240, PYS4265
1951226023	Young	Michael	MAT4105, PYS4160, INF4172, INF4223, ECO4215, INF4261
2960930024	Hernandez	Nicole	MAT4105, PYS4160, INF4172, INF4223, MAT4260, MNG4255
1950331025	King	Oscar	MAT4105, MEC4131, INF4172, INF4223, PYS4265, ECO4215
2950711026	Wright	Peggy	MAT4105, PYS4160, INF4172, INF4223, MAT4240, PHI4203
1960213027	Smith	Ruppert	MAT4105, PYS4160, INF4172, INF4223, PHI4203, INF4261
2960422030	Brown	Sybil	MAT4105, PYS4160, INF4172, INF4223, MNG4255, INF4261
1950121032	Johnson	Ted	MAT4105, MEC4131, INF4172, INF4223, PHI4203, PYS4265
1960907034	Sanchez	Victor	MAT4105, PYS4160, INF4172, INF4223, PYS4265, MAT4260
1950806036	Hernandez	Walter	MAT4105, PYS4160, INF4172, INF4223, MAT4260, ECO4215

Appendix B

Probability Theory

Key Concepts
- Axiomatic of probability theory
- Sample space, event, σ-algebra
- Probability measure
- Sylvester-Poincaré development
- Conditional probabilities, Bayes's theorem
- Random variable, cumulative distribution function
- Expected value
- Variance, standard deviation
- Laws of large numbers, central limit theorem

Part of model-based systems engineering relies on probability theory. This appendix recalls the main definitions and results of probability theory.

B.1 Axiomatic

As a branch of mathematics, modern probability theory is defined by means of an axiomatic. The axiomatic of probability theory has been proposed by Kolmogorov (Kolmogorov, 1933). It has the advantage to cover both the discrete and the continuous cases.

Definition B.1.1 Sample space. A *sample space* is the set of all the possible outcomes of a non-deterministic experiment. An experiment is *deterministic* if it gives always the same result in the same condition and *non-deterministic* otherwise.

In probability theory, the sample space is usualy called Ω.

Definition B.1.2 Event. An *event* over a sample space Ω is a subset of Ω. An event gather all possible outcomes of the experience that fullfil a certain property.

For the probability theory to be well defined, the set $\mathscr{A}$ of events should have a particular mathematical structure, namely it should be a Borel σ-algebra, or σ-algebra for short.

Definition B.1.3 σ-algebra. Let Ω be a set. A *σ-algebra* over Ω is a set $\mathscr{A} \subseteq 2^{\Omega}$, where 2^{Ω} denotes the *power set*, i.e. the set of subsets of $\mathscr{A}$, such that:

$\mathscr{A} \neq \emptyset$	$\mathscr{A}$ is not empty.
$\forall A \in \mathscr{A}, \Omega \setminus A \in \mathscr{A}$	$\mathscr{A}$ is closed by complementation.
If $\forall n \in \mathbb{N}$, $B_n \in \mathscr{A}$, then $\bigcup_{n\in\mathbb{N}} B_n \in \mathscr{A}$	$\mathscr{A}$ is closed by countable union.

The above definition implies that:
– The empty event $\emptyset$ and the total event Ω belong to $\mathscr{A}$.
– $\mathscr{A}$ is closed by countable intersection.
The pair $(\Omega, \mathscr{A})$ is a probabilisable space, i.e. it is possible to define a probability measure on it.

Definition B.1.4 Probability measure. Let Ω be a set and $\mathscr{A}$ be a σ-algebra over Ω. A *probability measure* p is a function from $\mathscr{A}$ to the real interval $[0,1]$ such that:
1. $\forall A \in \mathscr{A}$, $0 \leq p(A) \leq 1$ (*positivity*).
2. $p(\Omega) = 1$ (*unitary mass*).
3. If $\forall n \in \mathbb{N}$, $A_n \in \mathscr{A}$ and if moreover $\forall i, j \in \mathbb{N}$, $i \neq j$, $A_i \cap A_j = \emptyset$, then $p\left(\bigcup_{n\in\mathbb{N}} A_n\right) = \sum_{n\in\mathbb{N}} p(A_n)$ (*additivity*).

B.2 Additional Definitions and Properties

The above definition induces a number of well-known properties.

Proposition B.2.1 Basic properties of probability measures. Let p be a probability measure over the space $(\Omega, \mathscr{A})$. Then, the following equalities hold for all $A, B \in \mathscr{A}$.

$$\begin{aligned} p(\emptyset) &= 0 \\ p(\Omega \setminus A) &= 1 - p(A) \\ p(A \cup B) &= p(A) + p(B) - p(A \cap B) \end{aligned}$$

The above third equality is extensible to finite unions of events via the so-called Sylvester-Poincaré development.

Proposition B.2.2 Sylvester-Poincaré development. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A_1, A_2, \ldots A_n \in \mathscr{A}$. Then, the following equality holds.

$$p\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{k=1}^{n} \left((-1)^{k-1} \sum_{1 \leq i_1 < i_2 < \ldots < i_n \leq n} p(A_{i_1} \cap \ldots \cap A_{i_k}) \right)$$

The notion of independence plays an important role in probabilistic risk analysis. It is defined as follows.

Definition B.2.1 Independent events. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$. Then, A and B are *independent* if $p(A \cap B) = p(A) \times p(B)$.

The notion of conditional probability captures the idea of measuring the probability of occurrence of an event, given that another event occurred

Definition B.2.2 Conditional probability. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$ such that $p(B) \neq 0$. The *conditional probability* of A given B, denoted as $p(A \mid B)$, is defined as follows.

$$p(A \mid B) \stackrel{def}{=} \frac{p(A \cap B)}{p(B)}$$

It follows immediately from the definitions that if A and B are independent events, the following equality holds.

$$p(A \mid B) = p(A) \tag{B.1}$$

We shall conclude this section by the very useful Bayes's theorem, also called theorem about the probability of causes.

Theorem B.2.3 Bayes's theorem. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$. Then the following equality holds.

$$p(A \mid B) = \frac{p(B \mid A) \times p(A)}{p(B)}$$

B.3 Random Variables

Often, some numerical value calculated from the result of a non-deterministic experiment is more interesting than the experiment itself. These numerical values are called random variables. We shall introduce here only real-valued random variables, but the notion of random variables applies to any measurable set.

Definition B.3.1 Random variable. Let p be a probability measure over the space $(\Omega, \mathscr{A})$. A (real-valued) *random variable* X is a function from Ω into $\mathbb{R}$ such that:

$$\forall r \in \mathbb{R}, \{\omega \in \Omega : X(\omega) \leq r\} \in \mathscr{A}$$

The cumulative distribution function of a random variable is the probability that this random variable takes a value less or equal to a certain threshold.

Definition B.3.2 Cumulative distribution function. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let X be a random variable built over p. The *cumulative distribution function* of X is the function F_X from $\mathbb{R}$ into $[0,1]$ defined as follows (for all $r \in \mathbb{R}$).

$$F_X(r) \stackrel{def}{=} p(\{\omega \in \Omega : X(\omega) \leq r\})$$

Intuitively, the *expected value* a random variable X, also called the *expectation* of X, is the long-run average value of repetitions of the same experiment X represents.

In the finite or denumerable case, this is formalized as follows.

Definition B.3.3 Expected value (finite or denumerable case). Let X be a random variable with a countable set of finite outcomes $x_1, x_2, \ldots$, occurring with probabilities $p_1, p_2, \ldots$, respectively, such that the infinite sum $\sum_{i=1}^{\infty} |x_i| p_i$ converges. The *expected value* of X, denoted $E[X]$, is defined as

following series.

$$E(X) \stackrel{def}{=} \sum_{i=1}^{\infty} x_i \times p_i$$

In the infinite uncoutable case, the formal definition is as follows.

Definition B.3.4 Expected value (uncountable case). Let X be a random variable whose cumulative distribution function admits a density $f(x)$, then the *expected value* of X is defined as the following Lebesgue integral.

$$E(X) \stackrel{def}{=} \int_{\mathbb{R}} x f(x)\, dx$$

Here follows a basic property of expected value.

Proposition B.3.1 Basic property of expected value. Let X and Y be two random variables and $c \in \mathbb{R}$ be a constant. Then,

$$\begin{aligned} E[X+Y] &= E[X]+E[Y] \\ E[cX] &= cE[X] \end{aligned}$$

It is often of interest to know how closely a distribution is packed around its expected value. The variance provides such a measure.

Definition B.3.5 Variance. Let X be a random variable. The *variance* of X, denoted $\mathrm{Var}(X)$, is the expectation of the squared deviation of X from its expected value.

$$\mathrm{Var}(X) \stackrel{def}{=} E\left[(X - E[X])^2\right]$$

The standard deviation has a more direct interpretation than the variance because it is in the same units as the random variable. It is defined as follows.

Definition B.3.6 Standard deviation. Let X be a random variable. The *standard-deviation* of a random variable X, denoted $\sigma(X)$, if the square root of its variance.

$$\sigma(X) \stackrel{def}{=} \sqrt{\mathrm{Var}(X)}$$

B.4 Laws of Large Numbers and Theorem Central Limit

B.4.1 Laws of large numbers

The frequentist interpretation of probability states that if an experiment is repeated a large number of times under the same conditions and independently, then the relative frequency with which an event E occurs and the probability of that event E should be approximately the same.

A mathematical formulation of this interpretation is the law of large numbers, which exists under two forms: the weak law and the strong law.

The weak law of large numbers states that the sample average converges in probability towards the expected value.

Theorem B.4.1 Weak law of large numbers. Let $X_1, X_2 \ldots$ a denumerable family of random variables identically distributed with expected value $E(X_1) = E(X_2) = \ldots = \mu$. Let $\overline{X}_n$ be the average of the sample made of the n first variables, i.e.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

Then, for any positive number ε,

$$\lim_{n\to\infty} Pr\left(\left|\overline{X}_n - \mu\right| > \varepsilon\right) = 0$$

The strong law of large numbers states that the sample average converges almost surely to the expected value.

Theorem B.4.2 Strong law of large numbers. Let $X_1, X_2 \ldots$ a denumerable family of random variables identically distributed with expected value $E(X_1) = E(X_2) = \ldots = \mu$. Let $\overline{X}_n$ be the average of the sample made of the n first variables, i.e.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

Then,

$$\lim_{n\to\infty} Pr\left(\overline{X}_n = \mu\right) = 1$$

The weak law states that for a specified large n, the average of the sample $\overline{X}_n$ is likely to be near μ. Thus, it leaves open the possibility that $\left|\overline{X}_n - \mu\right| > \varepsilon$ happens an infinite number of times, although at infrequent intervals.

The strong law shows that this almost surely will not occur. In particular, it implies that with probability 1, for any $\varepsilon > 0$, there exists a n_0 such that for all $n > n_0$, $\left|\overline{X}_n - \mu\right| < \varepsilon$ holds.

There are special cases, for which the weak law is verified but not the strong one.

B.4.2 Theorem central limit

The central limit theorem states that, in some situations, when independent random variables are added, their properly normalized sum tends toward a normal distribution even if the original variables themselves are not normally distributed. This theorem plays a central role in probability theory because it implies that probabilistic and statistical methods that work for normal distributions can be applicable to many problems involving other types of distributions.

Theorem B.4.3 Theorem Central Limit. Let $X_1, \ldots, X_n$ be independent random variables having a common distribution with expectation μ and variance σ^2. Let $\overline{X}_n$ and Z_n defined as follows.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

$$Z_n \stackrel{def}{=} \frac{\overline{X}_n - \mu}{n/\sqrt{n}}$$

Let $\Phi(z)$ be the distribution function of the normal law $\mathcal{N}(0,1)$, i.e. the normal law of mean 0 and

variance 1. Then for all $z \in \mathbb{R}$,

$$\lim_{n \to \infty} Pr\left(Z_n \leq z\right) = \Phi(z)$$