Hands on Programming with Python

Theory & Practice

JOSE MARÍA ALVAREZ RODRÍGUEZ

ISBN: 9798656766715

DEDICATION

To my beloved family and friends.

CONTENTS

PROLOGUE

After more than 10 years teaching programming courses in different programming languages (C, C++, Java, Fortran and Python), I have realized what learning methods and explanations really work to teach programming concepts. There are many excellent books out there to explore a new programming language (syntax, semantics, libraries, etc.). Furthermore, there are many books focusing on cooking recipes for a specific programming language or showing the capabilities of some libraries. Gray literature such as blogs, web sites, tutorials, question/answers platforms are also a good source of knowledge. However, in just a few sources, one can find the motivation, concept and application of the programming concepts within a programming language.

In this book, programming concepts are theoretically introduced and explained. Then their application is presented in the context of a programming language (Python) to finally present examples of use. This four-stage method (problem statement, concept, Python implementation and examples of use) helps to solve the 5 W's + How questions when learning a new programming language from scratch. In terms of learning methodology, this first ten chapters corresponds to the theoretical concepts that must be complemented with more than 200 examples and exercises in the Lab-x chapters. Furthermore, all contents are also publicly available as Jupyter notebooks[1] and other materials such as Kahoot! Quizzes that are other good learning resources to establish a learning methodology encouraging a blended and self-paced learning process.

This book is structured through different chapters that introduce the basic building blocks in Programming with Python to beginners without any background in computer science and/or programming. This methodology including other learning activities and methods (e.g. project-based learning) has been properly applied to the Programming course in the context of the Engineering Physics degree within the Carlos III University of Madrid (Spain).

.

[1] https://github.com/chemaar/python-programming-course/tree/master/intro-programming

1 INTRODUCTION TO PROGRAMMING

1.1 Context

Is software eating the world?

The popular web pioneer *Mark Anderssen* pointed out and discussed this question in an article in the New York Times in 2011. He stated that everything is fueled by software. From a business to a personal perspective we can agree that this is completely true.

Could we find examples of our daily life in which software is participating somehow?

There are a lot of examples. Usually we understand software as an intangible asset (something that we cannot touch) that is present in computer applications like those we use in our laptop or mobile phones. However, there is much more at stake than the simple existence of software in our personal devices.

Can you think in some example?

Well, just looking around we can find cars, now smart cars that are connected to Internet and can communicate each other to collaborate, for instance to find a car parking, to control the car engine or even to see what is in the next corner. Tesla, Toyota or Continental are good examples of traditional companies that are becoming part of this new wave of smart cars.

Aircrafts also contain a good number of software pieces. In our own home we can find very good examples of software-based devices such as smart TVs. Again, homes on their own are now smart homes.

In our cities, there is also a growing commitment by governments to create new services based on software such as touristic routes or find the best place to eat. We, ourselves, have some software inside. For instance, we use smart bracelets, t-shirts or rings. Besides we have changed a lot our behavior and activities:

When was the last time you bought a music album? 5 years ago? Maybe for a present?

Do you go to a travel agency to make the reservation of a hotel or to buy ticket flights?

Have you realized that telephone boxes are now a point of interest to make pictures but not to call?

I guess that you all are now realizing that we have made an evolution controlled or driven by software. Traditional activities can now be made or enjoyed in a different way for instance:

Have you ever made your shopping basket via a web site?

Have you bought clothes in an on-line shop?

I guess, again, your answer, in a high percentage, is YES!

Indeed, we can agree that software is eating the world and changing our habits and traditions.

Is this new context or environment better or worse?

Well, it depends, we could discuss a lot about it but what it is completely true is that software really seems to be everywhere and that is why we are continuously listening terms such as smart X device, Internet of the Things, cloud computing, mobile sensing, big data, cyber security, etc.

Let me ask you a new question:

Do you think we are then living the fourth industrial revolution?

This phrase “industrial revolution” probably takes you to your foggy memories sitting in the high school in the history class. Was not something about agriculture or steam engines? Yes, there was and, at least, two centuries ago. However, many experts claim that we are living the fourth industrial revolution: cyber-physical systems (this is basically a catch-all term for talking about the integration of smart, internet-connected machines and human labor). In other words, Industrial Revolution 4.0 "is the production-side equivalent of the consumer-oriented Internet of Things, in which everyday objects from cars to thermostats to toasters will be connected to the internet."

When we can consider a new industrial revolution is emerging?

This is not an easy question to answer but we can say that an industrial revolution comes when something happened, a technological advance, and regarding our initial assumptions, everything changes. It is clear that industry 4.0 is here. All indicators suggest that we are going to dive into the age of the smart environments where objects communicate each other for our own benefit improving our workplace, home, city and daily life activities.

Furthermore, this new industry cannot be understood without software, so software is then: **MY PRECIOUS!**

As main conclusion after this introduction to the current impact of software in our lives, we can think about what it means.

We have seen some examples of business that have changed due to the use of software. We have new concepts such as cloud computing or cyber physical systems. We have services that have changed our way of communicating. We have wearables, software that is directly impacting in our personal health and what is most important: all software pieces share a lot of commonalities such as a platform to make comments, to rate the quality of a product or service (the like button is an example) or to represent information in a map.

1.2 Basic computer architecture

A computer system can be classified into two categories[1]:

1. A **desktop computer** that is any PC or laptop we use every day.
2. An **embedded system** that is a kind of software running on top of a hardware architecture for a specific type of device like a TV, a SCADA, a washing machine, a remote control, a toy, etc.

Anyway, any computer system has some common building blocks and principles of operation that are essentially the same. A common computer architecture comprises the next three elements or subsystems:

Processor (CPU: Central Processing Unit). It is the subsystem in charge of **executing** instructions. To measure the velocity of a processor we use Hertz (Hz). Currently, there are many types of processors depending on the way instructions are processed. Furthermore, GPUs (Graphics Processing Units) and FPUs (Floating Point Processing Unit) are gaining momentum due to their performance to process large amounts of data

Memory. It is the subsystem in charge of **storing** any type of item such as data and processes. To measure the capacity of a memory we use bytes (B). There are many types of memories: RAM (Random Access Memory), ROM (Read Only Memory), EPROM (Erasable Programmable Read-Only Memory), EEROM (Electrically Erasable Read-Only Memory) Flash, etc. each of them has different uses

Input/Output (I/O). It is the subsystem in charge of **coordinating the interaction**, in terms of input and output, between the different elements in the computer system. As examples of input devices, we can find a touchable screen, a keyboard, a mouse, etc. output devices: a screen, any display, a speaker, etc. and there are also devices that can be considered for both purposes such as a touchable screen.

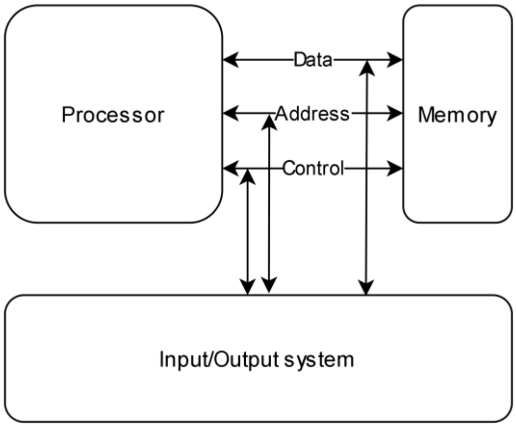


Figure 1 Basic computer system from the O'Reilly book: "Designing Embedded Hardware".

This type of computer architecture is also known as "Von-Neumann architecture[2]". Furthermore, any computer has another important element that is in charge of governing the hardware components: the **motherboard**.

A motherboard is the main printed circuit board (PCB) found in any computer system. It governs the connection and communication between the CPU, the memory and the I/O system.

1.3 What is programming?

Programming can be defined as the **engineering technique** to establish a set of statements that can be executed by a computer machine. In

2 https://www.sciencedirect.com/topics/computer-science/von-neumann-architecture

other words, programming is the art and science of writing computer programs. A computer program can then be defined as a set of sentences that will be executed by a computer machine to accomplish with a task taking an input, making a processing and producing some results.

In this manner, programming is an engineering technique to **code programs** that are **translated into some instructions** that machine (a physical machine) can **automatically** and systematically process.

Programming is part of a major process, the **Software Engineering process**, which among other activities comprises: analysis of the problem, design, implementation, testing, deployment, production, maintenance and retirement of a software system. The implementation of a software system will strongly rely on the programming of the system to be able to produce a program that can fulfill a set of pre-defined requirements.

A complete description of the art of computer programming can be found in the well-known book [2] of Prof. Donald Knuth ("The Art of Computer Programming").

Let's make a first program:

- **Input** two numbers: a and b with some constant values
- **Processing**: the program shall be able to add two constant numbers.
- **Output**: the program shall print out the result of adding two numbers.

```
  a = 2
  b = 3
  c = a + b
  print ("The sum of a {} + b {} is {}".format(a,
b,c))
```

In this very first program, we can see some interesting elements:

- Variables
- Operators
- An statement.
- A print sentence.

These are some of the elements we have to code programs that are part of a **programming language**.

We can put another more complex example in which we generate values to print the `sin(x)` and `cos(x)`.

```
  import matplotlib.pyplot as plt
  import numpy as np
  %matplotlib inline
  x = np.arange(0, 10, 0.1)
  y = np.sin(x)
  z = np.cos(x)

  plt.plot(x,y,x,z)
  plt.xlabel('Time')  # string must be enclosed w
ith quotes '  '
  plt.ylabel('sin(x) and cos(x)')
  plt.title('Plot of sin and cos')
  plt.legend(['sin(x)', 'cos(x)'])
  plt.show()
```

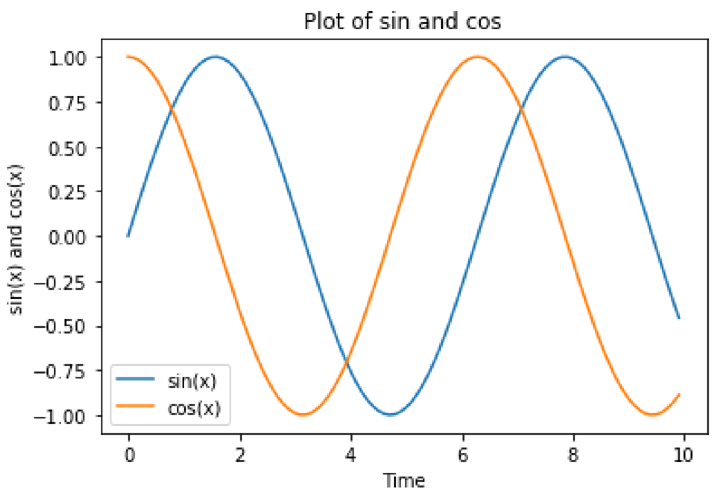


Figure 2 Example of output.

1.3.1 What is a programming paradigm?

A programming paradigm is a philosophy, style, or general approach to write and execute code.

There are different programming paradigms that depend on the way we express the instructions (the programming language) and how they are executed. A common classification divides the programming paradigms in two main styles:

Imperative: the programmer exactly indicates how the machine evolves and changes its state. It is like the extensional definition of a set (Which are the elements of the set?).

Declarative: the programmer declares which are the properties to process and execute functions, then the computer applies the paradigm to any input. Examples here are functional and logical programming. It is like the intensional definition of a set (which are the properties of the set to become a member?)

Later, in this chapter, we will see the notion of programming language. Programming languages are designed to support a programming style, although some languages like Python support different types.

Following, an example of adding up numbers of a list is showed.

```
#Sum the elements of a list
items = [1,2,3,4]

#Imperative programming
sum = 0
for item in items: #for each element in the lis
t, we aggregate values
    sum = sum + item

print("Imperative sum is: "+str(sum))
#Declarative programming (functional)

from functools import reduce
```

```
  #Syntax: reduce(function, sequence[, initial])
-> value
  #Given a function, add, and a sequence of eleme
nts, apply this function to all alements and retu
rn a result
  result = reduce(lambda x, y: x+y, items)
  print("Functional sum is: "+str(result))
```

1.4 What is an algorithm?

General speaking, an algorithm is a set of instructions that are defined to accomplish with a task in a finite time. It is a step-by-step method to solve a problem.

Can you think in examples of algorithms we do every day?

A more formal definition of an algorithm is as follows:

An algorithm is an ordered set of unambiguous, executable steps t hat defines a terminating process.

Building on this definition, which are the characteristics of an algorithm?

- It is **finite**. An algorithm must always finish, although the number of steps or the time to execute them can be potentially huge.
- It is **defined**. Each step of an algorithm is perfectly defined without any ambiguity.
- It has some **input**. Usually any algorithm will take some input data to process.
- It produces some result (**output**). After making the processing of the input data, the algorithm will provide some result.
- It is **effective** and (**efficient?**). The algorithm is expected to make a good use of the resources of a computer: memory, processor and input/output system. This means that we have to think in algorithms to consume as less resources as possible. In small problems, this is not maybe an issue, but in a large scale, it can have a large impact in terms of time or consumption of resources.

Let's put an example, what is the algorithm to calculate the factorial of an positive integer number?

Given a natural number n, n>=0, the factorial is calculated as follows: f(n) = n * f(n-1)

- Input: a natural number, although the algorithm must work for any number.
- Output: a natural number or an error message if the factorial cannot be calculated.

```
#Try to change the value of n and run the code
#Input
n = 5
factorial = 1
error = 0

#Processsing
if n < 0:
  error = 1
elif n == 0 or n == 1:
  factorial = 1
else:
  for i in range (1, n+1):
    factorial = factorial * i

#Output
if error:
  print("To calculate the factorial, n must be 
greater or equal 0")
else:
  print ("The factorial of {} is {}.".format(n, 
 factorial))
```

Finally, just a remark regarding the efficiency of algorithms, there is a way of measuring this characteristic by calculating the complexity in terms of the number of instructions (time) to be executed depending on some input (n). In general, there are different levels such as:

- Constant: O(k)
- Logarithmic: O(log n)
- Linear: O(n)
- Quadratic: O(n^2)
- ...
- Exponential: O(2^n)
- NP-Hard

1.5 What is a programming language?

A **programming language *can be defined as a* language** in which we can express sentences that can be then interpreted and/or executed by a computer machine. A programming language, as any other type of language, includes elements to build correct statements from lexical, syntactical and semantics point of views.

Let's think a bit about the **meaning of a language**, Which elements and processes can we find in any language?

- *Do we have a set of characters or symbols?*
- *Do we have a set of words (a lexicon) that is build using those characters?*
- *Do we have a set of rules or grammar to build correct statements?*
- *Do we have a way of interpreting the statements? What an adjective or and adverb means?*

Once, you have an answer for these questions, let's summarize in the following table a mapping between a natural language (like Spanish or English) and a programming language.

Table 1 Mapping between natural language and a programming language.

Natural language	**Example**	**Programmin g language**	**Example**
Character set	a, b, c, etc.	Character set	a, b, c, etc.
Words	"hello"	Tokens	print
Grammar (syntax)	<subject> <predicate> <object>	Grammar (syntax)	<identifier> = <expression>
Meaning (semantics)	"A way of greeting..."	Meaning (semantics)	"Assignment of a value to a variable"

As you can see, both have a lot of commonalities. We need characters to build words (that in computer programming are called **tokens**). There are some grammar rules that help us to "speak" properly and, finally, there are some semantics that allow us to have a common understanding of the statements.

What happened if we find an issue in any of these levels?

Character or symbol not recognized. This means we are using some character that does not belong to the language and, it is not a valid character. It cannot be use to build any word in that language.

Undefined token. This means we are using a word that does not exist in the dictionary. The word is not part of vocabulary.

Expected a . at the end of the statement. This means that we are not fulfilling the rules to write (structurally) a valid statement. The statement is not accepted by the language syntax.

The expression shall return an integer value. In natural language, the semantics focuses on the interpretation of the words and their combination in a sentence. It is assumed to be the same regardless who is reading the sentence. In a programming language, semantics means that the interpretation of the sentences must be correct as well. It is not possible to assign a string literal to an integer value; it does not make sense since they are not the same.

As a summary, a programming language is a formal language to express correct instructions that can be executed by a computer machine.

1.5.1 Types or programming languages

Firstly, it is necessary to understand that we are working in a logical/virtual level and the instructions that we code in a program will be translated into physical pulses.

There are many classifications of programming languages depending on this level of abstraction, programming paradigm, generation, compiled/interpreted etc.

1.5.1.1 Level of abstraction

Low level programming languages. There are programming languages which code is written in a code close to a machine language. An assembly programming language like X86 is an example.

Table 2 Example of assembly code.

```
.global main

       .text
main:

   # This is called by C library's >startup code

   # First integer (or pointer) parameter >in %rdi

       mov     $message, %rdi

   # puts(message)

       call    puts
       ret

   # Return to C library code
message:

   # asciz puts a 0 byte at the end
       .asciz "Hello, world!"
```

Source: x86 Assembly Language Programming

Programs create in assembly code have the following characteristics:

- They are *efficient* in terms of memory management and very fast.
- They take advantage of the *hardware architecture*.

- They can directly *manipulate elements* of the CPU and memory.

On the other hand, these programming languages bring some disadvantages:

- **Portability**, since they are machine and architecture specific cannot be easily ported to other architectures.
- **Productivity** of programmers is constrained. The development, debugging and maintenance processes are complicated and more prone error.
- **Dependency** on the knowledge that a programmer can have of the machine and computer architecture.

Mid-level programming languages. These programming languages have features from both low-level and high-level programming languages. This means it is possible to manage resources (like the memory) at a low-level but you also have primitives and a complete programming language (syntax and semantics) similar to a high-level programming language. As an example, the C programming language could be considered a mid-level programming language.

High-level programming languages. These languages are close to the human language in terms of having a lexicon, a syntax and a semantics. They have primitive to easily manage computer resources and the program flow. Here, Python, Java, C++ are just examples of high-level programming language.

The main advantages of these programming languages are (improvement of low-level programming languages):

- *High-level of abstraction* to manage computer resources and control flow.
- They are *machine-independent*.
- *Productivity*, they are easy to learn improving the productivity of programmers.

While the main disadvantages come:

- **Compilation time**. It will take some time to translate the high-level structures into machine instructions.
- **Execution or interpretation time**. Currently, compilers are superb tools that can optimize the code that is generated.

However, there are still routines that can be improved at a low-level.

- **Efficiency**. Since the number and type of instructions generated by the compiler can change, more instructions will be then executed.

In general, high-level programming languages are preferred for solving most of the problems we can find in many domains.

1.5.1.2 Programming paradigm

As it has been introduced in the previous section, there are two main types of paradigms:

- Imperative
- Declarative

1.5.1.3 Generation

In general, the generation of a programming language depends mainly on its level of abstraction and, in many cases, in the time it was designed. There are 5 different levels:

- 1G Languages. Low-level languages like machine language.
- 2G Languages. Low-level assembly languages used in kernels and hardware drives.
- 3G Languages. High-level languages like Python, C, C++, Java, Visual Basic and JavaScript.
- 4G Languages. Languages which statements are close human language such as SQL or MatLab (MatrixLaboratory).
- 5G Languages. Domain-specific programming languages like Prolog (for logics).

1.5.1.4 Type of program execution

Native (machine-dependent, executable file). The compiling and linking process will generate a native executable file that will be executed by the operating system.

Interpreted (an interpreter, a virtual machine). The compiling process will generate some intermediate language (like Bytecode in Java or IL en .NET) that is then executed by an interpreter or virtual machine. In this manner, we add a new layer of abstraction to our programs. It is not necessary to build different versions of a program for the different platforms, but, what it is necessary is an interpreter for each platform. Let's say we separate the execution of the program from the machine.

1.5.1.5 Type of management of data and operations

Procedural programming. The program is modularized in procedures that take as an input some data and return some result. The way we think in programs is in functions that may perform some operation. C, Pascal or even Python can be classified as procedural programming languages.

Object-oriented programming. In this case, we model a problem with objects that have some attributes (data) and methods (operations). When an object is created, it has a state that evolves (object lifetime) depending on the operations that are being executed. Java, C++, C# use this object abstraction to model solutions.

Functional programming. It is similar to procedural programming, but it is a kind of computing style in which the functions are formally defined avoiding side-effects. Haskell and Python are examples of this type of language.

Logic programming. There are languages that can be used to model a problem as a set of logic restrictions that are then solved by a reasoner. Prolog is an example of this type of language.

Prototype-oriented programming. Like object-oriented programming, here the programming languages define the notion of a class as a prototype that can receive any message for getting an attribute or a method. This prototype is similar to the notion of interface in object-oriented programming. In prototype-oriented programming, it is important how the invocations to methods or attributes are solved. JavaScript and Python are examples of this type of programming language.

1.5.1.6 Datatypes management

Strong typing. When we declare an element of the program like a variable, a function or a function parameter, we must establish a type (e.g. integer, float, date, etc.). In this manner, the compiler can check whether the program contains type error (usually checking the semantics) and, before, executing the program we already know: 1) the size that the program will initially take in the memory and 2) if the assignments and function invocations are correct. These programming languages are usually more secure than those with duck typing. They prevent some common errors. Java, C++ or C are examples of these programming languages.

Duck typing. The name "duck" comes from this sentence "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." If we apply this concept to types in a programming language, it means that we can declare elements of the program without indicating the type. The compiler will not detect typing errors, and, only during the execution errors will be detected throwing an exception. The strategy to assign a type are mainly two: 1) type of the value that is assigned for the first time to a variable and 2) each time that a value is assigned, the variable will take the type of that value. The programming languages with duck typing are more prone to errors. Python, Scala or Ruby are examples of this category.

```
#Example of a function in Python

def sum(a,b):
  return a + b

print("The sum is: "+str(sum (2,3)))

#Example of a class in Python

class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def say_hello(self):
    return ("I am {} and I am {} years old."
.format(self.name, self.age))

p = Person("Jose", 37)
print("Name: "+p.name)
print("Age: "+str(p.age))
print(p.say_hello())

#Example of functional programming in Python
```

```
values = [1,2,3,4,5,6]
#Leave only the values > 3 generating a new
list
filtered_values = [x for x in values if x>3]
print(filtered_values)

#Example of duck typing in Python

a = 5 #We know a is now of type integer
print(type(a))
a = "Hello" #Now a is of type string
print(type(a))
```

As a summary of this section, a programming language can be classified into many different categories depending on the aspect we are evaluating.

A programming language like Python is a high-level programming language that belongs to the 3-4 generation and includes capabilities for both imperative and declarative programming. It has an abstraction to create classes (prototype-style) and it contains some features for functional programming.

1.6 Foundations of problem solving: analysis, design, implementation and testing

Our main purpose in this book is to learn how to analyze problems and design and code solutions in Python.

However, the development of a software project comprises many processes, activities and tasks what is also known as software development lifecycle (SDLC). Let me give you some examples, if we are architects before starting to build a bridge or a building, we have first to design a solution and when everything is defined, we can start building.

Again, if we are a chef and we want to create a menu we have to make a plan, buy food and ingredients, prepare the kitchen utensils and finally, when everything is ready, we can start *cooking*. So, in software development we follow a similar approach. Before coding we have to carry out different activities: analyze the problem, design a solution, code the solution and test.

1.6.1 The software development lifecycle (SDLC)

Basically, the SDLC is the process, activities and tasks we do to deliver a software that is verified and validated. In other words, a working software that is reliable, robust, secure, etc. and makes what it must do. "*Do the right thing right*". Everything has a lifecycle, and, in the software world, we have the software lifecycle.

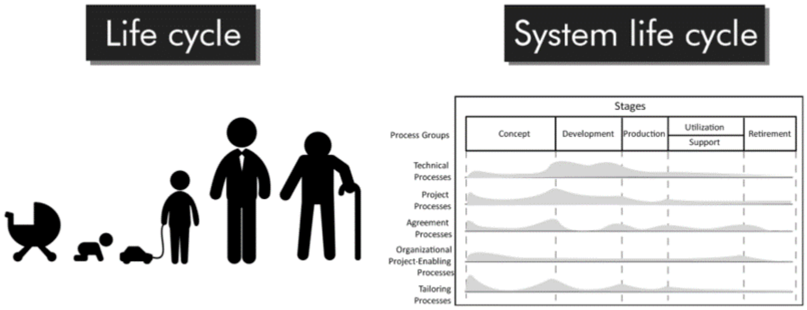


Figure 3 Software lifecycle.

There are many SDLCs but, in our context, we will focus on the basic SDLC, a kind of cascade covering: analysis, design, coding and testing.

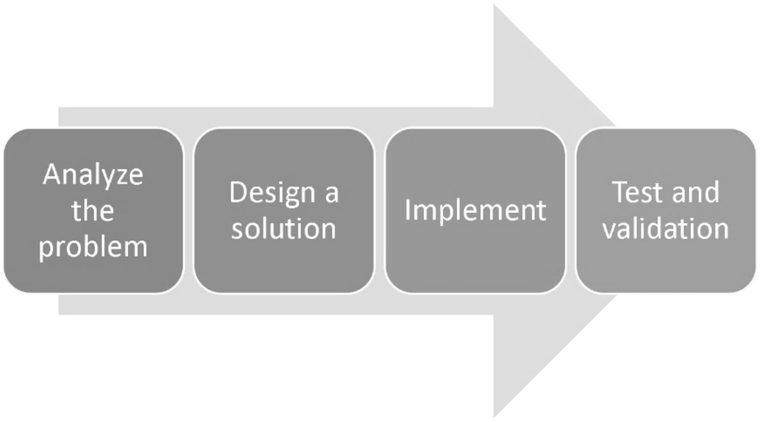


Figure 4 Software development lifecycle.

Analysis. We basically solve a question "What?". We must understand a problem, extract the required functionalities (and non-functional requirements such as performance, the "-ilities": usability, accessibility, etc.)

getting as an outcome a set of functional requirements. We work in the problem domain.

Design. In this stage, we solve a question "How?". We formulate a solution for the problem (we model a solution) or to meet the defined requirements. In the context of this book, a design would be a kind of pseudo-code or flow diagram in most of cases.

Coding. We implement the solution in a programming language.

Testing. We verify ("do the thing right") and validate ("do the right thing") that the program is working properly. There are many testing techniques but, basically, for us a kind of white-box and black-box testing is enough. White-box texting means that we can ensure that any sentence in the program has been executed and works properly. Black-box testing means that the functionalities of our program have been testing using techniques such as limit values.

```
#Given two integer numbers, make a program that adds both numbers and displays the result
#Analysis: it is the own statement of the problem
#Design: we declare two variables, we load values and we perform the operation (+) showing the result in the console
#Testing:
#Test Case description: we input 2 and 3, the program shall show 5


a = 2
b = 3
result = a+b
print(result)
```

1.7 Building a program: the compiling process

A compiler can be defined as follows:

A program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer.

A general compiling process comprises the next stages (they may change depending on the type of programming language, type of execution, etc.):

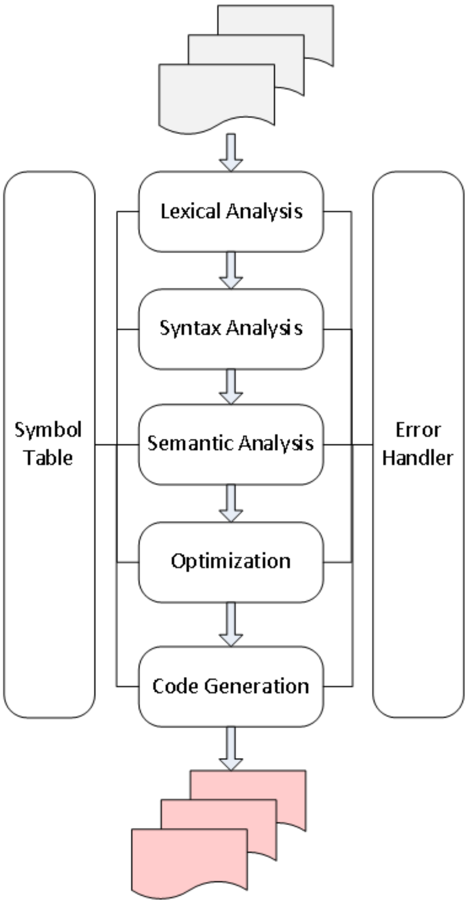


Figure 5 General compilation process.

1.8 Information encoding

Software represents an intangible asset that runs on top of a computer hardware, executing instructions and producing results. Here, we can raise some questions:

- How information is gathered from an user?
- How information is sent to an input/output device?
- How physical signals (from transistors, etc.) are converted into something digital that can be processed by a software system?

Information encoding is the process to encode/decode information within a computer system. This process is in charge of converting virtual data into physical signals that can be stored and processed in the hardware level. In the same manner, those signals produced by the hardware are processed to be converted in something interpretable by a software system.

The minimum unit of information is a bit (binary digit).

There are other scales of information depending on the number of bits (see the next table). In terms of abbreviation, B (Bytes) is used when it refers to 1024 bits. There are other cases in which B is used in lowercase letter, this means 1000 bits. So, 1 MB is 1024 KB while 1 Mb corresponds to 1024 Kb.

Table 3 Bits, bytes and units of measurement.

Name	**Abbreviation**	**Size**
Byte	B	8 bits
KiloByte	KB	2^{10}=1024 bytes
MegaByte	MB	2^{20} bytes
GigaByte	GB	2^{30} bytes
TeraByte	TB	2^{40} bytes
PetaByte	PB	2^{50} bytes
ExaByte	EB	2^{60} bytes
ZettaByte	ZB	2^{70} bytes
YottaByte	YB	2^{80} bytes

In general, computer hardware works in the binary world (0's and 1's) generating bits. This means that a signal (+-5V) generated by a device are interpreted as a 0 or a 1. These values are aggregated to be interpreted as an instruction, a data item, etc. So, there is some software that is in charge of interpreting signals (communication with the hardware, device driver). Then these sequences of 0's and 1's are interpreted and executed by other type of software (operating system) to be consumed by some user application. This trip can be also done in the other way around, an user application send some instruction and data to the operating system that delegates in some driver the communication with the device to finally do something (print some character in the screen). This flow of information is basically built on top of a set of characters that are used to represent everything (instructions and data). These characters are put together to build instructions or to be interpreted as numbers, etc. but, essentially, everything is a character, a byte (8 bits).

In this context, it is necessary to provide mechanisms to encode information as characters. To do so, there are character sets such as ASCII, ISO, etc. and algorithms to encode information such as UTF-8, UTF-16, etc. These characters correspond to a number that can be encoded in other format such as binary or hexadecimal. For instance, the letter 'A' in ASCII corresponds to the number 65 in decimal or 41h in hexadecimal format. So, to encode information, apart from having a character set, it is necessary to provide methods to represent information in binary, octal or hexadecimal.

In order to represent information in binary, we have first to think in the general form of a number N (e.g. 27 in 10 base):

$$N = \sum_{i=-q}^{p-1} a_i r^i$$

$$(N)_r = (a_{p-1} \; a_{p-2} \; a_{p-3} \; \dots a_0 \; a_{-1} \; a_{-2} \; a_{-3} \; a_{-q})$$

If we represent 27 in 10-based using this general representation form, we get the following:

- $(27)_{10} = (a_1 = 2 \;\; a_0 \; = 7 \; a_{-i} = 0)$
- $(27)_{10} = 2 \; * 10^1 + 7 \; * 10^0$

This number representation schema is also applicable to other bases. However, it is necessary to consider the type of symbols that are being used, whether the representation is human-readable and use in computers. In the following table, a summary of the decimal, binary, octal and hexadecimal bases is presented.

Table 4 Summary of number representation base and symbols.

Name	**Base**	**Symbols**	**Human-Readable**	**Use in computers**
Decimal	10	0, 1, … 9	Yes	No
Binary	2	0, 1	No	Yes
Octal	8	0, 1, … 7	No	No

Hexa-decimal	16	0, 1, … 9, A, B, … F	No	No

As a simple exercise, we can try to count using these different representation formats.

Table 5 Example of counting in different bases.

Decimal	**Binary**	**Octal**	**Hexadecimal**
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Once the general form of a number and the different base symbols are also known, it is convenient to explore the methods to encode numbers in different bases and make a round trip among them. Considering, the example (number 27 in 10 base), it is possible to represent that number in the different bases as it is presented below. So, the question is: How can we transform a number from one base to another?

$27_{10} = 11011_2 = 33_8 = 1B_{16}$

First things first, let's interpret numbers in different bases in decimal:

- **Convert binary to decimal:**
    - Multiply each bit by 2^n (source base) where **n** es the position of the bit.
    - Start from to **0** on the **right.**
    - Sum up the preliminary results.

11011_2	1	x 2^0	=	1
	1	x 2^1	=	2
	0	x 2^2	=	0
	1	x 2^3	=	8
	1	x 2^4	=	16
				27

- **Convert octal to decimal:**
    - Multiply each bit by 8^n (source base) where **n** es the position of the digit.
    - Start from to **0** on the **right.**
    - Sum up the preliminary results.

33_8	3	x 8^0	=	3
	3	x 8^1	=	24
				27

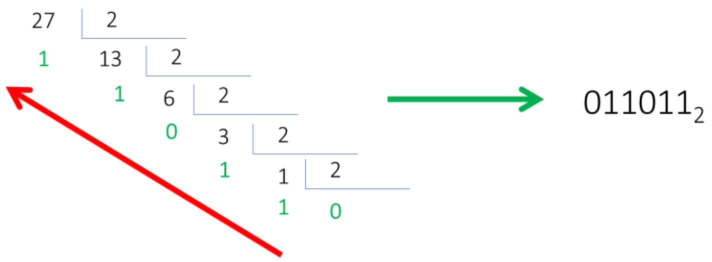
- **Convert hexadecimal to decimal:**

- Multiply each bit by 16^n (source base) where **n** es the position of the digit.
- Start from to **0** on the **right.**
- Sum up the preliminary results.

$1B_{16}$	B x 16^0 = 11 1 x 16^1 = 16 **27**

Let's try now to transform from one base to another.

- **Transform a decimal number to binary:**
  - Divide by the target base (2), annotate the remainder
  - First remainder is bit 0 .
  - Second remainder is bit 1 and so on.



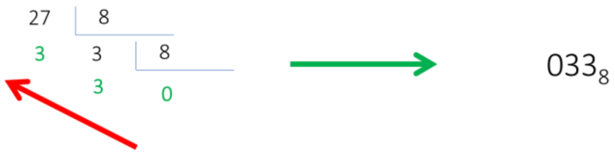
- **Transform a binary number to octal:**
  - Group bits in sets of three bits ($2^3 = 8$, 3 bits required), starting on right.
  - Convert to octal digits.

33_8	$011\ 011_2$

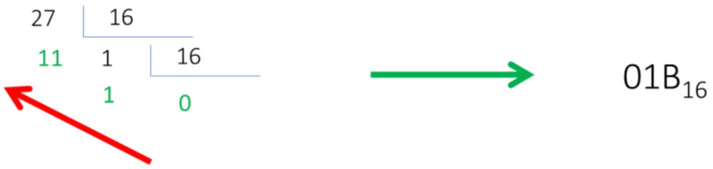
- **Transform a binary number to hexadecimal:**
  - Group bits in sets of four bits ($2^4 = 16$, 4 bits required), starting on right.
  - Convert to hexadecimal digits.

$1B_{16}$	$0001\ 1011_2$

- **Transform a decimal number to octal:**
    - Divide by the target base (8), annotate the remainder
    - First remainder is position 0
    - Second remainder is position 1
    - Etc.



- **Transform a decimal number to hexadecimal:**
    - Divide by the target base (16), annotate the remainder
    - First remainder is position 0
    - Second remainder is position 1
    - Etc.



Try to make the next exercise, converting numbers between different bases:

Table 6 Example of conversion between bases.

Decimal	Binary	Octal	Hexadecimal
135	10000111	207	87
89	1011001	131	59
39	100111	47	27
272	100010000	420	110

So far, we have seen how to transform and interpret numbers in different bases. However, this only works for natural numbers and, in a computer system, it is necessary to provide operations that work with integer and/or real numbers. To do so, there are other techniques and standards to encode this type of information. These representation schemes can be classified in two main groups: 1) fixed point representation and 2) floating point representation.

In the first case, fixed point representation, we can apply the general representation format of a number to encode information. We will use: m + n bits → m: integer part, q: scaling factor (power).

Given a number 0,625 in base 10 convert it to a binary representation s

- a_{-1} -> 0,625 x 2 = **1**,25
- a_{-2} -> 0,25 x 2 = **0**,5
- a_{-3} -> 0,5 x 2 = **1**,0 (Stop, a 0 has been reached in the real part)
- a_{-4} -> 0 x 2 = **0**

Let's interpret the sequence of bits:

- $(0{,}625)_{10} = (a_{-1} = 1 \ a_{-2} = 0 \ a_{-3} = 1)$
- $(0{,}625)_{10} = 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3} = {}^{1}/_{2} + {}^{1}/_{8} = 0{,}5 + 0{,}125$ = 0,625
- $(0{,}625)_{10} = (0{,}101)_{2}$

Try to make the next exercise, converting the following real numbers using the general representation format of a number:

Table 7 Example of conversion real numbers in a fixed-point representation scheme.

Decimal	**Binary**	**Octal**	**Hexadecimal**
2,5	10.1	2.4	2.8
38,65	100110.1010011001100110 0110011001100110	46.514631 46…	26.A6666666
12,5	1100.1	14.4	C.8
4,625	100.101	4.5	4.A

So, the next question is: how we can encode the sign? There are some methods to represent the **integer numbers**:

Signed-magnitude (easy for humans):

- Given **n** bits → 1 bit sign + (n-1) bits magnitude.
- Range: $-(2^{n-1}-1)\ to\ (2^{n-1}-1)$
- Example: 8 bits $(-2^{8-1}-1,\ 2^{8-1}-1)=(-127,\ 127)$
    - 27_{10} = **0**001 1011_2
    - -27_{10} = **1**001 1011_2
- Known-Problems
    - Ambiguity: 0 = 1000 0000 or 0 = 0000 0000
    - Overflow in arithmetic operations.

1's complement (self-delusion):

- It is the bitwise **NOT** applied to it — the "complement" of its positive counterpart.
- Range: $-(2^{n-1}-1)\ to\ (2^{n-1}-1)$
- Example 8 bits:
    - 27_{10} = **0**001 101**1**$_2$
    - -27_{10} = **1**110 010**0**$_2$
- Known-Problems
    - Ambiguity: 0 = 0000 0000 or -0 = 1111 1111

2's complement (easy for computing):

- Algorithm I:
    - Starting from the right, find the first '1'
    - Invert all of the bits to the left of that one.
    - Examples 8 bits:
        - 27_{10} = 0001 101**1**$_2$
        - -27_{10} = 1110 010**1**$_2$
        - 42_{10} = 0010 10**10**$_2$

        - $-42_{10} = 1101\ 01\mathbf{10}_2$

- Algorithm II:
    - Invert all the bits through the number (1’s complement)
    - Add 1.
    - Examples 8 bits:

$$
\begin{array}{ll}
27_{10} = 0001\ 1011_2 = & 1110\ 0100_2 \text{ (1's complement)} \\
 & + \\
 & 0000\ 0001_2 \\
\hline
 & 1110\ 0101_2 = -27_{10}
\end{array}
$$

$$
\begin{array}{ll}
42_{10} = 0010\ 1010_2 = & 1101\ 0101_2 \text{ (1's complement)} \\
 & + \\
 & 0000\ 0001_2 \\
\hline
 & 1101\ 0110_2 = -42_{10}
\end{array}
$$

Excess-Z:

- A value is represented by the unsigned number which is Z greater than the intended value.
    - 0 is represented by *Z*
    - $-Z$ is represented by all-zeros
- Generalization of 2’s complement-> Excess-2^{N-1}
- Example: 8 bits → Excess-127
    - $27_{10} = 27_{10}\ \mathbf{+\ 127} = 154 = 1001\ 1010_2$
    - $-27_{10} = -27_{10}\ \mathbf{+\ 127} = 100 = 0001\ 1111_2$
    - $1001\ 1010_2 = 154\ \mathbf{-\ 127} = 27_{10}$
    - $0001\ 1111_2 = 100\ \mathbf{-\ 127} = -27_{10}$

Let’s look to the next examples:

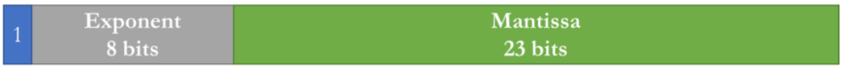
Table 8 Representation of integer numbers.

Number	Binary (unsigned)	Signed-magnitude	1's complement	2's complement	Excess-127
0	0000 0000	0000 0000	0000 0000	0000 0000	01111111
1	0000 0001	0000 0001	0000 0001	0000 0001	10000000
-1	N/A	1000 0001	1111 1110	1111 1111	0111 1110
127	0111 1111	0111 1111	0111 1111	0111 1111	1111 1110
-127	N/A	1111 1111	1000 0000	1000 0001	00000000
128	1000 0000	N/A	N/A	N/A	11111111
-128	N/A	N/A	N/A	N/A	N/A
255	1111 1111	N/A	N/A	N/A	N/A

On the other hand, it is also required to encode real numbers trying to keep the precision (avoiding issues when performing numerical calculations). To do so, there is standard to encode/decode real numbers: the IEEE-754 format.

- As it has been introduced, a number can be represented in floating point as a power of a base
    - 27_{10} = **$27*10^0$** or **$2,7*10^1$** or **$0,27*10^2$**
- Usually (but not always) the normalized form places the radix point immediately to the left of the leftmost, nonzero digit in the fraction: **$0,27*10^2$**

The IEEE-754 is a technical standard for floating-point computation established in 1985. The last version IEEE 754-2008 has been published in August 2008 (current). This format can make use of a different number of bits. In this section, the focus is on the simple precision format with 32 bits that are distributed as follows.



First, it is convenient to recall the notion of standard format of a number which may be either base 2 (binary) or base 10 (decimal):

- Each finite number is described by three integers:
    - *s* = a *sign* (zero or one),
    - *c* = a *significand* (or 'coefficient')
    - *q* = an *exponent.*
- The numerical value of a finite number is $(-1)^s \times c \times b^q$
    - *b* is the base (2 or 10).
    - For example, if the sign is 1 (indicating negative), the significand is 12345, the exponent is −3, and the base is 10, then the value of the number is −12.345.
- Two infinities: +∞ and −∞.
- Two kinds of NaN (Not a Number):
    - a quiet NaN (qNaN) and a signaling NaN (sNaN).
    - A NaN may carry a *payload* that is intended for diagnostic information indicating the source of the NaN.
    - The sign of a NaN has no meaning, but it may be predictable in some circumstances.

Keeping this in mind, and considering the IEEE-754 (32 bit), the steps to encode a number in this format are as follows:

1. Convert the number into to a binary form:
    - $-2345.125_{10} = -100100101001.001_2$
2. Normalized the number:
    - $-1\mathbf{00100101001.}001_2 = -1.00100101001001_2 \mathbf{* 2^{11}}$
3. Extract the values of the sign, exponent and mantissa:
    - $-1.00100101001001_2 \mathbf{* 2^{11}}$
        - Sign: -1
        - Exponent (excess-Z 127) : 11 + 127 = 138
        - Mantissa (in 23 bits): 00100101001001000000000
4. Represent according to the standard form:

1	10001010	00100101001001000000000

If we apply the steps in reverse order to decode the number and get the decimal value, we will proceed in the following manner:

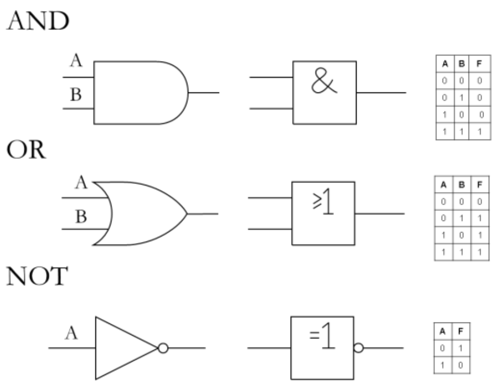
1. Extract the values of the sign, exponent and mantissa:
    - Sign: $(-1)^{1}$
    - Exponent: 10001010 = 138 – 127 = 11
    - Mantissa: **00100101001001**000000000
2. Normalized the number:
    - $-1.$ **00100101001001**$000000000_2 * 2^{11}$
3. Transform from binary to decimal:
    - -1**00100101001.001**000000000 = -2345.125_{10}

As an exercise: represent -12.62510 in single precision IEEE-754 format.

1	1000 0010	1001 0100 0000 0000 0000 000

1.9 Boolean algebra

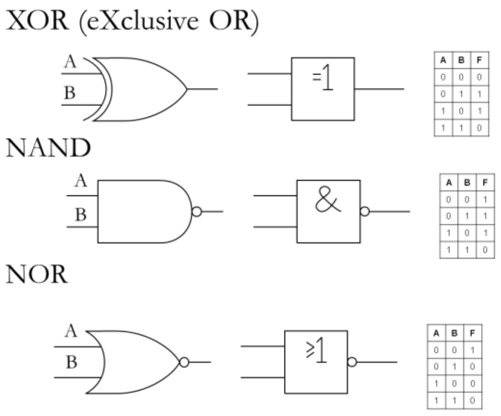
Logical operators and digital logical gates. In logical expressions, logical operators will be used to express conditions. The basic logical operators are: AND, OR and NOT. These operators have a truth-table and can be implemented, at a physical level, with digital logical gates as it is presented below.



A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

A	F
0	1
1	0



A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Monotone laws. The Boolean algebra represents the mathematical foundations for computational logic. These rules serve us to govern and express logical expressions that can be used in conditional statements or, in a more advanced setting, to perform reasoning processes. Monotone laws are those which change in the input does not imply a change in the output.

Table 9 Bool Algebra: monotone laws.

Associativity of ∨ (OR)	$x \lor (y \lor z)$	=	$(x \lor y) \lor z$
Associativity of ∧ (AND)	$x \land (y \land z)$	=	$(x \land y) \land z$
Commutativity of ∨	$x \lor y$	=	$y \lor x$
Commutativity of ∧	$x \land y$	=	$y \land x$
Distributivity of ∧ over ∨	$x \land (y \lor z)$	=	$(x \land y) \lor (x \land z)$
Identity for ∨	$x \lor 0$	=	x
Identity for ∧	$x \land 1$	=	x
Annihilator for ∧	$x \land 0$	=	0
Idempotence of ∨	$x \lor x$	=	x
Idempotence of ∧	$x \land x$	=	x
Absorption 1	$x \land (x \lor y)$	=	x
Absorption 2	$x \lor (x \land y)$	=	x
Distributivity of ∨ over ∧	$x \lor (y \land z)$	=	$(x \lor y) \land (x \lor z)$
Annihilator for ∨	$x \lor 1$	=	1

Nonmonotone laws

Table 10 Bool Algebra: nonmonotone laws.

Complementation 1	$x \land \neg x$	=	0
Complementation 2	$x \lor \neg x$	=	1
Double negation	$\neg\neg$x	=	x

De Morgan 1	$(\neg x) \wedge (\neg y)$	=	$\neg(x \vee y)$
De Morgan 2	$(\neg x) \vee (\neg y)$		$\neg(x \wedge y)$

Finally, some source code snippet to use different number representation schemes in Python is presented to show the notation used in the programming language.

```
#Binary: notice the format 0b
a = 0b11001
print(int(a))
#Decimal
a = 25
print(a)
#Octal: notice the format 0o
a = 0o31
print(int(a))
#Hexadecimal: notice the format 0x
a = 0x19
print(int(a))
```

1.10 The Python programming language

Python was designed in the late 1980s by Guido van Rossum (CWI, Google) inspired by the ABC programming language (a general-purpose language and environment created in the CWI, Netherlands).

Initially, the Python creator and other developers were thinking in a programming language that will fulfill some requirements (see the next section, The Zen of Python). The philosophy behind Python is that** code must be readable helping programmers to write clean code**. As we have introduced, Python is a general-purpose programming language that supports multiple programming paradigm: procedural, object-oriented, and functional programming. It also includes a large library of functionalities (modules that offer some specific capabilities).

There are Python interpreters for most of the operating systems and a community of developers maintains the Python interpreter CPython, an open source reference implementation. Currently, Python is in its version 3.8.1. The name of "Python" comes as a tribute to the Monty Python group.

The use of Python can be found in many domains and sectors:

- Google
- Facebook
- Instagram
- Spotify
- Quora
- Netflix
- Dropbox
- Reddit
- Microsoft

Last times, Python is keeping a good fight in the programming languages area to become the main programming language for data science. (see the next video)

1.10.1 The Zen of Python

"Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down."[3] These principles are guidelines to properly write Python code.

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.

[3] https://www.python.org/dev/peps/pep-0020/

- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one -- and preferably only one -- obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

1.11 Relevant common resources

- PEP (Python Enhancement Proposals) guidelines. Source: https://www.python.org/dev/peps/
- Python official tutorial. Source: https://docs.python.org/3/tutorial/index.html
- Real Python: a very complete blog including many tutorials in different topics. Source: https://realpython.com/
- Learn Python: a blog including interesting resources on Python programming concepts. Source: https://www.learnpython.org/
- Books: it is possible to find many programming books in Python. From basic concepts to cooking recipes or advance modules for specific purposes, the landscape of bibliographic resources is huge. To summarize reading list of relevant books is presented below:
    - "Programming in Python 3: a complete introduction to the Python language" [3].
    - "Fluent Python" [4].

- “Python Cookbook” [5].
- “Head first Python” [6].
- “Open Book Project-Python”. Source: http://openbookproject.net/thinkcs/python/english3e/

- Open education platforms such as edX, Coursera, Udemy or Udacity contains many courses on Python applied to different domains such as data science.

1.12 Quiz

1. **What CPU and RAM stands for?**

   a. Central Processing Unit and Random Access Memory

   b. Central Processing Unit and Read Access Memory

   c. Central Programming Unit and Random Access Memory

   d. Central Programming Unit and Read Access Memory

Answer: a

2. **An algorithm must be:**

   a. Ambiguous

   b. Finite

   c. Undefined

   d. Discrete

Answer: b

3. **According to the type of program execution, a programming language:**

   a. Functional

   b. Procedural

   c. Object-oriented

   d. Interpreted

Answer: d

2 PROGRAMMING ELEMENTS

2.1 Elements of a programming language

A program is composed of a set of different statements that perform some operation or algorithm taking some input and generating some output. During the design and implementation of a program, we will find different needs:

- Store (and recover) values of different type.
- Create expressions to perform some calculation.
- Assign values.
- Make decisions depending on some state.
- Repeat a set of statements until reaching some state.
- Define specific sub-programs or functions.
- Reuse existing functionalities.
- ...

To do so any programming language comprises a set of basic elements of programming.

2.1.1 Identifiers

2.1.1.1 Problem statement

Information is stored in memory slots. A memory slot is reference by memory address that usually is represented as an hexadecimal number: 0x0000F1A0. A memory address has not been designed to be used by humans (like programmers) and, unless, you are coding at a very low level, it is not easy to remember a memory address and to know what content and type is stored in such address.

Furthermore, when coding, we create new functionalities, elements, etc. that must be reused. So, we need a technique to be able to easily make references to those elements we have created.

2.1.1.2 Concept

To ease the management of memory resources and to be able to reuse parts of our program (e.g. variables, functions, etc.), we need a strategy to name it.

An identifier can be defined as human-readable name for an element within a program.

2.1.1.3 Application

It is possible to name almost any element within a program. For example:

- Constants
- Variables
- Functions
- Class attributes
- Class methods
- Modules
- ...

2.1.1.4 Identifiers in the Python programming language

In the Python programming language, an identifier is basically a sequence of case sensitive characters that must begin with a letter (a-z,A-B) or underscore (_) and follow by other similar characters with a reasonable length.

Some reserved words cannot be used as identifiers to avoid ambiguity.

Table 11 Lexical production rules.

```
  identifier   ::=  xid_start xid_continue*
id_start     ::=  <all characters in general categorie
s Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and characte
rs with the Other_ID_Start property>
id_continue  ::=  <all characters in id_start, plus ch
aracters in the categories Mn, Mc, Nd, Pc and others w
ith the Other_ID_Continue property>
xid_start    ::=  <all characters in id_start whose NF
KC normalization is in "id_start xid_continue*">
xid_continue ::=  <all characters in id_continue whose
 NFKC normalization is in "id_continue*">
```

- Lu - uppercase letters
- Ll - lowercase letters
- Lt - titlecase letters
- Lm - modifier letters
- Lo - other letters
- Nl - letter numbers
- Mn - nonspacing marks
- Mc - spacing combining marks
- Nd - decimal numbers
- Pc - connector punctuations

2.1.1.5 Examples in Python

```
this_is_my_identifier = 3
_another_identifier = 4
```

```
  This_is_my_identifier = 5
  other_4567890_identifier = 6
  id = 7 #id is a reserved word, and can lead to mi
sleading behaviors
  print(this_is_my_identifier)
  print(_another_identifier)
  print(This_is_my_identifier)
  print(other_4567890_identifier)
  print(id)
```

2.1.2 Constants

2.1.2.1 Problem statement

In many cases, we find domains in which some values are always the same. They will NOT change during the execution of a program. For instance, the value of PI, the number of weekdays, the number of months, a set of colors, a threshold value for a specific problem, etc.

Furthermore, these values are commonly used in different parts of a program, so we need a strategy to be able to update them without visiting all the statements in which they are being used.

2.1.2.2 Concept

To ease the management of constant values and to provide a strategy to make our source code more generic and cleaner, we can identify these values and name them with an identifier.

A constant is an identifier to access a memory area (reference) in which we will store a value that will NOT change during the execution of the program.

2.1.2.3 Application

The use of constants is clear when we use the same value along the code. In these cases, as a good programming practice, we should declare a constant (an identifier) and assign that value. Then, we can refer the constant name to access the value in the memory.

2.1.2.4 Constants in the Python programming language

In Python, we do not really have a "constant" as a concept. We have variables that are declared in a **global scope**. Following some Python-specific features are presented:

- A constant is an **identifier** (name) following the rules to write identifiers in the Python programming language[4].
- A constant has a **datatype** that indicates the size of the content in the memory and how to interpret the information stored in the memory.
- Although it is not mandatory, constants identifiers are usually written in **UPPERCASE** or **UPPER_CASE_WITH_UNDERSCORES** letters.
- In Python, there is no special management of constants. Constants are just variables that are **managed at a global scope**.
- As a good practice, a constant name should be **representative and explanatory** avoiding names such as "temp", "other", etc.

2.1.2.5 Examples in Python

```
#Define a constant

#Note that there is no indentation.

NUMBER_OF_DAYS = 7

print(NUMBER_OF_DAYS)
```

2.1.3 Variables

2.1.3.1 Problem statement

A program implements some algorithm taking as an input some data and generating some result. During this computation process, there is a need of storing values that can be reused for further computations where they will be updated.

[4] https://docs.python.org/3/reference/lexical_analysis.html#identifiers

2.1.3.2 Concept

To ease the management of variables values and to provide a strategy to store and retrieve them from the memory, we can identify and name them with an identifier.

A variable is an identifier to access a memory area (reference) in which we will store a value that can change during the execution of the program.

2.1.3.3 Application

The use of variables is clear when we need to store and retrieve a value.

2.1.3.4 Variables in the Python programming language

In Python, we have the notion of a variable as an identifier that can be declared at anytime, anywhere at a **local scope**. Following some Python-specific features are presented:

- A variable is an **identifier** (name) following the rules to write identifiers in the Python programming language.
- A variable has a **datatype** that indicates the size of the content in the memory and how to interpret the information stored in the memory.
- In Python, variables are **managed at a local scope**.
- As a good practice, a variable name should be **representative and explanatory** avoiding names such as "temp", "other", etc.
- In Python, any variable is **a reference to an object in the memory**.

2.1.3.5 Examples in Python

```
  my_integer_variable = 2
  print(my_integer_variable)
  #Reference to the memory: identity of the variabl
e
  print(id(my_integer_variable))
  my_integer_variable = 5
  print(id(my_integer_variable))
```

2.1.4 Datatypes

2.1.4.1 Problem statement

When using a constant or a variable, we need to know which type of data we are managing because of two main reasons:

- To exactly know what the size is required to store the value in the memory (and being able to recover from it).
- To exactly know which are the operations that can be made with that type of content.

2.1.4.2 Concept

A datatype indicates the type of content that is stored in some variable and, by extension, the size that must be allocated in the memory and the type and semantics of the operations that we can make with those values.

2.1.4.3 Application

When coding we will make use of the different statements to build our program.

2.1.4.4 Datatypes in the Python programming language

In Python, we have different datatypes:

- There are **simple datatypes** like integer, float or boolean.
- There are **complex datatypes** (objects) such as string, date or any other class.
- There are **user-defined datatypes** that are specific datatypes defined by the programmer using a combination of existing datatypes.
- Datatypes allow us to ensure that specific operations can be done with constants/variables.
- Variables can be **mutable** (the value that is referenced in the memory can change) or **immutable** (if a new value is assigned, a new memory area is allocated).
- In Python, any constant and/or variable has a type that corresponds to the current value that is pointed by the identifier (**dynamic typing**).

Following, we present a summary of the official hierarchy of datatypes[5] in Python (directly taken from the official reference):

- **None**. This type has a single value. There is a single object with this value. This object is accessed through the built-in name ***None***. It is used to signify the absence of a value in many situations. Its truth value is false.
- **Number**. These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. **Numeric objects are immutable; once created their value never changes.** Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.
    - Integers (int)
    - Booleans (bool)
    - Float (float)
    - Complex (complex)
- **Sequences**
    - **Immutable sequences**. An object of an immutable sequence type cannot change once it is created.
        - **Strings**. A string is a sequence of values that represent Unicode code points. All the code points in the range U+0000 - U+10FFFF can be represented in a string. Python doesn't have a char type; instead, every code point in the string is represented as a string object with length 1.
        - **Tuples**. The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions.
    - **Mutable sequences**. Mutable sequences can be changed after they are created.

[5] https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy

    - **Lists**. The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets.
    - **Set types**. These represent unordered, finite sets of unique, immutable objects.
    - **Mappings**. These represent finite sets of objects indexed by arbitrary index sets. The subscript notation a[k] selects the item indexed by k from the mapping a;
    - **Dictionaries**. These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key's hash value to remain constant.

- …
- **Modules**. Modules are a basic organizational unit of Python code and are created by the import system as invoked either by the import statement.

2.1.4.5 Examples in Python

```
a = 2
#A datatype is integer
print(type(a))
#...now we assign another value, with type string
a = "hello"
print(type(a))
```

2.1.5 Statements

2.1.5.1 Problem statement

A program is a set of statements. A statement represents a step to accomplish with some task or to reach some objective.

2.1.5.2 Concept

A statement is a grammatically correct sentence. There are different types of statements, but we can summarize them in the following list:

- ***Constant/variable/attribute*** declaration. It is an statement in which a new value is assigned to a variable.
- ***Assignments***. It is an statement in which a value is assigned to an identifier (variable, constant).
- ***Expressions***. It is a statement that performs some calculation with operators and operands. There are different types of expressions:
    - Arithmetical
    - Comparison
    - Logical
- ***Control flow statements***. There are two main types:
    - Conditional (if-else) statements.
    - Loops (for, while) statements.
    - Unconditional jumps (go to, pass, break) statements.
- ***Function declaration.***
- ***Function invocation.***
- ***Class declaration.***
- ***Method declaration.***
- ***Method invocation.***
- ...

2.1.5.3 Application

When coding we will make use of the different statements to build our program.

2.1.5.4 Statements in the Python programming language

In Python, as in any other programming language, a statement is a grammatically correct sentence.

2.1.6 Expressions

2.1.6.1 Problem statement

When building a program, we need to perform operations that will return some value.

2.1.6.2 Concept

As we have introduced, an expression is a type of statement to define an operation with **operators** and operands. There are different types of expressions:

- Arithmetical
- Comparison
- Logical

Operators. An operator is a symbol that serves us to build expressions combining different operands.

2.1.6.3 Application

In order to perform some operation, we will design expressions combining operators and operands.

2.1.6.4 Expressions in the Python programming language

Arithmetic operators. These are operators that perform some arithmetical operation returning a number.

Table 12 Arithmetic operators.

Operator	Type	Interpretation	Example
+	Unary	Only to complement the unary negation	+a
-	Unary	Unary negation	-a
+	Binary	Adds two expressions (e.g. two variables)	a + b
-	Binary	Subtracts two expressions (e.g. two variables)	a - b
*	Binary	Multiplies two expressions (e.g. two variables)	a * b
/	Binary	Divides (float value) two expressions (e.g. two variables)	a / b
//	Binary	Divides (integer two expressions (e.g. two variables)	a // b

%	Binary	Returns the remainder of two expressions (e.g. two variables)	a % b
**	Binary	Raise an expression to an exponent (e.g. two variables)	a ** b

Comparison operators. These are operators that perform a comparison returning a boolean value (True or False).

Table 13 Comparison operators.

Operator	Type	Interpretation	Example
==	Binary	True if the two expressions are **equal** (by value), False otherwise	a == b
!=	Binary	True if the two expressions are **NOT equal** (by value), False otherwise	a != b
>	Binary	True if one value is greater than the other (by value), False otherwise	a > b
<	Binary	True if one value is less than the other (by value), False otherwise	a < b
>=	Binary	True if one value is greater or equal than the other (by value), False otherwise	a >= b
<=	Binary	True if one value is less or equal than the other (by value), False otherwise	a <= b

It is possible to evaluate some expressions that are not boolean, as boolean values. Python has a perfectly defined strategy to evaluate as **FALSE** the following situations.

- The value False.
- Any numerical value that is zero (0, 0.0, 0.0+0.0j).
- An empty string.
- An instance of built-in composite datatype (such as list) which is empty.
- The value None.

In the case of comparing floating numbers, we need to be especially careful because of the representation error.

```
   r = 2.1 + 3.2
print(r == 5.3)
# Another way to compare float values is to consider som
```

```
e tolerance
print(abs(r - 5.3) < 0.0000001)
```

Logical operators. These operators serve us to compose logical expressions (make conditions). It is important to know the truth tables of each operator.

Table 14 Logical operators.

Operator	Type	Interpretation	Example
and	Binary	Logical AND, true if both are true, false otherwise.	a and b
or	Binary	Logical OR, true if any is true, false otherwise	a or b
not	Unary	Logical NOT, it negates the current logical value.	not a

The evaluation of a logical AND is short-circuited (once an operand is false, the interpreter will NOT continue evaluating the rest of operands.

Other interesting Python feature are **chained comparisons** in which we can express natural comparisons that are more complex.

```
   a = 2
b = 3
c = 8
a < b <= c
# This is similar to
a < b and b <= c
```

Bitwise operators. They manage operands as sequences of binary digits operating them bit by bit.

Table 15 Bitwise operators.

Operator	Type	Interpretation	Example
&	Binary	Each bit position is the AND operation between the bits at that position	a & b
|	Binary	Each bit position is the OR operation between the bits at that position	a
~	Unary	Bit negation	~ a
^	Binary	Each bit position is the XOR operation between the bits at that position	a ^ b
>>	Binary	Each bit is shifted right n places	a >> n

<<	Binary	Each bit is shifted left n places	a >> n

Identity Operators. is and **id** that determine whether the given operands have the same identity, they refer to the same object. This is not the same thing as equality, which means the two operands refer to objects that contain the same data (values) but are not necessarily the same object.

Operator precedence. When performing operations on expressions with operands and operators, it is necessary to know the operator precedence to properly calculate the value. The precedence and order of evaluation in **Python is similar to other languages: from the highest to lowest precedence and from the left to the right**. The precedence is something we can establish using parenthesis.

Table 16 Operator precedence.

Operator	Priority (highest)
**	exponentiation
+a, -b, ~b	unary operations
*, /, //, %	multiplication and division
+, -	addition and subtraction
+, -	addition and subtraction
<<, >>	bit shifts
&	bit and
^	bit xor
|	bit or
==, !=, <, <=, >, >=, is, is not	comparison
not	logical not
and	logical and
or	logical or

2.1.6.5 Examples in Python

See Lab-2-Programming elements notebook for the complete list of examples.

2.1.7 Other functional programming elements

Keywords. The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

Table 17 Other Python Keywords.

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

Built-in-functions. The Python interpreter has several functions and types built into it that are always available[6].

Delimiters. These are symbols/tokens that serve as delimiters in the grammar.

Table 18 Delimiters.

`(`	`)`	`[`	`]`	`{`	`}`	
`,`	`:`	`.`	`;`	`@`	`=`	`->`
`+=`	`-=`	`*=`	`/=`	`//=`	`%=`	`@=`
`&=`	`|=`	`^=`	`>>=`	`<<=`	`**=`	

2.1.8 Other non-functional programming elements

Comments. A comment starts with a hash character (#) that is not part of a string literal and ends at the end of the physical line.

- *Block comments*. Block comments generally apply to some (or all) code that follows them and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment). Paragraphs inside a block comment are separated by a line containing a single #.

- *Inline comments*. An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space. Inline comments are unnecessary and in fact distracting if they state the obvious.

Documentation. To write with documentation strings (a.k.a. "docstrings"), you may visit PEP 257.

[6] https://docs.python.org/3/library/functions.html

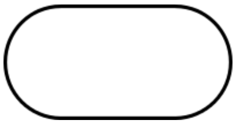
2.2 Representing a program: flow diagram

As it has been defined, a program is a set of statements that perform some operation. Basically, the set of statements represents a step-by-step method to solve a problem, a process.

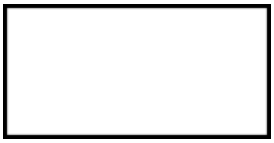
To represent a process, we have technique: the flow diagram. A flow diagram is just a chart in which we model which are the steps to complete a process.

In flowchart, we will find the next types of entities:

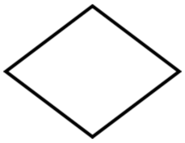
- **A rounded box:** representing the beginning or the ending of a program.



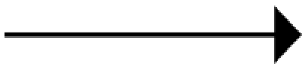
- **A square box:** representing a set of statements.



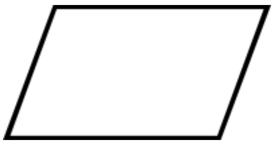
- **A decision point:** representing the evaluation of some condition.



- **An arrow:** representing the flow to one entity to another.



- **A parallelogram:** representing an input/output of the program.



There are other specific symbols that can be included in the flow diagram. However, in our case, this notation is more than enough to represent the flow of our programs.

As a good practice, before coding and in the design phase, we can use a flow diagram to design and describe the sequence of steps and tasks in our program.

Example

In the following flowchart, we design a program that asks the user to input some value and, then, we check whether the value multiplied by 3 is odd or even displaying a message.

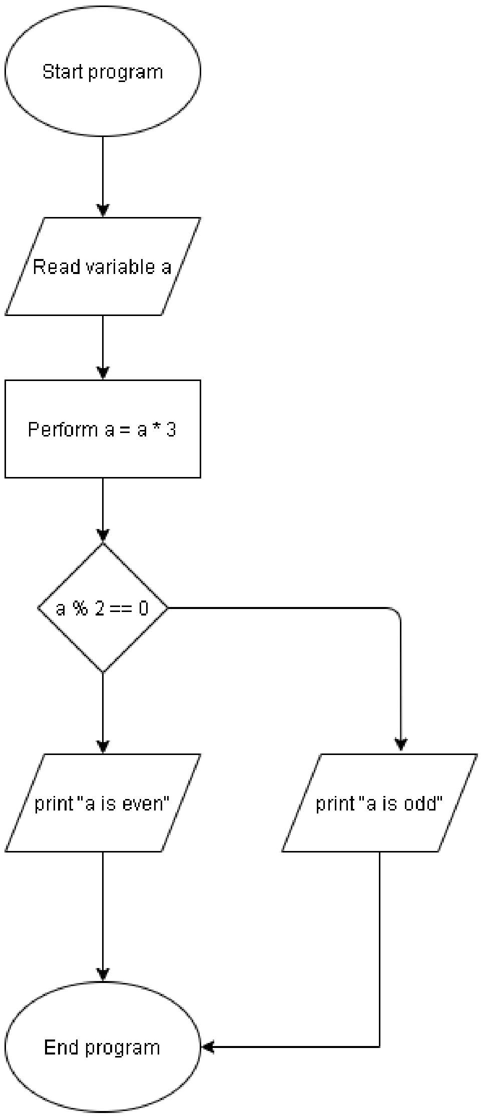


Figure 6 Example of flow diagram.

2.3 General structure of a program

2.3.1 The entry point: the `main` function

Any program has an **entry point**. This means the first instruction that will be executed by the interpreter.

In Python, this is the `main` function. To define, the main function, we declare the next statement:

```
# Other elements some imports and functions

if __name__=="__main__"_
  #sententences
```

In general, a Python program will have the next structure:

```

# Documentation

# import sections: see the next section

# function definitions

# main function

if __name__=="__main__"_
  #sententences

```

It is important to remark that the evaluation of a program in Python is "lazy", so until we do not go through some sentence, we cannot know whether it will behave properly.

2.4 Interesting links

ISO 5807:1985.Information processing — Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts

2.5 Quiz

1. **What is a variable?**
    a. It is an statement that performs some calculation with operators and operands.
    b. It is an identifier to access a memory area (reference) in which we will store a value that will change during the execution of the program.
    c. It is a literal value enclosed between " ".
    d. It is the type of content that is stored in the memory of a computer.

Answer: b

2. **What is the result of the next expression in Python 2*2**2+1?**
    a. a. 64
    b. b. 9
    c. c. 17
    d. d. 16

Answer: b

3. **Given a = 2, and b =5, What is the result of the following expression?**

    (a > 0) and (b <= 3) and (a > b%2)
    a. Runtime error
    b. Compiling error
    c. False
    d. True

Answer: c

4. **What is the result of evaluating the following expression 2 == "2"?**
    a. True
    b. Runtime error
    c. False
    d. Compilation error

Answer: c

5. **What is a Python built-in-function?**
    a. It is a function that is part of the Python library and available for all Python interpreters.
    b. It is a specific variable that we must import to be able to use it.
    c. It is a variable that is part of the Python library and available for all Python interpreters.
    d. It is a specific function that we must import to be able to use it.

Answer: a

6. **Given the next expression, which would be the result?**

```
3 // 2
```

    a. 1.5
    b. Runtime error
    c. 1
    d. Compilation error

Answer: c

3 CONTROL FLOW: CONDITIONAL STATEMENTS AND LOOPS

3.1 Control flow in a program

So far, we have seen that a program is a set of sequential steps (statements). In general, we have made some simple programs that get some input, perform some operation and display the result. However, the execution of a program is not always so easy and ideal. In many cases, it is necessary to check conditions and make decisions depending on the result. For instance, we may need:

- Ensure that some input value is in a proper range.
- After some computation, check what is current value of some variable and make a decision.
- Repeat set of statements until some condition is True (or False).
- ...

That is why, any programming language includes constructors to control the program flow. Basically, it implies that we are able to perform "jumps" in our program from one line to another depending on some condition. In this sense, we can classify these jumps into two main categories:

- **Conditional jumps**. In this case, and based on checking some condition, the program continues in some other statement.
- **Unconditional jumps**. In this second case, the program "jumps" unconditionally to some line without checking any condition. These jumps are usually known as "go-to". **They are completely not recommended**.

Let's put a simple example:

Write a program that asks for an integer number, checks whether is an even number and displays two types of message: "It is an even number" (if positive) or "It is an odd number" (if negative).

How can we proceed?

Use of conditional statements.

3.2 Conditional statements

A conditional statement is a kind of statement to control the flow of a program depending whether a condition is True or False.

We have different types of conditional statements (depending in how many branches are created when evaluating a condition):

- **Simple IF statement**. It checks a condition and, if it is True, do some actions, otherwise, the program will continue normally.
- **IF-ELSE statement**. It checks a condition and, if it is True, do some actions, otherwise, do other actions and, then the program will continue normally.
- **IF-ELSE multiple statement**. It checks a condition and, if it is True, do some actions, otherwise, check the rest of conditions until one of them is true, then the program will continue normally. In other programming languages, this type of conditional statement is also known as "switch".

3.2.1 Simple IF statement

3.2.1.1 Problem statement

There are many times in which we have to check some condition, and, in case, the condition is evaluated as True, a set of statements will be executed, and, if it is False, another set of statements will be then executed. For instance:

if the grade is greater or equal than 5, the course is passed.

3.2.1.2 Concept

The IF statement is a compound statement in which a condition is checked, and the control flow is divided into two branches:

- True branch: set of statements to be executed if the condition was True.
- False branch: set of statements to be executed if the condition was False.

Flow diagram:

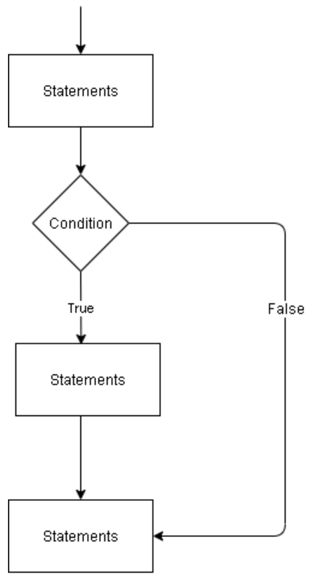


Figure 7 Simple IF statement flow diagram.

3.2.1.3 Application

The main application of the simple conditional statement is for checking conditions.

3.2.1.4 IF-ELSE statement in the Python programming language

In the Python programming language, the if-else statement is classified as a compound statement (because after its execution a new set of statements will be executed, or, in other words, a new indented block will follow to some of the branches). The grammar is as follows (in this case we will focus in the first case):

```
if_stmt ::=  "if" expression ":" suite
             ("elif" expression ":" suite)*
             ["else" ":" suite]
```

- expression is a conditional expression (logical operators and operands) that will be evaluated as True or False.
- suite is a set of **indented** statements.
- Although it is not necessary, sometimes it is good to enclose the expression between parenthesis for a better source code readability.

Grammar meaning,

- ()* means between 0-n repetitions of the statement (optional).
- [] means between 0-1 repetitions of the statement (optional).

3.2.1.5 Examples

```
grade = 6

if grade >= 5:
  print("You have passed the course...")
```

```
a = 4
b = 2

if a > b and a % 2 ==0:
  print("a is greater than b and even...")
```

```
a = 3
b = 2

if (a > b) and (a % 2 == 1):
  print("a is greater than b and odd...")
  print("...other statement in the if block...")

print ("Other statement...")
```

```
#Nesting if simple statements
a = 3
b = 2

if (a > b):
  print("a is greater than b...")
  if (a % 2 == 1):
    print("...and a is also odd...")
  print("...other statement in the suite of the first if...")
print("...main execution suite...")
```

3.2.2 IF-ELSE statement

3.2.2.1 Problem statement

As in the first case, there are many times in which we have to check some condition and, in case, the condition is evaluated as True, a set of statements will be executed, and, if it is False, another set of statements will be then executed. For instance:

If the grade is greater or equal than 5, the course is passed, otherwise the course is not passed.

3.2.2.2 Concept

The if-else statement is programming language constructor to give support to the creation of two execution branches:

- True branch: set of statements to be executed if the condition was True.
- False branch: set of statements to be executed if the condition was False.

Flow diagram:

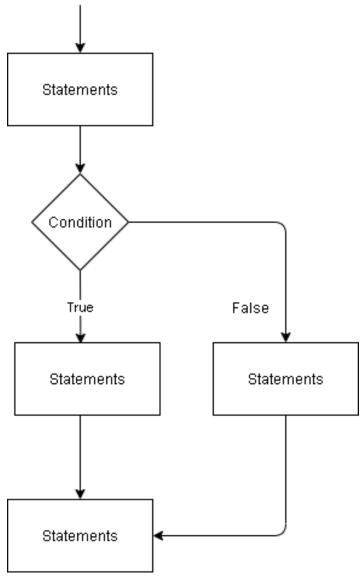


Figure 8 Simple IF-ELSE statement flow diagram.

3.2.2.3 Application

The main application of the if-else conditional statement is for checking conditions and provide two execution branches.

3.2.2.4 IF-ELSE statement in the Python programming language

The grammar is as follows (in this case we will focus in the third case):

```
  if_stmt ::=  "if" expression ":" suite
               ("elif" expression ":" suite)*
               ["else" ":" suite]
```

Again, here it is important to remark that **the suite of statements that follow the else clause must be properly indented**.

3.2.2.5 Examples

```
grade = 6
 if grade >= 5:
  print("You have passed the course...")
else:
  print ("You have NOT passed the course...")
```

3.2.3 Multiple IF-ELSE statement

3.2.3.1 Problem statement

In this case, instead of evaluating just one condition, we have a case in which we must make some actions depending on more than one condition.

For instance:

if the day is 1, print "Monday", if the day is 2, print "Tuesday", etc.

3.2.3.2 Concept

The if-elif-else statement is programming language constructor to give support to the creation of *n* execution branches:

- True branch: set of statements to be executed if the condition was True.
- False branch: set of statements to be executed if the condition was False.

Flow diagram, see Figure 9.

3.2.3.3 Application

The main application of the `if-elif-else` conditional statement is for checking multiple cases and provide more than two execution branches.

3.2.3.4 IF-ELIF-ELSE statement in the Python programming language

The grammar is as follows (in this case we will focus in the second case):

```
if_stmt ::=  "if" expression ":" suite
             ("elif" expression ":" suite)*
             ["else" ":" suite]
```

Again, here it is important to remark that **the suite of statements that follow the else clause must be properly indented**.

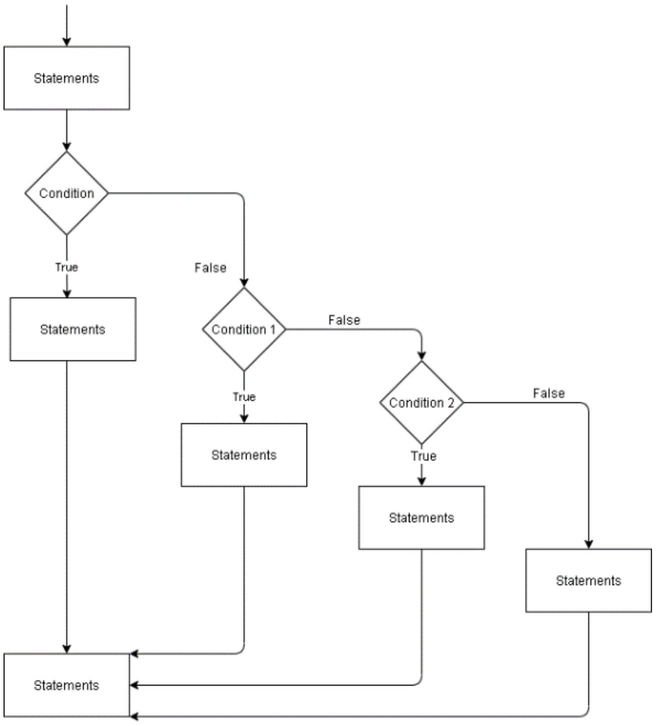


Figure 9 Multiple IF-ELSE statement flow diagram.

3.2.3.5 Examples

```
day = 2


if day == 1:
  print ("Monday")
elif day == 2:
  print ("Tuesday")
elif day == 3:
  print ("Wednesday")
  elif day == 4:
    print ("Thursday")
  elif day == 5:
    print ("Friday")
  elif day == 6:
    print ("Saturday")
  elif day == 7:
    print ("Sunday")
  else:
    print("That number of day has not a name...")


  #Sometimes this it not an elegant way of writing
Python code...but it is just an explanatory example
.
```

3.2.4 Other types of conditional statements and examples

In many programming languages, and in Python as well, there are some simplified forms of the `if-else` statements. Although, they are grammatically correct, they also make the code less readable, so, I personally advise not to use them unless it is completely necessary.

Saving code lines is not an objective and to write this type of statements does not make your code more efficient or "pythonic".

A good use case of this type of statements is debugging.

- **In-line simple if statement.**

```
   "if" expression ":" suite
 "if" expression: suite 1; suite 2; ...; suite n
```

```
  a = 2
  b = 3
  if a < b: print("This is..."); print(a)
```

- **Conditional expression.** This type of statement was introduced by the Python creator, Guido, in the PEP 308. It is also known as "ternary operator".

   - These expressions are a good manner of assigning a value to a variable under certain conditions, like a lambda function.

```
expression if expression else expression
```

```
age = 20
category = "minor" if age < 18 else "adult"
print(category)
```

- **pass statement**. This is a null statement; it does not make anything. Usually, we write pass when we want to allocate some code, but we do not know yet which would be.

```
a = 2
if a % 2 == 0:
  pass #Source code is not decided yet...
else:
  print("This is not True...")
```

3.3 Loops

As we have introduced in the previous chapter, we have made some simple programs that get some input, ***check some condition***, perform some operation and display the result. However, the execution of a program is not always so easy and ideal. In many cases, it is necessary to repeat a set of actions and make decisions depending on the result. For instance, we may need:

- Aggregate values.
- Search for something.
- Show a set of elements.
- Process a set of elements.
- ...

Again, any programming language includes constructors to control the program flow. Basically, it implies that we are able to perform "jumps" in our program from one line to another depending on some condition.

Let's put a simple example:

Write a program that calculates the average grade of the students.

- How can we proceed?

Use of loop statements.

A loop statement is a kind of statement to control the flow of a program and repeat a set of statements until a condition is True or False, or, until a set of elements have been visited.

We have different types of loop statements:

- **While statement**. It firstly checks a **stop condition**, and, if it is True, a set of statements are executed. Then, the stop condition is checked again. This process is repeated until the condition is False. This loop can potentially be infinite (due to bugs, errors or other coding issues).
- **For statement**. It iterates over a set of elements and, once, all the elements have been visited the loop ends. However, there are different types of **for** loops:
  - For statement (fully equivalent to the `while` statement). In this case, the for loop has another syntax but, the semantics and execution, is completely equivalent to a `while` loop. This type of loop is NOT available in the Python programming language. Take a look to the next loop in the Java programming language.

```
  for (int i = 0; i<5; i++)
System.out.println(i)
```

  - For statement to iterate over an *iterable* sequence. In this case, the for statement allows us to iterate over a set of elements and, the loop will end, once all elements have been visited.

3.3.1 WHILE statement

3.3.1.1 While statement

There are many times in which we must repeat a set of actions until some condition is True or False. In the case of the while loop, the loop will be executed while the stop condition is True. For instance:

While it is not the end of the list of grades, get the grade at position i and aggregate.

3.3.1.2 Concept

The WHILE statement is a compound statement in which a condition is checked and, if it is True, a set of statements are executed. Then, the stop condition is checked again.

- True branch: set of statements to be executed if the condition was True.
- The while loop will be executed between [0−N] times.

Flow diagram:

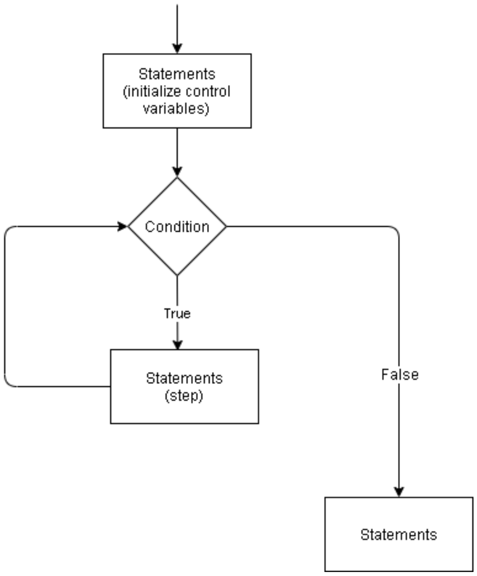


Figure 10 WHILE statement flow diagram.

- The while loop is composed of the next elements:
  - **Control variables**. These are the variables participating in the stop condition. **It is necessary to always initialize these variables before entering in the loop**.
  - **Stop condition**. This is a logical expression.
  - **Step**. This is a set of statements that updates the control variables in each iteration.

3.3.1.3 Application

The main application of the while loop is the repetition of a set of statements while the stop condition is True.

3.3.1.4 WHILE statement in the Python programming language

In the Python programming language, the while statement is classified as a compound statement (because after its execution a new set of statements will be executed, or, in other words, a new indented block will follow to some of the branches).

The grammar is as follows (in this case we will focus in the first case):

```
   while_stmt ::=  "while" assignment_expression ":" suit
e
                ["else" ":" suite]
```

- `assignment_expression` is a conditional expression (logical operators and operands) that will be evaluated as True or False.
- `suite` is a set of **indented** statements.

Following, the official definition is presented:

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates. A `break` statement executed in the first suite terminates the loop without executing the `else` clause's suite. A continue statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

Grammar meaning,

- ()* means between 0-n repetitions of the statement (optional).
- [] means between 0-1 repetitions of the statement (optional).

In addition, the while statement in the Python programming language allows us to include an `else` statement that will be executed after finishing the loop. **This statement will NOT be executed when the loop is terminated by a break statement**.

3.3.1.5 Examples

```
i = 0
while i < 5:
  print(i)
  i = i + 1


while i < 5:
  print(i)
  i = i + 1
else:
  print("Done")
```

3.3.2 FOR statement

3.3.2.1 Problem statement

In this case, we have a set of elements that we want to visit (iterate over) and perform some operation (usually, aggregation, searching, etc.).

For instance: *For each student, print the ID, name and grade.*

3.3.2.2 Concept

The `for` statement is programming language constructor to give support to the iteration over an iterable sequence of elements.

- Definitions:
  - **Iterable**: it is a collection of objects such as a list, a set or a tuple. If an object is "iterable", it means that can be used in an iteration process.
  - **Iterator**: if an object is iterable, then, an iterator is returned. An iterator is a pointer to the next element in the iterable object. The pointer is updated every time is invoked. Internally, there is a function `iter` to get the iterator of an iterable object. Then, the function `next()` will be invoked to the next element.

Flow diagram:

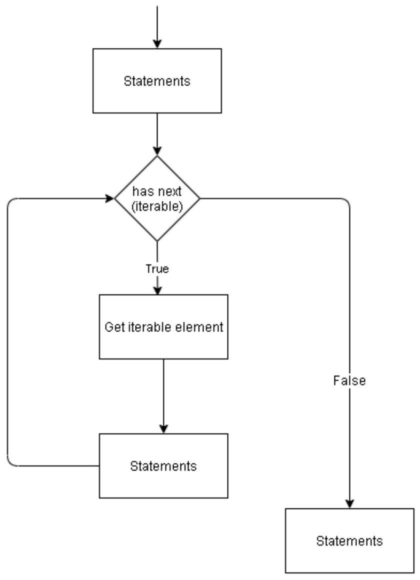


Figure 11 FOR *statement flow diagram.*

3.3.2.3 Application

The main application of the for-loop statement is for providing a common set of operations to a set of values. As it has been commented before, aggregation, searching for some value, replacing a value, etc.

3.3.2.4 FOR statement in the Python programming language

The grammar is as follows:

```
  for_stmt ::=  "for" target_list "in" expression_list "
:" suite
                ["else" ":" suite]
```

Again, here it is important to remark that **the suite of statements that follow the for clause must be properly indented**.

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments, and then the `suite` is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a StopIteration exception), the `suite` in the else clause, if present, is executed, and the loop terminates.

- In addition, the `for` statement in the Python programming language allows us to include an `else` statement that will be executed after finishing the loop. A `break` statement executed in the first suite terminates the loop without executing the else clause's suite. A `continue` statement executed in the first suite skips the rest of the suite and continues with the next item, or with the else clause if there is no next item.

Remark from the official documentation: there is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

```
for x in a[:]:
  if x < 0: a.remove(x)
```

3.3.2.5 Examples

```
for i in range(0,5):
   print (i)
```

3.3.3 Generating an iterable sequence: the range class

- `range(start, stop[, step])` (simplified definitions from the official documentation).
    - The arguments to the range constructor must be integers (either built-in int or any object that implements the `__index__` special method). If the step argument is omitted, it defaults to 1.
    - Ranges containing absolute values larger than sys.maxsize are permitted but some features (such as len()) may raise OverflowError.

  - Ranges implement all of the common sequence operations except concatenation and repetition (due to the fact that range objects can only represent sequences that follow a strict pattern and repetition and concatenation will usually violate that pattern).

- Parameters:
  - `start`: the value of the start parameter (or 0 if the parameter was not supplied).
  - `stop`: the value of the stop parameter.
  - `step`: the value of the step parameter (or 1 if the parameter was not supplied).

Remark: *The advantage of the range type over a regular list or tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).*

```
  #Examples of range


  # 10 numbers between 0 and 9: 0,1,2,3,4,5,6,7,8,9
  for i in range (0,10):
    print(str(i)+",",end="")
  print()


  #Even numbers between 0 and 10 (not included): 0,
2,4,6,8
  for i in range (0, 10, 2):
    print(str(i)+",",end="")
  print()
  #10 numbers between 10 and 0 (not included): 10,9
,8,7,6,5,4,3,2,1
  for i in range (10, 0, -1):
    print(str(i)+",",end="")
  print()
  #Ranges can be sliced
  # 5 numbers between 0 and 5 (not included): 0,1,2
,3,4,
  for i in range (0,10)[:5]:
    print(str(i)+",",end="")
  print()

  # we reverse here the range: 9,8,7,6,5,4,3,2,1,0,
  for i in range (0,10)[::-1]:
    print(str(i)+",",end="")
  print()
```

3.4 Other statements: `break` and `continue`

We have seen before that in Python we have two ways of stopping a loop:

- If it is a while loop through the stop condition (when it is false).
- If it is an iteration loop (for), the loop will end once the iterator has no elements to return.

However, it is also possible to modify the behavior of the loop which means stopping the loop and have a kind of internal control flow through the next two statements.

- **break statement**. The break statement, like in C, breaks out of the innermost enclosing for or while loop.
- **continue statement**. The continue statement, also borrowed from C, continues with the next iteration of the loop.

```
  #Prints only the first even number
  for n in range(1, 10):
    if n % 2 == 0:
      print(n)
      break
```

```
#Prints only the odd numbers
for n in range(1, 10):
  if n % 2 == 0:
    continue
  else:
    print(n)
```

3.5 Efficient creation of iterators

Learn more: https://docs.python.org/3/library/itertools.html

3.6 Interesting links

- https://docs.python.org/3/reference/compound_stmts.html
- https://www.python.org/dev/peps/pep-0308/

3.7 Quiz

1. **Is a conditional statement a compound statement?**
   a. Yes, a conditional statement can comprise more than one condition.
   b. Yes, a conditional statement can comprise several elif statements.
   c. Yes, after the condition one single statement can be written.
   d. Yes, after the condition a block of statements can be written.

Answer: d

2. **Is it mandatory to include an else block after any if statement?**
   a. Yes, if the expression to be evaluated is multiple.
   b. No, only if there is at least one elif block.
   c. No, it is not mandatory.
   d. Yes, it is always mandatory.

Answer: c

3. **Given the next source code snippet, which would be the result?**

```
a = 2
b = 3
if b > a:
    if  b % 2 == 0:
        print("b")
    elif a % 2 != 0:
        print("a")
    else:
        print("ba")
```

   a. b
   b. ba
   c. Compilation error
   d. Runtime error

Answer: b

4. **Given the next configuration of the function range: range (-1, 10, 2), Which would be the result of iterating over that range and print each value?**
   a. 1, 3, 5, 7, 9, 11
   b. 1, 3, 5, 7, 9, 10
   c. -1, 1, 3, 5, 7, 9
   d. -1, 1, 3, 5, 7, 9, 10

Answer: c

5. **Which is the result of the next loop?**

```
i = 0

while i < 10:

    if i % 2 == 0:

        print(i)

    else:

        continue

    i = i + 1
```

a) 1 and infinite loop

b) Compilation error

c) 0 and infinite loop

d) 0, 2, 4, 6, 8

Answer: c

6. What is the meaning of the sentence break?

a. It completely breaks the execution of a loop.

b. It evaluates the stop condition of a loop.

c. It breaks the current iteration of a loop and starts again from the first loop statement.

d. It checks a condition within a loop.

Answer: a

4 DATA STRUCTURES

4.1 Introduction

In this chapter, we introduce the use of data structures. More specifically, a detailed explanation of the selection, use and implementation of arrays and strings is presented.

So far, we have designed and implemented programs that take some simple input, perform some operations and return some value. To do so, we were declaring some variables (2-3?) and work with them. However, we should ask ourselves what happened when we have a larger input.

Let's put a simple example:

Write a program that prints the name and grade of all students.

How can we proceed?

- Shall we create 72 variables for storing names and grades?
- How can we perform operations on the variables like calculating the average grade or count the number of grades >k?
- How can we scale the program to support the whole set of university students?

To properly manage the data is used by a program (as input, as intermediary results or as output), we have to organize and structure data to be able to easily process all data items used by program according to a set of requirements (the selection of a proper data structure depends on many factors that we will summarize later on this chapter).

Selection and use of a proper data structure.

4.2 Data structures: context and definitions

First, it is important to recall the notion of a program. When we develop a system, we are going to implement some algorithms that will take some input (data), perform some operations and produce some output.

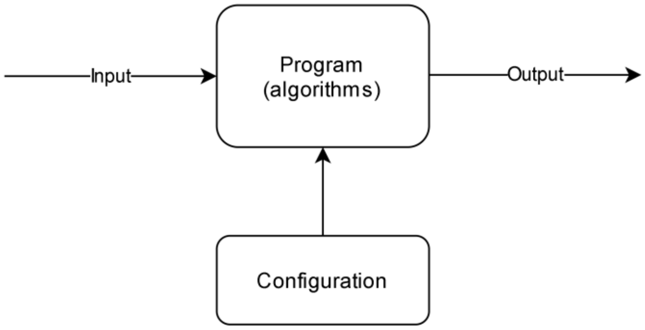


Figure 12 Basic structure of a software program.

In order to organize, manage and store, the input data, the intermediary results and the results, we have to first think in needs related to the management of data. To do so, a proper identification of the type of data we have and the type of operations we need to perform is critical to provide a good implementation of the problem we are solving.

A data structure is the way we organize, structure and store data within a program.

The next figure shows the main relationships between data, data structure, data type and variable. In general, we have **data** (e.g. a list of grades), that, depending on the problem, can be conceptualized in some **data structure** (e.g. a vector of numbers) and, then, we can define a **specific data type** to implement that data structure (e.g. a list of numbers). Finally, we can create **variables** of that new data type.

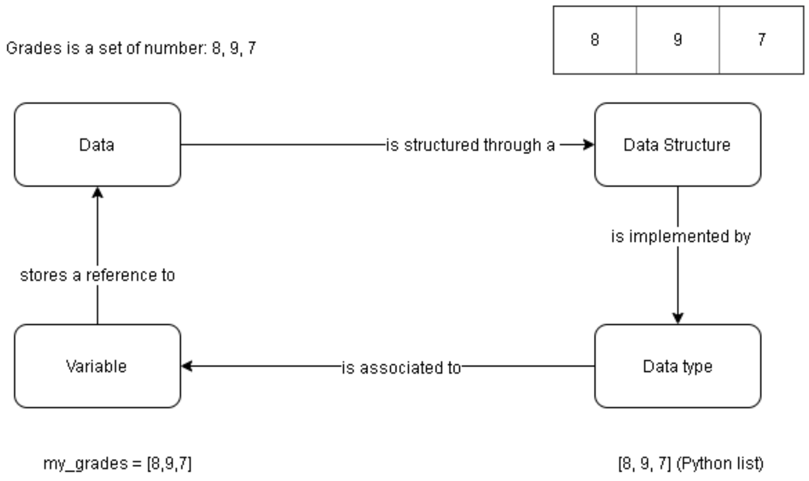


Figure 13 Data, Data structure, Datatype and variable relationships.

Conceptually speaking, we should apply the next steps to identify a proper data structure:

1. Identify the type of data. E.g. a sequence of numbers, records, etc.
2. Identify the structure to organize data. E.g. a vector, a set, etc.
3. Identify the target operations to perform. E.g. search, access element by element, etc.
4. Study the cost (and scalability) of the target operations in the different data structures. Usually, this evaluation requires knowledge about the different data structures and their spatial and temporal complexity for each of the target operations.
5. Select the most efficient data structure.

In our case, we will focus in the first three steps trying to directly map our necessities to a set of predefined data structures.

4.2.1 Common data structures

There are a set of well-known data structures that every programmer should know:

- Vector (array). It is used to represent a finite set of elements of the same type. A vector can have more than 1 dimension. A 2-d dimension vector is a matrix or a table.

- List. It is used to represent an infinite collection of elements.
- Set. It is used to represent an infinite set of elements.
- Dictionary. It is used to represent elements indexed by some field.
- Tree. It is used to represent an hierarchy of elements in which there is one root node and each node has only one parent node.
- Graph. It is used to represent relationships between elements. It is a generalization of a tree.

Then, these data structures can be implemented by a combination of specific data types. For instance, a dictionary can be implemented through a hash table that internally uses a linked list.

4.3 Objects and references

In the first chapter of this course, we saw that programming languages can be classified depending on the type of programming paradigm (e.g. functional, object-oriented, etc.).

The Object-Oriented Programming (OOP) is a paradigm in which we represent the data and operations of our problem and solution making an exercise of abstraction by defining classes including attributes and operations (methods). Complex datatypes and user-defined datatypes are usually defined using objects. This means we define a class with the required attributes and operations to manage the entities of our domain. Following, the main definitions for the OOP paradigm are presented:

What is a class**?**

A class is an abstraction of a (real/virtual) entity defined by a set of attributes (data/features) and a set of capabilities/functionalities/operations (methods).

As an example, let suppose we have to store the information about a person (id and name) and provide some capabilities (speaking and running). We can define a class Person with these attributes and capabilities.

```
class Person:
  id = ""
  name = ""

  speaking():
    #do speaking

  running(velocity):
    #do running
```

A class defines a category of objects and contains all common attributes and operations.

What is an object**?**

An object is an instance, a realization, of a class. It is the realization of a class with specific values for each attribute and a shared behavior.

Following with the example, we can now define an instance of a Person, John, with some id (1).

What is an attribute**?**

An attribute is a feature/characteristic/property shared by a set of objects.

In the Python programming language, an attribute can be accessed using the next syntax:

```
instance_name.attribute
```

What is a method**?**

A method is a capability/operation/functionality/behavior shared by a set of objects.

In the Python programming language, a method can be invoked using the next syntax:

```
instance_name.method_name(parameters)
```

In our context, we only need to know these basic definitions since our data structures in Python will be implemented through classes (like the class list) and, by extension, we need to know how to invoke a method, how to access an attribute, etc.

Finally, there are 4 principles of the OOP that are relevant and enumerated below:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

4.4 The array (vector) data structure

4.4.1 Problem statement

There are many problems in which we must manage a collection of elements of the same type. For instance, if we must store and perform some operation on a set of grades. How can we do it? Declaring n variables? How can we ensure an unified management of all data items?

In general, the issues we have to tackle are:

- Organize and structure a finite collection of data items
- Same type of data items
- Provide operations that must work with all the data items

4.4.2 Concept

According to these needs, we have a conceptual data structure that is the **vector** (or array). A vector is a data structure to organize, store and exploit a collection of data items. A vector has some characteristics:

- Finite collection of elements
- Same type of elements
- Fixed size

4.4.3 Application

When we have a collection of data items of the same type, we may want to perform some operations:

- Access an element (and all)
- Iterate over the collection of elements
- Search for an element
- Filter by a condition
- Sort the elements by some criteria
- Implement some aggregation operators: sum, min, max, count, size, etc.

4.4.4 Array data structure implementation in the Python programming language

In the Python programming language, there is no vector or array data structure as in other programming languages. So, to implement a conceptual vector, we can use the data type `list`.

The class list in Python: `class list([iterable]):`

- Lists may be constructed in several ways:
  - Using a pair of square brackets to denote the empty list: `[]`
  - Using square brackets, separating items with commas: `[a], [a, b, c]`
  - Using a list comprehension: `[x for x in iterable]`
  - Using the type constructor: `list()` or `list(iterable)`

The constructor builds a list whose items are the same and in the same order as iterable's items. iterable may be either a sequence, a container that supports iteration, or an iterator object. If iterable is already a list, a copy is made and returned, similar to `iterable[:]`.

- Mutability. A list in Python is mutable.
- Size. A list in Python is dynamic. Although, this is not what we expect from a vector, it is a feature that we may know when using the class list in Python.
- Types of the elements. A list in Python can contain elements of different types. As before, this is a feature that does not correspond to the conceptual view of a vector.
- Indexing and slicing. A list in Python can be accessed by position (index) or by slicing the list into a chunk.
  - An index is an integer expression. To access an element by an index, we must use the brackets: mylist[position]. It is also possible to access elements by using a negative index since there is a double indexing.

Positive index	**0**	**1**	**2**
List content	5	6	7
Negative index	**-3**	**-2**	**-1**

- An slice follows the same notation and has the same meaning as a range.
- Nested lists. A list in Python can contain elements that are lists. In this manner, we can implement n dimensional vectors.
- Operators. Some operators can be applied to lists
    - "+" which has the meaning of concatenation.

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

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

-
    - "*" which has the meaning of concatenating n times the list elements.

```
[1,3,4]*4

[1, 3, 4, 1, 3, 4, 1, 3, 4, 1, 3, 4]
```

4.4.5 Examples of the main functions and methods

Given a list L, some of the main methods to work with a list are presented below (→ return value):

- len: len(L) → number of elements of the list.
- append: L.append(object) → None -- append object to end.
- insert: L.insert(index, object) →insert object before index. Index is a position.
- index: L.index(value, [start, [stop]]) → integer -- return first index of value. Raises ValueError if the value is not present.
- count: L.count(value) → integer -- return number of occurrences of value.
- copy: L.copy()→ list -- a shallow copy of L. A new object is created.
- reverse: L.reverse() →reverse *IN PLACE*. "In place" means that the list is modified.
- sort: L.sort(key=None, reverse=False) → None -- stable sort *IN PLACE*

- remove: L.remove(value) → None -- remove first occurrence of value. Raises ValueError if the value is not present.
- clear: L.clear() → None -- remove all items from L.

Other interesting functions are `extend`, `push` and `pop` (operations for the management of a list with different input/output strategies).

```
#Creating a list
values = [1,2,3]
#Accessing elements

#Prints 1
print(values[0])

#Length
#Prints 3
print(len(values))

#Iterating

#By value
for v in values:
  print(v)
#By index
for i in range (len(values)):
  print(values[i])

#Adding an element
values.append(100)

#Creating a copy
other_values = values.copy()
```

```
#Counting elements
values.count(3)

#Reversing the list
values.reverse()

#Removing an element
values.remove(1)

#Removing all elements
values.clear()

#Slicing
#[start, stop, step]
```

```
#Listing list methods and get the method defin
ition
dir([])
help([].append)
```

4.5 The array of characters (string) data structure

4.5.1 Problem statement

Any exchange of information between the program and any other entity uses strings (text sequences) as units of information. When an user inputs something, before being interpreted (as a number for instance), the program and libraries receives a string.

When a program reads some input data from a file, service or database, the information is encoded as strings. When something is displayed on the screen, strings are used.

So, in general, everything is a string that can be then interpreted as anything else (like a number, a set of elements, etc.). That is why, we need means to easily manage sequences of characters.

4.5.2 Concept

According to these needs, we have a conceptual data structure that is the **string**. A string is a data structure to organize, store and exploit a collection of characters. Conceptually speaking, a string is a vector of characters, so it shares most of the vector characteristics.

- Finite collection of elements (characters)
- Same type of elements (character)
- Fixed size

4.5.3 Application

Since a string is a kind of vector, most of the applications are similar but focusing on the use of characters.

- Access an element (and all)
- Iterate over the collection of elements
- Search for a character or substring
- Filter by a condition
- Sort the strings by some criteria
- Implement some aggregation operators: sum, min, max, count, size, etc.

4.5.4 String data structure implementation in the Python programming language

Textual data in Python is handled with str objects, or strings. Strings are immutable sequences of Unicode code points. String literals are written in a variety of ways:

- Single quotes: 'allows embedded "double" quotes'
- Double quotes: "allows embedded 'single' quotes".
- Triple quoted: '''Three single quotes''', """Three double quotes"""

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

String literals that are part of a single expression and have only whitespace between them will be implicitly converted to a single string literal. That is, ("spam " "eggs") == "spam eggs".

The `class string` in python is initialized through the next method: `class str(object=b'', encoding='utf-8', errors='strict')`.

- Mutability. A string in Python is **immutable**.
- Size. A string in Python is not dynamic, since the addition of new characters implies the creation of a new instance. So, theoretical speaking this means that the same string cannot append new characters. However, from the developer point of view, it is possible to add characters to an existing string.
- Types of the elements. A string in Python can contain characters.
- Indexing and slicing. A string in Python can be accessed by position (index) or by slicing the string into a chunk.
    - An index is an integer expression. To access an element by an index, we must use the brackets: string[position]. It is also possible to access elements by using a negative index since there is a double indexing.

Positive index	**0**	**1**	**2**
String "Two"	T	w	o
Negative index	**-3**	**-2**	**-1**

    - An slice follows the same notation and has the same meaning as a range.

- Operators. Some operators can be applied to strings:
    - "+" which has the meaning of concatenation.

```
"Hello" + "World"

"HelloWorld"
```

- "*" which has the meaning of concatenating n times the string characters.

```
"Hello"*2

"HelloHello"
```

4.5.5 Examples of the main functions and methods

Given a string S, some of the main methods to work with a list are presented below (→ return value):

- len: len(S) → number of characters of the string.
- capitalize: S.capitalize() → str. Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.
- count: S.count(sub[, start[, end]]) → int. Return the number of non-overlapping occurrences of substring sub in string S[start:end].

Optional arguments start and end are interpreted as in slice notation.

- endswith: S.endswith(suffix[, start[, end]]) → bool. Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position.

suffix can also be a tuple of strings to try.

- find: S.find(sub[, start[, end]]) → int. Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

- format: S.format(*args, **kwargs) → str. Return a formatted version of S, using substitutions from args and kwargs.

The substitutions are identified by braces ('{' and '}').

- index: S.index(sub[, start[, end]]) → int. Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation. Raises ValueError when the substring is not found.

- isalnum: S.isalnum()→ bool. Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.
- isalpha: S.isalpha() → bool. Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.
- isdecimal: S.isdecimal() → bool. Return True if there are only decimal characters in S, False otherwise.

- isdigit: S.isdigit() → bool. Return True if all characters in S are digits and there is at least one character in S, False otherwise.
- isidentifier: S.isidentifier() → bool. Return True if S is a valid identifier according to the language definition.
- Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".
- islower: S.islower() → bool. Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.
- isnumeric: S.isnumeric() → bool. Return True if there are only numeric characters in S, False otherwise.
- isspace: S.isspace() → bool. Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.
- isupper: S.isupper() → bool. Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.
- join: S.join(iterable) → str. Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.
- lower: S.lower() → str. Return a copy of the string S converted to lowercase.
- replace: S.replace(old, new[, count]) → str. Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.
- split: S.split(sep=None, maxsplit=-1) → list of strings. Return a list of the words in S, using sep as the delimiter string.

If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.

- splitlines: S.splitlines([keepends]) → list of strings. Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
- startswith: S.startswith(prefix[, start[, end]]) → bool. Return True if S starts with the specified prefix, False otherwise. With optional start,

test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

- strip: S.strip([chars]) → str. Return a copy of the string S with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

- title: S.title() → str. Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.

There are other methods that are specific implementations of replace, index, find, etc.

```
#Some examples of string method invocation
name = "Mary"
print(len(name))

print(name.count("a"))

#Formatting
print("mary".capitalize())

#Is methods
print("123".isalnum())
print("A".isalpha())
print("1".isdigit())
print("def".isidentifier())
print(name.islower())
print(name.isupper())
print(" ".isspace())

print(" ".join(["Mary", "has", "20", "years"]))
#Checking values
print(name.startswith("M"))
print(name.endswith("ry"))

#Finding
print(name.find("r"))
print(name.index("r"))

#Replace
print(name.replace("M","T"))

#Splitting
print("Mary has 20 years".split(" "))

print("    Mary    ".strip())
```

```
#Listing string methods and get the method def
inition
dir("")
help("".strip)
```

4.6 Tuples

4.6.1 Problem statement

In the previous sections, the problem of unifying the management of a collection of items has been addressed through the creation of lists. A list serves us to conceptually implement a kind of vector. However, in the context of Python, the theoretical features of a vector such as ordered accessed, fixed size and same type of elements, are not directly provided by the type list. Furthermore, when using a list in Python we have to consider the mutability aspect.

According to this context, there are two main factors that can lead us to think in other type of data type to allocate and manage data items fulfilling the following requirements:

- Keep the order of elements

- Ensure a type that is immutable
- Ensure a structure that is fixed size

For instance, let's suppose we have to represent data about a person. A person has three fields: name, age and id. We know the number, type and order of fields will not change over time. So, how can we proceed? So far, we could think in a list associating each position to some field. However, this approach is quite weak. We cannot ensure the content of each field and we cannot access by using a name (just an index). Furthermore, new fields could be added changing the expected behavior of our data structure.

In other programming languages, it is possible to find the notion of "record" or "struct". This is a kind of user-defined datatype to represent and organize data of an entity, accessing fields by name. Close to this notion, we have the datatype Tuple.

4.6.2 Concept

In order to address these necessities, there is a type Tuple, available in many programming languages, that allows us to represent information as a finite sequence of fields that can be accessed in an ordered manner.

4.6.3 Application

When we have to represent a predefined set of fields keeping the order and access the elements by an index, we can think in using a Tuple. A tuple is commonly used when it is necessary to package the parameters of a function or to return several values from a function. Furthermore, a tuple is also a sequence of elements, so, most of the operations available for a list are still available:

- Access a field (and all)
- Iterate over the collection of fields
- Search for a field
- Filter by a condition
- Implement some aggregation operators: sum, min, max, count, size, etc.

4.6.4 Tuple implementation in the Python programming language

In the Python programming language, a built-in datatype tuple is available. According to the official documentation,

*Tuple*s are immutable sequences, typically used to store collections of heterogeneous data (such as the 2-*tuple*s produced by the *enumerate()* built-in). *Tuple*s are also used for cases where an immutable sequence of homogeneous data is needed (such as allowing storage in a *set* or *dict* instance).

The class tuple in Python: `class tuple([iterable])` may be constructed in several ways:

- Using a pair of parentheses to denote the empty tuple: `()`
- Using a trailing comma for a singleton tuple: `a,` or `(a,)`
- Separating items with commas: `a, b, c` or `(a, b, c)`
- Using the `tuple()` built-in: `tuple()` or `tuple(iterable)`

The constructor builds a tuple whose items are the same and in the same order as iterable's items. iterable may be either a sequence, a container that supports iteration, or an iterator object.

If iterable is already a tuple, it is returned unchanged. For example, tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3). If no argument is given, the constructor creates a new empty tuple, ().

Note that it is actually the comma which makes a tuple, not the parentheses. The parentheses are optional, except in the empty tuple case, or when they are needed to avoid syntactic ambiguity.

Furthermore, tuples implement all of the common sequence operations. On the other hand, there are situations in which we may want to access fields by name instead of using a number. To do so, we have the notion of ***namedtuple***.

As an important remark, tuples have the following characteristics:

- Mutability. A tuple in Python is **immutable**. This has several side-effects; **it is NOT possible to add new elements and to modify the type or value of a field**.
- Size. A tuple in Python is fixed-size.

- Types of the elements. A tuple in Python can contain elements of different types.
- Indexing and slicing. A tuple in Python can be accessed by position (index) or by slicing the tuple into a chunk.
    - An index is an integer expression. To access an element by an index, we must use the brackets: mytuple[position]. It is also possible to access elements by using a negative index since there is a double indexing.

Positive index	**0**	**1**	**2**
List content	5	6	7
Negative index	**-3**	**-2**	**-1**

    - A slice follows the same notation and has the same meaning as a range.

- Nested tuples. A tuple in Python can contain elements that are tuples.
- Operators. Some operators can be applied to tuples.
    - "+" which has the meaning of concatenation.

```
(1, 3, 4) + (2, 5)

(1, 3, 4, 2, 5)
```

    - "*" which has the meaning of concatenating n times the list elements.

```
(1,3,4)*4

(1, 3, 4, 1, 3, 4, 1, 3, 4, 1, 3, 4)
```

4.6.5 Examples of the main functions and methods

Given a tuple T, some of the main methods to work with a tuple are presented below (→ return value).

- len: len(T) → number of elements of the tuple.

- count: T.count(value) → integer -- return number of occurrences of value.
- index: T.index(value, [start, [stop]]) → integer -- return first index of value. Raises ValueError if the value is not present.

```
#Create
my_tuple = ("Jose", 8) #name and grade
print(my_tuple)

my_other_tuple = tuple(["Jose", 8])
print(my_other_tuple)
#Tuple -->Record | Card

#Access
#Sequence
for field in my_tuple:
  print(field)

print(len(my_tuple))
#Searching for a value within a tuple
print("Jose" in my_tuple)
print(my_tuple*3)# repeat 3 times the tuple
print(my_tuple + my_other_tuple) #concat

#Slice
print(my_tuple[::-1])
#Add this comma-->create a tuple of one element
t = (2,)
#syntax to de-ambiguate
print(len(t))
```

Following an example of a simulated *namedtuple* is presented:

```
#cars: brand, model
```

```
brand, model = (0,1) #unpackaging: brand-->0, model-->1
car = ("Peugeot", "307")
#Avoid literal values
print(car[brand]) #brand-->0
print(car[model]) # model-->1
print(car[0]) #brand-->0
print(car[1]) # model-->1
#It is not possible to add elements to a tuple
#car.append("")

#It is not possible to add elements
#car[0] = 1

#it is not possible to assign new values / modifiy the type of an element
```

In the previous example, we have assigned a name to an index position. However, we can also use the constructor *namedtuple* as follows:

```
#NamedTuple:
# (name, age, address) --> Person
import collections
#Create a new data type with the name Person-->store data
Person = collections.namedtuple("Person","name age address")
p = Person("Jose",37,"") #This is the creation of a namedtuple called Person
print(p)
print(p[0])
#List of tuples of type Person
```

```
  people = [Person("Jose",37,""), Person("David",18
,""), Person("Alfonso",18,"")]
  for person in people:
    print(person[0])
```

Note that the creation of the *namedtuple* contains two main parameters: 1) the name of the tuple and 2) the list of fields. Then, we can create "named" tuples and accessed the fields by name.

4.7 Sets

4.7.1 Problem statement

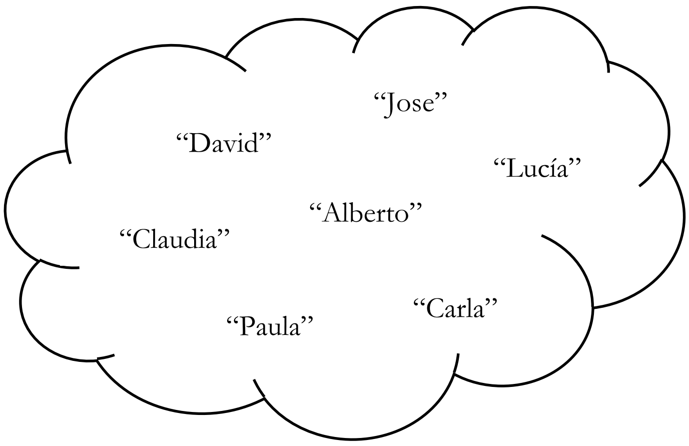
Although lists, strings and tuples allow us to organize and structure a collection of elements under specific necessities, there is a common operation that many times it is necessary to perform: ensure the uniqueness of items. In other words, a collection of unique elements is required to perform some operations. For instance, given a list of students we want to calculate the number of different names.

To do so, it is possible to implement such operations using a list and a temporary structure (an auxiliary list in which before inserting an element we check if the element already exits). However, there are some other operations that may have some relevance like union, intersection or difference between two collections. In this case, it would be better to use a specific data structure that provides these capabilities as built-in functionalities.

4.7.2 Concept

With the aim of providing collections without duplicate values, the type set, inheriting the description and capabilities from the mathematical notion of set, can be found in some programming languages.

A set is an **unordered** collection of **unique** elements.



4.7.3 Application

Collection of non-ordered elements in Python.

4.7.4 Set implementation in the Python programming language

In the Python programming language, a built-in datatype set is available. According to the official documentation,

A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

The class set in Python may be constructed in two main ways:

- Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary.

The set in Python is **mutable**. If necessary, it is possible to create a **frozen set**.

4.7.5 Examples of the main functions and methods

Given a set S, some of the main methods to work with a set are presented below (→ return value).

- len: len(S) → number of elements of the set.
- add: S.add(v) → adds an element to the set.
- clear: S.clear() → removes all elements from this set.

- difference: S.difference(S1) → returns the difference of two or more sets as a new set. (i.e. all elements that are in this set but not the others.). It is also possible to use the operator -.
- discard: S.discard(v) → removes an element from a set if it is a member.
- intersection: S.intersection(S1) → returns the intersection of two sets as a new set. It is also possible to use the operator &.
- isdisjoint: S.isdisjoint(S1) → returns True if two sets have a null intersection.
- issubset: S.issubset(S1) → reports whether another set contains this set.
- issuperset: S.issuperset(S1) → reports whether this set contains another set.
- remove: S.remove(v) → removes an element from a set; it must be a member.
- union: S.union(v) → returns the union of sets as a new set. It is also possible to use the operator |.

The set specific operations are also provided with an "update" version.

Following an example of using a set in Python is presented:

```
#Sets

#Create a set
my_set = {1, 2, 3}
print(my_set)
my_set = set()
#Adding
for i in range(3):
  my_set.add(i)
print(my_set)
#Sequence
for item in my_set:
  print(item)
```

```
#Remove
print(my_set)
if 3 in my_set: #to protect the KeyError, in case
the element does not exist within the set
  my_set.remove(3)
my_set.remove(2)
print(my_set)

#Operations: set-specific
s1 = {1, 2, 3}
s2 = {5, 2, 6}
s3 = {1}

print(s1-s3) #Operator
print(s1.difference(s3)) #method invocation
print(s1 & s2)
print(s1.intersection(s2))
print(s1 | s2)
print(s1.union(s2))


s4 = set() #empty set
print(s1-s4)
```

4.8 Dictionaries

4.8.1 Problem statement

Lists are a data structure to organize and manage a collection of items. The access to the elements is using an index (or slice) that usually is an integer expression. However, there are many problems in which it is necessary to access elements using another type of index (not just an integer number). For instance, let's suppose we have a list of students with the personal information and we want to get the information of a student using some string identifier. How can we do that?

In general, it would be possible to use different data structures keeping a mapping between the identifier and the position in a list, but this type approach will imply the necessity of quite a lot "glue code" to have consistency between both data structures.

In some programming languages, there is also a notion of "associative array" that is pretty much what is required. Furthermore, the cost (in terms of time) to access elements or keep this consistency between different data structures will imply an extra set of operations.

4.8.2 Concept

In order to provide a data structure in which is possible to access elements by using as a key any object and using the minimum number of operations (constant time), most of programming languages provide a data structure called dictionary (map or hashtable). The behavior usually is the same in many programming languages but internally, the implementation can change.

Conceptually speaking, a dictionary is table in which we have two fields: 1) the key and 2) the value.

Key	Value
"1"	Person ("1",18, David)
"2"	Person ("2",18, Claudia)
…	…

The implementation of a dictionary must consider several aspects:

- The key must be "*hashable*". An object is *hashable* if there is a function hash that can generate an unique identifier from the object.
- A *hash function* is a function that taking as input some data it generates an unique identifier. A hash function shall be **uniform**. A good hash function should map the expected input as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability. This means that two different objects should not generate the same hash (not always possible). Examples of hash functions are those used in cryptography such as the MD5 function.
    - Input: Hello

  - Your Hash: 8b1a9953c4611296a827abf8c47804d7

Furthermore, a hash function should be **not reversible** (once you know the hash, it should be impossible to get the input) and it should be efficient.

- A dictionary can be implemented using a hash table in which it is possible to configure: the hash function and the load factor. The **load factor is calculated as the number of keys divided by the total capacity**. At the implementation level, the size of the table should be chosen to get a load factor less than 1 (avoiding too many conflicts when hashing objects).

So, according to this view, a dictionary is based on hashing some object to have a unique identifier for some value and providing search capabilities in terms of time (constant time).

4.8.3 Application

The main application of the dictionaries is the provision of a data structure to efficiently store values that are identified by some key.

4.8.4 Dictionary implementation in the Python programming language

In the Python programming language, a built-in datatype dict is available. According to the official documentation,

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary.

The class dict in Python may be constructed in two main ways:

- By placing a comma-separated list of key: value pairs within braces, for example: {'1': “Jose”, '2': “Claudia”}.
- By the **dict** constructor.

Note that there are some implications in the use of dictionaries in Python:

- A key must be an **immutable** object. So, it is not possible that the hash can change over time.
- The **key object must be hashable**. This means that the object must implement the internal method `__hash__`.
- A **value can be any object**.

- **Dictionaries** in Python are **un-ordered**. However, there is an implementation of ordered dictionaries that can be used to keep the order of the different items.

Internally, a Python dictionary contains three objects:

- **Keys**: a set of objects.
- **Values**: a collection of values.
- **Items**: a collection of <k,v> (key, value) pairs.

4.8.5 Examples of the main functions and methods

Given a set D, some of the main methods to work with a dictionary are presented below (→ return value).

- List: list(d) → returns a list of all the keys used in the dictionary d.
- Len: len(d) → returns the number of items in the dictionary d.
- Access to an element: d[key] → returns the item of d with key key. Raises a KeyError if key is not in the map.
- Modify an element → d[key] = value: sets d[key] to value.
- Remove an element: del d[key] → removes d[key] from d. Raises a KeyError if key is not in the map.
- Check if key exists → key in d: returns True if d has a key *key*, else False.
- Check if key not exists → key not in d: equivalent to not key in d.
- Iterate: iter(d) → returns an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).
- Clear: clear() → removes all items from the dictionary.
- Copy: copy() → returns a shallow copy of the dictionary.
- Get item: get(key[, default]) → returns the values for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.
- Items: items() → returns a new view of the dictionary's items ((key, value) pairs).
- Keys: keys() → returns a new view of the dictionary's keys.

- Pop: pop(key[, default]) → if key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.
- Values: values() → returns a new view of the dictionary's values.

Note that many methods accessing to elements of a dictionary can raise an exception (a runtime error). To avoid this situation, it is recommendable to check whether the key exists before getting or updating its value.

```
#Common errors


#Error 1: key is not hashable-->the key is mutable
table = {}
bad_key = [1,2,3]
good_key = (1,2,3)
table[good_key] = ""


#Error 2: access and element that does not exist-->key error
table = {"1":"foo"}
#table["2"] #key error
#del table["2"] #key error
```

In the following code snippet, some of the main methods are outlined:

```
#Create a dictionary
student_grades = dict()
print(type(student_grades))
#Add element
student_grades["123456789"] = 9
student_grades["987654321"] = 8
print(student_grades)
#Access an element
print(student_grades["123456789"])
print(student_grades.get("123456789"))
```

```
  #Iterate
  #Over the keys
  for student_id in student_grades.keys():
    print(student_id)
    print(student_grades[student_id])

  #Over the values
  for student_grade in student_grades.values(): #It
erable
    print(student_grade)

  #Over the items, unpackaging key and valud
  for student_id, student_grade in student_grades.i
tems():
    print("ID: ", student_id," grade: ",student_id)

  #Removing elements

  #Remove element
  del student_grades["987654321"]
  print(student_grades)
  student_grades.clear()

  #Common methods for length, etc.
  print(len(student_grades))
  print(len(student_grades.keys()))
  print(len(student_grades.values()))

  #Checking the existence of a key before accessing
  student_grades["123456789"] = 9
  input_id = ""
```

```
#input_id = input("Introduce your id: ")
if input_id in student_grades: #IMPORTANT! Protec
ting the access to the dictionary items
  print(student_grades[input_id])
else:
  print("Your id does not exist")

#Initialize
student_grades = {'1': 9, '2': 8} # {key:value, k
ey:value}
print(student_grades)
student_grades = dict({'1': 9, '2': 8})
```

Following two examples using dictionaries are presented:

```
#Write a program that reads an input string, and
displays the frequency of each word within the inpu
t string

#input string: this is an input string containing
 this is an input

#output (not in this order):
# this: 2
# is: 2
# an: 2
# input: 2
# string: 1
# containing: 1
freq = dict()
string = input("Write your string: ")
#Split
lista = string.split(" ") #list of words ["","",""
"]
```

```
#Count frequency
for word in lista:
  if word in freq:
    freq[i]=freq[i]+1
  else:
    freq[i]=1

for word,frequency in freq.items():
  print("{}: {}".format(word,frequency))


#Write a program to store the data about demograp
hy in different regions and to
#display some report.
#Demography: number_of_women, number_of_men, avg_
age
#Input:

#Madrid = 100, 95, 45
#CyL = 80, 86, 55
#Extremadura = 75, 72, 57

#Output:

#Report:
#N° of women in Spain
#N° of men in Spain
#Avg. age in Spain

demography = {"Madrid":(100,95,45),"CyL":(80,86,5
5),"Extremadura":(75,72,57)}
women_spain=0
```

```
  men_spain=0
  avg_age_spain=0

  for info in demography.values():
    (women,men,age)=info #Unpackage a tuple info(1,
2,3)
    #women = info[0]
    #men = info[1]
    #age = info[2]
    women_spain +=women
    men_spain +=men
    avg_age_spain +=age

  avg_age_spain = avg_age_spain/len(demography.keys
())
  print("Report:\nWomen in Spain: {}\nMen in Spain:
 {}\nAvg. age in Spain: {}".format(women_spain,men_
spain,avg_age_spain))
```

4.9 Data structures comparison

In the following table, a comparison of the different data structures and their features is presented.

Table 19 Data structure features comparison.

Criteria	List	String	Set	Tuple	Dictionary
Elements	0..n	0..n	0..n	0..n (fixed size)	0..n
Type of elements	Any and mixed	Characters	Any and mixed	Any and mixed	Any
Ordered access	Yes by index	Yes by index	No	Yes by index	No (ordered dictionary)

Sorted	No	No	No	No	No
Mutable	Yes	No	Yes (No, unless the set is frozen)	No	Yes
Iterable	Yes	Yes	Yes	Yes	Yes

Table 20 Data structure time complexity of common operations.

Data structure/ Operation	**Add (not sorted)**	**Remove**	**Search (one element)**	**Search all**
List	O(1)	O(1)	O(n)	O(n)
String	O(1)	O(1)	O(n)	O(n)
Tuple	O(1) new tuple created	N/A	O(n)	O(n)
Set	O(n)	O(n)	O(n)	O(n)
Dictionary	O(1)	O(1)	O(1)	O(n)

4.10 Comprehension list, set and dictionary

In this chapter, an introduction to some of the main built-in data structures in Python has been done. However, in regard to data structure creation, there are some methods that can be used to initialize lists, sets and dictionaries in a Pythonic way. Comprehension list, set and dictionary are rapid (and more declarative) ways to create these data structures when there is some initial input over which we have to perform some operation.

The general syntax to create lists, sets and dictionaries using comprehension techniques follows the next grammar rules:

- `[expression for element in collection if expr]`
- `{expression for element in collection if expr}`
- `{(expression, expression) for element in collection if expr}`

It is also possible to add more complex and nested expressions for filtering data. However, in order to keep the readability of the code, it is a good programming practice to keep these structures “clean” and “clear”.

```
  #List
  grades = [5,6,7,8]
  #Comprehension list applying a bonus of the 25% o
f the initial grad.
  updated_grades = [g+(g*0.25) for g in grades]
  print(updated_grades)
  #Comprehension list applying a bonus under some c
ondition.
  updated_grades = [g+(g*0.25) for g in grades if g
 >= 7]
  print(updated_grades)

  #Set
  repeated_names = ["Jose", "Mary", "Jose", "Claudi
a"]
  #Comprehension set
  name_set = {name for name in repeated_names}
  print(name_set)

  #Dict
  names = ["Jose", "Mary", "Claudia", "Antón"]
  #Comprehension dict by mapping grades and names,
the use of the zip function will be also valid.
  names_grades = { (names[i], grades[i]) for i in r
ange(len(names))}
  print(names_grades)
```

4.11 Interesting links

- https://docs.python.org/3/reference/compound_stmts.html
- https://docs.python.org/3/library/stdtypes.html#str
- https://docs.python.org/3/library/stdtypes.html#textseq
- https://realpython.com/list-comprehension-python/

4.12 Quiz

1. **Select the correct statement:**
   a. Lists and tuples are immutable.
   b. Lists and tuples are mutable.
   c. Strings and tuples are immutable.
   d. Lists and strings are immutable.

Answer: c

2. **Given the next statements, select the CORRECT answer:**

```
t = (1)
print(t[0])
```

   a. The program is correct and displays "1".
   b. There is a runtime error.
   c. There is a compilation error.
   d. The program is correct and displays "(1)".

Answer: b

3. **Is it possible to iterate over the keys of a dictionary?**

   e. Yes
   f. It is only possible to iterate over the items.
   g. It is only possible to iterate over the values.
   h. No

Answer: a

For more questions, see also the online Python Bingo[7].

[7] https://slides.com/josemariaalvarez/python-bingo

5 SUBPROGRAMS: PROCEDURES AND FUNCTIONS

5.1 Introduction

A software program is basically a black box that receives some input (data and configuration), perform some operation(s) and generate some output (results). In order to implement a software program, a development lifecycle is followed to specify, develop, test, deploy, maintain and retire the program. In this lifecycle, the first stage is specification (after a business motivation) in which analysts and other stakeholders define the functionalities of the software product or service. Afterwards, a detailed designed is done prior to the implementation. Depending on the type of development methodology these stages are executed in different manner. However, a complete, consistent and correct specification is always required (even in products where we have to find out the necessities).

In this book, the focus is on the development stage applying an engineering technique: programming. The specification or requirements are already there, and the effort relies on the design (not too much) and implementation stages. So far, the capabilities of a programming language have been unveiled focusing on the management of data (organization, structure and storage) and, in the flow control. However, the black box, the algorithms, the operation center, where the engineering happens, is still a bit transparent. In this black box, operations are implemented to provide some capability that will be a combination of functionalities working together to produce some result.

Furthermore, these functionalities are, in many cases, candidates to be reused by other programs, e.g. the absolute value function, so, when designing functions, it is necessary to think in the reusability factor of the function.

On the other hand, if a function has been implemented and tested:

- Why should not reuse it?
- Why to reinvent the wheel? (only reinvent the wheel to make a better wheel)
- Why to pass again for the whole development process of that function?

Due to all these reasons, it is important to modularize our programs to provide specific functions that are reusable.

5.2 Functions and procedures

5.2.1 Problem statement

There are many cases in which some source code is repeated in different lines. This leads us to think in the possibility of creating some functionality to wrap this code in something that can be reusable. Obviously, the first type of code reusing is copy/paste but a functionality (simple or complex) is a candidate to be reused in case we can isolate such functionality and assign a single responsibility (recommended).

Furthermore, when program complexity is increased, we must add methods that can make the code more modular, understandable and robust.

With the aim of improving the reusability factor of our code, reuse existing pieces of code, making our code more modular and understandable, we find the use of functions as a basic technique to improve the code quality and to separate responsibilities.

Two main principles are addressed with functions:

- Don’t Repeat Yourself (DRY).
- Don’t Reinvent the Wheel.

5.2.2 Concept

A first definition of a function can be found in mathematics.

In mathematics, a function is a binary relation over two sets that associates to every element of the first set exactly one element of the second set. (Wikipedia)

From this definition, we can derive several elements of a function:

- A function has a name that denotes the binary relation.
- A function has some input set.
- A function has some output set.
- A function relates the input with the output set (an application).

Following this definition, we can define the factorial function:

$$\text{fact: } N \rightarrow N$$

- *fact* is the name of the function.
- A non-negative integer number is the input set.
- A non-negative integer number is the output set.
- There is relationship (an application) between the input and the output:
    - *fact(n) = n * fact(n-1) if n > 1*
    - *fact(n) = 1 if n in {0,1}*

In computer science, programming, there is also a mapping between these definitions:

Table 21 Mapping between a function in mathematics and programming.

Mathematical concept	**Programming concept**
Function name	Function name
Input set (dimension)	Input parameters (domain): number and type of parameters
Output set (dimension)	Return value (range): number and type of returned values

So, in general, the definition of a function in programming is similar to the concept in mathematics. An application that relates some input to some output. However, in programming, there is also some relevant remarks:

- **There are functions that does not return any value (procedures).**
- There are programming languages which functions are pure (e.g. Haskell), no side effects.
- A function in programming can generate side effects (modifications in input parameters and/or global variables).
- A function in programming has a signature, an unique identifier.
- A function in programming is some executable code in which we have to consider how the parameters are passed.

From these remarks, two new definitions can be made:

- **Function signature**. It is an unique identifier that comprises the **function name and the number and type of parameters**. Depending on the programming language the function signature can change enabling the possibility of adding new arguments (overloading).
- **Parameters**. There are two types of parameters: **formal parameters** (part of the function definition) and **actual parameters** (those that are passed when the function is invoked).

Recursive functions

A recursive function is a function which either calls itself or is in a potential cycle of function calls.

A recursive function shall have:

- A basic case in which the function is solved (a value is returned).
- A general case in which the function is invoking itself.

Recursive functions are considered not very efficient in terms of execution time and memory consumption (too many function calls and not reuse of previous results in many cases, e.g. Fibonacci). However, many problems in computer science and algorithms are expressed in terms of recursive functions.

5.2.3 Application

The main application of functions is to increase the reusability factor of existing pieces of code to make our programs more modular, reusable, understandable and robust.

5.2.4 Function implementation in the Python programming language

Once a formal definition of a function has been made, it is necessary to map this definition to the specific syntax Python.

A function is an executable statement defined in some scope in which there is a binding between a block of code and a name. A function is also a type ("a wrapper around the executable code for the function"), we can declare variables of type "function".

According to the official documentation, a function definition defines a user-defined function object:

```
    funcdef                ::=  [decorators] "def" funcname "(" [parameter_list] ")"
                                ["->" expression] ":" suite
    decorators             ::=  decorator+
    decorator              ::=  "@" dotted_name ["(" [argument_list [","]] ")"]
NEWLINE
    dotted_name            ::=  identifier ("." identifier)*
    parameter_list         ::=  defparameter ("," defparameter)* "," "/" [","
[parameter_list_no_posonly]]
                                | parameter_list_no_posonly
    parameter_list_no_posonly ::=  defparameter ("," defparameter)* [","
[parameter_list_starargs]]
                                | parameter_list_starargs
    parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**"
parameter [","]]]
                                | "**" parameter [","]
    parameter              ::=  identifier [":" expression]
    defparameter           ::=  parameter ["=" expression]
    funcname               ::=  identifier
```

Not: The function definition does not execute the function body; this gets executed only when the function is called.

Following with the factorial function example, it is possible to define a function as follows:

```
def factorial(n):
    if n==0 or n == 1:
        return 1
    else:
        fact = 1
        for v in range(1,n+1):
            fact = fact * v
        return fact
* if n<0 the function will not work, a new case must
be defined.

def name (list of parameters):
     block of code
   return (optional)
```

On the other hand, Python functions (or methods) can have different types of parameters. More specifically, according to the official documentation, the next types of parameters can be found:

positional-or-keyword: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

positional-only: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /,
positional_or_keyword): ...
```

keyword-only: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

var-positional: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

var-keyword: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Applying the different types of parameters in Python, we introduce here the following examples of use.

```
#Positional Parameters
def add(a,b):
  return a+b


#Optional Parameters
def add_3(a,b=3):
  return a+b


def add_k(a,k=0):
  return a+k


#Keyword Parameters
def keyword_params(*args):
  for i in args:
    print(i)


#Dictionary Parameters
def dict_params(**kwargs):
  for k, v in kwargs.items():
    print(k,":", v)


#Invoke a function
print(add(2,3))
print(add_3(2))
print(add_k(3))
keyword_params(["a",2])
dict_param ={"name":"Jose"}
dict_params(**dict_param)
```

Note: all parameters after an optional parameter shall be also optional.

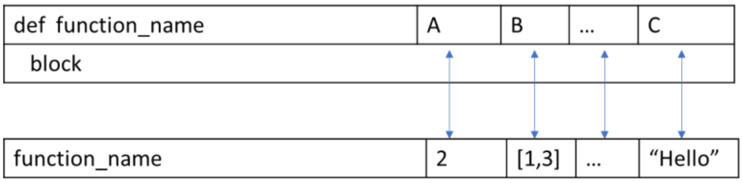
Parameter evaluation (from the official documentation)

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same "pre-computed" value is used for each call.

This is especially important to understand when a default parameter is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default value is in effect modified.

Parameter mapping (formal and actual parameters):

When invoking a function there is internal mapping between the formal parameters and the actual parameters.



More specifically, in the memory, the parameters are stacked and, the first part of the function is in charge of assigning the values to the different formal parameters.

In Python, all parameters are passed by reference (a memory address).

Other programming languages also support the pass by value in which instead of a memory address a copy of the value is passed to the function.

In general, this second strategy is less efficient in terms of memory usage. According to the following table, there is a function that takes 3 parameters as an input. When the function is invoked, the memory address of the actual parameters is stacked to be recover and assigned to the formal parameters in the function body.

Other functions	**Memory address**
Main function …	
Variables	
2	20
[1,3]	24
“Hello”	32
…	
function_name	
a	
b	
c	
Function code	
…	
Stack	
…	
32 (immutable)	
24 (mutable)	
20 (immutable)	
Heap …	

Parameters which type is mutable could be modified within the function.

```
   #Pass parameters by reference
   def my_max(a, b):
     print(id(a)) #identifier of the parameter
     print(id(b)) #identifier of the parameter
     a = 1000 #a is a number-->Inmutable--
>there is a new assignment a new space in the memor
y is allocated
     if a > b:
       return a
     else:
        return b

    def modify (alist):
      print(id(alist))
      #Append a new element-->modify
      alist.append(1000)
      #return is optional --
> functions without a return statement are called p
rocedures



    if __name__=="__main__":
      a = 2
      b = 3
      print(id(a)) #identifier of the variable
      print(id(b)) #identifier of the variable
      my_max(a, b)
      print(a)
      mylist = [1,3,4]
      print("Before calling...")
      print(mylist)
      print(id(mylist))
      modify(mylist)
      print("After calling...")
      print(mylist)
      #Pure functions-->no side effects
```

Parameter annotations (documentation)

Internally, annotations are added as a dictionary to the function object. The reference for annotations is defined in the PEP 3107[8].

[8] https://www.python.org/dev/peps/pep-3107/

```
  #Annotations can be just string values.
  def my_add(a: '<a>', b: '<b>') -
> '<return_value>':
    return a+b

  print(my_add.__annotations__)

  #Annotations can also include types. However, thi
s is only documentation. it does not impose any res
triction on the parameters.
  def my_add2(a: int, b: int) -> float:
    return a+b

  print(my_add2.__annotations__)
```

5.2.5 Lambda functions

A **lambda function** is an anonymous function declared online.

- A lambda function can take any number of arguments.
- A lambda function can only return one expression.
- A lambda function is a Python function, so anything regarding parameters, annotations, etc. are applicable to lambda functions.

The theory behind lambda functions comes from the "Lambda Calculus". According to the official documentation[9], the Lambda functions follow the next grammar:

```
lambda_expr    ::=    "lambda"    [parameter_list]    ":"
expression      lambda_expr_nocond      ::=      "lambda"
[parameter_list] ":" expression_nocond
```

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression lambda parameters: expression yields

9 https://docs.python.org/3/reference/expressions.html#grammar-token-lambda-expr

a function object. The unnamed object behaves like a function object defined with:

```
def (parameters):
return expression
```

Lambda functions are mainly used in the following scenario:

- *Simple functions that we want to apply inline and we do not plan to reuse.*

Lambda functions also come with some drawbacks:

- Syntax can be complex; it is not so intuitive as a regular function.
- Readability and understandability of the source code become complex.
- Need of thinking in a functional way (not intuitive when coming from imperative program.

In the Python PEP8 document recommends the following:

Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

Yes:

```
def f(x): return 2*x
```

No:

```
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically 'f' instead of the generic ''. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression).

However, Lambda functions are elegant to solve specific problems and parametrize some expressions in functional programming.

```
(lambda x, y: x + y)(2, 3)
```

5.2.6 High-order functions

A high-order function is a function that:

- takes a function as a parameter or

- returns a function.

For instance, we are going to define a function that receives as parameters:

- The function, f, to be applied to.
- The list of elements, alist and returns, a list of the values after applying f to each of the elements in alist.

In this case, the example will return the square of the elements of the list.

- Input: f my own function to calculate the square of a number, and the list [1,2,3].
- Output:

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

Modify the program to make a function that adds 2 to each of the elements of the list.

- Output: [3, 4, 5]

```
#We define a function that takes as a parameter a function f and a list, a list,
#then applies the function f to any element in the list.
def apply_f_to_list(f, alist):
  results = []
  for v in alist:
    value = f(v)
    results.append(value)
  return results


def my_square(n):
  return n**2


def add_2(n):
  return n+2
```

```
if __name__=="__main__":
  values = [1,2,3]
  results = apply_f_to_list(my_square,values)
  print(results)
  results = apply_f_to_list(add_2,values)
   print(results)
```

5.2.7 Modules

According to the official documentation[10], a module can be defined as follows:

- **A module is a file containing Python definitions and statements.** The file name is the module name with the suffix `.py` appended.
- **A module can contain executable statements as well as function definitions.** These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables.

When a module named "`mymodule`" is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `mymodule.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
- The installation-dependent default.

It is possible to import modules in different manners:

[10] https://docs.python.org/3/tutorial/modules.html

- import <module_name> (as <alias>)
    - import numpy
    - import numpy as np
- from <module_name> import <a,b,c>
    - from typing import cast, Any, Callable
    - from typing import *
- from <module_name> import <a> (as <alias>)
    - from typing import cast as cs

Following an example of a module, `myfactorials,` containing two functions.

```
def factorial_r(n):
    if n<0:
        return -1
    elif n==1 or n==0:
        return 1
    else:
        return n*factorial_r(n-1)



def factorial(n):
    if n==0 or n == 1:
        return 1
    else:
        fact = 1
        for v in range(1,n+1):
            fact = fact * v
        return fact
```

The module is imported and used from a main function.

```
import myfactorials as mf
if __name__=="__main__":
    print(mf.factorial_r(5))
```

As a final remark, Python comes with a library of standard modules, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls.

5.3 Interesting links

- PEP 3107 - Function Annotations.
- PEP 484 - Type Hints. Definition of a standard meaning for annotations: type hints.
- PEP 526 - Syntax for Variable Annotations. Ability to type hint variable declarations, including class variables and instance variables

5.4 Quiz

1. **Can a function have nested blocks of code?**

   a. Yes, it is part of program and we can use all elements of the programming language.

   b. No, it is only allowed one block per function.

   c. Yes, but only in recursive functions.

   d. No, unless the function is declared within a module.

Answer: a

2. **What is the function signature?**

   a. It is the combination of the name, number and type of the parameters and return type.

   b. It is the combination of the name and number of parameters.

   c. It is the combination of the name and annotations.
   d. It is the combination of the name and return type.

Answer: a

3. **Which are the types of parameters in Python functions?**
   a. Positional parameters and annotations.
   b. Keyword parameters.
   c. Annotations and documentation.
   d. Positional, default and value/keyword parameters.

Answer: d

6 OBJECT ORIENTED PROGRAMMING

6.1 Object-oriented Programming

The Object-Oriented Programming (OOP) paradigm is a huge topic which in some sense is out of scope of this book. However, it is convenient to introduce the main notions about OOP, since everything in Python is an object. It is also true that there is some debate in the management of objects in Python, since there are some similarities between prototype-oriented programming and the use of objects in Python.

In this chapter, OOP is introduced from a syntax point of view. This chapter does not cover: Object-Oriented Analysis or Design (logical and behavioral design), modelling languages such as UML (Unified Modelling Language) or advanced software design.

6.1.1 Problem statement

In the previous sections, a complete programming paradigm has been introduced to develop software programs. On the one hand, data is managed through data structures that organizes and stores the input data, the temporary results and the output data. On the other hand, functions are used to consume data and to provide capabilities that will generate results.

However, there are other programming paradigms. OOP (Object Oriented Programming) is a complete programming paradigm that makes use of an abstraction (classes and objects) to model both problems and solutions in a software environment. Instead of separating data and functions, objects combine both creating abstractions of some entity.

For instance, in case we must develop a software system to manage the student's grades, we can address the problem in two different manners:

- Creating data structures to manage the information about students and grades. Afterwards, functions to exploit this information will be designed separating data from functionalities.
- Defining domain objects with attributes and methods (capabilities) that will interact each other passing messages and exchanging data to perform some operation.

6.1.2 Concept

We recall here the definitions introduced in a previous chapter.

- What is a **class**?

A class is an abstraction of a (real/virtual) entity defined by a set of attributes (data/features) and a set of capabilities/functionalities/operations (methods). A class defines a category of objects and contains all common attributes and operations.

- What is an **object**?

An object is an instance, a realization, of a class. It is the realization of a class with specific values for each attribute and a shared behavior.

- What is an **attribute**?

An attribute is a feature/characteristic/property shared by a set of objects.

- What is a **method**?

A method is a capability/operation/functionality/behavior shared by a set of objects. It is possible to identify the next types of methods:

- **Constructor**. It is a method used to initialize the state of an object. It is invoked when the object is created.
- **Destructor**. It is a method used to release the resources in use by an object. It is invoked when the object is removed or has finished the object lifecycle.

- **Setter**. It is a method to access and to set a value to some object attribute.
- **Getter**. It is a method to access and to get the value of some object attribute.
- **Business logic**. It is a method that delivers some object-specific functionality.

Furthermore, there are 4 principles of the OOP that are relevant:

- **Abstraction**. Abstraction refers to the process where only "relevant" data is represented in the classes hiding unnecessary details that are out of scope of our problem or solution.
- **Encapsulation**. Encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them. It is possible to define three levels of visibility: public, protected and private.
- **Inheritance**. Inheritance is a mechanism, an object designed technique in which some data or behavior is inherited from one class to another creating a taxonomy through relationships (*is-a*). The use of inheritance will generate a set of categories. There is also some discussion[11] between the use of inheritance vs composition/delegation (for further reading).
- **Polymorphism**. Polymorphism refers to the possibility that an object can behave in different manner (multiple forms). There are different types of polymorphism: subtyping (method overriding when a parent method is overridden providing a new behavior) and parametric (method overloading when a method changes the input parameters).

Finally, the **SOLID** principles are also a good reference to design classes[12]:

- *The **Single Responsibility Principle**: a class should have one, and only one, reason to change.*

[11] https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

[12] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

- *The* ***Open Closed Principle****: you should be able to extend a classes behavior, without modifying it.*
- *The* ***Liskov Substitution Principle****: derived classes must be substitutable for their base classes.*
- *The* ***Interface Segregation Principle****: make fine grained interfaces that are client specific.*
- *The* ***Dependency Inversion Principle****: depend on abstractions, not on concretions.*

6.1.3 Application

The main application of OOP is the design and implementation of solutions using objects as an abstraction. The software program is comprised of objects (instances of a class) that interact each other to provide a solution.

6.1.4 Object-oriented programming in the Python programming language

A class definition in Python follows the next syntax:

```
class ClassName:

    <statement-1>

    .

    .

    .

    <statement-N>
```

Class definitions, like function definitions (def statements) must be executed before they have any effect. In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful.

As an initial example, let suppose we have to store the information about a person (id and name) and provide some capabilities (speaking and running). We can define a class Person with these attributes and capabilities.

```
class Person:
  id = ""
  name = ""
  speaking():
    #do speaking

  running(velocity):
    #do running
```

Following with the example, we can now define an instance of a Person, John, with some id (1).

In the Python programming language, an attribute can be accessed using the next syntax:

```
instance_name.attribute
```

In the Python programming language, a method can be invoked using the next syntax:

```
instance_name.method_name(parameters)
```

self object. When defining a class, we can make a reference to *my* own attributes or methods using the implicit object "self". Furthermore, the self object is always declared as the first parameter of the instance methods. This implicit reference to myself is mainly used to de-ambiguate the names of attributes, variables/parameters and functions in some scope. At a low level, a method is a function which first parameter is the self object reference. There is no object abstraction when executing a program (at least not in Python), the execution is purely procedural invoking functions.

```
  #Class and object


  #class <CLASSNAME>:
  #   STATEMENTS
  #STATEMENT: METHOD DECLARATION (A FUNCTION DEFINE
D IN THE CONTEXT OF SOME CLASS)


  #To represent people, name and age (data)

#Capabilities (methods/verbs):

  #speak
  #run

  class Person: #make an abstraction of the data an
d capabilities (functions) of a set of objects
    #Constructor
    #self-->implicit object
    def __init__(self, name, age):
      print("I am being created...")
      self.name = name
      self.age = age
      print(type(self))
    #Destructor
    def __del__(self):
      print("The object is being removed...")
    #Setting and getting attributes of the class
    #Encapsulation: visibility--Python is public
    def set_name(self,new_name):
      self.name = new_name

    def get_name(self):
      return self.name
    #Specific capabilities
    def speak(self, words):
      for w in words:
        print("Speaking: ",w)
    def run(self, velocity):
      print("Running at: ", velocity)
    #Override existing/inherited methods
    def __str__(self):
      return "I am {} and I am {} years old".format
(self.name, self.age)
```

```
 #
 #identifier = class_name(parameters)
 mary = Person("Mary",25) #Constructor

 #Notation to access class members: attributes and
methods
 # class_instance.class_member

 print(mary.get_name())
 mary.set_name("Mary bla bla bla")
 print(mary.get_name())

 print(mary.name)
 print(mary.age)


 mary.speak(["Hello", "class"])
 mary.run(30)

 #dir(mary)
 print(mary)
```

6.2 Interesting links

- https://docs.python.org/3/tutorial/classes.html
- https://realpython.com/python3-object-oriented-programming/

6.3 Quiz

1. **Is the same a class than an object?**

   a) Yes, both are the same.

   b) A class is a kind of blueprint to create instances (objects).

c) An object is a kind of blueprint to create instances (classes).

d) A class is a set of objects.

Answer: b

2. **Which is NOT a principle of Object Oriented Programming?**

a) Inheritance

b) Encapsulation

c) Abstraction

d) Composition

Answer: d

3. **What is "self" in the context of Object-Oriented Programming in Python?**

a) It is a reference to the implicit constructor within a class.

b) It is a reference to the Python GIL.

c) It is a reference to the Python Garbage Collector.

d) It is a reference to the implicit object within a class.

Answer: d

7 SEARCHING AND SORTING

7.1 Introduction

The implementation of algorithms requires the use of different programming techniques to mainly represent, consume and produce data items.

- Data structures allow us to properly conceptualize the structure and organization of the data that is managed by our programs (as input, as intermediary results or as output).
- Functions allow us to fulfill two important requirements in any program:
    - The DRY (Do not Repeat Yourself) principle.
    - Improve the reusability of the code (implicitly including reliability if the function has been properly verified and validated).
    - Improve the readability and understandability of the code.
- Objects are a conceptualization of a domain in which we express data (attributes) and operations (functions) modelling entities

(classes) that represent the relevant entities in both in the problem and solution domain.

These three elements allow us to manage the data is managed in a program and to implement business logic through functions and methods.

However, the functions or methods (in case of objects) that offer some capability usually implement some complex business logic that requires different types of algorithms.

There are different types of algorithms depending on the structure, their formal definition, their control flow or how they manage data. As an example, we can classify algorithms in a category as follows:

- **Recursive algorithms**: trying to solve a base case and, then, make a recursion to solve the general case. E.g. Factorial.
- **Dynamic programming algorithms**: remembering past results (when there is an overlapping) and using them to find new results. E.g. Fibonacci numbers.
- **Backtracking algorithms**: trying to find a solution applying a depth-first recursive search until finding a solution. E.g. The famous n queens.
- **Divide and conquer algorithms**: dividing the problem into smaller problems that can be then solved recursively to finally combine solutions and deliver a general solution. E.g. Binary search.
- **Greedy algorithms**: finding not just a solution but the best one. E.g. Count money using the fewest bills and coins.
- **Branch and bound algorithms**: calculating the probability of getting a solution for optimization problems. E.g. The famous TSP (Travelling salesman problem) problem.
- **Brute force algorithms**: testing all possible solutions until a satisfactory solution is found. It makes use of heuristics and optimization techniques. E.g. Break a password.
- **Randomized algorithms**: getting a solution by running a randomized number of times to make a decision. E.g. Quicksort and pivot selection.

The key point is how the algorithm tackle the solution of a problem and how the algorithm makes use of basic functionalities like searching and/or sorting to prepare the data and generate (intermediary and final) solutions.

That is why, it is necessary to know and govern basic algorithms for searching and sorting data under different data structures to be able to define and design more complex algorithms. In general, the design of algorithms makes use of different types of algorithms. In other words, more complex algorithms are built on top of other more basic algorithms.

For instance, an advanced artificial intelligence technique makes use of another type of algorithm (e.g. branch and bound) that, at the same time, makes use of other techniques such as searching and sorting. So, an algorithm relies on other layers of algorithms.

In this chapter, some basic searching (find an element in a collection) and sorting (ordering a collection of values) algorithms are presented. More specifically, the following algorithms are explained.

- Linear search.
- Binary search.
- Bubble sort.
- Insertion sort.
- Selection sort

Obviously, there are many other algorithms in both fields, and, this chapter only pretends to introduce the foundations of these techniques.

Furthermore, many programming languages and libraries are already providing these types of algorithms. However, it is important to know the foundations of these techniques to be able to design our own algorithms.

Finally, some introduction to the evaluation of algorithms in terms of time and space complexity is also outlined.

7.2 Searching algorithms

We all know the notion search:

- There is a catalogue of items.
- There is a query.
- An algorithm tries to match the items according to this query.
- The results (positions or the items) are returned.

In the field of Computer Science and algorithms, searching is quite similar.

- Given a list of items and a target element, the searching function looks for the target element in the list of items and returns a value (the item, the position or a boolean value).

7.2.1 Linear Search

7.2.1.1 Problem definition

There are many cases in which we need to look for a value in some collection. For instance, given a person id and a list of students, the program shall return the student details.

In a simpler way, given a number and a collection of numbers, the program shall return whether the element exists within the collection.

- Given the list: [2,8,3,9]
- Target number: 2
- Result: True

7.2.1.2 Concept

Linear search is the most basic technique for searching. Basically, it iterates over a collection until finding the target value. Then, there are different strategies:

- Find first.
- Find last.
- Find all.

In linear search, no assumption is taken (like elements must be sorted).

In regard to the time/temporal complexity, the linear search cannot be considered very efficient.

- Best case: O(1), the first element visited by the algorithm is the target element.
- Worst case: O(n), being n the number of elements in the collection. For instance, if the last element is the target or if the target does not exist.
- Average case: O(n), being n the number of elements in the collection. Although the target can be found in any position

between (0,n) if it is not found in the first position, we can assume the algorithm will iterate over n elements.

The notion of linearity for some algorithm is not generally bad. However, as the input grows the time to search will be also increased in linear time. So, an algorithm that can perfectly work for thousands of items, if the problem scales in one order of magnitude, the time can dramatically be increased.

In other algorithms with quadratic or event exponential time complexity, this situation is directly unsustainable, and we should re-think our data structures and algorithms.

7.2.1.3 Application

The linear search represents the basic and general algorithm for looking up elements in a collection.

It only requires a collection of items, a target (e.g. query or value) and strategy (first, last or all) and the algorithm will iterate over all the elements until finding the target.

Find first:

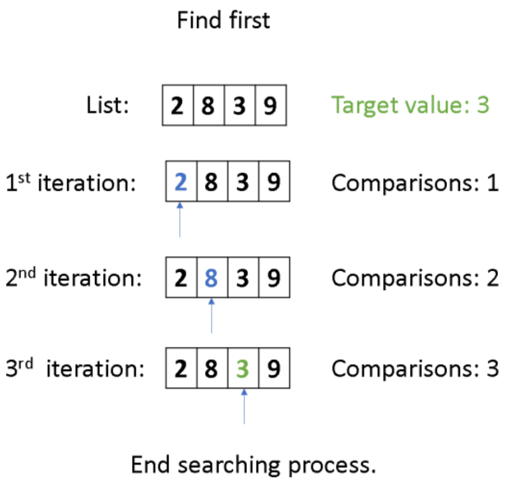


Figure 14 Find first example.

Find last:

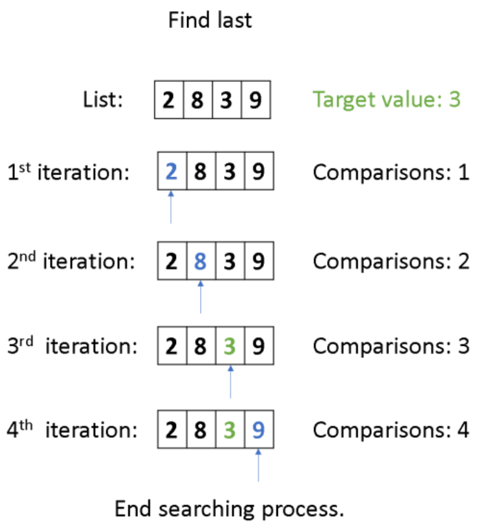


Figure 15 Find last example.

Find all:

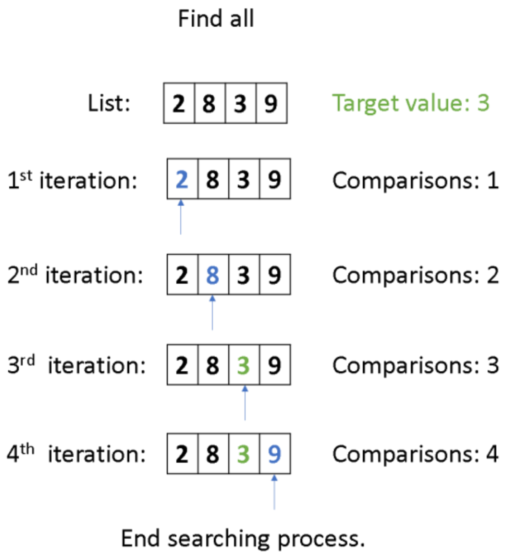


Figure 16 Find all example.

7.2.1.4 Python implementation

Following, some examples of linear search are presented:

```
#Linear search examples

def linear_search_first(values, target):
  found = False
  i = 0
  while not found and i<len(values):
    found = values[i] == target
    i += 1
  return found

def linear_search_last(values, target):
  found = False
  i = len(values)-1
  while not found and i>=0:
    found = values[i] == target
    i -= 1
  return found

def linear_search_all(values, target):
  i = 0
  while i<len(values):
    found = values[i] == target
    if found:
      print("Found at position: ",i)
    i += 1
  return

print(linear_search_first([2,8,3,9], 3))
print(linear_search_last([2,8,3,9], 3))
linear_search_all([2,8,3,9], 3)
```

7.2.1.5 Visual animations

You can find some interesting visual representation of the algorithms in the following links:

- https://www.cs.usfca.edu/~galles/visualization/Search.html
- https://visualgo.net/en

7.2.2 Binary Search

7.2.2.1 Problem definition

Sometimes we face problems in which we can apply a technique of divide and conquer. In other words, we can reduce the solution space applying some rule.

For instance, if we have a deck of sorted cards and someone asks for some card, we can easily look up that card without checking all the cards. Try to think a bit in how you internally proceed:

1. You know the card you are looking for.
2. You know that the deck is sorted (perhaps is new).
3. You make an approximation to look up the target card.
4. If you are "lucky", you will match the card at the first attempt, otherwise, you will make an internal calculation to approximate where the card could be.
5. You will repeat the steps from 2 to 4 until getting the card.

In general, everybody will proceed in this manner (unless you have a lot of time to review all the cards). Here, the question is:

What are we actually doing?

Basically, we are optimizing how we look up for an item by reducing the number of items we have to review. Since, the deck is sorted we can even discard part of the cards in each attempt.

As a simple experiment, try to sort a deck of cards and look for one specific card (measure the time). Afterwards, repeat the same experiment after shuffling the cards (measure the time again). Write down your feeling!

7.2.2.2 Concept

Binary search or half-interval search is a kind of searching technique that is based in the previous concept:

- In each iteration, we reduce the possibilities by discarding half of the problem.

This technique works quite well for many problems, but it has a very strong assumption: **the list must be sorted according to some criteria**. Furthermore, we need access to the elements using an index.

Given a list l and a target value v, the algorithm works as follows:

- It takes two indexes: min and max. Initially, min=0 and max=len(l)
- It calculates the middle, (min−max)/2 element of a list.
- It compares this element with the target element.
- If both are equal, then we have found v and we can stop.
- If v>l[middle], update index min=middle+1
- If v<l[middle], update index max=middle−1
- The algorithm will stop if min=max or v is found.

In regards to the time/temporal complexity, the binary search is quite efficient (but we need a sorted list).

- Best case: O(1), the first element visited by the algorithm is the target element.
- Worst case: O(log_n), being n the number of elements in the collection. The log complexity comes from the height of a full balanced binary tree (that is actually the intrinsic structure of calls that is generated).
- Average case: O(log_n), being n the number of elements in the collection.

This algorithm represents a very good option if we have a sorted list.

7.2.2.3 Application

As we have stated before, if we have searching problems containing a sorted collection with indexed access to elements, we can approach the problem with a binary search technique.

Binary search example:

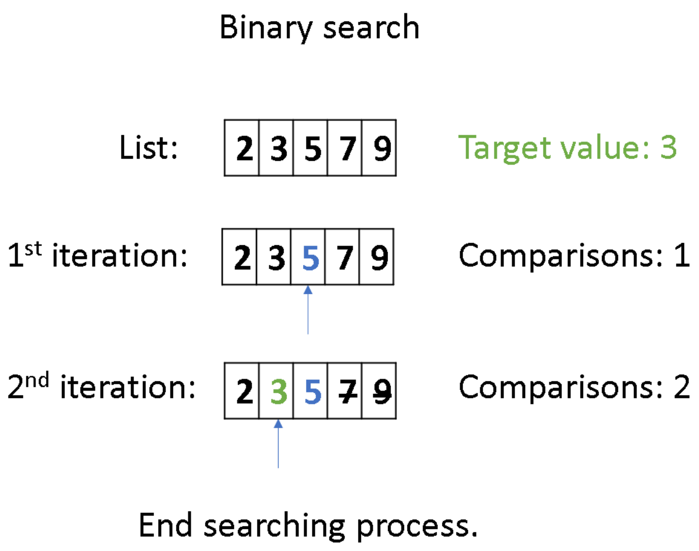


Figure 17 Binary search example.

7.2.2.4 Python implementation

```
 #See animation: https://www.w3resource.com/python-exercises/data-structures-and-algorithms/python-search-and-sorting-exercise-1.php
 def binary_search(values, target):
   first = 0
   last = len(values)-1
   found = False
   while first <= last and not found:
     mid = (first + last)//2
     if values[mid] == target:
       found = True
     else: #Discard half of the problem
       if target < values[mid]:
         last = mid - 1
```

```
        else:
            first = mid + 1
    return found

print(binary_search([1,2,3,5,8], 6))
print(binary_search([1,2,3,5,8], 5))
```

7.3 Sorting algorithms

As in searching, we all know the notion sorting a collection:

- There is a catalogue of items.
- There is some criteria to order the items.
- Al algorithm tries to swap elements until getting the proper order.

In the field of Computer Science and algorithms, sorting is quite similar.

- Given a list of items and some criteria (single or multiple), the sorting function looks for comparing the elements according to criteria and put them in the proper order.

The main consideration we need to know when sorting is that elements must be **comparable**. In practice, it means that we can apply the logical operators for comparison.

7.3.1 The Bubble sort algorithm

7.3.1.1 Problem definition

Let's suppose we have to generate a report of the list of students sorted by the family name in descending order. How can we approach this problem?

In general, we have to compare the items in the list, in this case family names, and swap them until we can ensure that the last element is sorted.

7.3.1.2 Concept

The bubble sort is a very basic sorting algorithm based on comparing adjacent items and swap them if they are not in the proper order. The algorithm will repeat the process until all elements are sorted (and somehow have been compared each other).

In the following pseudo-code, the bubble sort algorithm is presented. The algorithm starts taking one element and compares it against all others swapping if necessary.

```
   for i from 1 to N
     for j from 0 to N-1
          if a[j]>a[j+1]
             swap(a[j], a[j+1])
```

In regards of time complexity, the bubble sort algorithm behaves with a very poor performance because it requires comparing all elements against all elements. In a list of n elements, each element is compared against the other n−1 elements.

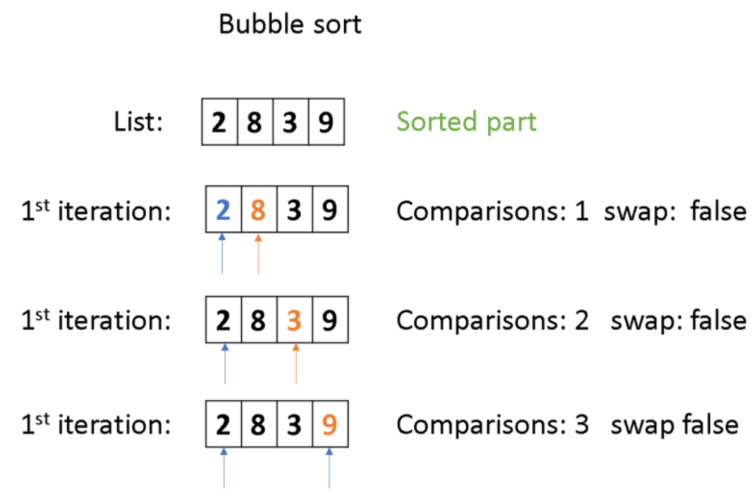
- Best, Worst and Average case: O(n2), being n the number of elements in the list. In general, since we have two nested loops the time complexity can be calculated by multiplying the time complexity of the first loop (n) times the time complexity of the second loop (n).

However, there are variants of the bubble sort that tries to reduce the number of comparisons (and swaps) stopping when in one iteration there is no swap (the list is sorted). Anyway, the worst-case time complexity still remains O(n2).

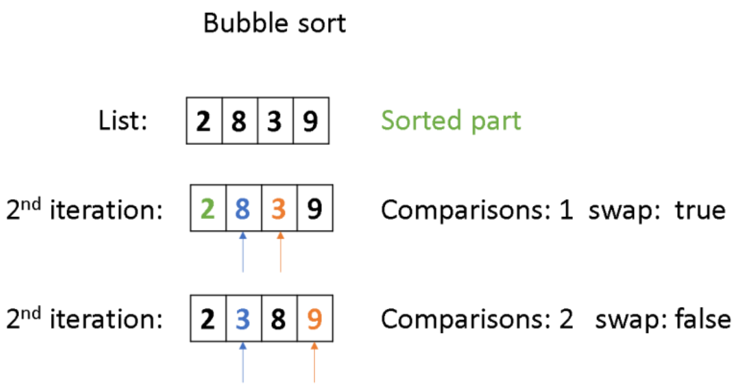
7.3.1.3 Application

The application of the bubble sort algorithm is justified when we have to sort a small list of items, otherwise the time to sort will be dramatically increased and other algorithms should be considered.

Bubble sort example:



End first iteration.



End second iteration.

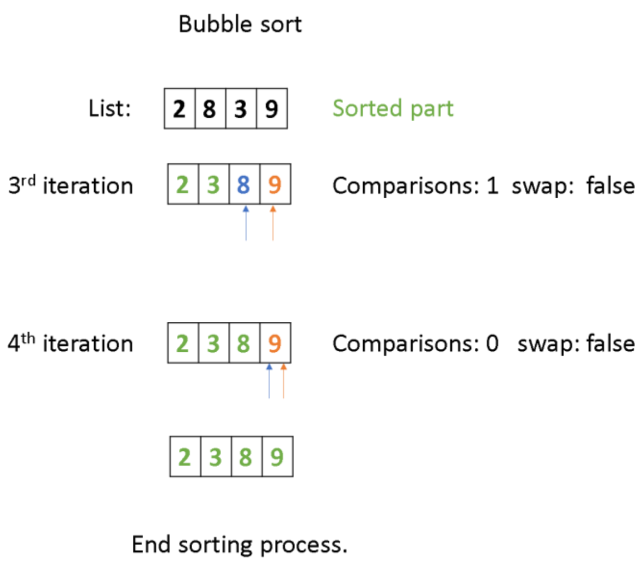


Figure 18 Bubble sort example.

7.3.1.4 Python implementation

```
  #See more: https://www.w3resource.com/python-
exercises/data-structures-and-algorithms/python-
search-and-sorting-exercise-4.php
  def bubble_sort(values):
      n = len(values)
      for i in range(n):
          for j in range(n-i-1):
              if values[j] > values[j+1]:
                #Swap
                values[j], values[j+1] = values[j+1
], values[j]

  values = [22, 8, 33, 12, 14, 43]
  bubble_sort(values)
  print(values)
```

7.3.2 The Selection sort algorithm (only information)

7.3.2.1 Problem definition

As we have introduced before, any time we have to sort a list of items according to some criteria, we should think in a sorting algorithm. If the list starts to become very large, we can optionally think in other algorithms rather than the bubble.

7.3.2.2 Concept

The selection sort algorithm is a sorting algorithm that in each iteration looks for the minimum element in the right-hand side of the unsorted list and swap it with the element after the last sorted element (initially the first element).

Intrinsically, the algorithm generates two sublists (managed by an index): the sublist of sorted elements and the remaining list.

In the following pseudo-code, the selection sort algorithm is presented.

```
for i in len(lst):
  min_index = i
  for j in (i+1, max)
        if lst[min_index] > key
             min_index = j
  swap(lst[min_index],lst[i])
```

In regards to the time/temporal complexity, the selection sort does not behave to much better in comparison to the bubble sort. It only tries to improve the number of swaps in the worst case.

- Best case, Worst case and Average case: O(n2) comparisons and n swaps.

There are other variants of the selection sort that can improve the efficiency.

7.3.2.3 Application

The main situation in which we should think in using the selection sort algorithm is when memory is limited since it does not required any extra resource to store temporary results.

Selection sort example:

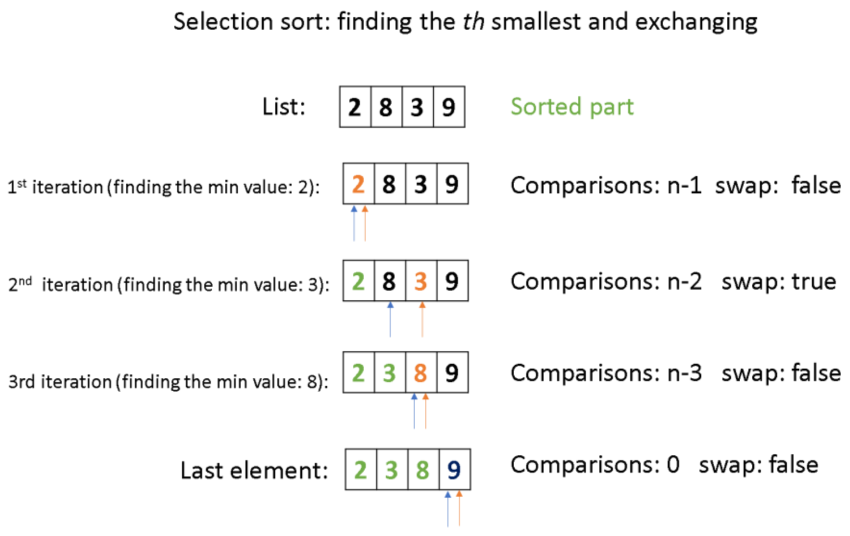

Selection sort: finding the th smallest and exchanging
List: 2 8 3 9
Sorted part
1st iteration (finding the min value: 2): 2 8 3 9
Comparisons: n-1 swap: false
2nd iteration (finding the min value: 3): 2 8 3 9
Comparisons: n-2 swap: true
3rd iteration (finding the min value: 8): 2 3 8 9
Comparisons: n-3 swap: false
Last element: 2 3 8 9
Comparisons: 0 swap: false

Figure 19 Selection sort example.

7.3.2.4 Python implementation

Selection: "*Given a list, take the current element and exchange it with the smallest element on the right hand side of the current element.*"

```
  def selection_sort(values):
    n = len(values)
    for i in range(n):
      min_idx = i
      for j in range(i+1, n):
          if values[min_idx] > values[j]:
              min_idx = j
      #Swap with the minimum value position
      values[i], values[min_idx] = values[min_idx],
 values[i]
   values = [22, 8, 33, 12, 14, 43]
  selection_sort(values)
  print(values)
```

7.3.3 The Insertion sort algorithm (only information)

7.3.3.1 Problem definition

See problem definition before.

7.3.3.2 Concept

The insertion sort algorithm is a sorting algorithm that in each iteration is building a sorted list by moving the elements until finding the right position.

In the following pseudo-code, the insertion sort algorithm is presented.

```
for i from 1 to N
 key = a[i]
 j = i - 1
 while j >= 0 and a[j] > key
    a[j+1] = a[j]
    j = j - 1
 a[j+1] = key
```

In regard to the time/temporal complexity, the insertion sort does not behave to much better in comparison to the previous ones.

- Best case: O(n) comparisons and constant swaps.
- Worst case: O(n2) comparisons and swaps.
- Average case: O(n2) comparisons and swaps.

The insertion sort algorithm is quite simple and easy to implement, this is the main advantage.

7.3.3.3 Application

The main situation in which we should think in using the insertion sort algorithm is again when the memory is limited.

Insertion sort example:

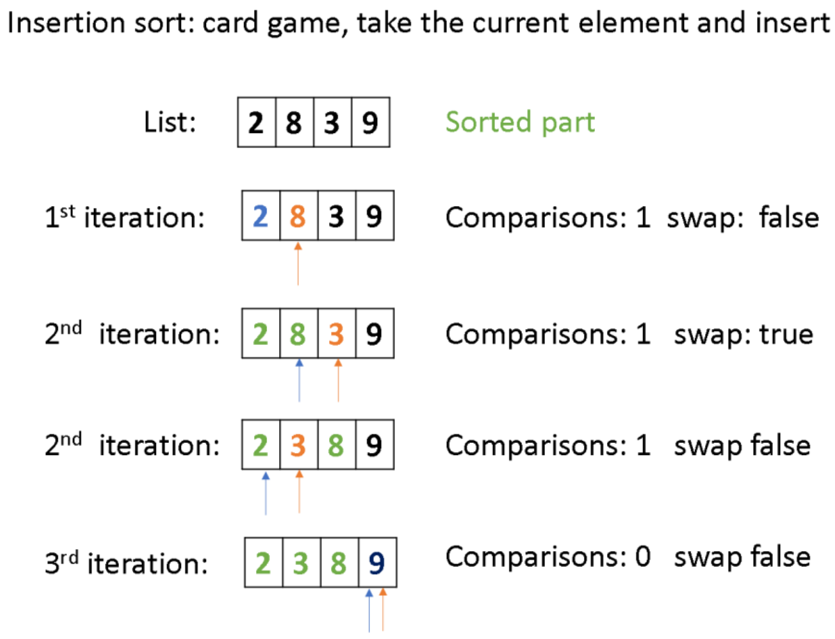

Insertion sort: card game, take the current element and insert
List: 2 8 3 9
Sorted part
1st iteration: 2 8 3 9
Comparisons: 1 swap: false
2nd iteration: 2 8 3 9
Comparisons: 1 swap: true
2nd iteration: 2 3 8 9
Comparisons: 1 swap false
3rd iteration: 2 3 8 9
Comparisons: 0 swap false

Figure 20 Insertion sort example.

7.3.3.4 Python implementation

Insertion: "*Given a list, take the current element and insert it at the appropriate position of the list, adjusting the list every time you insert. It is similar to arranging the cards in a Card game.*"

```
  def insertion_sort(values):
    n = len(values)
    for i in range(1, n):
      key = values[i]
      # Move elements of arr[0..i-
1], that are  greater than key, to one position ahe
ad of their current position
      j = i-1
      while j >=0 and key < values[j] :
        values[j+1] = values[j]
        j -= 1
      values[j+1] = key
```

```
values = [22, 8, 33, 12, 14, 43]
insertion_sort(values)
print(values)
```

7.4 Evaluation of algorithms: time complexity

In computer science, time complexity refers to the amount of time required to execute an algorithm. Usually, it is estimated by counting the number of basic operations (constant execution time).

To express the time complexity of an algorithm, we use the Big O notation. This is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Its main application is the classification of algorithms depending on how the runtime increases when the input size grows. There are some established time complexities, take a look to this cheat sheet[13].

To evaluate the time complexity, we usually define three cases (best, worse and average). However, we should always consider the worst case to really provide a realistic value of the maximum execution time.

- Best case: represents the minimum amount of time required for inputs of a given size.
- Worst case: represents the maximum amount of time required for inputs of a given size.
- Average case: represents the average amount of time required for inputs of a given size.

[13] https://www.bigocheatsheet.com/

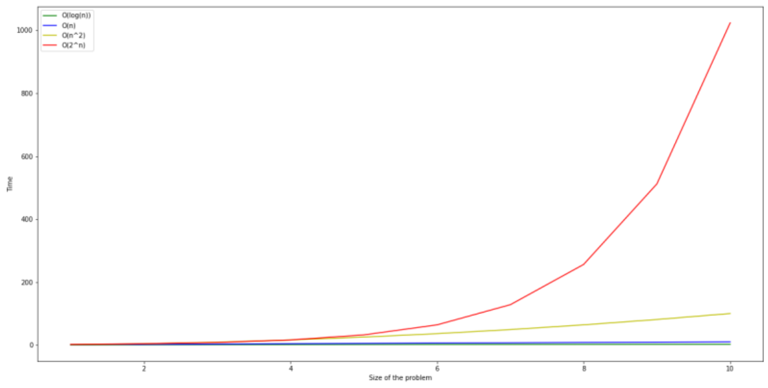


Figure 21 Generation of time complexity with a Python program.

```
import matplotlib.pyplot as plt
import numpy as np
import math
x = np.linspace(1,10,10)
logv = np.vectorize(math.log)
linear = np.vectorize(lambda x: x)
quadratic = np.vectorize(lambda x: x**2)
exponential = np.vectorize(lambda x: 2**x)
#Figure
fig = plt.figure()
plt.figure(figsize=(20,10))
log_label, = plt.plot(x,logv(x), 'g',  label='O(log(n))')
linear_label, = plt.plot(x,linear(x), 'b',   label='O(n)')
quad_label, = plt.plot(x,quadratic(x), 'y', label='O(n^2)')
exp_label, = plt.plot(x,exponential(x), 'r', label='O(2^n)')
plt.legend(handles=[log_label, linear_label, quad_label, exp_l
abel])
plt.xlabel('Size of the problem')
plt.ylabel('Time')
plt.show()
```

The calculation of time complexity is an interesting and formal area in computer science. In this course, it is only necessary to know that there is a way of measuring and comparing the complexity of algorithms in terms of execution time.

Some typical time complexities can be calculated as follows:

- **Constant**: O(1). In the following example, the number of instructions that are executed is constant (4) and does not depend on the input size.

```
a = 2
b = 3
c = a * b
print(c)
```

- **Logarithmic**: O(log_n). In the following example, the size of the problem is reduced in each iteration. We are discarding half of the problem in each step.

```
i = n
while i > n:
  #Do constant operations
  i = i // n
```

- **Linear: O(n).** In the following example, the number of instructions to be executed will depend on the input size, n.

```
i = 0
while i < n:
  #Do constant operations
  i = i + 1
```

- **Quadratic:** O(n^2). In the following example, the number of instructions to be executed will depend again on the input size, n. Nested loops are examples of quadratic complexities

```
for i in range(n):
   for j in range (i):
     #Do constant operations
```

In general, there are some rules to reduce the time complexity expressions and keep only an upper limit. For instance, if we have the following complexities, the estimations would be:

- O(k)→O(1)
- O(kn)→O(n)
- O(n+m)→O(n)
- ...

Which would be the time complexity of the next program?

```
for i in range(n):
  print(i)

i = 0
while i<n:
  print(i)
  i = i + 1
```

Solution: the first loop iterates n times performing one operation. Then, the second loop iterates again n times performing 3 operations.

n∗1+n∗2=3n,→O(3n)→O(n)

7.5 Interesting links

- https://python-textbok.readthedocs.io/en/1.0/Sorting_and_Searching_Algorithms.html
- https://runestone.academy/runestone/books/published/pythonds/SortSearch/toctree.html
- https://www.oreilly.com/library/view/python-cookbook/0596001673/ch02.html

7.6 Quiz

1. **Can we perform linear search over a sorted list?**
   a. True
   b. False

Answer: a

2. **A binary search will always perform in log n time.**
   a. Yes, but if and only if the input is sorted.

b. Yes, but if and only if the input is sorted in ascending order.
c. Yes, but if and only if the input is unsorted.
d. Yes, it does not depend on the input.

Answer: a

3. **What is the main drawback of the bubble sorting algorithm?**
    a. The time complexity is n^2 (quadratic) due to the Python implementation.
    b. It only works with lists of integers.
    c. The time complexity is n^2 (quadratic) due to: two nested loops and many comparisons and swaps.
    d. It cannot be applied to dictionary values.

Answer: c

8 RESOURCE MANAGEMENT

8.1 Introduction

A software development process must carefully consider the resources required to run a program. Usually, at the very first stages of development, the use of resources, the performance, the throughput of a software are not always critical. These are non-functional requirements that are, many times, considered when the program is already providing the desired functionalities and it is closed to be transferred to an operational environment (dealing with real workloads).

Physical resources are a cornerstone to ensure that the functionality is properly delivered. Flexibility, scalability and other non-functional aspects are not always tested but they are critical when our software must evolve.

There are many factors that can potentially impact the use of resources in our software: the architecture, the users, the operational environment, etc. Most of them must be, theoretically, stated in the non-functional requirements, designed using the proper notation in UML (e.g. physical deployment diagram) and implemented through the proper technologies (e.g. gateways, protocols, etc.) and programming models (e.g. push, event-oriented, etc.). Although non-functional requirements are very relevant to the success of a software program, the implementation of such requirements is sometimes not prioritized, delegating them to a secondary position.

However, the history has demonstrated that these non-functional aspects of a system are, at least, as important as the functionalities. If a capability is delivered but it cannot be properly used, what are we creating?

As examples of how systems have evolved overtime considering non-functional aspects, we can list some interesting well-known social network platforms such as Twitter[14], Linkedin[15] or Whatsapp[16]. They basically applied the principle: **"Think big, start small and keep growing"**. The functionality was almost the same but, the success of their platforms implied a great effort on the non-functional aspects driving them to a success

However, in this chapter, we focus more on the low-level aspects of resource management. Those that can be managed at the programming level. Any software platform (a process) will consume physical resources: processing units, memory, input/output devices (e.g. network). In this context, programming, as an engineering technique, must consider the proper use of these three categories of resources. Good programming techniques will help us to make a proper use of computer resources. Then, other engineering methods such as software design and software architecture methods will also address the problem from other high-level perspectives.

The experience has demonstrated that bad coding practices or small issues in a program can lead us to further and unexpected huge problems in a larger setting. We all have experienced these situations:

- Too much time to run some program because the CPU is completely busy running another process. (Processor).
- Lack of memory because a program is allocating more and more memory while it is running. (Memory)
- Fail to save a file because it is being used by another program, although the program is not anymore running. (Input/Output system)

These situations lead us to think in the origin of such issues. In general, if we assume that the program does not have any special requirement (e.g. CPU, memory or input/output), then, there is a potential problem in the

[14] https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

[15] https://engineering.linkedin.com/architecture/brief-history-scaling-linkedin

[16] http://highscalability.com/blog/2014/2/26/the-whatsapp-architecture-facebook-bought-for-19-billion.html

implementation, at the coding level: a memory leak, a zombie process, a file that has not been closed, a bad use of the programming language, etc.

That is why, in this chapter, the focus relies on understanding the resources a program can use, identifying potential points of interest and implementing Python programs more efficiently. To do so, the chapter is divided into two main subsections:

- Process management.
- Memory management.
- Input/Output system. This last topic will be also presented in the next chapter.

8.2 Process management

A process is a program in execution that is consuming computer resources. The source code is loaded in the main memory, there is a stack to pass parameters to a function and to leave the result, a heap to create new objects, etc. At the same time, the code is executed consuming CPU cycles, new processes and threads can be created to parallelize the processing and the input/output system is used to gather inputs from an user or an external service and to persist results. When a program is running, resources are consumed, and CPU time is one of them.

8.2.1 Problem statement

CPU time is a high-valuable resource. It allows us to execute programs that will provide some functionalities. If we run many programs at the same time, we will potentially face time execution issues. If our program is not properly coded and abuses of the CPU time, other programs will be affected. So, here the question is if we can understand and prevent a bad use of the CPU time by applying small improvements in our code.

8.2.2 Concept

Firstly, it is necessary to understand how a program is assigned to a CPU slice. There are some strategies to schedule programs (e.g. First Come, First Served) and parameters to prioritize one process over another. To do so, the operating system keeps a process table with metainformation of each process under execution and plans the schedule according to different parameters.

From our programing perspective, it is interesting to know this metainformation about our processes to profile our programs and detect potential issues in the time that our code is executing. To do so, there are APIs and profiling tools that can help us to detect bottlenecks.

Obviously, process management is an advanced topic in the field of Operating Systems so, the intention is to show what we can know from a coding perspective to, at least, get some awareness of the decisions we are making when implementing a program.

8.2.3 Application

The main application of a proper process management is to identify those parts of code that can be taken more time than necessary due to a bad programming practice (e.g. a loop that iterates over a full list when it could finish before).

8.2.4 Python implementation

In Python, like in other interpreted programming languages, the virtualized execution of programs is a relevant aspect. Everything is actually managed by the interpreter that can make decisions about our program, e.g. code optimization. However, it is important to remark some modules and functionalities that can help us to explore our programs and to run profiling tools: `psutils`[17], `dis`[18] and `profiler`[19]:

8.3 Memory management

Memory is one of the key computer resources. Depending on the programming language and the type of execution, the memory will be managed at different levels: program, interpreter or operating system.

8.3.1 Problem statement

When developing a software system, the proper use of memory is critical to avoid problems related to memory leaks (the use of memory is growing due to a bad management of resources, the memory that is allocated is not properly released). This situation is especially relevant in programming languages such as C, C++, Pascal or Fortran where dynamic memory must be managed by the developer.

[17] https://psutil.readthedocs.io/en/latest/

[18] https://docs.python.org/3/library/dis.html

[19] https://docs.python.org/3/library/profile.html

In this sense, it is possible classify memory according to different aspects. From a developer point of view, the implementation of a software system shall have a clear view of the type of memory that is in use:

- Static memory, containing the program code, functions, constants or variables.
- Dynamic memory, containing the space to allocate new objects, etc.

Again, depending on the programming language and the execution type, the strategies to manage the memory will dramatically change. That is why, it is convenient to be able to solve the following questions:

- What is memory?
- Which are the types of memory?
- Why do we need memory?
- What happened with variables?
    - When are they created?
    - Whey are they deleted?
- How memory is allocated?
- How memory is released?

8.3.2 Concept

Memory management is the process of ensuring the proper allocation and release memory blocks during the execution of program.

As it has been previously introduced, the memory management will depend in many factors:

- Type of programming language: static vs dynamic typing.
- Type of execution: native vs interpreted.
- Type of programming language paradigm: procedural vs objects.

8.3.3 Application

The main application of memory management lies on the proper allocation and release of program resources. When a program is running (a process), the memory is used to first load the program including all static configuration:

- Main program and functions code.
- Data: constants and variables (static) with different data types.
- Stack: memory space for function calls.
- Heap: memory space to dynamically allocate space for new data items.

Depending on the programming language, it is very important to understand how the memory is managed (specially the stack and heap spaces).

8.3.4 Python implementation

8.3.4.1 Memory architecture

In Python, the memory management process depends on the implementation of the interpreter. In our context, every behavior refers to the default implementation of the Python interpreter in C: **CPython**. Other interpreters, such as Jython, IronPython, etc. could change some of the strategies to execute Python programs. However, the notions of memory management will remain similar.

Firstly, it is interesting to see the stack of memory elements in CPython. Here, it is convenient to describe the different layers:

- Object-specific memory: memory blocks allocated for the program variables (with different data types).
- Python core: memory blocks for the elements of the Python standard library.
- Python object allocator: creation and deletion of objects (the "magic").
- Python's raw memory allocator: interpreter management of memory.
- Underlying general-purpose allocator: implementation in the C programming language of the memory allocation and release.
- OS-specific Virtual Memory Manager: memory management of the operating system: different strategies to bring memory pages, etc.
- Physical memory: hardware in which everything is stored.

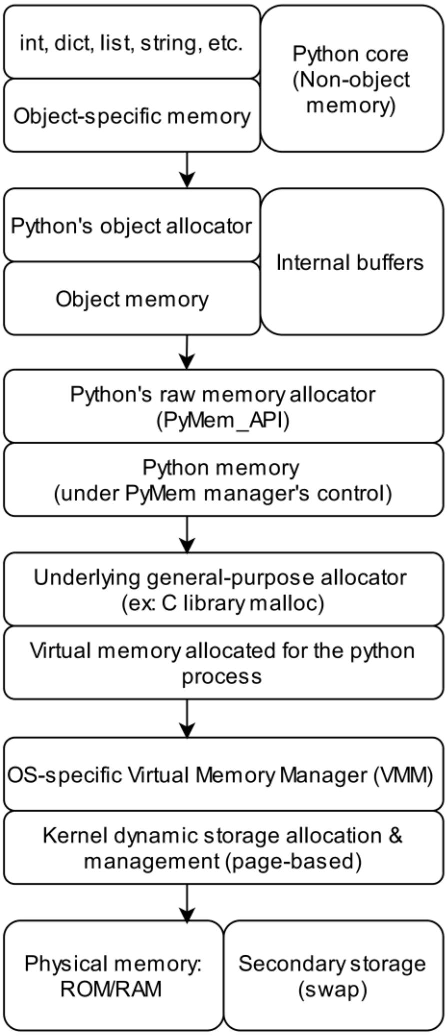

int, dict, list, string, etc.
Object-specific memory
Python core (Non-object memory)
Python's object allocator
Object memory
Internal buffers
Python's raw memory allocator (PyMem_API)
Python memory (under PyMem manager's control)
Underlying general-purpose allocator (ex: C library malloc)
Virtual memory allocated for the python process
OS-specific Virtual Memory Manager (VMM)
Kernel dynamic storage allocation & management (page-based)
Physical memory: ROM/RAM
Secondary storage (swap)

Figure 22 Memory layers[20] in Python.

8.3.4.2 Memory structure

In the CPython interpreter, the memory is structured according to three elements:

---

20 https://github.com/python/cpython/blob/ad051cbce1360ad3055a048506c09bc2a5442474/Objects/obmalloc.c#L534

- **Block**: a portion of memory that can keep only one Python object of a fixed size.
    - ***The size of the block can vary from 8 to 512 bytes and must be a multiple of eight (i.e., use 8-byte alignment).***
- **Pool:** a composition of blocks of the same size. The size of a pool is commonly equals to the memory page (4KB). See following, the pool implementation in the C programming language.

```
   /* Pool for small blocks. */
struct pool_header {
    union { block *_padding;
            uint count; } ref;          /* number of all
ocated blocks    */
    block *freeblock;                   /* pool's free l
ist head         */
    struct pool_header *nextpool;       /* next pool of
this size class  */
    struct pool_header *prevpool;       /* previous pool
       ""        */
    uint arenaindex;                    /* index into ar
enas of base adr */
    uint szidx;                         /* block size cl
ass index        */
    uint nextoffset;                    /* bytes to virg
in block         */
    uint maxnextoffset;                 /* largest valid
 nextoffset      */
};
```

Each pool has three states:

1. used — partially used, neither empty nor full.
2. full — all the pool's blocks are currently allocated.
3. empty — all the pool's blocks are currently available for allocation.

- **Arena:** a chunk of 256kB memory allocated on the heap, which provides memory for 64 pools. Arenas are organized into a doubly linked list called usable_arenas. Arenas are sorted according to the number of free pools. This notion of "free" is at Python level (no operating system).

8.3.4.3 The Python object: GIL (Global Interpreter Lock) and GC (Garbage Collector)

Everything in Python is an object.

At the low level, in the CPython interpreter there is a struct (a record) which is used by every other object in CPython.

- `ob_refcnt`: reference count. This counter is used for the garbage collection.
- `ob_type`: pointer to another type. The type of of tha Python object (e.g. int, str, dict, etc.)

There are two elements that have a very important impact in the memory management:

- **The Global Interpreter Lock (GIL)**. It is a solution to protect the access to shared resources (e.g. memory). For instance, two threads trying to rewrite a variable value will be locked and the requests will be serialized. The impact of the GIL is that actually there is no parallelization at the thread level (processes are **single-thread**). This is a major topic of discussion: how to manage race condition in Python.
- **The Garbage Collector (GC)**. It is a functionality that basically counts the number of references of an object to decide whether the memory can be released (reference count is 0). There are many ways of implementing garbage collection strategies (e.g. generations). In our context, it is interesting to know when an object increases the number of refences:
    - Assign it to another variable.
    - Pass the object as an argument.
    - Include the object in a list.

Finally, there are some lessons or key points about the Python memory management strategy:

- Python completely abstracts the memory management. Developers can work at a very high-level abstraction (e.g. no need to specify the bytes to allocate or release).
- The memory release is done by the GC. However, the implementation of this functionality cannot be easily configured (it will depend on the interpreter).

In general, it is necessary to know how things work behind the scenes. However, unless we work on compilers, program optimization or interpreters, we do not really need to manage these low-level features.

8.4 Input/Output system

This topic will be presented in the next chapter with special focus on files.

8.5 Interesting links

- https://cython.readthedocs.io/en/latest/src/tutorial/memory_allocation.html
- https://rushter.com/blog/python-memory-managment/
- https://rushter.com/blog/python-garbage-collector/
- https://realpython.com/python-memory-management/

8.6 Quiz

1. **Everything in Python is an object.**
   a. True
   b. False

Answer: a

2. **What is the meaning of GIL?**
   a. Global Internal Lock
   b. Generic Interpreter Lock
   c. Global Interpreter Lock
   d. Generic Internal Lock

Answer: c

3. **When the number of references to a Python object is increased?**

   a. All answers are correct.
   b. Pass the object as an argument.
   c. Include the object in a list.
   d. Assign it to another variable.

Answer: a

9 INPUT/OUTPUT SYSTEM

9.1 Introduction

In the first chapter an introduction to the main elements of a Von-Neumann computer was done. CPU, Memory and the Input/Output (I/O) system are the main building blocks of computers based on this architecture.

In the case of the I/O system, it can be defined as the subsystem in charge of the management of "*relationships*" between the software and hardware. There are many I/O devices classified into three major categories: input, output and input/output. All of them have a similar structure:

A hardware that serves us to gather and to put something from/to the real world. For instance, the action of pushing a key in the keyboard, the management of a print head, the sound of some speaker, etc.

A software driver that serves us to communicate the hardware with the operating system. This software provides a set of primitives to manage the physical device and implements a set of other primitives to communicate with a specific operating system.

In the operating system, there is usually another layer called the "HAL" (Hardware Abstraction Layer) that is in charge of abstracting the management of hardware devices.

Following with the previous example, when someone press a key in the keyboard, a code is sent through the internal buses to the CPU through an interruption ("A new input has been detected."). Then, the operating system is able to interpret that instruction (controller/driver) getting from the CPU the generated interruption: the source and content (part of the memory where the device is leaving data). Finally, this new input is interpreted and passed to the active program or the operating system by itself.

This is a basic flow of execution and interaction with the I/O system. At the programming level, programming languages commonly provides high-level APIs ("Application Programming Interfaces") to interact with I/O devices. These APIs include capabilities to:

- **Connect** to a device: locking mode.
- **Operate** with the device: reading and writing.
- **Disconnect** from the device: release the resource.

9.2 The file subsystem

The file subsystem is the component of the operating system in charge of managing files. A file is basically an interface that serves us to store and to retrieve information in some data structure provided by the operating system through a set of system calls. There are different types of file systems such as NTFS, FAT33, EXT3, etc.

Files are used to store information when persistence capabilities are required or when the memory required in a program exceeds the computer memory. A file usually has some metadata (e.g. creation date, owner, etc.), data (file contents in binary or text) and some pre-defined delimiters (e.g. end of line and end of file) that may change depending on the operating system. Directories (folders), links and files by itself are the main types of files.

In all programming languages, there is always an interface (an object or a set of functions) to interact with the file subsystem. In general, these capabilities are provided, at the low-level, by the operating system. The basic flow of working with files is as follows:

Open the file. In general, it is necessary to indicate the path of the file, the mode (binary, text, reading, writing) and the encoding (sometimes optional, by default the one provided by the operating system).

Do operations. Depending on the mode, the file content will be processed: character by character, line by line, block by block, etc. until reaching some flag (end of line, end of file). On the other hand, if the operation is for writing, the operations will be similar but for writing.

Close file. When opening a file, a computer resource is being locked for our program. Once, the processing has finished, it is necessary to release the resource again to the operating system.

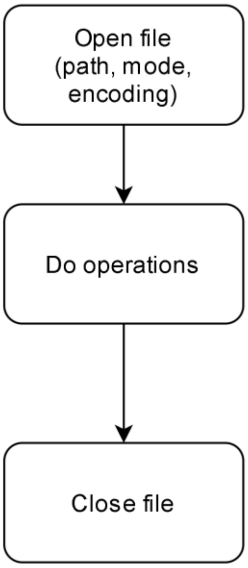


Figure 23 Basic flow to work with files.

It is possible to classify files according to different criteria:

- Type of content: binary (bytes) vs text (structured and un-structured).
- Type of interaction: writing, reading or appending.
- Type of processing: char, line or block.
- Type of file: directory, link or file content.

As a final remark, working with files is like having a pointer to a specific position in a file that it is changed when some operation is done (reading, writing, seeking, etc.). In this manner, it is possible to navigate over the file contents making a sequential or random access. For instance, when opening a file in appending mode, the pointer is situated at the end of the file (to start appending contents).

9.3 Text-based files

9.3.1 Problem statement

As it has been introduced, files are used to persist information when the size of data is greater than the main memory or when we need to recover data after an execution of a program. This second case represents the most common need when running a program. After performing some operations, it is necessary to save the state of the program (e.g. a MSWord document, a video under edition, etc.) to later recover the previous state. Furthermore, we have also seen that there are different types of files depending on the content, processing or interaction. In this context, the use of text-based files is a common practice when it is necessary to enable a mechanism to edit file contents with a simple text editor.

For instance, let's suppose we save the state of a program to manage student's grades (student ID, grade) generating a set of tuples separated by commas. Can we update the grade of the student without running the program again? In this case, it would be possible to just open the file and change some of the values. However, the direct modification of a file can lead us to introduce some typo, error or inconsistency in the contents.

Text-based files are a good mechanism to persist information that could be changed by a third-party program like a text-editor. However, it is necessary to consider other aspect regarding the structure of the file. A text-based file can be classified depending on its internal structure:

- Unstructured: text lines (raw text).
- Structured: following some format that can be customized or standardized. In the first case, we can create a file for which we establish a program-specific structure, or we can use a predefined format such as CSV (Comma Separated Values), XML (eXtended Markup Language) or JSON (Javascript Object Notation). Anyway, the two main operations for saving and loading the file will need to understand the structure and the semantics (interpretation of the fields) of the contents to properly populate data in memory.

9.3.2 Concept

Text-based files provide an interface to serialize/de-serialize data in files that can be modified with a text editor. Depending on the structure and format, different file handlers may be used. Furthermore, the generation of the input and processing of the output will rely on the format and semantics of the contents.

In regard to the processing of file contents, it is necessary to establish how the file will be processed: char by char, line by line, etc. In general, a file will be processed populating instances in some data structure in memory such as a list, dictionary or object.

9.3.3 Application

The main application of text-based files is the serialization of program data in a plain text file that can be modified using any text-editor. On the other hand, it is possible to easily read a text file using different processing strategies and applying rules to interpret the content. However, when reading/writing text-based files, it is convenient to understand the underlying semantics of the structure (if any). For instance, each line corresponds to a record in which the first 4 characters represents an unique identifier, then there is a separator (“;”) and finally, there is a name until finding the end of the line.

9.3.4 Text-based files in the Python programming language

Python provides an interface within the standard library to handle files. The file object allows us to interact with the file system through common operations like: *open, close, read, readlines, readline, write, seek, etc.* Furthermore, there are 3 standard files for input (`sys.stdin`), output (`sys.stdout`) and error (`sys.stderr`). Reading/Writing from the standard input/output files is similar to read/write to a text-based file.

In order to handle a file in Python, it is necessary to follow the previous process.

Open the file indicating the operation mode.

- `open`: returns a file object.

Table 22 Python file handling modes.

Mode	Description
'r'	Default mode. Opens a file for reading.
'w'	Opens a file for writing. If the file does not exist, it will create a new one. If the file exists, it truncates the contents.

'a'	Open file in append mode to add contents. If the file does not exist, it creates a new one.
't'	This is the default mode. It opens the file in text mode.
'b'	Open the file in binary mode.
'+'	This will open a file for reading and writing.

Do operations depending on the operation mode and content (binary vs text).

- `read`: returns a specified number of bytes from the file.
- `readline`: reads one entire line from the file.
- `readlines`: returns a list containing lines from the file.
- `next`: returns a next line from the file.
- `write`: writes a string to the file.
- `writelines`: writes a sequence of strings to the file.
- `flush`: flushes the write buffers of the file.
- `tell`: returns the file's current position.
- `seek`: sets the file's current position.

Close the file releasing the resource.

- `close`: flushes and closes the file.

As an example of using files a Python, check the next example.

```
if __name__ == "__main__":
    msgs = ["Hello", "Mary", "How are you?"]
    path = "messages.txt"
    file = open(path,"w")
    for msg in msgs:
        file.write(msg+"\n")
file.close()
```

Other methods and libraries such as the CSVDictWriter, CSVDictReader, etc. are presented in the volume II of this book.

The with Python-specific statement also allows us to avoid the need of closing the file descriptor. However, as a general good programming practice, we should explicitly close the file to release resources to the operating system.

In the volume II, examples of reading and writing text-based files in different formats are provided. Furthermore, it is convenient to remark that Python objects can be directly serialized in some text-based format but, depending on their composition, a specific class encoder will be necessary. For instance, in the following example, a JSON encoder is implemented to serialize a composite object. Attributes of type "`datetime.datetime`" will be serialized as a string while the rest of the attributes will be serialized accessing all elements in the internal dictionary of any Python object.

```
class MyEncoder(JSONEncoder):
        def default(self, o):
            if isinstance(o, datetime.datetime):
                return o.__str__()
            else:
                return o.__dict__
```

9.4 Binary-based files

9.4.1 Problem statement

There are certain situations in which we want to write and read data that cannot be modified by third-party tools. In this manner, it is possible to ensure the consistency of file contents. For instance, the edition of an MP4 can only be done by programs that are able to process this standard format. If we open the file with another program, like a text-editor, we will only see strange characters that represent a bunch of bytes. Although the file can be modified, it will not be easy to add bytes to a binary file without losing consistency.

9.4.2 Concept

Binary-based files provide an interface to serialize/de-serialize binary data in files that cannot be modified with a text editor, only with the specific program. More specifically, a binary file is designed to fulfill the following requirements:

- To avoid unexpected modifications of the file contents.

- To preserve file content consistency.
- To optimize how contents are stored and processed.

9.4.3 Application

The main application of binary-based files is the serialization of program data in a non-modifiable and customized structured format. On the other hand, it is possible to easily read a binary file using different processing strategies. However, when reading/writing binary-based files, it is also convenient to understand the underlying semantics of the structure (if any). For instance, first 4 bytes corresponds to an integer number, next 255 bytes to a string, etc.

9.4.4 Binary-based files in the Python programming language

The use of binary files in Python requires the use of a flag (b), see *Table 22*, for opening the file. Then, it is possible to read, write, seek a position, etc. Although, the management of bytes at a low level is possible, the `pickle` module in Python has been developed to ease the interaction with binary data through two main operations (dump and load).

9.5 Interesting links

- https://docs.python.org/3/tutorial/inputoutput.html
- https://python-reference.readthedocs.io/en/latest/docs/file/
- https://realpython.com/read-write-files-python/

9.6 Quiz

1. **What is the return value of the method "readlines"?**
   a. A list of characters.
   b. A string buffer.
   c. A list of strings.
   d. A list of bytes.

Answer: c

2. **Which of the following statements is used to open a file "messages.txt" for only writing in text format?**

   a. outfile = open("messages.txt", “w”)

   b. outfile = open("messages.txt", “r”)

   c. outfile = open("messages.txt", “wb”)

   d. outfile = open("messages.txt", “rw”)

Answer: a

3. **What is the sys.stdout?**

   a. The standard input file.

   b. The standard input/output file.

   c. The standard error file.

   d. The standard output file.

Answer: d

10 TYPES OF ERRORS, TESTING, DEBUGGING AND ERROR HANDLING

10.1 Introduction

When creating a software program applying a software development lifecycle, our main objective is to "*do the right thing right*". In other words, implement what is expected by the user/client and in a correct manner. *Do the thing right* means that our system is verified, while *Do the right thing* means that our system is validated. However, software programs sometimes fail due to different factors. In general, we can speak about the concept of "*software bug*". The main questions are:

What is a software bug?

It is a defect, error, failure or fault that has been introduced in our program and produces an incorrect or unexpected result.

Why a software can contain bugs?

Software engineers and programmers are humans. They are not perfect, and they make mistakes so, when we are programming, we can introduce defects because of a lack of understanding, communication, experience, time and many other factors.

Timeframe and last-minute changes. Usually the time to perform some task is underestimated. No realistic development times or lack of experience can lead us to likely introduce errors in our code. Moreover, before delivering some software, there is a bad practice of quickly introduce new features without following our software development process. This again will likely imply an incorrect behavior of our software.

Poor coding practices and design/testing skills. This is mainly related to our way of facing a programming problem. Lack of experience in the programming language, use of the tools, etc. will lead us to produce low quality code and, again, defects will likely emerge.

So, if a program is not free of bugs…we need methods and techniques to be able to find and fix bugs.

10.2 Types of errors

In order to provide high-quality software, the detection of errors is key activity. So, the first step lies on classifying the type of errors we can find when developing a software system. A common taxonomy of errors classifies them into these categories:

- Type of error.
    - Syntax error, due to a wrong application of the language grammar
    - Semantic error, due to a wrong interpretation of the language semantics.
- Time.
    - Compilation error, they occur in compilation time.
    - Runtime error, they occur in runtime.
- Nature[21].

[21] "Design by Contract". http://se.inf.ethz.ch/~meyer/publications/old/dbc_chapter.pdf

- Exception, "*the occurrence of an abnormal condition during the execution of a software element*". For instance, an unexpected network error.
- Failure, *"the inability of a software element to satisfy its purpose"*. Failures will likely lead us to an exception. For instance, an out of memory issue or a time out.
- Defect (bug), *"the presence in the software of some element not satisfying its specification."* For instance, when coding some operation, we use a wrong operator (e.g. * instead of +).

To prevent this type of errors there are different tools (the own compiler) and techniques such as defensive programming, contract-based designed, etc. However, in this chapter, we will focus on applying the principles of Test-Driven Development (TDD) [7] [8] to ensure the reliability and quality of our source code. This means that once we have a specification, we first start writing unit/functional tests to later code the functionality. In this manner, it is possible to have a measure of the working functionalities and to ensure the code still works after some change. The very first step to put this approach in action is to know the basic testing techniques such as unit tests.

10.3 Testing techniques

10.3.1 Problem statement

Software is not free of bugs so, methods for testing are completely necessary to ensure the reliability of a software system minimizing. Many testing techniques and methods are available [9] [10] to verify and validate the capabilities of a system. Regarding the type of methods to test, we can find a first classification:

- **White-box** testing consisting in ensuring the proper behavior of the internal structure of the code. This type of testing serves us to check the control flow statements, the different branches of execution, etc. As an example, if we have a decision point (an if-else statement), we must ensure that tests run over both block of statements creating test cases with specific inputs. In this manner, it is also possible to establish a degree of statement coverage. The techniques to perform these tests are usually unit tests.
- **Black-box** testing consisting in ensuring the functionalities of a software system as a black box (something that takes some input and generates some output). Examples of black-box testing are the boundary value analysis checking the domain and range of

functionalities. Again, some of the techniques to perform this type of testing are unit tests.

Regarding the type of technique, there are again many techniques available that can be classified as follows. The techniques mainly take a processing unit (a function, a process, etc.), an aspect (a functional or non-functional requirement, etc.) and a time to run (building, integration, deployment, etc.). Then, these techniques can be implemented with different technological frameworks, e.g. Junit in Java or `unittest` in Python.

- Unit test
- Integration test
- System test (functional, performance, security, usability, accessibility etc.)
- Acceptance test, Deployment test, Regression test, etc.

As a remark, many methodologies are now focusing on putting tests first like eXtremme Programming or Agile. Furthermore, good programming practices [11] such as design patterns or architectural frameworks also help to code software that can be easily verified.

10.3.2 Concept

With the aim of ensuring the reliability and quality of our software, testing methods and techniques must be applied to verify and validate the different requirements of our specification.

As stated in the book "The Practical Testing Pyramid"[22], it is important to provide strategies for testing and with different levels of abstractness. However, the cornerstone relies on group software tests and to mainly address two challenges: **automation** and **optimization** of the testing processes.

The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups.

…

It shows which kinds of tests you should be looking for in the different levels of the pyramid and gives practical examples on how these can be implemented

[22] https://martinfowler.com/articles/practical-test-pyramid.html

To implement a testing environment, we will follow the next process:

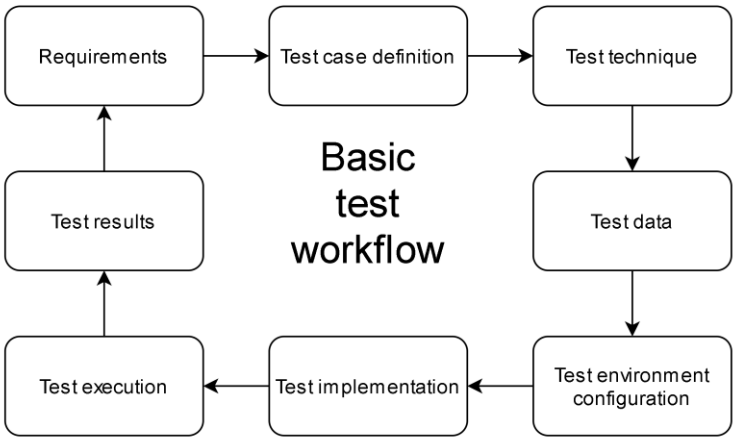


Figure 24 Basic test workflow.

10.3.3 Application

The verification and validation of software systems has gained a lot of attraction. There are many techniques and methods. It is a complex but necessary process. In the context of this book, testing is a method to ensure the reliability of our software. At least, unit tests must be provided to ensure the proper behavior of the different functionalities. Other more advanced testing techniques or methods could be applied to integrate our software in a higher degree system. However, the programs expected as an outcome of this first contact with programming are not so complex, so, learning about the principles of test-driven development is good enough at this stage.

10.3.4 Testing in the Python programming language

The basic library to perform unit tests in Python is the `unittest` module (inspired by the Junit framework for Java). This library includes methods to define test cases through the extension of a base class (TestCase).

According to the official documentation, the following concepts are available in this module:

- test fixture: *it represents the preparation needed to perform one or more tests, and any associated cleanup actions.*
- test case: *it is the individual unit of testing. It checks for a specific response to a particular set of inputs*
- test suite: *it is a collection of test cases, test suites, or both.*

- test runner: *it is a component which orchestrates the execution of tests and provides the outcome to the user.*

Some of the main methods to define the process of testing are defined in the *TestCase* class. Following the official documentation, the next type of assertions (check of a condition) can be defined.

Table 23 TestCase methods and checkings.

Method	**Check**
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

As an example, design and implement a Python program that provides these 4 functions:

- A function that given a filename, it reads all contents returning a list of strings (each string represents a line). Use the string method "strip" to remove the trailing character "\n" at the end of the line.
    - Input file content:

```
Hello Mary
Hello Peter
Mary
```

    - Expected output: ["Hello Mary","Hello Peter","Mary"]

- A function that given a list of strings, it calculates the frequency of each word returning a dictionary of words and frequencies.
    - Input: ["Hello Mary","Hello Peter","Mary"]
    - Expected output: {'Hello': 2, 'Mary': 2, 'Peter': 1}
- A function that given a dictionary of word-frequency, it returns a sorted list of tuples (sorted descending by frequency).
    - Input: {'Hello': 2, 'Mary': 2, 'Peter': 1}
    - Expected output: [('Hello', 2), ('Mary', 2), ('Peter', 1)]

```
import unittest

def load_file(filename):
    content = []
    if filename:
        file = open(filename,"r")
        content = file.readlines()
        file.close()
        content = [line.strip() for line in conte
nt]
    return content

def calculate_frequencies (contents):
    word_frequencies = {}
    if contents:
        for line in contents:
            for word in line.split(" "):
                if word in word_frequencies:
                    word_frequencies[word] += 1
                else:
                    word_frequencies[word] = 1
    return word_frequencies
```

```
  def sort_word_frequencies(word_frequencies):
      sorted_frequencies = []
      if word_frequencies:
          sorted_frequencies = sorted(word_frequenc
ies.items(),key=lambda k: k[1], reverse=True)
      return sorted_frequencies

  class AppTest(unittest.TestCase):

      def test_load_file(self):
          contents = load_file("input.txt")
          self.assertListEqual(["Hello Mary","Hello
 Peter","Mary"], contents)

      def test_calculate_frequencies(self):
          contents = ["Hello Mary","Hello Peter","M
ary"]
          word_frequencies = calculate_frequencies(
contents)
          self.assertDictEqual({'Hello': 2, 'Mary':
 2, 'Peter': 1},word_frequencies)

      def test_sort_word_frequencies(self):
          word_frequencies = {'Peter': 1, 'Hello':
2, 'Mary': 2}
          word_frequencies_sorted = sort_word_frequ
encies(word_frequencies)
          self.assertListEqual([('Hello', 2), ('Mar
y', 2), ('Peter', 1)], word_frequencies_sorted)

  if __name__=="__main__":
      unittest.main()
```

10.4 QAOps: Automated Quality Testing for DevOps

The increasing adoption of "Development" and "Operations" (DevOps) [12] [13] as methodology to develop software-based systems (Plan→Code→Build→Test→Release→Deploy→Operate→Monitor→Plan→…) has implied the identification of some impediments [14] and the re-definition of some roles such as the responsibilities of Quality Assessment (QA) teams.

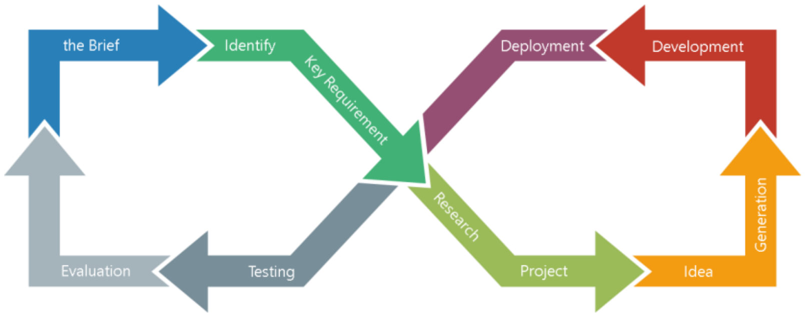


Figure 25 DevOps process.

DevOps is not just a technical approach but a philosophy, a culture of how to address the major challenge of delivering timely and reliable software products and services. In general, this may impact in the number of releases, the time to market and the lowering of failure rates. DevOps processes are based on the 6 C's (continuous): planning, development, integration, testing, monitoring and delivery. In each of these subprocesses, testing play a major role:

- Continuous **planning and development**: all development actors may collaborate to define a clear plan for releasing new features.
- Continuous **integration**: one of the main objectives of this stage is the identification of bugs improving software quality.
- Continuous **testing**: automation of tests along the process lifecycle identifying the methods, techniques and tools to automate and optimize the verification and validation processes.
- Continuous **monitoring**: ensuring how the results are reported.
- Continuous **delivery**: again, automation of the stages of building and testing.

- Continuous **deployment**: preparing the new built passing a first pre-test phase

In order to enable engineering environments that fulfill these 6 C's, it is completely required the coordination and collaboration of different roles and teams. It is also remarkable the role of **testing** as a primitive to accept changes and to ensure the quality of new developments.

Many development methodologies such as Agile or TDD [7] [8]/BDD [15] (Test Driven Development/Behavior-Driven Development) focuses on the concept of "test-first". However, in the frame of DevOps [16], these methodologies represent a way of working while DevOps, as stated before, it is a kind of philosophy/culture. In the same manner, there are many methodologies and testing techniques with some support in IT tools. So, the relevance of testing in DevOps relies on addressing challenges in the next aspects:

- Definition of testing strategies: types and methods.
- Implementation of the testing strategies through standardized toolchain.
- Definition of quality metrics.

In the first case, it is possible to find different types of testing: unit, functional, system, exploratory, regression, compatibility, security, acceptance, etc. The implementation of these techniques can be done with different technologies and managed by tools

The combination of "Development" and "Operations" through the automation of a common lifecycle: implementation of a source code piece, building and deploying a solution in a testing environment, execution of test cases and, finally, deployment into production, focuses on two factors: ability to automate tasks and ability to ensure what is being added pass all required tests. In this context, the notion of continuous development, integration and delivery possess some requirements in the lifecycle of a product or a service. In general, everything should be standardized (e.g. dependency management) and automated (e.g. managed by specific tools).

In order to evaluate the notion of automation and quality in a DevOps chain, it is necessary to redefine the responsibilities of a QA team. Commonly, a QA team creates a testing environment in which they deploy the updated version of the product or a service to run functional and regressions tests. Since time is a critical variable for many DevOps stages (e.g. time to build, time to delivery, time to run test cases, time to deploy, mean time between failures, time), this old-fashioned of deployment management is directly impacted by the time to perform these tasks. Shifting the development lifecycle to a DevOps methodology requires that QA is also shifted to align their efforts to the DevOps stages. They need to ensure the automation of test cases execution (configuration, data management, execution, clean up, etc.). They also need to design standardized testing environments and ensure every task is executed under the DevOps continuous integration lifecycle, see the next Figure about the stages of DevOps. This situation implies that the work of a QA team is more focused now on the configuration and coordination within the whole lifecycle than in the own testing needs.

That is why, it is possible to distinguish some key points to improve QA practices:

- Creation of a DevOps testing strategy: this activity requires the collaboration and coordination with the development team. The objective is the identification of critical parts for specific builds of a subsystem or component.
- Definition of a test execution plan and contexts: a test execution plan shall be defined to optimize the time to execute test cases. It is necessary to define different contexts depending on the subsystem or component. Although the testing of the whole system is necessary, a good test execution plan can optimize the time to run tests ensuring a degree a coverage.
- Definition and generation of test cases: building on the previous steps, test cases must be defined considering their nature (e.g. unit, functional, system, regression, etc.) and the implications on the whole system.
- Selection of automation techniques: DevOps is all about automation. The proper identification and selection of automation techniques (e.g. cross-platform testing), etc. is critical to ensure the quality of the system under development.
- Implementation of automation techniques through a well-defined toolchain: depending on the testing and automation techniques, an identification and selection of the technological support is completely relevant. On the one hand, tools for the management of test execution and, on the other hand, technology to run everything (e.g. use of microservices architectures).

- Configuration management: since a new good number of tools is introduced and the target execution environments of the product or service may vary a lot, it is necessary to enable mechanisms that can ease the automatic configuration of the testing process.

- Reporting strategy: once everything is up and running and results of the testing process are generated, it is necessary to identify and establish a strategy to report those issues that may be found after running the process. For instance, what is a blocking a problem? How a critical software bug is reported? Can all of them be automated?

- Other stages may cover metric definition and evolution.

To implement these strategies there are many tools in the market:

- Landscape of DevOps tools[23].

- Azure DevOps[24].

10.5 Debugging

The best definition of the relationship between coding and debugging was made by Dijkstra, a pioneer computer scientist in many research fields of computer science.

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."

And there is another related quote:

"If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with."

It seems simple: "The art of finding and fixing software bugs"

In general, IDEs provide capabilities for debugging. When a program is under a debugging process, it is possible to control how the program runs. This means that it is possible to add breakpoints, to execute the source code statements step by step and, to watch the status of our variables. So, let's define the main concepts:

[23] https://noise.fresh8.co/continuous-delivery-tool-landscape-cac734d4b5e7

[24]https://docs.microsoft.com/en-us/azure/devops/test/overview?view=azure-devops

- **Breakpoint**: It is an intentional pause in a program. A break point is established to stop the program in a certain line and inspecting the status of the variables or continue the execution step by step. In general, breakpoints are situated in those lines of code that contain some suspect statement, there is an incorrect behavior. As a final comment, it is also possible to use more advanced features such as conditional breakpoints

- **Execution step by step**. This means that the program is executed line by line or sentence by sentence, we trace our program. In this way, it is possible to easily inspect all variables in some scope and check the behavior of a sentence or a method.

Here, it is also possible to defined different modes of tracing a program:

  - Step-into: when a method is going to be invoked and we want to debug the code of the method.
  - Step-over: when a method is going to be invoked and we are not interested in the implementation details of the method.
  - Step return: we do not want to execute the method step by step and we run the entire method until the return statement.
  - Resume run the program "normally".

- **Call stack**. This is the stack of subroutines (methods) that have been invoked.

10.6 Error handling

10.6.1 Problem statement

As it has previously introduced, there are different types of errors. In some cases, technology (e.g. the selected programming language) will allow us to easily solve some of them, e.g. compiling errors in strong and static typing programming languages like Java. Testing techniques can also help us to verify and validate the behavior of our software system. However, other unexpected errors are hard to find due to its nature and we will need methods to manage and to react to these situations. For instance, a network timeout can happen when we are consuming web services. So, how can we handle these unexpected situations?

10.6.2 Concept

Exceptions, runtime unexpected errors, are the type of problem that needs some handling mechanism. To do so, modern programming languages such as Python or Java include a mechanism, a constructor, to handle exceptions.

An exception is an error raised in execution time leading our program to an uncertain state.

In order to handle exceptions, it is possible to protect those parts of code candidates to raise an error. E.g. opening a non-existing file, accessing to a network service that is not available, an out of memory exception, etc. To manage exceptions, we will find three main parts:

- Protected code (try-block). Usually, this is the part of our source code that could generate an exception.
- Exception handler (exception-block). This is a part of our code to handle the error and to make some decision when the error is raised. It is possible to define handlers for each type of exception.
- Finally block. This is a part of our code that will be executed in any case to clean up the execution.

So, the next questions are:

- *Which parts of the source code should be protected?*

In this case, we must evaluate which parts are more suspicious to raise an exception. To do that, we can check the methods we are using to find out if they will raise an exception. Furthermore, there are usual suspects like operations on files or databases, connections to external services, etc. that we have to carefully manage.

- *Where do we need to handle the exception?*

There is no rule of thumb. However, exceptions must be caught in the place they occur and, then depending on the type of error we can follow two main strategies:

  - Handle the exception in place and continue the normal execution of the program but without results in the operation.
  - Catch the exception and raise again to be managed in other part of the source code. This is a common practice. Exceptions are caught where they are raised but, the real

handling is done in other part of the source code that is in charge of reacting and making decisions about errors.

Exceptions and their management come with some computational cost (e.g. time to create the exceptions, use of memory, execution of the handlers, etc.). So, it is necessary to carefully handle how exceptions are created, handled and thrown. In some cases, a better and simple option lies on returning some empty element or an error code.

10.6.3 Application

The use of exception is a common practice to handle error in software programs. In order to protect our code of unexpected errors, we can enclose suspected source code parts within a try-block.

10.6.4 Error handling in the Python programming language

In Python, there is a compound statement to manage exceptions.

```
  try_stmt  ::=  try1_stmt | try2_stmt
  try1_stmt ::=  "try" ":" suite
                 ("except"   [expression   ["as"
identifier]] ":" suite)+
                 ["else" ":" suite]
                 ["finally" ":" suite]
  try2_stmt ::=  "try" ":" suite
               "finally" ":" suite
```

This type of statement has different blocks:

- The try-block in which the source code is protected. If an exception is raised, the handlers, specified in the except clauses, will be activated.
- The except-block in which one or more exception handlers can be specific. If there is no exception, these handlers will be skipped and the finally-block will be executed. If there is an exception, the system will try to match the type of exception with the handler to execute the proper source code to manage the exception. If there is no except-block expression that matches the type of exception, the finally-block will be executed.

Inheritance between exceptions must be carefully considered when specifying exception handlers.

- The finally-block in which some actions are coded to be executed in any case.

Regarding the information about exceptions, when exception is raised some complementary information is generated. This information can be accessed can be accessed via sys.exc_info(). sys.exc_info(). It returns a 3-tuple consisting of the exception class, the exception instance and a traceback object. Take to look to the next basic examples to handle exceptions in the Python programming language.

- General exception handler.

```
try:
  2/0
except:
  print("This is the general exception handler")
```

- Specific exception handler.

```
try:
  2/0
except ZeroDivisionError as zde:
  print("This is the Zero division error handler")
except:
  print("This is the general exception handler")
```

- Specific exception handler with a finally block.

```
try:
  2/0
except ZeroDivisionError as zde:
  print("This is the Zero division error handler")
finally:
  print("Finally block")
```

- Specific exception handler getting information with the traceback module.

```
import traceback
try:
  2/0
except ZeroDivisionError as zde:
  traceback.print_exc()
finally:
  print("Finally block")
```

- Specific exception handler getting information with the sys module.

```
import sys
try:
  2/0
except ZeroDivisionError:
  print("Exception: ",sys.exc_info()[0],"occured.")
finally:
  print("Finally block")
```

10.7 Interesting links

- https://docs.python.org/3/library/unittest.html
- https://docs.python.org/3/reference/compound_stmts.html#grammar-token-try-stmt
- https://docs.python.org/3/reference/executionmodel.html#exceptions

11 TIPS PYTHON BUILTIN FUNCTIONS

11.1 Context

Following, we present some of the main built-in functions in Python. These functions are available at any time for any type of object in Python. However, the behavior can change depending on the input parameters.

First, let's discover which built-in functions are available. In general, they can be classified into some categories:

- Input/Output: input/print
- Basic operations: len
- Basic math functions: abs, pow
- Aggregation operators: min, max, sum
- Type casting: str, int, float, complex, bool
- Object identification: id, hash
- Help functions: help, dir, etc.

The rest of functions will be in our interest in other cases.

```
dir(__builtin__)
help(print)
```

11.2 `print` function

- Grammar: `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`
- Print objects to the text stream file, separated by sep and followed by end. sep, end, file and flush, if present, must be given as keyword arguments.
- Source: https://docs.python.org/3/library/functions.html#print

```
  print("Hello, no return", end="")
  print("...in the same line...")
  print("New line")
  #Including characters such as tabulator and retur
n
  print("\t This is a tabulator with two returns ex
tra \n\n ")
  print("other line")
```

```
  print("Hello", "I am Paul", "and I am", str(30), 
"years old.")
  #Separating input tokens with a hash
  print("Hello", "I am Paul", "and I am", str(30), 
"years old.", sep="#")
```

```
  #Formmatting the output with str.format
  #Naming parameters
  print("Hello I am {name} and I am {age} years old
.".format(name="Paul",age=30))
  #Positional parameters
  print("Hello I am {} and I am {} years old.".form
at("Paul",30))
```

```
  #Formatting numbers
  #%[flags][width][.precision]type
```

```
print("%6.3f" % (3.14516))
print("{0:8.3f}".format(3567.14516))
```

11.3 `input` function

- Grammar: `input([prompt])`
- If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised.
- Source: https://docs.python.org/3/library/functions.html#input

```
a = input(">>>>")
print(a)
```

11.4 `str` function

- Grammar: `str(object=b'', encoding='utf-8', errors='strict')`
- Return a string version of object. If object is not provided, returns the empty string. Otherwise, the behavior of str() depends on whether encoding or errors is given, as follows.
- Source: https://docs.python.org/3/library/stdtypes.html#str

```
print(str(4))
print(str([1,2]))
print(str())
print(str(True))
print(str(None))
```

11.5 `len` function

- Grammar: `len(s)`
- Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).
- Source: https://docs.python.org/3/library/functions.html#len

```
len("Hello")
len(4)
```

11.6 `abs` function

- Grammar: `abs(s)`
- Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.
- Source: https://docs.python.org/3/library/functions.html#abs

```
abs(-5)
```

11.7 `max` function

- Grammar: `max(arg1, arg2, *args[, key])`
- Return the largest item in an iterable or the largest of two or more arguments.
- Source: https://docs.python.org/3/library/functions.html#max

```
max(2,3,4)
```

11.8 `min` function

- Grammar: `min(arg1, arg2, *args[, key])`
- Return the smallest item in an iterable or the smallest of two or more arguments.
- Source: https://docs.python.org/3/library/functions.html#min

```
min(1,2,3)
```

11.9 `pow` function

- Grammar: `pow(base, exp[, mod])`
- Return base to the power exp; if mod is present, return base to the power exp, modulo mod (computed more efficiently than pow(base, exp) % mod). The two-argument form pow(base, exp) is equivalent to using the power operator: base**exp.
- Source: https://docs.python.org/3/library/functions.html#pow

```
pow(2,3)
```

11.10 `sum` function

- Grammar: `sum(iterable, /, start=0)`
- Sums start and the items of an iterable from left to right and returns the total. The iterable's items are normally numbers, and the start value is not allowed to be a string.
- Source: https://docs.python.org/3/library/functions.html#sum

```
sum([1,2,3])
```

11.11 `float` function

- Grammar: `float([s])`
- Return a floating point number constructed from a number or string s.
- Source: https://docs.python.org/3/library/functions.html#float

```
float("5.2")
```

11.12 `int` function

- Grammar: `int([s])`
- Return an integer object constructed from a number or string x, or return 0 if no arguments are given.
- Source: https://docs.python.org/3/library/functions.html#int

```
int("1")
```

11.13 `complex` function

- Grammar: `complex([real[, imag]])`
- Return a complex number with the value real + imag*1j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns 0j.
- Source: https://docs.python.org/3/library/functions.html#compl ex

```
complex(1+2j)
```

11.14 `bool` function

- Grammar: `bool([x])`
- Return a Boolean value, i.e. one of True or False. x is converted using the standard truth testing procedure. If x is false or omitted, this returns False; otherwise it returns True.
- Source: https://docs.python.org/3/library/functions.html#bool

11.15 `id` function

- Grammar: `id(object)`
- Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.
- Source: https://docs.python.org/3/library/functions.html#id

```
a = 2

id(a)
```

11.16 `hash` function

- Grammar: `hash(object)`
- Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).
- Source: https://docs.python.org/3/library/functions.html#hash

```
text = "This is an string"

hash(text)
```

11.17 Interesting links

- https://docs.python.org/3/library/functions.html

12 LAB 1-SETTING UP THE ENVIRONMENT AND GETTING STARTED WITH PROGRAMMING ELEMENTS IN PYTHON

In this first chapter, we will review some of the main Python programming elements. In order to have an official reference of the syntax and semantics of the language, please, always refer to the official documentation: "The Python Language Reference".

12.1 Setting up the environment

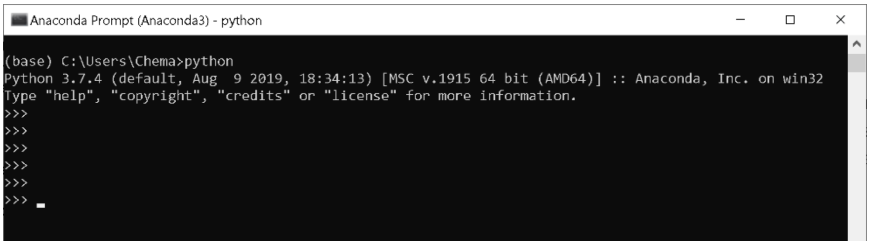
Once we know the process of compiling and execution of a Python program (theoretically speaking), it is time to start coding simple programs and setting up our development environment. We need to configure the next things in our machine:

- A Python compiler and interpreter.
- An integrated development environment (IDE). This is **not completely necessary** since we could write source code in any text-processor such as Notepad++. However, an IDE eases the tasks of: project management, program coding (syntax highlighting,

debugging, etc.) and program construction and execution.

To prepare this environment, we have different options:

1. **Manual Python local installation** (Python 3 + Spyder). Make an installation of the Python compiler and interpreter (isolated). Afterwards, we download and installed an IDE like Spyder or Pycharm.
   - To do so, we have select our platform and the last version of the Python distribution in the official web page.
     - Spyder or
     - PyCharm.
   - To check the installation, we can open a console a type "python".
   - If we need to install extra packages, we can use the pip (Python Package Index) utility as a command line tool (here, it is showed the Python interpreter within Anaconda).

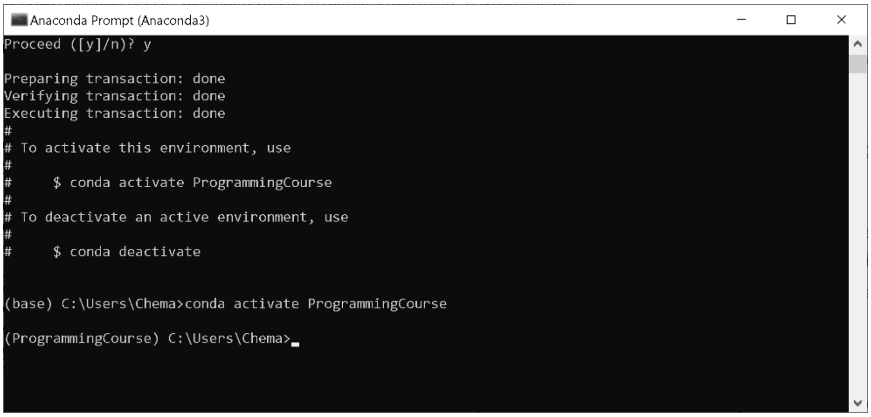


```
(base) C:\Users\Chema>python
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>>
>>>
>>>
>>>
```

Figure 26 Python running from the console.

2. **Managed Python installation** (Anaconda, recommended). Anaconda is a meta-manager of Python tools that already includes the Python interpreter, the IDE, etc. You can download Anaconda from here, depending on your machine (~700MB).
   - We can prepare an isolated conda environment (by default we have the base environment) for our work and, thus, it is possible to manage all dependencies we have. To do so, we have to open the Anaconda console (it is also possible through the Anaconda graphical user interface).

```
conda create --name ProgrammingCourse
conda activate ProgrammingCourse
```


Anaconda Prompt (Anaconda3)

```
Proceed ([y]/n)? y

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate ProgrammingCourse
#
# To deactivate an active environment, use
#
#     $ conda deactivate


(base) C:\Users\Chema>conda activate ProgrammingCourse

(ProgrammingCourse) C:\Users\Chema>_
```

Figure 27 Anaconda console and commands to create a new environment.

In this manner, we can already run from the console any Python program through the interpreter executing one of the following commands:

```
python #to open the interpreter, Ctrl+d to exit
python file.py #to run a script
```

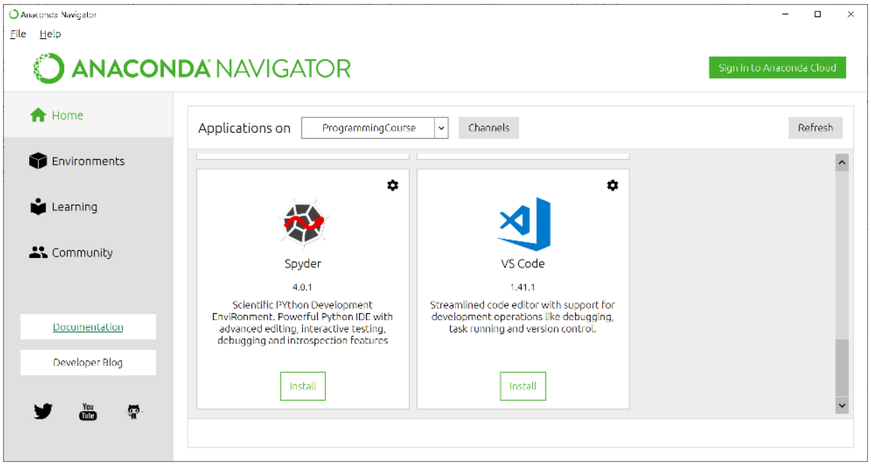
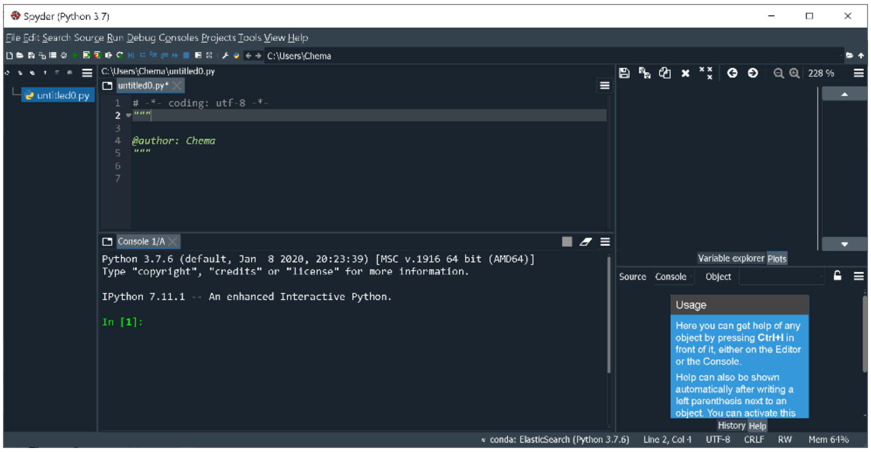

Anaconda Navigator
File Help
ANACONDA NAVIGATOR
Sign in to Anaconda Cloud
Home
Environments
Learning
Community
Documentation
Developer Blog
Applications on
ProgrammingCourse
Channels
Refresh
Spyder
4.0.1
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Install
VS Code
1.41.1
Streamlined code editor with support for development operations like debugging, task running and version control.
Install

Figure 28 Anaconda navigator.

Even better, we can now launch the Anaconda Navigator, select and launch Spyder and we have everything ready to code with an IDE (the first time, it will take some time to configure the environment). It is important to remark that inside Spyder we have as interpreter IPython (Interactive Python), an enhanced version of the Python interpreter.


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\Chema
untitled0.py
C:\Users\Chema\untitled0.py
untitled0.py*
Console 1/A
228 %
Variable explorer Plots
Source Console Object
Usage
History Help
conda: ElasticSearch (Python 3.7.6) Line 2, Col 4 UTF-8 CRLF RW Mem 64%

```
# -*- coding: utf-8 -*-
"""

@author: Chema
"""
```

```
Python 3.7.6 (default, Jan  8 2020, 20:23:39) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.11.1 -- An enhanced Interactive Python.

In [1]:
```

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this

Figure 29 Spyder IDE.

and now, **we are ready to code!**

12.2 Writing Python code

12.2.1The entry point: the main function

In Python, this is the `main` function. To define, the main function, we declare the next statement:

```
# Other elements some imports and functions

if __name__=="__main__"
  #sententences
```

In general, a Python program will have the next structure:

```
# Documentation

# import sections: see the next section

# function definitions

# main function

if __name__=="__main__"
  #sententences
```

12.2.2 Code layout (PEP8)

The following guidelines has been summarized from the PEP8 official document.

- **Indentation**. Use 4 spaces per indentation level.
- **Line size**. Limit all lines to a maximum of 79 characters.
- **Long blocks**. For flowing long blocks of text with fewer structural restrictions (docstrings or comments), the line length should be limited to 72 characters.
- Blocks

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

- **Blank Lines.**
- **Encoding**. Code in the core Python distribution should always use UTF-8 (or ASCII in Python 2).
- **Imports**. Imports should usually be on separate lines:
    - Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
    - Imports should be grouped in the following order:
        1. Standard library imports.
        2. Related third party imports.
        3. Local application/library specific imports. You should put a blank line between each group of imports.
    - Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):

```
"""This is the example module.


This module does stuff.
"""
```

```
__all__ = ['a', 'b', 'c']
__version__ = '0.1'
__author__ = 'Cardinal Biggles'
```

12.3 Exercises

12.3.1 Important read

So far, we know the basic elements of programming to mainly build expressions that can be computed. In order to make the exercises, we will need the use of some functions, so you have to *make an act of faith* and just use them (mechanically) as it is indicated.

- string literal: it is a literal value enclosed between " ". We can concatenate string values with the operator +.
- `input(message)`: it prints out in the standard output the value of message (a string), asks the user to input some value and returns a string representation of the input.
  - `input("What is your name?")`
  - `input("What is your age?")`
- `print(element)`: it prints out in the standard output (the console) the string version of the element. In the case of integer numbers, we must first convert the number to string with the next function.
  - `print("Hello")`→Hello
  - `print (str(5))`→ 5
- `str(element)`: returns a string representation of the argument passed as a parament.
  - `str(5)` → "5"
  - `str(2+3)` →"5"
- `len(element)`: returns the length of the element passed as an argument.
  - *len("Hello")* → 5
- `int(string_literal)`: tries to convert a string literal to an integer number. It can raise a conversion error.
  - `age = int(input ("What is your age?"))`

- `float(string_literal)`: tries to convert a string literal to float number. It can raise a conversion error.
  - `t = float(input ("What is the outside temperature?"))`

Tip

We will make use of the function print. It has the following syntax:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, fl
ush=False)
```

- Objects to be printed using a separator and ending with "enter".
- The file in which objects are printed is the default output (console).
- We can use % to concatenate strings.

We can format a bit the output as follows:

- Creating a template in which the parameters are enclosed between `{}`:`print("Hello {} ".format("Paul"))`
- For floating numbers, establishing the number of digits in the integer (n) and decimal part (m): %n.mf
  - `%[flags][width][.precision]type`
- Learn more in this tutorial.

```
print("Hello {} ".format("Paul"))
print("Hello {}, you are {} years old.".format("P
aul", 30))
print ("This is a float number {0:8.5f}".format((
1/3)))
print ("%.5f" % (1/3))
b = 1/3
```

12.3.2 List of exercises

1. **Write a program that declares a variable and assigns the value 10, then print out the value on the screen.**

   - Input: 10

- Expected output:

```
The value is: 10
```

```
a = 10
print("The value is: "+str(a))
```

2. **Write a program that declares and adds two integer variables (2, 10), then print out the result on the screen.**

- Input: 2 and 10
- Expected output:

```
The result is: 12
```

```
a = 2
b = 10
result = a + b
print("The result is: "+str(result))
```

3. **Write a program that declares and multiplies two integer variables (4, 15), then print out the result on the screen.**

- Input: 4 and 15
- Expected output:

```
The result is: 60
```

```
a = 4
b = 15
result = a * b
print("The result is: "+str(result))
```

4. **Write a program that asks the user to input an integer value, then print out the input value multiplied by 2.**

- Input: an integer number, 4

- Expected output:

```
The result is: 8
```

```
value = int(input("Please, enter a number..."))
result = value * 2
print("The result is: "+str(result))
```

5. **Check the type of different variables using the function `type`.**

- Input:

```
a = 2
b = 3
c = a
char = 'a'
string_value = "hello"
bool_value = True
float_value = 2.3
```

- Expected output:

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

```
a = 2
b = 3
c = a
char = 'a'
string_value = "hello"
bool_value = True
float_value = 2.3
print(type(a))
print(type(b))
print(type(c))
print(type(char))
print(type(string_value))
print(type(bool_value))
print(type(float_value))
```

6. **Write a program that asks your name and prints out "Hello " #name#**

   - Input: Paul
   - Expected output:

```
Hello: Paul
```

```
name = input("What is your name? ")
print ("Hello: "+name)
```

7. **Write a program that asks your name and prints out the length of the name.**

   - Input: Paul
   - Expected output:

```
Your name: Paul has 4 characters.
```

```
name = input("What is your name? ")
l_name = len(name)
print ("Your name: "+name+" has "+str(l_name)+"
characters.")
```

8. **Write a program that asks your age and prints out how many years are until 100.**

   - Input: 25
   - Expected output:

```
Until 100 there are 75 years.
```

```
  age = int(input("What is your age? "))
  years_until_100 = 100-age
  print ("Until 100 there are "+str(years_until_10
0)+" years.")
```

9. **Write a program that asks the year of birth and prints out your age.**

- Input: 1983
- Expected output:

```
  You are 37 years old.
```

```
  year_birth = int(input("Introduce the year of bi
rth..."))
  age = 2020-year_birth
  print ("You are "+str(age)+" years old.")
```

10. **Write a program that asks the temperature in Kelvin degress and prints out the temperature in Celsius. T(°C)=T(K)−273.15**

- Input: 300
- Expected output:

```
  The temperature in Celsius is: 26.850000000000023
```

```
  t_kelvin = float(input("Introduce the temperatur
e in Kelvin..."))
  t_celsius = t_kelvin-273.15
  print ("The temperature in Celsius is: "+str(t_c
elsius))
```

11. **Write a program that asks a number and prints True if the number is odd, and False otherwise.**

- Input: 2020
- Expected output:

```
   The number 2020 is odd: True
```

- Input: 2019
- Expected output:

```
   The number 2019 is odd: False
```

```
  number = int(input("Introduce an integer number:
 "))
  is_even = (number % 2 == 0)
  print("The number "+str(number)+" is odd: "+str(
is_even))
```

12. **Write a program that asks for a year and prints True if the year is a leap year, and False otherwise. In the Gregorian calendar, a leap year follows these rules:**

    - The year can be evenly divided by 4;
    - If the year can be evenly divided by 100, it is NOT a leap year, unless;
    - The year is also evenly divisible by 400. Then it is a leap year.

- Input: 2020
- Expected output:

```
   The year 2020 is leap: True
```

- Input: 2019
- Expected output:

```
   The year 2019 is leap: False
```

```
  year = int(input("Introduce a year: "))
  is_leap_year = (year % 4 == 0)
  is_leap_year = is_leap_year and (year % 100 != 0
)
```

```
    is_leap_year = is_leap_year or (year % 400 == 0)
    print("The year "+str(year)+" is leap: "+str(is_
leap_year))
```

13. Write a program calculates the area of a triangle.

- Input: b = 2, h = 3
- Expected output:

```
The area is: 3.0
```

```
b = 2
h = 3
area = (b*h)/2


print("The area is: "+str(area))
```

14. Write a program to calculate the Euclidean distance between two points.

- Use the function `math.sqrt(x)` to calculate the root square of a number.

```
    Syntax: math.sqrt(x)

Parameter:
x is any number such that x>=0

Returns:
It returns the square root of the number
passed in the parameter.
```

- Input: (2.0, 3.0) (4.0, 5.0)
- Expected output:

```
2.8284271247461903
```

```
x1 = 2.0
y1 = 3.0
```

```
x2 = 4.0
y2 = 5.0
distance = math.sqrt((x1-x2)**2 + (y1-y2)**2)
distance
```

15. **Write a program that asks the temperature in Farenheit and prints out the temperature in Celsius. T(C)=(T(F)−32)*5/9**

- Input: 100
- Expected output:

```
Temperature in Celsius is 37.77777777777778
```

```
t = float(input("Temperature: "))
t_celsius = ((t - 32)*5)/9;
print("Temperature in Celsius is " + str(t_celsi
us));
```

16. **Write a program to swap the values of two variables.**

- Input: a = 2, b = 3
- Expected output:

```
a = 3 b = 2
```

```
a = 2
b = 3
print("a = "+str(a)+" b = "+str(b))
c = a
a = b
b = c
print("a = "+str(a)+" b = "+str(b))
```

17. **Write a program to calculate the thermal sensation.**

```
Ts=13,12+0,6215∗T−11,37∗V0,16+0,3965∗T∗V0,16
```

- Input: t = 10.0, v = 5.0
- Expected output:

```
Temperature = 10.0
Velocity = 5.0
Thermal sensation  = 9.755115709161835
```

```
  t = 10.0
  v = 5.0
  ts = 13.12 + 0.6215*t -
(11.37*(v** 0.16))+(0.3965*t*(v**0.16))
  print("Temperature = " + str(t))
  print("Velocity = " + str(v))
  print("Thermal sensation  = " + str(ts))
```

18. Write a program to calculate the values r and theta required to transform cartesian coordinates to polar.

   - r=(√x2+y2)
   - θ=tan−1(y/x)
- Input: x = 2, y = 3
- Expected output:

```
r     = 3.605551275463989
theta = 0.982793723247329
```

```
  x = 2.0
  y = 3.0
  r = math.sqrt(x*x + y*y)
  theta = math.atan2(y, x)
  print("r     = " + str(r))
  print("theta = " + str(theta))
```

19. Write a program that calculates the roots following the quadratic formula:

- o x=−b±b2−4ac√2a.
- o Use the function `math.sqrt(x)`

- Input: a = 4, b = 5, c = 1
- Expected output:

```
The roots are:
-0.25
-1.0
```

```
a = 4
b = 5
c = 1
bac=b**2-4*a*c
#This will not work if the roots are not real
root_positive = (-b + math.sqrt(bac))/(2*a)
root_negative = (-b - math.sqrt(bac))/(2*a)
print("The roots are:")
print (str(root_positive))
print (str(root_negative))
```

20. Write and check the difference between 1/3 and 1//3?

```
1/3
```

```
1//3
```

21. Write and check the result of comparing 5 == "5".

```
5=="5"
```

22. **A person drops a ball from a window. After 20.0 seconds, the ball hits the ground. What was the velocity of the ball the instant before hitting the ground?**

```
v = v0 + at
```

- Input: data from the statement
- Expected output:

```
Velocity: 196.0
```

```
t = 20
a = 9.8
vi = 0
vf = vi + a*t
print("Velocity: "+str(vf))
```

23. **Write a program to test the** augmented assignment. **What is the result of the following program?**

```
a = 2
a += 3
print(a)
a *= 2
print(a)
```

24. **Write a program to calculate the area of a circle of radius r.**
    1. **Use the constant** `math.pi`

- Input: r = 3
- Expected output:

```
28.274333882308138
```

```
r = 3
area = math.pi * r**2
```

```
print(area)
```

25. **Review and test the functions in the math module.** Check the next link: https://docs.python.org/3/library/math.html

```
#Ask Python about the module
dir(math)
```

```
help(math.sqrt)
```

12.4 Relevant resources

- Python official grammar
- PEP 8 Style Guide for Python Code
- https://edabit.com/challenges/python3
- https://exercism.io/tracks/python/exercises
- https://www.practicepython.org/

13 LAB 2-CONTROL FLOW: CONDITIONAL STATEMENTS

In this chapter, we propose and solve some exercises about conditional statements.

As a good programming practice, our test cases should ensure that all branches of the code are executed at least once.

13.1 List of exercises

1. **Write a program that asks the user for an integer numbers and prints out a message indicating whether the input is greater than 10.**

- Input: 15
- Expected output:

```
The input 15 is greater than 10.
```

```
a = int(input("-->"))
if a > 10:
    print("The input {} is greater than 10.".format
(a))
```

2. **Write a program that asks the user for two integer numbers and prints out the greater value.**

- Input: 4 and 15
- Expected output:

```
   The result is: 15
```

```
  a = int(input("-->"))

  b = int(input("-->"))

  if a > b:

    print("a is greater..."+str(a))

  else:

    print("b is greater..."+str(b))
```

3. **Modify the program in 2 to detect when the input values are equal.**

- Input: 4 and 4
- Expected output:

```
   Both values are equal.
```

```
  a = int(input("-->"))

  b = int(input("-->"))

  if a > b:

    print("a is greater..."+str(a))

  elif a < b:

    print("b is greater..."+str(b))

  else:

    print ("Both values are equal.")
```

4. **Write a program that asks the user for an integer number and prints the absolute value of the input value.** *Do not use the built-in function abs()*

- Input: -3
- Expected output:

```
The absolute value of -3 is 3.
```

```
value = int(input("Introduce an integer number: "
))
abs_value = value
if value < 0 :
  abs_value = value * -1
print("The absolute value of {} is {}.".format(va
lue, abs_value))
```

5. **Write a program that asks the user for two float numbers and prints out the greater value.**

- Input: 3.5, 4.5
- Expected output:

```
4.5 is greater than 3.5
```

```
a = float(input("Introduce the first number: "))
b = float(input("Introduce the second number: "))
if a < b: #Change to compare properly float numbe
rs abs(a - b) < 0.0000001
  print("{} is greater than {}".format(b,a))
else:
  print("{} is greater than {}".format(a,b))
```

6. **Write a program that asks the user for 3 integer numbers and prints out the greater value.**

- Input: 4, 6, 2

- Expected output:

```
6 is the greatest value.
```

```
a = int(input("Introduce the first number: "))
b = int(input("Introduce the second number: "))
c = int(input("Introduce the third number: "))

greatest = c

if a > b > c:
  greatest = a
elif b > a and b > c:
  greatest = b

print("{} is the greatest value.".format(greatest
))
```

7. **Write a program that asks for a year and prints True if the year is a leap year, and False otherwise. In the Gregorian calendar, a leap year follows these rules (see exercise 12 in Lab 2):**

   - The year can be evenly divided by 4;
   - If the year can be evenly divided by 100, it is NOT a leap year, unless;
   - The year is also evenly divisible by 400. Then it is a leap year.

   - Input: 2020
   - Expected output:

```
The year 2020 is leap.
```

- Input: 2019
- Expected output:

```
The year 2019 is not leap.
```

```
year = int(input("Introduce a year: "))
is_leap_year = (year % 4 == 0)
is_leap_year = is_leap_year and (year % 100 != 0)
is_leap_year = is_leap_year or (year % 400 == 0)
if is_leap_year:
  print("The year "+str(year)+" is leap.")
else:
  print("The year "+str(year)+" is leap.")
```

8. **Write a program that asks the user for an integer number and prints out whether or not it is an even number.**

- Input: 4
- Expected output:

```
4 is an even number.
```

```
value = int(input("Introduce a value: "))
if value % 2 == 0:
  print("{} is an even number.".format(value))
```

9. **Write a program that asks the user for a day number and prints the day name.**

- Input: between 1-7
- Expected output:

```
Monday-Tuesday-etc.
```

```
day = int(input("Introduce the day number: "))
name = "No day"
if day == 1:
  name = "Monday"
```

```
elif day == 2:
  name = "Tuesday"
elif day == 3:
  name = "Wednesday"
elif day == 4:
  name = "Thursday"
elif day == 5:
  name = "Friday"
elif day == 6:
  name = "Saturday"
elif day == 7:
  name = "Sunday"

print(name)
```

10. **Write a program that asks the user for her name an prints the number of characters.**

- Input: Jose, a valid name must have more than 2 characters (otherwise the program shall print "Your name is too short")
- Expected output:

```
Your name has 4 characters
```

```
name = input("What is your name?")
if len(name) > 2:
  print("Your name has {} characters.".format(len
(name)))
else:
  print("Your name is too short.")
```

11. Write a program that given a quantitative grade, prints out the qualitative value.

- A if input in [9-10]
- B if input in [7-9)
- C if input in [5-7)
- D if input < 5

- Input: 7.5 (test limit values like 7, 5, 9, 10 as an input)
- Expected output:

```
Your grade is B.
```

```
a = float(input("Introduce your grade as a number
: "))
grade = ""
if a < 5:
  grade = "D"
elif a < 7:
  grade = "C"
elif a < 9:
   grade = "B"
else:
  grade = "A"

print("Your grade is {}.".format(grade))
```

12. Write a program that given an integer numbers, prints out if it is divisible by 5 and 11.

- Input: 55
- Expected output:

```
The number 55 is divisible by 5 and 11.
```

```
value = int(input())
```

```
  if value % 5 == 0 and value % 11 == 0:
    print ("The number {} is divisible by 5 and 11.
".format(value))
```

13. **Write a program to check that a character is a letter in uppercase.**

Make use of these functions:

- value.isupper() -->Returns True if the value is in uppercase.
- value.isalpha()-->Returns True if the value is a letter.

- Input: A
- Expected output:

```
The character A is in uppercase.
```

```
  value = input()
  is_upper = len(value) == 1 and value.isupper() an
d value.isalpha()
  if is_upper:
    print("The character {} is in uppercase.".forma
t(value))
  else:
    print("It is not.")
```

14. **Write a program that given a month name, prints the number of days (if the name is not correct, it shall print out some error message).**

- Input: January
- Expected output:

```
  January has 31 days.
```

```
  month = input("Introduce the month name: ")
  n_days = 0
  if month == "January" or month == "March" or mont
h == "May" or month == "July" or month == "August"
or month == "October" or month == "December":
    n_days = 31
  elif month == "April" or month == "June" or month
 == "September" or month == "November":
    n_days = 30
  elif month == "February":
    n_days = 28

  if n_days != 0:
    print(month + " has {} days.".format(n_days))
  else:
    print("The month name is not recognized.")
```

```
  #More Python
  month = input("Introduce the month name: ")
  n_days = 0
  if month in ["January", "March", "May", "July", "
August", "October", "December"]:
    n_days = 31
  elif month in ["April", "June", "September", "Nov
ember"]:
    n_days = 30
  elif month == "February":
    n_days = 28

  if n_days != 0:
    print(month + " has {} days.".format(n_days))
  else:
    print("The month name is not recognized.")
```

15. **Write a program that given the 3 angles of triangles, checks whether is valid (the sum must be 180).**

- Input: 45, 45, 90
- Expected output:

```
The triangle: 45, 45, 90 is valid.
```

```
a1 = int(input("Introduce the first angle: "))
a2 = int(input("Introduce the second angle: "))
a3 = int(input("Introduce the second angle: "))


if (a1+a2+a3 == 180):
  print("The triangle: {}, {}, {} is valid.".form
at(a1,a2,a3))
else:
  print("The triangle is not valid.")
```

16. **Write a program that given the 3 integer numbers, it prints out the sorted values (in descending order) using conditional statements.**

- Input: 3, 5, 2
- Expected output:

```
5, 3, 2
```

```
a = 3
b = 5
c = 2


if a > b:
  if a > c:
    if b > c:
      print("{} {} {}".format(a,b,c))
    else:
      print("{} {} {}".format(a,c,b))
  else:
     print("{} {} {}".format(c,a,b))
else:
  if b > c:
    if c > a:
      print("{} {} {}".format(b,c,a))
    else:
      print("{} {} {}".format(b,a,c))
  else:
    print("{} {} {}".format(c,b,a))
```

14 LAB 3-CONTROL FLOW: LOOPS

In this chapter, we propose and solve some exercises about loops.

As a good programming practice, our test cases should ensure that all branches of the code are executed at least once.

14.1 List of exercises

1. **Write a program that prints out the numbers from 0 to 10 (only even numbers). Use both: while and for loops.**

- Input: no input
- Expected output:

```
0
2
4
6
8
10
```

```
i = 0
while i <= 10:
  print(i)
  i = i + 2


for i in range(0,12,2):
  print(i)
```

2. **Write a program that prints out the numbers from 0 to 10 (only odd numbers). Use both: while and for loops.**

- Input: no input
- Expected output:

```
1
3
5
7
9
```

```
  i = 1
  while i < 11:
    print(i)
    i = i + 2



  for i in range(1,11,2):
    print(i)
```

3. **Write a program that prints out the numbers from 10 to 0 (only odd numbers). Use both: while and for loops.**

- Input: no input
- Expected output:

```
9
7
5
3
1
```

```
i = 9
while i >= 0:
  print(i)
  i = i - 2



for i in range(9,0,-2):
  print(i)
```

4. **Write a program to calculate the factorial of an integer number. Use both: while and for loops.**

    - factorial (n) = n * factorial (n-1)

- Input: an integer number, 5
- Expected output:

```
120
```

```
number = 5
fact = 1

if number < 0:
  fact = -1
elif number == 0 or number == 1:
  fact = 1
else:
  for i in range(1,number+1):
    fact *= i

print(fact)
```

5. **Write a program to calculate the exponentiation of a number with base b, and exponent n, both: while and for loops.**

- **base^{exponent}=base*base*base…*base**

- Input: base 2, exponent 3
- Expected output:

```
8
```

```python
base = 3
exponent = 3
mipow = 1

if exponent > 0:
  for i in range(0,exponent):
     mipow *= base

print(mipow)
```

6. **Given an integer number, n, make the sum of all the numbers from 0 to n.**

- Input: 5
- Expected output:

```
15
```

```python
n = 5
add_n = 0
for i in range(0,n+1,1):
  add_n += i

print(add_n)
```

7. **Given an integer number, n, print a cross of radius n as follows.**

- Input: 9

- Expected output:

```
 *  .  .  .  .  .  .  .  *
 .  *  .  .  .  .  .  *  .
 .  .  *  .  .  .  *  .  .
 .  .  .  *  .  *  .  .  .
 .  .  .  .  *  .  .  .  .
 .  .  .  *  .  *  .  .  .
 .  .  *  .  .  .  *  .  .
 .  *  .  .  .  .  .  *  .
 *  .  .  .  .  .  .  .  *
```

```python
n = 9
for i in range(0, n, 1):
  for j in range (0, n, 1):
    if (i==j or n-i-1 == j):
      print(" *", end="")
    else:
      print(" .", end="")
  print("")
```

8. **Given an integer number, n, print a wedge of stars as follows.**

- Input: 5
- Expected output:

```
*****
****
***
**
*
```

```python
n_stars = 5
for i in range (n_stars, 0, -1):
  for j in range (0, i, 1):
    print("*", end="")
  print("")
```

```python
#Pythonic
for i in range (n_stars, 0, -1):
    print("*"*i)
```

9. Given an integer number, n, print a wedge of stars as follows.

- Input: 5
- Expected output:

```
*
**
***
****
*****
```

```python
n_stars = 5
for i in range (0, n_stars+1, 1):
    print("*"*i)
```

10. Make a program to display the multiplication table (1-10).

- Input: no input
- Expected output:

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

2      4      6      8      10     12     14     16     18     20

3      6      9      12     15     18     21     24     27     30

4      8      12     16     20     24     28     32     36     40

5      10     15     20     25     30     35     40     45     50

6      12     18     24     30     36     42     48     54     60

7      14     21     28     35     42     49     56     63     70

8      16     24     32     40     48     56     64     72     80

9      18     27     36     45     54     63     72     81     90

10     20     30     40     50     60     70     80     90     100
```

```
for i in range(1,11,1):
  for j in range (1, 11, 1):
    print(i*j, end = "")
    print("\t", end = "")
  print("")
```

11. Write a program to detect if a number is a prime number.

- Input: 5, 8
- Expected output:

```
The number 5 is prime.
The number 8 is not prime.
```

```
n = 1
n_divisors = 1
divisor = 2
while divisor < n and n_divisors <= 2:
  if n % divisor == 0:
    n_divisors = n_divisors + 1
  divisor = divisor + 1

if n_divisors > 2:
  print("The number {} is not prime.".format(n))
else:
  print("The number {} is prime.".format(n))
```

12. Write a program to sum all odd numbers between n and 100 (included).

- Input: 11
- Expected output:

```
   The sum between 11-100 is 4995.
```

```
  top = 100
  n = 11
  add_top = 0
  for value in range(11, top+1):
      add_top = add_top + value

  print("The sum between {}-
{} is {}".format(n,top,add_top))
```

13. Write a program to show the square of the first 10 numbers.

- Input: no input
- Expected output:

```
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
```

```
  for i in range(0, 11):
    print("The square of {} is {}".format(i,i**2))
```

14. Write a program to show the Fibonnacci sequence of a given number n.

```
fibonacci(n) =  fibonacci (0) = 0
                fibonacci (1) = 1
fibonacci(n) =  fibonacci(n-1) + fibonacci (n-2)
```

- Input: a positive number n
- Expected output:

```
0
1
1
2
3
```

```
numbersToGenerate=5
fn2 = 0
fn1 = 1
print(fn2)
print(fn1)
for i in range(2, numbersToGenerate):
  fcurrent = fn1 + fn2
  temp = fn1
  fn1 = fcurrent
  fn2 = temp
  print(fn1)
```

15. Write a program to check if an integer number is a palindrome.

- Input: 121
- Expected output:

```
The number 121 is palindrome: True
```

```
number = 121
palindrome = number
reverse = 0
while (palindrome != 0):
  remainder = palindrome % 10
  print(remainder)
  reverse = reverse * 10 + remainder
  print(reverse)
  palindrome = palindrome // 10
```

```
    print(palindrome)
    print("---")

  print("The number "+str(number)+" is palindrome: 
"+str((number==reverse)))
```

16. **Write a program to check if an integer number of three digit is an Armstrong number.**

    - *An Armstrong number of three digit is a number whose sum of cubes of its digit is equal to its number.*

- Input: $153=1^3+5^3+3^3$ or 1+125+27=153
- Expected output:

```
  The number 153 is an Armstrong number.
```

```
  number = 153
  initial_number = number
  result = 0
  if number>=100 and number <= 999:
    while number != 0:
      remainder = number % 10
      result = result + remainder**3
      number = number // 10
    if result == initial_number:
      print("The number {} is an Armstrong number."
.format(result))
    else:
      print("The number {} is NOT an Armstrong numb
er.".format(result))
  else:
    print("Number is not valid.")
```

17. Write a program to show the first N prime numbers.

- Input: N = 10
- Expected output:

```
The number 1 is prime.
The number 2 is prime.
The number 3 is prime.
The number 5 is prime.
The number 7 is prime.
The number 11 is prime.
The number 13 is prime.
The number 17 is prime.
The number 19 is prime.
The number 23 is prime.
```

```
counter = 0
MAX_PRIMES = 10
n = 1

while counter < MAX_PRIMES:
  n_divisors = 1
  divisor = 1
  #Loop if n is prime
  while divisor < n and n_divisors <= 2:
    if n % divisor == 0:
      n_divisors = n_divisors + 1
    divisor = divisor + 1
  #End loop if n is prime
  #Step second loop
  if n_divisors <= 2:
    print("The number {} is prime.".format(n))
    counter = counter + +1

  n = n + 1
```

18. Write a program to calculate the binomial coefficient:

$$\binom{m!}{n!} = \frac{m!}{n!\ (m-n)!}$$

- Input: m = 10, n = 5
- Expected output:

```
252
```

```
m = 10
n = 5
factorialmn = 1
factorialn = 1
factorialm = 1

if (m < n):
  print("M must be >= n");
else:
  #Factorial m
  if m == 0 or m == 1:
    factorialm = 1
  else:
    i = m
    while i > 0:
      factorialm = i * factorialm
      i = i - 1
  #Factorial n
  if n == 0 or n == 1:
    factorialn = 1
  else:
    i = n
```

```
      while i>0 :
        factorialn = i * factorialn
        i = i - 1
    #Factorial m-n
    if (m - n == 0 or m - n == 1):
      factorialmn = 1
    else:
      i =  m - n
      while i>0 :
        factorialmn = i * factorialmn
        i = i - 1
    print("The result is: ",(factorialm / (factoria
ln * factorialmn)))
```

19. Write a program to print a Christmas tree of 15 base stars.

- Input: base_stars = 15
- Expected output:

```
      *
     ***
    *****
   *******
  *********
 ***********
*************
     ***
     ***
     ***
```

```
  base_stars = 15
  half_blank_spaces = 0;
  for i in range(1,base_stars, 2):
    half_blank_spaces = (base_stars-i) // 2
    for j in range(0, half_blank_spaces):
      print(" ",end="")
    for k in range(0, i):
      print("*",end="")
    for j in range(0, half_blank_spaces):
      print(" ",end="")
    print("")

  half_blank_spaces = (base_stars//2)-1;
  for i in range(3):
    for j in range(0,half_blank_spaces):
      print(" ",end="")
    for k in range(0, 3):
      print("*",end="")
    for j in range(0, half_blank_spaces):
      print(" ",end="")
    print("")
```

20. Write a program to print an * in the upper diagonal of an implicit matrix of size n.

- Input: n = 3
- Expected output:

```
* * *
. * *
. . *
```

```
  n = 3
  for i in range(1,n+1):
    for j in range (1,n+1):
      if (i <= j):
        print("*", end = "")
      else:
        print(".", end = "")
    print("")
```

21. Write a program to print an * in the lower diagonal of an implicit matrix of size n.

- Input: n = 3
- Expected output:

```
* . .
* * .
* * *
```

```
n = 3
for i in range(1,n+1):
  for j in range (1,n+1):
    if (i >= j):
      print("*", end = "")
    else:
      print(".", end = "")
  print("")
```

22. Write a program to print a diamond of radius n.

- Input: n = 7
- Expected output:

```
 . . . * . . .
 . . * * * . .
 . * * * * * .
 * * * * * * *
 . * * * * * .
 . . * * * . .
 . . . * . . .
```

```
n = 7
mid = n//2
lmin=0
rmax=0
```

```
if (n % 2 != 0):
  for i in range(0,n):
    if i <= mid:
      lmin=mid-i
      rmax=mid+i
    else:
      lmin=mid-(n-i)+1
      rmax=mid+(n-i)-1
    for j in range (0,n):
      if j>=lmin and j<=rmax:
        print(" *", end = "")
      else:
        print(" .", end = "")
    print("")
```

23. Write a program that displays a table of size (n x n) in which each cell will have * if i is a divisor of j or "" otherwise.

- Input: n = 10
- Expected output:

```
* * * * * * * * * * 1
* *   *   *   *   * 2
*   *     *     *   3
* *   *       *     4
*       *         * 5
* * *     *         6
*           *       7
* *   *       *     8
*   *           *   9
* *     *         * 10
```

```
N = 10
for i in range (1, N+1):
  for j in range (1, N+1):
    if (i%j == 0 or j%i == 0):
      print(" *", end = "")
    else:
      print("  ", end = "")
  print(" ",i)
  print("")
```

24. Write a program to display a menu with 3 options (to say hello in English, Spanish and French) and to finish, the user shall introduce the keyword "quit". If any other option is introduced, the program shall display that the input value is not a valid option.

- Input: test the options
- Expected output:

```
----------MENU OPTIONS----------
1-Say Hello!
2-Say ¡Hola!
3-Say Salut!
> introduce an option or quit to exit...
```

```
  option = "1"
  while option != "quit":
    print("----------MENU OPTIONS----------")
    print("1-Say Hello!")
    print("2-Say ¡Hola!")
    print("3-Say Salut!")
    option = input("> introduce an option or quit t
o exit...")
    if option == "1":
      print("Hello!")
    elif option == "2":
      print("¡Hola!")
    elif option == "3":
      print("Salut!")
    elif option == "quit":
      print("...finishing...")
    else:
      print("Not a valid option: ", option)
```

25. **The number finder: write a program that asks the user to find out a target number. If the input value is less than the target number, the program shall say "the target value is less than X", otherwise, the program shall say "the target value is greater than X. The program shall finish once the user finds out the target number.**

- Input: target = 10
- Expected output:

```
Introduce a number: 3
The target value is greater than  3
Introduce a number: 5
The target value is greater than  5
Introduce a number: 12
The target value is less than  12
Introduce a number: 10
```

```
  target = 10
  found = False
  while (not found):
    value = int(input("Introduce a number: "))
    if target < value:
      print("The target value is less than ", value
)
    elif target > value:
      print("The target value is greater than ", va
lue)
    else:
      found = True
  if found:
    print("You have found out the target value!")
```

26. Modify the program in 20 to give only 5 attempts.

- Input: target = 10
- Expected output:

```
Introduce a number: 6
The target value is greater than  6
Introduce a number: 7
The target value is greater than  7
Introduce a number: 12
The target value is less than  12
Introduce a number: 9
The target value is greater than  9
Introduce a number: 11
The target value is less than  11
You have consumed the max number of attempts.
```

```
  target = 10
  found = False
  attempts = 0
  MAX_ATTEMPTS = 5
  while (not found and attempts < MAX_ATTEMPTS):
    value = int(input("Introduce a number: "))
    if target < value:
      print("The target value is less than ", value
)
    elif target > value:
      print("The target value is greater than ", va
lue)
    else:
      found = True
    attempts = attempts + 1
  if found:
    print("You have found out the target value!")
  else:
    print("You have consumed the max number of atte
mpts.")
```

27. Write a program to print the following pattern:

```
1
22
333
4444
55555
```

```
  for i in range (1,6):
    print(str(i)*i)
```

28. Write a program to find greatest common divisor (GCD) of two numbers.

- Input: x = 54, y = 24
- Expected output:

```
  The GCD of 54 and 24 is: 6
```

```
  #Python version in the math library
  def gcd(a, b):
      """Calculate the Greatest Common Divisor of a
 and b.

      Unless b==0, the result will have the same si
gn as b (so that when
      b is divided by it, the result comes out posi
tive).
      """
      while b:
          a, b = b, a%b
      return a
```

```
  x = 54
  y = 24
  gcd = 1
  #find x divisor
  current_divisor = 1
  if x != 0 and y != 0:
    if x > y:
      top = y
    else:
      top = x
    while (current_divisor < top):
      if (x % current_divisor == 0) and (y % curren
t_divisor == 0):
        gcd = current_divisor
      current_divisor = current_divisor + 1

    print("The GCD of {} and {} is: {}".format(x,y,
gcd))
  else:
    print("The input must be different to 0.")
```

29. Write a program that counts the number of primer numbers between 1-MAX.

- Input: MAX = 100
- Expected output:

```
   The number of primer numbers between 1 and 100 is 26.
```

```
  MAX = 100
  n_primes = 0
  n = 1

  while n < MAX:
    n_divisors = 1
    divisor = 1
    #Loop if n is prime
    while divisor < n and n_divisors <= 2:
      if n % divisor == 0:
        n_divisors = n_divisors + 1
      divisor = divisor + 1
    #End loop if n is prime
    #Step second loop
    if n_divisors <= 2:
      n_primes = n_primes + 1

    n = n + 1

  print("The number of primer numbers between 1 and
 {} is {}.".format(MAX, n_primes))
```

30. Write a program that sums all primer numbers between 1-MAX.

- Input: MAX = 100
- Expected output:

```
  The sum of all primer numbers between 1 and 100 is 106
1.
```

```
  MAX = 100
  sum_primes = 0
  n = 1
```

```
while n < MAX:
  n_divisors = 1
  divisor = 1
  #Loop if n is prime
  while divisor < n and n_divisors <= 2:
    if n % divisor == 0:
      n_divisors = n_divisors + 1
    divisor = divisor + 1
  #End loop if n is prime
  #Step second loop
  if n_divisors <= 2:
    sum_primes = sum_primes + n

  n = n + 1

print("The sum of all primer numbers between 1 an
d {} is {}.".format(MAX, sum_primes))
```

31. Write a program to approximate the value of e using the Taylor series for ex.

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

- Input: x = 1, MAX (N) = 100
- Expected output:

```
The value of e is 2.7182818284590455
```

```
sum_e = 0
MAX = 100
```

```
x = 1
for n in range (1, MAX+1):
 factorial_n = 1
 for i in range(1,n):
   factorial_n = i * factorial_n
 sum_e = sum_e + (1/factorial_n)

print("The value of e is {}".format(sum_e))
```

32. Write a program to detect whether a number is a perfect number.

"A number is perfect if the addition of all positive divisors is equal to the number."

- Input: n = 6
- Expected output:

```
The number 6 is perfect.
```

```
n = 6
perfect = 0
for i in range (1,n):
  if n%i == 0:
    perfect = perfect + i

if perfect == n:
  print("The number {} is perfect.".format(n))
else:
  print("The number {} is NOT perfect.".format(n)
)
```

15 LAB 4-DATA STRUCTURES: LISTS AND STRINGS

In this chapter, we propose and solve some exercises about basic data structures implemented through Python lists and strings.

In these exercises, we can always proceed solving the problems in a generic way or taking advantage of Python capabilities. As a recommendation, first, try the generic way (applicable to any programming language) and, then, using Python

As a good programming practice, our test cases should ensure that all branches of the code are executed at least once.

15.1 List of exercises

1. **Write a program that creates a list of n numbers initializing each position with a value (the numeric value of such position). Display the list in the console.**

- Input: 5
- Expected output:

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

```
values = []
n = 5
for i in range(n):
  values.append(i)
print (values)


#Python way
values = [x for x in range(n)]
print(values)
```

2. **Write a program that given a list of n numbers, calculates and displays the length of the list.**

- Input: [3,4,5,6]
- Expected output:

```
   The length of the list is: 4
```

```
values = [3,4,5,6]
length = 0
for v in values:
  length = length + 1
print("The length of the list is: ", length)
#Python way
print("The length of the list is: ", len(values)
)
```

3. **Write a program that given a list of n numbers, calculates and displays the max value within the list and its position.**

- Input: [8, 1, 9, 2]
- Expected output:

```
The max value is: 9 in position: 2.
```

```
values = [8,1,9,2]
if values and len(values) > 0:
  max_value = values[0]
  position = 0
  for i in range(1, len(values)):
    if values [i] > max_value:
      max_value = values [i]
      position = i
print("The max value is {} in position: {}".form
at(max_value, position))


#Python way to find the max value
max_value = max(values)
position = values.index(max_value)
print("The max value is {} in position: {}".form
at(max_value, position))
```

4. **Write a program that given a list of n numbers, calculates and displays the min value within the list and its position.**

- Input: [8, 1, 9, 2]
- Expected output:

```
The min value is: 1 in position: 1.
```

```
values = [8,1,9,2]
if values and len(values) > 0:
  min_value = values[0]
  position = 0
  for i in range(1, len(values)):
    if values [i] < min_value:
        min_value = values [i]
        position = i
  print("The min value is {} in position: {}.".for
mat(min_value, position))

  #Python way to find the max value
  min_value = min(values)
  position = values.index(min_value)
  print("The min value is {} in position: {}.".for
mat(min_value, position))
```

5. **Write a program that given a list of n numbers, calculates and displays the sum of its elements.**

- Input: [8, 1, 9, 2]
- Expected output:

```
The sum is: 20.
```

```
values = [8,1,9,2]
sum_values = 0
for v in values:
  sum_values = sum_values + v
print("The sum is {}.".format(sum_values))

#Python way to find the max value
print("The sum is {}.".format(sum(values)))
```

6. **Write a program that given a list of n numbers and a target number k, counts the number of occurrences of k in the list.**

- Input: [8, 1, 9, 1], k=1
- Expected output:

```
The number 1 has 2 occurrences.
```

```
values = [8,1,9,1]
k = 1
occurrences = 0
for v in values:
  if k == v:
    occurrences += 1
print("The number {} has {} occurrences.".format
(k, occurrences))

#Python way
occurrences = values.count(k)
print("The number {} has {} occurrences.".format
(k, occurrences))
```

7. **Write a program that given a list of n numbers and a target number k, returns and displays the position of the first apparition of the number k.**

- Input: [8, 1, 9, 1], k=1
- Expected output:

```
The number 1 occurs first in position 1.
```

```
values = [8,1,9,1]
k = 1
first_occur = -1
i = 0
size = len(values)
while i<size and first_occur == -1:
  if values[i] == k:
    first_occur = i
  i += 1
  if first_occur != -1:
    print("The number {} occurs first in position
{}.".format(k, first_occur))


  #Python way
  print("The number {} occurs first in position {}
.".format(k, values.index(k)))
```

8. **Write a program that given a list of n numbers and a target number k, returns and displays the position of the last apparition.**

- Input: [8, 1, 9, 1], k=1
- Expected output:

```
  The number 1 occurs last in position 3.
```

```
  values = [8,1,9,1]
  k = 1
  last_occur = -1
  i = len(values)-1
  while i>=0 and last_occur == -1:
    if values[i] == k:
      last_occur = i
    i -= 1


  if last_occur != -1:
    print("The number {} occurs last in position {
}.".format(k, last_occur))
```

9. **Write a program that given a list of n numbers and a target number k, returns and displays a list of all positions in which the value k occurs.**

- Input: [8, 1, 9, 1], k=1

- Expected output:

```
The number 1 occurs in [1, 3].
```

```
values = [8,1,9,1]
positions = []
k = 1
size = len(values)
for i in range(size):
  if values[i] == k:
    positions.append(i)
print("The number {} occurs in {}.".format(k,pos
itions))
```

10. Write a program that given a list of n numbers, creates a new list in reverse order.

- Input: [7, 5, 9, 2]
- Expected output:

```
The reverse list is [2, 9, 5, 7].
```

```
values =  [7, 5, 9, 2]
reverse_values = []
for i in range(len(values)-1, -1, -1):
  reverse_values.append(values[i])

print("The reverse list is {}.".format(reverse_v
alues))

#Python way
reverse_values = values [::-1]
print("The reverse list is {}.".format(reverse_v
alues))
```

```
  #print("The reverse list is {}.".format(values.re
verse()))
```

11. **Benchmarking: compare your previous code against the Python versions. See the next example.**

```
  #Module to use timers
  import time

  values = [8,1,9,2]

  #Start the timer
  t = time.time()

  min_value = values[0]
  position = 0
  for i in range(1, len(values)):
    if values [i] < min_value:
      min_value = values [i]
      position = i

  print("\n\tMy min version--
>time Taken: %.5f sec" % (time.time()-t))

  #Python way to find the max value

  #Start the timer
  t = time.time()

  min_value = min(values)
  position = values.index(min_value)


  print("\n\tPython version--
>time Taken: %.5f sec" % (time.time()-t))
```

12. **Write a program that given a list of n numbers, v, and an scalar number k, returns and displays the scalar product of k·v.**

- Input: [7, 5, 9, 2], k = 3
- Expected output:

```
The scalar product of 3 and [7, 5, 9, 2] is [21, 15, 27,
 6].
```

```
  vector = [7, 5, 9, 2]
  k = 3
  result = []
  for v in vector:
    result.append(k * v)
  print("The scalar product of {} and {} is {}.".f
ormat(k,vector, result))


  #Python way
  result = [k * v for v in vector]
  print("The scalar product of {} and {} is {}.".f
ormat(k,vector, result))
```

13. **Write a program that given two lists of n numbers, returns and displays the vector product of l1·l2.**

- Input: [7, 5, 9, 2], [4, 5, 6, 7]
- Expected output:

```
The vector product of [7, 5, 9, 2] and [4, 5, 6, 7]  is
[28, 25, 54, 14].
```

```
  l1 = [7, 5, 9, 2]
  l2 = [4, 5, 6, 7]
  result = []
  if len(l1)==len(l2):
    for i in range(len(l1)):
      result.append(l1[i]*l2[i])
  print("The vector product of {} and {} is {}.".f
ormat(l1,l2,result))
```

14. **Write a program that given two lists of n numbers, returns all combination of pairs (cartesian product).**

- Input: [7, 5, 9, 2], [4, 5, 6, 7]
- Expected output:

```
The cartesian product is: [[7, 4], [7, 5], [7, 6], [7, 7
], [5, 4], [5, 5], [5, 6], [5, 7], [9, 4], [9, 5], [9, 6
], [9, 7], [2, 4], [2, 5], [2, 6], [2, 7]].
```

```
  l1 = [7, 5, 9, 2]
  l2 = [4, 5, 6, 7]
  result = []
  for v in l1:
    for w in l2:
      result.append([v,w])

  print("The cartesian product is: {}.".format(res
ult))

  #Python way returning tuples
  result = [(x, y) for x in l1 for y in l2]
  print(result)
```

15. Write a program that given a list of n numbers, returns and displays the average of the list numbers.

- Input: [4, 5, 6, 7]
- Expected output:

```
The average of [4, 5, 6, 7] is 5.5.
```

```
  l1 = [4, 5, 6, 7]
  sum_l1 = 0
  for v in l1:
    sum_l1 +=v
  if len(l1) >0:
    avg = sum_l1/len(l1)
    print("The average of {} is {}.".format(l1, av
g))

  #Python way
  if len(l1) > 0:
    avg = sum(l1)/len(l1)
    print("The average of {} is {}.".format(l1, av
g))
```

16. Write a program that given a list of n numbers and a number k, returns the first k numbers.

- Input: [4, 5, 6, 7], k = 2
- Expected output:

```
   [4,5]
```

```
  l1 = [4, 5, 6, 7]
  k = 2
  result = []
  if k <= len(l1):
```

```
  for i in range(k):
    result.append(l1[i])
print(result)


#Python way


result = l1[:k:]
print(result)
```

17. Write a program that given a list of n numbers and a number k, returns the last k numbers.

- Input: [4, 5, 6, 7], k = 2
- Expected output:

```
[7,6]
```

```
l1 = [4, 5, 6, 7]
k = 2
result = []
if k <= len(l1):
  for i in range(len(l1)-1, k-1, -1):
    result.append(l1[i])
print(result)


#Python way


result = l1[len(l1)-1 : k-1 : -1]
print(result)
```

18. Write a program that given a list of n numbers, returns a new list containing in the same position the factorial of that value (if the value is < 0, the return value will be -1) .

- Input: [5, 0, 1, -1]
- Expected output:

```
[120, 1, 1, -1]
```

```python
v = [5, 0, 1, -1]
factorials = []
for value in v:
  if value == 0 or value == 1:
    fact_value = 1
  elif value >1:
    fact_value = 1
    for i in range(1, value+1):
      fact_value = fact_value * i
  else:
    fact_value = -1
  factorials.append(fact_value)

print(factorials)
```

19. Write a program that given a list of n numbers, returns a new list without the repeated numbers.

- Input: [4, 5, 5, 6, 6, 8]
- Expected output:

```
[4, 5, 6, 8]
```

```python
values = [4, 5, 5, 6, 6, 8]
clean_values = []
for v in values:
  found = False
  pos = 0
  while not found and pos < len(clean_values):
    found = clean_values[pos] == v
    pos = pos + 1
  if not found:
    clean_values.append(v)

print(clean_values)

#Python way

clean_values = []
for v in values:
  if clean_values.count(v) == 0:
    clean_values.append(v)
print(clean_values)
```

20. Write a program that given two lists of n numbers, returns the union of both lists.

- Input: [4,5,6] [5,7,8,9]
- Expected output:

```
[4, 5, 6, 7, 8, 9]
```

```
v1 = [4,5,6]
v2 = [5,7,8,9]
union = []
for v in v1:
  union.append(v)
for v in v2:
  found = False
  pos = 0
```

```
    while not found and pos < len(union):
      found = union[pos] == v
      pos = pos + 1
    if not found:
      union.append(v)
  print(union)

  #Python way
  union = []
  union.extend(v1)
  for v in v2:
    if union.count(v) == 0:
      union.append(v)
  print(union)
```

21. Write a program that given two lists of n numbers, returns the intersection of both lists.

- Input: [4,5,6] [5,7,8,9]
- Expected output:

```
[5]
```

```
v1 = [4,5,6]
v2 = [5,7,8,9]
intersection = []
for v in v1:
  if v2.count(v) != 0:
    intersection.append(v)

print(intersection)
```

22. **Write a program that asks the user for n numbers and returns a sorted list.**

    - Use the function `insert(pos, value)`

- Input: 6, 4, 8
- Expected output:

```
[4, 6, 8]
```

```
n = 3
values = []
for i in range(n):
  number = input("Introduce a number: ")
  #Find position
  pos = 0
  while pos < len(values) and values[pos] < numb
er:
    pos = pos + 1
  values.insert(pos, number)
print(values)
```

23. **Write a program that asks the user for n numbers and returns a sorted list in descending order.**

    - Use the function `insert(pos, value)`

- Input: 6, 4, 8
- Expected output:

```
[8, 6, 4]
```

```
n = 3
values = []
for i in range(n):
  number = input("Introduce a number: ")
  #Find position
```

```
    pos = 0
    while pos < len(values) and values[pos] > numb
er:
      pos = pos + 1
    values.insert(pos, number)
  print(values)
```

24. Write a program that given a list of n numbers and a parameter k, creates chunks of k elements.

- Input: [1,2,3,4,5,6,7,8,9], k = 3
- Expected output:

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

```
values = [1,2,3,4,5,6,7,8,9]
k = 3
chunks = []
i = 0
size = len(values)
while i < size:
  chunk = []
  for j in range(i, k+i):
    chunk.append(values[j])
  chunks.append(chunk)
  i = i + k
print(chunks)

#Python way
chunks = []
size = len(values)
for i in range(0, size, k):
    chunks.append(values[i:k+i])

  print(chunks)

  chunks.clear()
  size = len(values)
  chunks = [values[i:k+i] for i in range(0, size,
k)]
  print(chunks)
```

25. **Write a program that given a number n and an initial value k, creates a list of size n with all positions having the initial value.**

- Input: n = 10, k = -1
- Expected output:

```
   [-1, -1, -1, -1, -1]
```

```
  n = 5
  k = -1

  list_values = []
  for i in range(n):
    list_values.append(k)

  print(list_values)

  #Python way
  list_values = [k for i in range(n)]
  print(list_values)
```

26. Write a program that given a string, displays the length of the string.

- Input: Hello
- Expected output:

```
The length of Hello is 5.
```

```
value = "Hello"
value_size = 0
for i in value:
  value_size = value_size + 1
print("The length of {} is {}.".format(value, va
lue_size))

#Python version

print("The length of {} is {}.".format(value, le
n(value)))
```

27. Explore the string object methods.

```
dir ("")
```

```
dir("")
```

28. Write a program that given a string, displays the string in reverse order.

- Input: Hello
- Expected output:

```
Hello and olleH
```

```
value = "Hello"
new_value = ""
```

```
  for i in range(len(value)-1, -1, -1):
    new_value = new_value + value[i]
  print("{} and {}".format(value, new_value))


  #Python version
  new_value = value[::-1]
  print("{} and {}".format(value, new_value))
```

29. Write a program that given a string, displays whether the string is a palindrome.

- Input: anna
- Expected output:

```
  anna is a palindrome True.
```

```
  value = "anna"
  is_palindrome = value == value[::-1]
  print("{} is a palindrome {}.".format(value, is_
palindrome))
```

30. Write a program that given a string, displays the string in uppercase letters.

  - Make use of function `ord(char)` -->ASCII number of the char.

- Input: This is a string
- Expected output:

```
  THIS IS A STRING
```

```
  value = "This is a string"
  upper_value = ""
  for v in value:
    ord_v = ord(v)
    if ord_v >= 97 and ord_v <= 122:
      upper_value = upper_value + chr(ord_v-32)
    else:
       upper_value = upper_value + v
  print(upper_value)


  #Python version
  print(value.upper())
```

31. Write a program that given a string, displays the string in lowercase letters.

- Input: THIS IS A STRING
- Expected output:

```
this is a string
```

```
value = "THIS IS A STRING"
lower_value = ""
for v in value:
  ord_v = ord(v)
  if ord_v >= 65 and ord_v <= 90:
    lower_value = lower_value + chr(ord_v+32)
  else:
     lower_value = lower_value + v
print(lower_value)


#Python version
print(value.lower())
```

32. Write a program that given a string and a char separator, returns a list of the words. (Tokenizer)

- Input: Anna,21,Programming (char separator ,)

- Expected output:

```
["Anna", "21", "Programming"]
```

```
value = "Anna,21,Programming"
sep = ","
tokens = []

current_value = ""
for v in value:
  if v == ",":
    tokens.append(current_value)
    current_value = ""
  else:
    current_value = current_value + v
tokens.append(current_value)

print(tokens)

#Python version

tokens = value.split(sep)
print(tokens)
```

33. Write a program that given a list of strings, returns a list with the size of each string.

- Input: ["Anna", "21", "Programming"]
- Expected output:

```
[4, 2, 11]
```

```
tokens = ["Anna", "21", "Programming"]
```

```
sizes = []
for t in tokens:
  sizes.append(len(t))
print(sizes)

#Python version
sizes = [len(x) for x in tokens]
print(sizes)
```

34. Write a program that given a string and a number k, returns a list of chunked strings of size k.

- Input: "This is a very looooong string" and k=3
- Expected output:

```
['Thi', 's i', 's a', ' ve', 'ry ', 'loo', 'ooo', 'ng
', 'str', 'ing']
```

```
values = "This is a very looooong string"
k = 3
chunks = []
i = 0
size = len(values)
while i < size:
  chunk = ""
  for j in range(i, k+i):
    chunk = chunk + values[j]
  chunks.append(chunk)
  i = i + k
print(chunks)

#Python way
chunks = []
```

```
  size = len(values)
  for i in range(0, size, k):
    chunks.append(values[i:k+i])

  print(chunks)

  chunks.clear()
  size = len(values)
  chunks = [values[i:k+i] for i in range(0, size,
k)]
  print(chunks)
```

35. Write a program that given a string, returns a trimmed string (removing blank spaces at the beginning and at the end) and separating words with just one blank space.

- Input: " Hello Mary "
- Expected output:

```
Hello   Mary   has 18 characters.
Hello Mary has 10 characters.
```

```
  value = "    Hello   Mary   "
  print("{} has {} characters.".format(value, len(
value)))
  trim_value = ""
  state = 1
  pos = 0
  for v in value:
    if v != " " and state == 1:
      trim_value = trim_value + v
      state = 2
      previous = True
    elif v == " " and state == 2 and previous and
pos < len(value)-2:
      trim_value = trim_value + v
      previous = False
    elif v != " " and state == 2:
      trim_value = trim_value + v
      previous = True
    pos = pos + 1

  print("{} has {} characters.".format(trim_value,
 len(trim_value)))

  #Python version
  trim_value = (value.strip())
  clean_value = ""
  previous = False
  for v in trim_value:
    if v != " ":
      clean_value = clean_value + v
      previous = True
    elif v == " " and previous:
      clean_value = clean_value + v
      previous = False

  print("{} has {} characters.".format(clean_value
, len(clean_value)))
```

36. Write a program that given a string, an input character and a replacement character, returns a string replacing all occurrences of input character by the replacement character.

- Input: "Hello", input = "l", replacement ="t"
- Expected output:

```
Hello is now Hetto.
```

```
value = "Hello"
input_char = "l"
replacement = "t"
new_value = ""
for v in value:
  if v == input_char:
    new_value = new_value + replacement
  else:
    new_value = new_value + v
print("{} is now {}.".format(value, new_value))


#Python version


new_value = value.replace(input_char, replacement
)
print("{} is now {}.".format(value, new_value))
```

37. Write a program that given a string, counts and displays the number of unique characters.

- Input: "Hello"
- Expected output:

```
The number of unique characters is 4.
```

```
value = "Hello"
```

```
  unique_chars = ""
  for v in value:
    if unique_chars.count(v) == 0:
      unique_chars = unique_chars + v
  print("The number of unique characters is {}.".f
ormat(len(unique_chars)))
```

38. Write a program that given a list of strings and a char separator, displays a message containing each string separated by separator.

- Input: ["Hello", "Mary,", "How", "are", "you?"], separator = "#"
- Expected output:

```
Hello#Mary,#How#are#you?
```

```
words = ["Hello", "Mary,", "How", "are", "you?"]
separator = "#"
message = ""
pos = 0
for w in words:
  if pos < len(words)-1:
    message = message + w + separator
  else:
    message = message + w
  pos = pos + 1
print(message)


#Python version
print("#".join(words))
```

39. Write a program that given a string and and input pattern (another string), checks if the string starts with the input pattern.

- Input: "Hello", pattern="He"
- Expected output:

```
True
```

```
value = "Hello"
pattern = "He"
size_value = len(value)
size_p = len(pattern)
match = False
if size_p < size_value:
  match = True
  i = 0
  while match and i < size_p:
    match = match and (pattern[i] == value[i])
    i = i + 1
print(match)

#Python version
print(value.startswith(pattern))
```

40. Write a program that given a string and an input pattern (another string), checks if the string ends with the input pattern.

- Input: "Hello", pattern="lo"
- Expected output:

```
True
```

```
value = "Hello"
```

```
  pattern = "lo"
  size_value = len(value)
  size_p = len(pattern)
  match = False
  if size_p < size_value:
    match = True
    i = 0
    while match and i < size_p:
      match = match and (pattern[i] == value[size_
value-size_p+i])
      i = i + 1
  print(match)

  #Python version
  print(value.endswith(pattern))
```

41. Write a program that given a string, filters all characters that are not numbers.

- Use the function `value.isdigit()`

- Input: "He2l3l4o5"
- Expected output:

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

```
  value = "He2l3l4o5"
  numbers = []
  for v in value:
    if v.isdigit():
      numbers.append(v)
  print(numbers)
```

```
#Python version
numbers = [x for x in value if x.isdigit()]
print(numbers)
```

42. Write a program that given a list of integers and a value k, filters all numbers that are less than k.

- Input: [4, 15, 9, 21], k=10
- Expected output:

```
[4, 9]
```

```
values = [4, 15, 9, 21]
filtered = []
k = 10
for v in values:
  if v < k:
    filtered.append(v)
print(filtered)


#Python version
filtered = [x for x in values if x < k]
print(filtered)
```

43. Write a program that given a list of integers and a value k, removes the first apparition of the value from the list.

- Input: [4, 15, 9, 21], k=15
- Expected output:

```
[4, 9, 21]
```

```
values = [4, 15, 9, 21]
k = 21
found = False
```

```
i = 0
while not found and i<len(values):
  found = values[i] == k
  i = i + 1
if found:
  del values[i-1]

print(values)


#Python version
values = [4, 15, 9, 21]
del values[values.index(k)]
print(values)
```

44. Write a program that asks the user for introducing k numbers and creates a list following a LIFO (Last Input First Output) strategy. Then, the program must extract and remove the elements following this strategy.

- Input: 4, 5, 6, 7 (k=4)
- Expected output:

```
Stack: [4, 5, 6, 7]
Extracting: 7, Stack: [4, 5, 6, 7]
Extracting: 6, Stack: [4, 5, 6]
Extracting: 5, Stack: [4, 5]
Extracting: 4, Stack: [4]
```

```
k = 4
stack = []
for i in range(k):
  value = int(input("Introduce value: "))
  stack.append(value)
print("Stack: {}".format(stack))
```

```
while len(stack) > 0:
  v = stack[len(stack)-1]
  print("Extracting: {}, Stack: {}".format(v, st
ack))
  del stack[len(stack)-1]

#Python version
k = 4
stack = []
for i in range(k):
  value = int(input("Introduce value: "))
  stack.append(value)

while len(stack) > 0:
    print("Extracting: {}, Stack: {}".format(sta
ck.pop(), stack))
```

45. Write a program that asks the user for introducing k numbers and creates a list following a FIFO (First Input First Output) strategy. Then, the program must extract and remove the elements following this strategy.

- Input: 4, 5, 6, 7 (k=4)
- Expected output:

```
Queue: [4, 5, 6, 7]
Extracting: 4, Queue: [4, 5, 6, 7]
Extracting: 5, Queue: [5, 6, 7]
Extracting: 6, Queue: [6, 7]
Extracting: 7, Queue: [7]
```

```
k = 4
queue = []
for i in range(k):
  value = int(input("Introduce value: "))
  queue.append(value)
print("Queue: {}".format(queue))

while len(queue) > 1:
  v = queue[0]
  del queue[0]

print(stack[0])
```

15.2 Quick questions

46. Define the lists x and y as lists of numbers.

- x = [2, 4, 6, 8]
- y = [1, 3, 5, 7]

- What is the value of 2*x?
- What is the result of x+y?
- What is the result of x-y?
- What is the value of x[1]?
- What is the value of x[-1]?
- What is the value of x[:]?
- What is the value of x[2:4]?
- What is the value of x[1:4:2]?
- What is the value of x[:2]?
- What is the value of x[::2]?
- What is the result of the following two expressions? x[3]=8

47. Define a string x = "Hello"

- What is the value of 4*x?
- What is the value of x[1]?
- What is the value of x[-1]?
- What is the value of x[::2]?
- What is the value of x[::-1]?

```
x = [2, 4, 6, 8]
y = [1, 3, 5, 7]
print(2*x)
print(x+y)
#print(x-y) #Raise an error
print(x[1])
print(x[-1])
print(x[:])
print(x[2:4])
print(x[1:4:2])
print(x[:2])
print(x[::2])
```

```
x = "Hello"
print(4*x)
print(x[1])
print(x[-1])
print(x[::2])
print(x[::-1])
```

15.3 Quick review of list and string methods

15.3.1 List: some relevant methods

```
#Review of list methods
#Create a list

values = []

#Append an element

values.append(1)
```

```
  print(values)

  #Access an element

  print(values[0])

  #Get number of elements

  len(values)
  print(len(values))

  #Count the number of elements

  print(values.count(1))

  #Slicing [start:end:step], default start = 0, end
= len(list), step = 1

  #First k elements, k = 1
  print(values[:1])

  #List sort

  print(values.sort())

  #List reverse

  values.reverse()

  #Remove an element
```

```
del values [0]

#Remove all elements

values.clear()
```

15.3.2 String: some relevant methods

```
#Review of string methods

value = "   Hello, Mary   "

print(len(value))

#Accessing

print(value[5])

#Cleaning
print(value.strip())

#Modifying values
print(value.upper())
print(value.lower())

#Finding and replacing
print("Hello".startswith("He"))
print("Hello".endswith("lo"))
print(value.find("H"))
print(value.replace("  ","#"))
```

```
#Check values

print("1".isdigit())

print("a".isalpha())

#Tokenizing
print(value.split(","))
```

16 LAB 5-DATA STRUCTURES: TUPLES, SETS AND DICTIONARIES

In this chapter, we propose and solve some exercises about basic data structures implemented through Python lists (matrix), tuples, sets and dictionaries.

In these exercises, we can always proceed solving the problems in a generic way or taking advantage of Python capabilities. As a recommendation, first, try the generic way (applicable to any programming language) and, then, using Python

As a good programming practice, our test cases should ensure that all branches of the code are executed at least once.

16.1 List of exercises

1. **Write a program that a given a number n∈(0,10) creates and displays a square matrix initialized with all positions to 1.**

- Input: 3

- Expected output:

```
111
111
111
```

```
  matrix = []
  n = int(input("Introduce the value of n: "))
  if n>0 and n<10:
    for i in range(n):
      row = []
      for j in range (n):
        row.append(1)
      matrix.append(row)
    #Manual display
    for i in range(n):
      for j in range(n):
        print(matrix[i][j], end="", sep=",")
      print("")
  #Pythonic version: comprehension lists
  matrix = [ [1 for i in range(n)] for j in range(
n)]
  print(matrix)
```

2. **Write a program that given a matrix, displays whether is a square matrix.**

- Input: reuse the code before to create a matrix
- Expected output:

```
  n = 3
  matrix = [ [1 for i in range(n)] for j in range(
n)]
  is_square = True
  if matrix and len(matrix)>0:
    n_rows = len(matrix)
    n_cols = len(matrix[0]) #dimension
    is_square = n_cols == n_rows
    i = 1
    while is_square and i<n_rows:
      is_square = is_square and n_cols == len(matr
ix[i])
      i = i + 1
    print("The matrix is square: ", is_square)
```

3. **Write a program that a given a matrix (n x m) with random numbers between (0,10), calculates and displays the transpose matrix.t(j,i)=m(i,j).**

- Input: n = 3, m = 2

```
[10, 0, 0]
[5, 1, 0]
```

- Expected output:

```
[10, 5]
[0, 1]
[0, 0]
```

- Dependencies: To create the matrix with random numbers, we are going to introduce the library of random numbers in Python.

```
  import random
  print(random.random()) # Random float x, 0.0 <=
x < 1.0
  print(random.uniform(0, 10))  # Random float x,
0.0 <= x < 10.0
  print(random.randint(0, 10))  # Integer from 0 t
o 10, endpoints included
```

```
  print(random.randrange(0, 101, 2))  # Even integ
er from 0 to 100
  print(random.choice('AEIOU')) #Get a randomized
element
  print(random.shuffle([1,2,3])) #Shuffle numbers
of a list
  print(random.sample([1, 2, 3],  2))
```

```
  #Create a matrix
  n = 3 #columns
  m = 2 #rows
  matrix = [ [random.randint(0, 10) for i in range
(n)] for j in range(m)]
  print(matrix)
  print("Matrix")
  for row in range(m):
    print(matrix[row])

  #Transposing
  t_matrix = []
  for i in range(len(matrix[0])): #the new matrix
has m rows
    t_row = []
    for j in range(len(matrix)): #and n columns
      t_row.append(matrix[j][i])
    t_matrix.append(t_row)

  print("Transposed matrix")
  for row in range(len(t_matrix)):
    print(t_matrix[row])
```

```
#Pythonic way
print("(Python version) Transposed matrix")
t_matrix = [[matrix[j][i] for j in range(len(mat
rix))] for i in range(len(matrix[0]))]
for row in range(len(t_matrix)):
  print(t_matrix[row])
```

4. **Write a program creates and displays an identity matrix of dimension n ∈ (0,10).**

- Input: n=5
- Expected output:

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

```
matrix = []
n = int(input("Introduce the value of n: "))
if n>0 and n<10:
  for i in range(n):
    row = []
    for j in range (n):
      if i == j:
        row.append(1)
      else:
        row.append(0)
    matrix.append(row)
  for row in range(len(matrix)):
    print(matrix[row])
```

5. **Write a program that makes the sum of two matrix A and B. The resulting matrix C is one in which each element c(i,j)=a(i,j)+b(i,j).**

- Input:

```
A:

[2, 1, 10]
[3, 1, 1]

B:
[2, 9, 1]
[10, 7, 6]
```

- Expected output:

```
[4, 10, 11]
[13, 8, 7]
```

```
  n = 3 #columns
  m = 2 #rows
  A = [ [random.randint(0, 10) for i in range(n)]
for j in range(m)]
  B = [ [random.randint(0, 10) for i in range(n)]
for j in range(m)]
  C = []
  print("A")
  for row in range(len(A)):
    print(A[row])


  print("B")
  for row in range(len(B)):
    print(B[row])
  if len(A) == len(B) and len(A[0])==len(B[0]):
    for i in range(len(A)):
      row = []
      for j in range(len(A[0])):
        row.append(A[i][j] + B[i][j])
      C.append(row)

    print("C")
    for row in range(len(C)):
      print(C[row])
  else:
    print("The number of rows and columns must be
 the same.")

  #Pythonic way
  print("(Python version) C")
  C = [ [A[i][j] + B[i][j] for j in range(len(A[0]
))] for i in range(len(A)) ]
  for row in range(len(C)):
    print(C[row])
```

6. **The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970.**

Write a program to implement the Game of Life. The program will receive as an input the size of the board, n, and the number of generations. In each generation, the board will be updated following the next rules:

- A living cell with less than 2 living neighbors →die
- A living cell with 2 or 3 living neighbors → survives
- A living cell with more than 3 living neighbors → die
- A dead cell with exactly living neighbors → new born, alive

Improvement: the simulation will end after n generations or when there is no change.

```
  import random
  import time

  import matplotlib.pyplot as plt

  def plot_board(board):
      rows = len(board)
      cols = rows
      plt.figure(figsize = (cols, rows))
      plt.axes(aspect='equal')
      for i, j in [(i, j) for i in range(rows) for
 j in range(cols)]:
          if board[i][j]:
              plt.plot(i, j, 'o', color='black', m
arkeredgecolor='w', ms=10)
      plt.show()



  def demo_pattern():
      return [
          [ True,  True,   True,   True,   False,
False,  True,   False,  False,  False],
          [ False, True,   True,   False,  False,
True,   True,   True,   False,  False],
          [ True,  True,   False,  True,   False,
True,   False,  False,  True,   True],
          [ False, True,   False,  False,  False,
False,  False,  False,  False,  False],
          [ False, True,   True,   True,   False,
True,   True,   True,   False,  False],
          [ True,  False,  True,   True,   False,
True,   True,   True,   True,   True],
          [ True,  False,  True,   False,  False,
True,   True,   False,  True,   False],
          [ True,  False,  False,  True,   False,
False,  False,  True,   False,  False],
          [ True,  False,  False,  True,   False,
True,   True,   True,   True,   False],
          [ False, False,  True,   True,   False,
False,  False,  False,  True,   True]
      ]

  def pretty_print(board):
      if board:
          for row in range(len(board)):
              for column in range(len(board[0])):
                  if board[row][column]:
                      print("\t\u2665", end = "")
                  else:
                      print("\t\u271D", end = "")
              print()

  def show(board):
      if board:
          for row in range(len(board)):
              print(board[row])

  def count_alive_neighbours(board, row, col):
      alive = 0
      if board and row >= 0 and row < len(board) a
nd col >= 0 and col < len(board[0]):
          nrows = len(board)
          ncols = len(board[0])
          alive +=  (1 if board[row][col+1] else 0)
   if col+1<ncols else 0
          alive +=  (1 if board[row][col-
1] else 0)   if col-1>=0 else 0
          alive +=  (1 if board[row+1][col] else 0)
   if row+1<nrows else 0
          alive +=  (1 if board[row-
1][col] else 0)   if row-1>=0 else 0
          alive +=  (1 if board[row-1][col-
1] else 0) if row-1>=0 and col-1>=0 else 0
          alive +=  (1 if board[row-
1][col+1] else 0) if row-
1>=0 and col+1<ncols else 0
          alive +=  (1 if board[row+1][col-
1] else 0) if row+1<nrows and col-1>=0 else 0
          alive +=  (1 if board[row+1][col+1] else
0) if row+1<nrows and col+1<ncols else 0
      return alive

  if __name__== "__main__":
      n = 5
      generations = 10
      current_generation = [ [random.choice([True,F
alse]) for j in range(n)] for i in range(n) ]
      alive_neighbours = 0
      alive_cells = 0
      for gen in range(generations):
          print("Generation ", gen)
          pretty_print(current_generation)
          #plot_board(current_generation)
          alive_cells = 0
      #A living cell with less than 2 living neighb
ors-->die
      #A living cell with 2 or 3 living neighbors--
>survives
      #A living cell with more than 3 living neighb
ors->die
      #A dead cell with exactly living neighbors-
>new born, alive
          new_generation = current_generation.copy(
)
          size = len(current_generation)
          for i in range(size):
              for j in range(size):
                  alive_neighbours= count_alive_nei
ghbours(current_generation, i, j)
                  if current_generation[i][j]:
                      if alive_neighbours<2:
                          new_generation [i][j] = F
alse
                      elif (alive_neighbours>=2 and
 alive_neighbours<=3):
                          new_generation [i][j] = c
urrent_generation[i][j]
                      elif alive_neighbours>3:
                          new_generation [i][j] = F
alse
                  else:
                      new_generation [i][j] = alive
_neighbours == 3
```

```
          #time.sleep(0.2)
          #The new generation becomes the current g
eneration
          current_generation = new_generation.copy(
)
```

7. Write a program to manage a shopping cart with the following features.

- There is a list of products: rice (1 euro per package), apple (0.30 euro per unit) and milk (1 euro).
- The program shall ask the user to introduce a product in the shopping cart indicating the number of units. Each product can be added just once.
- The program shall display a menu with the following options:

```
  Shopping Cart
--------------------
1-
See shopping cart. (Shows the current shopping cart and
calculates the cost.)
2-
See products. (Shows the list of products and its unit c
ots.)
3-
Add product. (Asks the user for a product and a number o
f units. if the product is none, the program will come b
ack to the menu)
4-Make the order. (Clears the cart.)
5-Exit. (The program ends.)
```

```
 if __name__== "__main__":
   NAME, COST = (0,1)
   products = [("rice", 1), ("apple", 0.30), ("
milk", 1)]
   shopping_cart = set()
   option = ""

   while option != "5":
          print("\n\nShopping cart manager\n")

          print("1-See shopping cart.")
          print("2-See products.")
          print("3-Add product.")
          print("4-Make the order.")
          print("5-Exit.")
          option = input("Introduce your option: 
")
          if option == "1":
              if len(shopping_cart) == 0:
                  print("The shopping cart is emp
ty")
              else:
                  for product, quantity in shopping
_cart:
                      print("Product: {}, units: 
{}, total cost: {}".format(product[NAME], quantity
, quantity*product[COST]))
          elif option == "2":
                  for product in products:
                      print("Product: {}, cost: {
}".format(product[NAME], product[COST]))
          elif option == "3":
              added = False
              while not added:
                  p = input("Introduce product na
me or (none to come back to the menu): ")
                  added = p == "none"
                  if not added:
                      q = int(input("Introduce qua
ntity: "))
                      selected_products = [item for
 item in products if item[NAME] == p]
                      if len(selected_products) >
0:
                          selected_product = select
ed_products[0]
                      else:
                          selected_product = None
                      if selected_product and q > 0
 :
                          if len([ item for item i
n shopping_cart if selected_product[NAME] == item[0
][NAME]]) == 0:
                              shopping_cart.add((se
lected_product,q))
                              added = True
                          else:
                              print("The product i
s already in the cart.")
                      else:
                          print("The product name
or the quantity is not correct.")
          elif option == "4":
              print("The order has been processed
.")
              shopping_cart.clear()
          elif option == "5":
              print("The program is going to end.
")
          else:
              print("Invalid option")
```

8. Implement the shopping cart using dictionaries.

```
  if __name__== "__main__":

      products = { "rice": 1, "apple": 0.30, "milk
":1}
      shopping_cart = {}
      option = ""

      while option != "5":
          print("\n\nShopping cart manager\n")

          print("1-See shopping cart.")
          print("2-See products.")
          print("3-Add product.")
          print("4-Make the order.")
          print("5-Exit.")
          option = input("Introduce your option:
")
          if option == "1":
              if len(shopping_cart) == 0:
                  print("The shopping cart is emp
ty")
              else:
                  for product in shopping_cart.keys
():
                      product_cost = products[produ
ct]
                      quantity = shopping_cart[prod
uct]
```

```
                      print("Product: {}, units:
{}, total cost: {}".format(products[product], quan
tity, quantity*product_cost))
          elif option == "2":
                  for product, cost in products.ite
ms():
                      print("Product: {}, cost: {
}".format(product, cost))
          elif option == "3":
              added = False
              while not added:
                  p = input("Introduce product na
me or (none to come back to the menu): ")
                  added = p == "none"
                  if not added:
                      q = int(input("Introduce qua
ntity: "))
                      if q > 0:
                          shopping_cart[p] = q
                          added = True

                  else:
                      print("The product name or
the quantity is not correct.")
          elif option == "4":
              print("The order has been processed
.")
              shopping_cart.clear()
          elif option == "5":
              print("The program is going to end.
")
          else:
                print("Invalid option")
```

9. Implement the TIC, TAC, TOE game.

- The program shall display the board in each iteration.
- The program shall ask the user for the coordinates to situate a value.

```
  def print_board(board):
      size = len(board)
      for row in range(size):
          for col in range(size):
              print(str(board[row][col])+"\t|", en
d= "")
          print()

  if __name__ == "__main__":
      size = 3
      board = [ ["" for j in range(size)] for i in
 range(size)]
      current_player = "X"
      other_player = "O"
      end_game = False
      situated = 0
      while not end_game:
          print("Turn of player: "+current_player)
          print_board(board)
          set_position = False
          while not set_position:
              x =int(input("Select position x:"))
```

```
               y =int(input("Select position y:"))
               if x>=0 and x<=size and y>=0 and y<=s
ize and board[x][y] == "":
                   #Place
                   board[x][y] = current_player
                   set_position = True
                   situated = situated + 1
               else:
                   print("The position is already
set.")
           #Check if current player is winner by row
s
           winner = False
           row = 0
           while not winner and row<size:
               winner = board[row].count(current_pla
yer) == size
               row = row + 1
           #Check if current player is winner by col
s
           col = 0
           while not winner and col<size:
               row = 0
               matches = 0
               while not winner and row < size:
                   if board[row][col] == current_pla
yer:
                       matches = matches + 1
                   row = row + 1
               col = col + 1
               winner = matches == size
```

```
          #Check if current player is winner in mai
n diagonal
          matches = 0
          if not winner:
             for i in range(size):
                 if board[i][i] == current_player:
                     matches = matches + 1
             winner = matches == size
          #Check if current player is winner in sec
ondary diagonal
          if not winner:
              matches = 0
              for i in range(size):
                  if board[i][size-i -
1] == current_player:
                      matches = matches + 1
              winner = matches == size
          end_game = winner or situated == 9
          current_player, other_player = other_play
er, current_player

      if winner:
          print("The winner is: ", other_player)
      else:
          print("Draw")
      print_board(board)
```

10. **Implement the previous program making use of just one vector and slicing capabilities of Python lists.**

```
  def print_board(board):
      for i in range(n):
          print(board[n*i:n*(i+1)])

  if __name__=="__main__":
      n = 3
      size = 9
      board = ["" for x in range(n*n)]
      current_player = "X"
      other_player = "O"
      end_game = False
      situated = 0
      while not end_game:
         print("Turn of player: "+current_player)
         print_board(board)
         set_position = False
         while not set_position:
             x =int(input("Select position x:"))
             y =int(input("Select position y:"))
             if x>=0 and y >= 0 and (x*n+y)<size an
d board[x*n+y] == "":
                  #Place
                  board[x*n+y] = current_player
                  set_position = True
                  situated = 0
             else:
                  print("The position is already
set.")
             #Check if current player is winner by
rows
             winner = False
             i = 0
             while not winner and i<n:
                 winner = board[n*i:n*(i+1)].count(
current_player) == n
                 i = i + 1
             #Check if current player is winner by
cols
             i = 0
             while not winner and i<n:
                 winner = board[i:size:n].count(cur
rent_player) == n
                 i = i + 1

             if not winner:
                 #Check if current player is winner
 in the main diagonal
                 winner = board[:size:n+1].count(cu
rrent_player) == n
             if not winner:
                 #Check if current player is winner
 in the secondary diagonal
                 winner = board[n-1:size-1:n-
1].count(current_player) == n
         end_game = winner or situated == 9
         current_player, other_player = other_playe
r, current_player
      if winner:
          print("The winner is: ", other_player)
      else:
          print("Draw")
      print_board(board)
```

17 LAB 6-FUNCTIONS

In this chapter, we propose and solve some exercises about functions in Python.

In these exercises, we can always proceed solving the problems in a generic way or taking advantage of Python capabilities. As a recommendation, first, try the generic way (applicable to any programming language) and, then, using Python

As a good programming practice, our test cases should ensure that all branches of the code are executed at least once.

In the specific case of functions, we have always to keep in mind the next key points:"

- Design the functions defining a proper domain and range.
- Think in the pre-conditions to execute the function.
- Design (pure) functions without side-effects.

17.1 List of exercises

1. **Write a function that defines a set of input parameters and displays their identifiers, types and values. Invoke this function from the main function.**

- Input: my_function(1,"Hello",[1,2,3])

- Expected output:

```
1 of type  <class 'int'>  with id:  10914496
Hello of type  <class 'str'>  with id:  139654081206232
[1, 2, 3] of type  <class 'list'>  with id:  13965408135
6488
```

- Make use of the Python functions:

```
type(object)
id(object)
```

```
def my_fun(value, message, alist):
    print(value, "of type ", type(value), " with id
: ", id(value))
    print(message, "of type ", type(message), " wit
h id: ", id(message))
    print(alist, "of type ", type(alist), " with id
: ", id(alist))


if __name__=="__main__":
    my_fun(1,"Hello", [1,2,3])
```

2. **Write a function that takes two arguments, at lest one parameter with a default boolean value True, and prints out the values of all parameters.**

- Input:
    - `default_parameters(1)`
    - `default_parameters(1, False)`
- Expected output:

```
1
True
1
False
```

```
def default_parameters(a, b = True):
    print(a)
    print(b)
if __name__=="__main__":
    default_parameters(1)
    default_parameters(1, False)
```

3. **Write a function that takes three arguments (integer, string and list), modifies the value of such argument (displaying the id) and checks the value in the invocation point (displaying the id again).**

- Input:

```
a = 2
msg = "Hello"
alist = [1,2,3]
my_fun(a, msg, alist)
```

- Expected output (the ids may change):

```
10914528
139654071637640
139654071236424
Call function...
10914528
4
139654071637640
Other
139654071236424
[1, 3]
After calling function...
2
Hello
[1, 2, 3]
```

```
def my_fun(a,msg,alist):
    print(id(a))
    a = 4
    print(a)
    print(id(msg))
    msg = "Other"
    print(msg)
    print(id(alist))
    alist = [1,3]
    print(alist)

if __name__=="__main__":
    a = 2
    msg = "Hello"
    alist = [1,2,3]
    print(id(a))
    print(id(msg))
    print(id(alist))
    print("Call function...")
    my_fun(a, msg, alist)
    print("After calling function...")
    print(a)
    print(msg)
    print(alist)
```

4. **Write a function that takes a parameter (a list), appends a new element and prints out the content of the list in the invocation point.**

- Input: [1,2,3], a new element 4 is added.
- Expected output:

```
Before calling...
Hello
After calling...
Hello Mary
```

```
def add_element(alist):
    alist.append(4)
```

```
if __name__=="__main__":
  alist = [1,2,3]
  print("Before calling...")
  print(alist)
  add_element(alist)
  print("After calling...")
  print(alist)
```

5. **Write a function that takes a parameter (a string), appends a new string and prints out the content of the string in the invocation point.**

- Input: "Hello", a new string " Mary".
- Expected output:

```
Before calling...
Hello
After calling...
Hello
```

```
def add_message(msg):
  msg = msg + " Mary"

if __name__=="__main__":
  msg = "Hello"
  print("Before calling...")
  print(msg)
  add_message(msg)
  print("After calling...")
  print(msg)
```

6. **Write a function that takes two integer numbers and returns the addition of both numbers (an integer number).**

- Input: my_add(1,2).
- Expected output:

```
3
```

```
def my_add(a,b):
  return a+b


if __name__=="__main__":
  print(my_add(1,2))
```

7. **Write a function to compare two integer numbers. The function shall return:**

- 0 if both values are equal.
- 1 if the first parameter is greater than the second.
- -1 if the second parameter is greater than the first.

- Input:
  - `are_equal(1,2)`
  - `are_equal(2,1)`
  - `are_equal(1,1)`

- Expected output:

```
1
-1
0
```

```
def are_equal(a,b):
  if a == b:
    return 0
  elif a > b:
    return 1
  else:
    return -1
```

```
if __name__=="__main__":
  print(are_equal(1,2))
  print(are_equal(2,1))
  print(are_equal(1,1))
```

8. **Write a function to implement the absolute value of an integer number.**

- Input:
    - my_abs(5)
    - my_abs(-5)
- Expected output:

```
5
5
```

```
def my_abs(a):
  return ( -a if a<0 else a)
if __name__=="__main__":
  print(my_abs(5))
  print(my_abs(-5))
```

9. **Write a function that takes as an argument one tuple packing argument (*args) and diplays the values.**

- Input:
    - my_f(2,"Hello",[2,3])
- Expected output:

```
2
Hello
[2,3]
```

```
def my_f(*args):
  if args:
    for v in args:
      print(v)
if __name__=="__main__":
  my_f(2,"Hello",[2,3])
```

10. Write a function that takes as an argument one dictionary argument (kwargs) and diplays the values.**

- Input:
  - my_f(name="Mary", age=25)
- Expected output:

```
Key:  name , value:  Mary
Key:  age , value:  25
```

```
  def my_f(**kwargs):
    if kwargs:
      for k,v in kwargs.items():
        print("Key: ", k, ", value: ", v)
  if __name__=="__main__":
    my_f(name="Mary", age=25)
```

11. Write a function, is_leap, that given a year number, returns whether is a leap year (reuse the code in the previous chapter).

- Input: -1
- Expected output:

```
   False
```

- Input: 2019
- Expected output:

```
   False
```

- Input: 2020
- Expected output:

```
   True
```

```
def is_leap(year):
  is_leap_year = False
  if year >= 0:
    is_leap_year = (year % 4 == 0)
    is_leap_year = is_leap_year and (year % 100 !
= 0)
    is_leap_year = is_leap_year or (year % 400 ==
 0)
  return is_leap_year


if __name__=="__main__":
  print(is_leap(-1))
  print(is_leap(2019))
  print(is_leap(2020))
```

12. **Check the next function in which a documentation string is added.**

- The documentation is enclosed between `"""`.
- The documentation string is multiline.
- The documentation is string is situated after the function signature.
- The documentation can be checked in two manners:
- help(function_name): displays the function signature and the doc string.
- `function_name.__doc__`: returns the doc string.
- The reference for documentation strings is defined in the next PEP: https://www.python.org/dev/peps/pep-0257/.

```
def my_sum(alist):
  """Returns the sum of a list of numbers."""
  aggregated = 0
  if alist:
    for v in alist:
      aggregated += aggregated + v
  return aggregated
```

```
def my_sum2(alist):
  """Returns the sum of a list of numbers.

     Keyword arguments:
     alist: is a non null list of integer numbers
.

     Last update: January 2020
  """
  aggregated = 0
  if alist:
    for v in alist:
      aggregated += aggregated + v
  return aggregated

help(my_sum)
print(my_sum.__doc__)
help(my_sum2)
```

13. **Check the next function in which** metadata **to describe the function is added.**

- Internally, the annotations are added as a dictionary to the function object.
- The reference for annotations is defined in the next PEP: https://www.python.org/dev/peps/pep-3107/

```
#Annotations can be just string values.
def my_add(a: '<a>', b: '<b>') -
> '<return_value>':
  return a+b
```

```
print(my_add.__annotations__)


#Annotations can also include types. However, this is only documentation. it does not impose any restriction on the parameters.
def my_add2(a: int, b: int) -> float:
  return a+b


print(my_add2.__annotations__)
```

14. Write a program to calculate the factorial of an integer number.

- factorial (n) = n * factorial (n-1)

- Input: an integer number, 5
- Expected output:

```
120
```

```
def factorial(number):
  if number < 0:
    fact = -1
  elif number == 0 or number == 1:
    fact = 1
  else:
    fact = 1
    for i in range(1,number+1):
      fact *= i
  return fact


if __name__=="__main__":
  print(factorial(5))
```

15. Write a program to calculate the factorial of an integer number using a recursive function.

- factorial (n) = n * factorial (n-1)

A recursive function has two main parts:

- The basic case. In the factorial function, when n=0 or n=1.
- The recursive case. In the factorial function, when n>1.

- Input: an integer number, 5
- Expected output:

```
120
```

```
def factorial(number):
  if number < 0:
    return -1
  elif number == 0 or number == 1:
    return 1
  else:
    return number * factorial(number-1)

if __name__=="__main__":
  print(factorial(5))
```

16. Write a function to calculate the exponentiation of a number with base b, and exponent n.

$\text{base}^{\text{exponent}}$=base*base*base...*base

- Input: base 2, exponent 3
- Expected output:

```
8
```

```
def my_pow (base, exponent):
  mypow = 1
  if exponent > 0 :
      for i in range(0,exponent):
        mypow *= base
  return mypow

if __name__=="__main__":
  print(my_pow(2,3))
```

17. Write a function to detect if a number is a prime number.

- Input: 5, 8
- Expected output:

```
The number 5 is prime.
The number 8 is not prime.
```

```
def is_prime(n):
  n_divisors = 1
  divisor = 2
  while divisor < n and n_divisors <= 2:
    if n % divisor == 0:
      n_divisors = n_divisors + 1
    divisor = divisor + 1
  return n_divisors > 2

if __name__=="__main__":
  n = 8
  if is_prime(n):
    print("The number {} is not prime.".format(n)
)
  else:
      print("The number {} is prime.".format(n))
```

18. **Write a function to sum up the Fibonacci sequence of the first n numbers.**

```
fibonacci (0) = 0

fibonacci (1) = 1

fibonacci (n) = fibonacci(n-1) + fibonacci (n-2)
```

- Input: a positive number n=4
- Expected output:

```
Fibonacci of: 0 , sequence:  [0] , sum =  0
Fibonacci of: 1 , sequence:  [0, 1] , sum =  1
Fibonacci of: 2 , sequence:  [0, 1, 1] , sum =  2
Fibonacci of: 3 , sequence:  [0, 1, 1, 2] , sum =  4
Fibonacci of: 4 , sequence:  [0, 1, 1, 2, 3] , sum =  7
```

```
  def fibonacci_seq(n):
    fib_seq = []
    if n == 0:
      return [0]
    elif n == 1:
      return [0, 1]
    else:
      fn2 = 0
      fn1 = 1
      fib_seq.append(fn2)
      fib_seq.append(fn1)
      for i in range(2, n+1):
       fcurrent = fn1 + fn2
       fib_seq.append(fcurrent)
       temp = fn1
```

```
        fn1 = fcurrent
        fn2 = temp
      return fib_seq

  if __name__=="__main__":
    for n in range(5):
      seq = fibonacci_seq(n)
      print("Fibonacci of:",n,", sequence: ", seq,"
, sum = ",sum(seq))
```

19. Write a recursive function to calculate the next Fibonacci number of the first n numbers.

- Input: a positive number n=4
- Expected output:

```
Fib at position  0  is  0
Fib at position  1  is  1
Fib at position  2  is  1
Fib at position  3  is  2
Fib at position  4  is  3
```

```
  def fibonacci_r(n):
    if n == 0:
      result = 0
    elif n == 1:
      result = 1
    elif n > 1:
      result = fibonacci_r(n-1) + fibonacci_r(n-2)
    return result

  if __name__=="__main__":
    for n in range(5):
      print("Fib at position ",n," is ",fibonacci_r
(n))
```

20. **Write a function to calculate the binomial coefficient (reuse your previous factorial function):**

$$\binom{m!}{n!} = \frac{m!}{n!\ (m-n)!}$$

- Input: m = 10, n = 5
- Expected output:

```
   252
```

```
  def factorial(number):
    if number < 0:
      return -1
    elif number == 0 or number == 1:
      return 1
    else:
      return number * factorial(number-1)
  def combinatorial_number(m, n):
    factorialm = factorial(m)
    factorialn = factorial(n)
    factorialmn = factorial(m-n)
    return (factorialm / (factorialn * factorialmn)
)
  if __name__=="__main__":
    print(combinatorial_number(10,5))
```

21. **Write a program to display a menu with 3 options (to say hello in English, Spanish and French) and to finish, the user shall introduce the keyword "quit". If any other option is introduced, the program shall display that the input value is not a valid option.**

Refactor your previous code to provide a function that displays the menu and returns the selected option.

- Input: test the options
- Expected output:

```
----------MENU OPTIONS----------
1-Say Hello!
2-Say ¡Hola!
3-Say Salut!
> introduce an option or quit to exit...
```

```
def menu():
  option = "-1"
  while option not in {"1","2","3","quit"}:
    print("----------MENU OPTIONS----------")
    print("1-Say Hello!")
    print("2-Say ¡Hola!")
    print("3-Say Salut!")
    option = input("> introduce an option or quit
to exit...")
  return option

if __name__=="__main__":
  option = "-1"
  while option != "quit":
    option = menu()
    if option == "1":
      print("Hello!")
    elif option == "2":
      print("¡Hola!")
    elif option == "3":
      print("Salut!")
    elif option == "quit":
      print("...finishing...")
    else:
      print("Not a valid option: ", option)
```

22. Write a function to detect whether a number is a perfect number.

A number is perfect if the addition of all positive divisors is equal to the number.

- Input: n = 6
- Expected output:

```
   The number 6 is perfect.
```

```
  def is_perfect(n):
    perfect = 0
    for i in range (1,n):
      if n%i == 0:
        perfect = perfect + i
    return perfect == n

  if __name__=="__main__":
    n = 6
    if is_perfect(n):
      print("The number {} is perfect.".format(n))
    else:
      print("The number {} is NOT perfect.".format(
n))
```

23. Write a function to calculate the length of a sequence. If the list is None, the function shall return -1.

- Input: [3,4,5,6]
- Expected output:

```
The length of the list is: 4
```

```python
def my_len(alist):
  if alist:
    size = 0
    for v in alist:
      size += 1
  else:
    size = -1
  return size


if __name__=="__main__":
  length = my_len([1,2,3,4])
  print("The length of the list is: ", length)
```

24. Write a function that given a list of n numbers, calculates and returns the max value within the list and its position.

- Input: [8, 1, 9, 2]
- Expected output:

```
The max value is: 9 in position: 2.
```

```python
def max_position(values):
  max_value = -1
  position = -1
  if values and len(values) > 0:
    max_value = values[0]
    position = 0
    for i in range(1, len(values)):
      if values [i] > max_value:
        max_value = values [i]
        position = i
    return (max_value, position)


  if __name__=="__main__":
    values = [8,1,9,2]
    max_value, position = max_position(values)
    print("The max value is {} in position: {}".for
mat(max_value, position))
```

25. Write a function that given a list of n numbers and a target number k, counts the number of occurrences of k in the list.

- Input: [8, 1, 9, 1], k=1
- Expected output:

```
   The number 1 has 2 occurrences.
```

```
  def my_count(values, k):
    occurrences = 0
    for v in values:
      if k == v:
        occurrences += 1
    return occurrences



  if __name__=="__main__":
    values = [8,1,9,1]
    k = 1
    count = my_count(values,k)
    print("The number {} has {} occurrences.".forma
t(k, count))
```

26. Write a module named `my_list_functions.py` that contains functions for:

- Counting the number of occurrences of a value k in a list.
- Finding the position of the first/last/all apparition (this shall be a parameter) of the value k.
- Creating a new list in reverse order.
- Returning the first/last (this shall be a parameter) k numbers of the list.
- Making the union of two lists.
- Making the intersection of two lists.
- Creating chunks of k elements.

These functions will be invoked from a program including the module with the next directive:

```
from my_list_functions import *
```

```
#Content of the file my_list_functions.py
def count(alist, k):
    occurrences = 0
    if alist:
        for v in alist:
            if k == v:
                occurrences += 1
    return occurrences


def find_last(values, k):
    last_occur = -1
    if values:
        i = len(values)-1
        while i>=0 and last_occur == -1:
          if values[i] == k:
            last_occur = i
          i -= 1
    return last_occur
```

```
def find_first(values, k):
    first_occur = -1
    i = 0
    if values:
        size = len(values)
        while i<size and first_occur == -1:
          if values[i] == k:
            first_occur = i
          i += 1
    return first_occur

def find_all(values, k):
    positions = []
    if values:
        size = len(values)
        for i in range(size):
            if values[i] == k:
                positions.append(i)
    return positions

def find(values, k, strategy=0):
    if strategy == 0:
        return [find_first(values, k)]
    elif strategy == 1:
        return [find_last(values, k)]
    else:
        return find_all(values, k)

def reverse(values):
        if values:
            return values[::-1]
        return []

    def take_first(values, k):
        if values and k>0:
            return values[:k:]
        return []

    def take_last(values, k):
        if values and k>0:
            return values[len(values)-1:len(values)-
k-1:-1]
        return []

    def union(l1, l2):
        union_list = []
        if l1 and l2:
            union_list.extend(l1)
            for v in l2:
                if union_list.count(v) == 0:
                    union_list.append(v)
        return union_list

    def intersection(l1, l2):
        intersection_list = []
        if l1 and l2:
            for v in l1:
                if l2.count(v) != 0:
                    intersection_list.append(v)
        return intersection_list


    def chunks(values, k):
        chunks = []
        if values and k > 0:
            size = len(values)
            chunks = [values[i:k+i] for i in range(0,
 size, k)]
        return chunks
```

```
    #Content of the file app.py: both files must be i
n the same directory
    from my_list_functions import *


    if __name__=="__main__":
        values = [8,1,9,1]
        k = 1
        print("All the assertions must be true") #Cha
nge to assert
        print(count(values, k) == 2)
        print(len(find(values, k)) == 1)
        print(len(find(values, k, 1)) == 1)
        print(len(find(values, k, 2)) == 2)
        print(values[::-1] == reverse(values))
        print(len(take_first(values, 2)) == 2)
        print(len(take_last(values, 2)) == 2)
        print(union([4,5,6], [5,7,8,9]) == [4, 5, 6,
7, 8, 9])
        print(intersection([4,5,6], [5,7,8,9]) == [5]
)
        print(len(chunks(values,2)) == 2)
```

27. Refactor the TIC, TAC, TOE game implementing the following functions.

- The program shall have a function to get the position to situate a new value.
- The program shall have function to detect whether a player is winner by rows.
- The program shall have function to detect whether a player is winner by columns.
- The program shall have function to detect whether a player is winner in the main diagonal.
- The program shall have function to detect whether a player is winner in the secondary diagonal.
- The program shall have function to print the board.

```
#Version using only one vector to store the posit
ions


def print_board(board):
    for i in range(n):
        print(board[n*i:n*(i+1)])


def get_free_position(board):
    set_position = False
    x = -1
    y = -1
    size = len(board)
    n = len(board) // 3
    while not set_position:
        x =int(input("Select position x:"))
        y =int(input("Select position y:"))
        if x>=0 and y >= 0 and (x*n+y)<size and b
oard[x*n+y] == "":
            set_position = True
            else:
                print("The position is already set."
)
        return (x,y,)

    def is_winner_by_rows(board, current_player):
        i = 0
        winner = False
        n = len(board) // 3
        while not winner and i<n:
            winner = board[n*i:n*(i+1)].count(current
_player) == n
            i = i + 1
        return winner

    def is_winner_by_cols(board, current_player):
        winner = False
        i = 0
        size = len(board)
        n = len(board) // 3
        while not winner and i<n:
            winner = board[i:size:n].count(current_pl
ayer) == n
            i = i + 1
        return winner

    def is_winner_main_diagonal(board, current_player
):
        size = len(board)
        n = len(board) // 3
```

```
      return board[:size:n+1].count(current_player)
 == n

  def is_winner_secondary_diagonal(board, current_p
layer):
      size = len(board)
      n = len(board) // 3
      return board[n-1:size-1:n-
1].count(current_player) == n


  if __name__=="__main__":
      n = 3
      #Management in just one vector
      board = ["" for x in range(n*n)]
      current_player = "X"
      other_player = "O"
      end_game = False
      situated = 0
      while not end_game:
         print("Turn of player: "+current_player)
         print_board(board)
         (x,y) = get_free_position(board)
         board[x*n+y] = current_player
         situated += 1
         winner = is_winner_by_rows(board, current_
player) or is_winner_by_cols(board, current_player)
 or is_winner_main_diagonal(board, current_player)
or is_winner_secondary_diagonal(board, current_play
er)
         end_game = winner or situated == 9
```

```
          current_player, other_player = other_playe
r, current_player
       if winner:
           print("The winner is: ", other_player)
       else:
           print("Draw")
       print_board(board)
```

28. As a refactoring exercise, take exercises of previous chapters and select the parts of code that can be a function, e.g. string functions, vector operations, matrix operations, etc.

17.2 Advanced function concepts

High-order functions. A high-order function is a function that:

1. takes a function as a parameter or
2. returns a function.

For instance, we are going to define a function that receives as parameters:

- The function, f, to be applied to.
- The list of elements, alist

and returns, a list of the values after applying f to each of the elements in alist.

In this case, the example will return the square of the elements of the list.

- Input: f my own function to calculate the square of a number, and the list [1,2,3].
- Output:

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

- Modify the program to make a function that adds 2 to each of the elements of the list.
    - Output: [3, 4, 5]

```
  #We define a function that takes as a parameter a
 function f and a list, a list,
  #then applies the function f to any element in th
e list.
  def apply_f_to_list(f, alist):
    results = []
    for v in alist:
      value = f(v)
      results.append(value)
    return results

  def my_square(n):
    return n**2

  def add_2(n):
    return n+2

  if __name__=="__main__":
    values = [1,2,3]
    results = apply_f_to_list(my_square,values)
    print(results)
    results = apply_f_to_list(add_2,values)
    print(results)
```

Lambda functions. A **lambda function** is an anonymous function declared online.

- A lambda function can take any number of arguments.
- A lambda function can only return one expression.
- A lambda function is a Python function, so anything regarding parameters, annotations, etc. are applicable to lambda functions.

The theory behind lambda functions comes from the "Lambda Calculus". According to the official documentation, the Lambda functions follow the next grammar:

```
lambda_expr    ::=    "lambda"    [parameter_list]    ":"
expression     lambda_expr_nocond     ::=     "lambda"
[parameter_list] ":" expression_nocond
```

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression lambda parameters: expression yields a function object. The unnamed object behaves like a function object defined with:

```
def (parameters):

return expression
```

Lambda functions are mainly used in the following scenarios:

- Simple functions that we want to apply inline and we do not plan to reuse.

Lambda functions also come with some drawbacks:

- Syntax can be complex; it is not so intuitive as a regular function.
- Readability and understandability of the source code become complex.
- Need of thinking in a functional way (not intuitive when coming from imperative program.

The Python PEP8 document recommends the following:

Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

Yes:

```
def f(x): return 2*x
```

No:

```
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically 'f' instead of the generic ''. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression).

However, Lambda functions are elegant to solve specific problems and parametrize some expressions in functional programming.

```
    (lambda x, y: x + y)(2, 3)
```

```
   #We can assign the lambda function to a variable that will become a variable of type function.
   f_add_two_numbers = (lambda x, y: x + y)
   print(type(f_add_two_numbers))
   result = f_add_two_numbers(2,3)
   print(result)
```

```
   values = [1,2,3]
   #Let's make the function squares inline with a lambda expression
   results = apply_f_to_list(lambda x: x**2 ,values)
   print(results)
```

17.2.1 Functional programming in Python: main functions.

- `map`. Given a function, f, and a sequence, S, it returns a new iterator, f(S), where each element in f(S) is the result of applying the function f to each element in S.
- `reduce`. Given an operator, p, and a sequence, S, it returns a value, v, after aggregating the items in S using the operator p.
- `filter`. Given a filter function, filter, and a sequence, S, it returns a new iterator, filtered(S), where each element in filtered(S) meets the conditions established in the filter filter.
- `zip`. *The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.* (source: W3C Schools)

```
   #Applying a function through the map
   values = [1,2,3]
   squared_values = [x for x in map(lambda x: x**2, values)]
```

```
  print(squared_values)
```

```
  import functools
  import itertools
  import operator
  #Aggregating values with reduce: sum of values
  values = [1,2,3]
  print (functools.reduce(lambda a,b : a+b, values)
)

  #Get the max of a list
  print (functools.reduce(lambda a,b : a if a > b e
lse b, values))

  #Count words frequency
  words = [('grape', 2), ('grape', 3), ('apple', 5)
, ('apple', 1), ('banana', 2)]
  word_frequency = {key: sum(list([v[1] for v in gr
oup])) for key, group in itertools.groupby(words, o
perator.itemgetter(0))}
  print(word_frequency)
```

17.3 Relevant resources

- Interesting discussion about Lambda expressions: https://treyhunner.com/2018/09/stop-writing-lambda-expressions/
- Functional programming in Python: https://docs.python.org/3/howto/functional.html
- The Zip function: https://realpython.com/python-zip-function/

18 LAB 7-OBJECTS

In this chapter, we propose and solve some exercises about objects in Python. In the specific case of objects, we have always to keep in mind the next key points:"

- Apply theoretical concepts of Object-Oriented Programming (OOP)
- Analyze a problem
- Extract entities/classes and individuals
- Extract methods and properties
- Design a set of classes according to the analysis
- Implement the previous design

Given a statement of a problem, follow these steps to apply the principles of OOP.

- Identify classes (commonly nouns).
- Identify individuals (names or specific nouns).
- Identify properties (attributes that define a class).
    - Class properties
    - Object properties
- Identify methods (actions that can/must be performed by a class).
    - Class methods
    - Object methods
- Identify relationships among classes.

- Design the Python classes.
- Implement the Python classes.

18.1 List of exercises

1. **Define a class Person with two attributes: name (string) and age (integer). Create instances of Person changing these values and print out the instances.**

- Input: me = Person("Jose", 37)
- Expected output:

```
Person with name:  Jose  and age:  37
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

if __name__=="__main__":
  me = Person("Jose", 37)
  print("Person with name: ",me.name," and age: "
, me.age)
```

2. **Include a new method `__str__(self)` to return a string representation of the Person.**

- Input: me = Person("Jose", 37)
- Expected output:

```
Person with name Jose and age 37
```

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  if __name__=="__main__":
    me = Person("Jose", 37)
    print(me)
```

3. **Include a new method `speak(self, words)` that displays the words passed as parameter .**

- Input: speak(["Hello", "World"])
- Expected output:

```
   Person with name Jose and age 37
Speaking
Hello
World
```

```
  class Person:
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def speak(self, words):
      print("Speaking")
      for w in words:
        print(w)
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  if __name__=="__main__":
    me = Person("Jose", 37)
    print(me)
    me.speak(["Hello", "World"])
```

4. **Create a new instance attribute to store the current evolution with the value "HOMO SAPIENS".**

```
  class Person:
    evolution = "HOMO SAPIENS"
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def speak(self, words):
      print("Speaking")
      for w in words:
        print(w)
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  if __name__=="__main__":
    me = Person("Jose", 37)
    other = Person("Laura", 38)
    print(me.evolution)
    print(other.evolution)
    me.evolution = "OTHER"
    print (me.evolution)
    print(other.evolution)
```

5. **Create an extension of the class Person to represent a SuperHero. A SuperHero is a person with the capability of flying until X meters. Create a method to set this property.**

```
  class Person:
    evolution = "HOMO SAPIENS"
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def speak(self, words):
      print("Speaking")
      for w in words:
        print(w)
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  class SuperHero(Person):
    def __init__(self):
      self.name = "Super"
      self.age = 0
      self.flying_meters = 0
    def fly(self):
      print("Flying at ", self.flying_meters)
    def set_flying_meters(self, meters):
      if meters > 0:
        self.flying_meters = meters

  if __name__=="__main__":
    me = SuperHero()
    print(me)
    me.set_flying_meters(100)
    me.fly()
```

6. **Override the function `__str__(self)` for the class SuperHero.**

```
  class Person:
    evolution = "HOMO SAPIENS"
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def speak(self, words):
      print("Speaking")
      for w in words:
        print(w)
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  class SuperHero(Person):
    def __init__(self):
      self.name = "Super"
      self.age = 0
      self.flying_meters = 0
    def fly(self):
      print("Flying at ", self.flying_meters)
    def set_flying_meters(self, meters):
      if meters > 0:
        self.flying_meters = meters
    def __str__(self):
      return "SuperHero with name {} and age {}".fo
rmat(self.name, self.age)

  if __name__=="__main__":
    me = SuperHero()
    print(me)
```

7. Create two private attributes for the class Person.

```
class Person:
  __surname = "private surname"
  __other_attr = 0
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def get_surname(self):
    return self.__surname

if __name__=="__main__":
  me = Person("Jose", 37)
  print(me.get_surname())
```

8. Implement the methods `__eq__` and `__hash__`.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __eq__(self, other):
      return other and self.name == other.name an
d self.age == other.age

  def __ne__(self, other):
    return not self.__eq__(other)

  def __hash__(self):
      return hash((self.name, self.age))
```

```
if __name__=="__main__":
  me = Person("Jose", 37)
  other = Person("Jose", 37)
  print(id(me))
  print(id(other))
  print(me == other)
```

9. Implement the method `__del__`.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
  def __del__(self):
    print("Deleting person...")

if __name__=="__main__":
  me = Person("Jose", 37)
  del me
```

10. Solve the next problem creating Python classes.

- The company Chocomize (http://www.chocomize.com/) that enable us the possibility of creating our customized bar chocolates has requested an application to manage orders.
- A client (with a name) that is identified by an unique id, makes an order that can include only a type of bar chocolate (and quantity).
- An order is identified by an unique id and it has a cost or prize.
- A customized bar chocolate is also identified by an unique id. The bar chocolate is comprised of different toppings (type and quantity) and can be black, milk or white chocolate (each type has different unit cost).
    - Bar_chocolate_prize=unit_cost_of_chocolate*quantity+s

um(toppings_prize)

  - Topping_prize=unit_cost_of_topping*quantity

- In this first version we only need to support two types of toppings (Nuts and Fruits), basically: Hazel Nuts, Walnuts and Cranberries. Each topping is also identified by an unique id and it includes a natural language description and an unit cost.

```
  class Client:
      def __init__(self, identifier, name):
          self.identifier = identifier
          self.name = name


  class Order:
      def __init__(self, identifier, quantity, bar_
chocolate):
          self.identifier = identifier
          self.quantity = quantity
          self.bar_chocolate = bar_chocolate


      def calculate_prize(self):
          return self.quantity*self.bar_chocolate.c
alculate_cost()


  class ClientOrder:
      def __init__(self, client, order):
          self.client = client
          self.order = order


  class BarChocolate:
      def __init__(self, chocolate, quantity, toppi
ngs_order):
          self.chocolate = chocolate
```

```
            self.quantity = quantity
            self.toppings_order = toppings_order

        def calculate_cost(self):
            cost = 0
            cost = cost + self.quantity * self.chocol
ate.unit_cost()
            for topping_order in self.toppings_order:
                cost = cost + topping_order.calculate
_cost()
            return cost

    class Chocolate:
        def __init__(self, name, cost):
            self.name = name
            self.cost = cost
        def unit_cost(self):
            return self.cost

    class ToppingOrder:
        def __init__(self, topping, quantity):
            self.topping = topping
            self.quantity = quantity
        def calculate_cost(self):
            return self.topping.unit_cost() * self.qu
antity

    class Topping:
        def __init__(self, name, cost):
            self.name = name
            self.cost = cost

        def unit_cost(self):
            return self.cost

    class Nuts (Topping):
        pass

    class Fruits (Topping):
        pass


    if __name__=="__main__":
        #Data
        black = Chocolate("black",1)
        white = Chocolate("white",2)
        nuts_topping = Nuts("nuts", 1)
        apple_topping = Fruits("apple",2)
        #Client
        client = Client(1,"Jhon");
     #We are going to create an order of 100 black 
bar chocolate
    #including all topings in the same quantity (2)
    #We create our list of toppings. Initially it i
s empty.
        toppings_order = []
        toppings_order.append(ToppingOrder(nuts_toppi
ng, 2))
        toppings_order.append(ToppingOrder(apple_topp
ing, 2))
```

```
      bar_chocolate = BarChocolate(black, 10, toppi
ngs_order)

      #Make the order
      quantity = 100
      order = Order("1", quantity, bar_chocolate)
      client_order = ClientOrder(client,order)

      print("Client order cost: ", client_order.ord
er.calculate_prize()," cost units.")
```

18.2 Relevant resources

- Classes tutorial in Python: https://docs.python.org/3/tutorial/classes.html
- Objects and inheritance discussion: https://fuhm.net/super-harmful/

19 LAB 8-SEARCHING AND SORTING

In this chapter, we propose and solve some exercises about searching and sorting algorithms in Python.

19.1 List of exercises

1. **Implement the linear search algorithm for finding the first apparition of an element v in a list of random integer values. Make the next experiment:**

   - Define a small list (5 values) containing the element in the first position and log the execution time.
   - Define a small list (5 values) containing the element in the last position and log the execution time.
   - Define a large list (1000000 values) containing the element in the last position and log the execution time.
   - Define a large list (1000000 values) containing the element in the last position and log the execution time.
   - Repeat the experiment, with different list sizes, to test if the execution time is linear.

Example output:

```
1-Searching first in an small list...

    Time Taken: 0.00007 sec

2-Searching last in an small list...

    Time Taken: 0.00005 sec

3-Searching first in a large list...

    Time Taken: 0.00005 sec

4-Searching last in a large list...

    Time Taken: 0.14423 sec
```

Use the next functions:

- `randrange(n)` from module random.
- `time()` from module time.

```
import time
from random import randrange
def linear_search_first(values, target):
  found = False
  i = 0
  while not found and i<len(values):
    found = values[i] == target
    i += 1
  return found

def create_list(n):
 return [randrange(n) for i in range(n)]

#Small list
small_list = create_list(5)
target = small_list[0]
```

```
  print("1-Searching first in an small list...")
  t = time.time()
  linear_search_first(small_list, target)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))
  target = small_list[len(small_list)-1]
  print("2-Searching last in an small list...")
  t = time.time()
  linear_search_first(small_list, target)

  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))
  #Large list
  large_list = create_list(1000000)
  target = large_list[0]
  print("3-Searching first in a large list...")
  t = time.time()
  linear_search_first(large_list, target)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))

  target = large_list[len(large_list)-1]
  print("4-Searching last in a large list...")
  t = time.time()
  linear_search_first(large_list, target)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))
```

2. **Repeat the previous experiment with a binary search algorithm adding a new test case:**

- The target element is in the middle position of the list.

Expected output:

```
1-Searching first in an small list...

    Time Taken: 0.00004 sec

2-Searching last in an small list...

    Time Taken: 0.00005 sec

3-Searching middle in an small list...

    Time Taken: 0.00005 sec

4-Searching first in a large list...

    Time Taken: 0.00007 sec

5-Searching last in a large list...

    Time Taken: 0.00006 sec

6-Searching middle in a large list...

    Time Taken: 0.00007 sec
```

```
import time


def binary_search(values, target):
  first = 0
  last = len(values)-1
  found = False
  while first <= last and not found:
    mid = (first + last)//2
    if values[mid] == target:
      found = True
    else: #Discard half of the problem
      if target < values[mid]:
        last = mid - 1
        else:
            first = mid + 1
    return found

def create_list(n):
  return [i for i in range(n)]

#Small list
small_list = create_list(5)

target = small_list[0]
print("1-Searching first in an small list...")
t = time.time()
binary_search(small_list, target)
print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))

target = small_list[len(small_list)-1]
print("2-Searching last in an small list...")
t = time.time()
binary_search(small_list, target)
print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))

target = small_list[len(small_list)//2]
print("3-Searching middle in an small list...")
t = time.time()
binary_search(small_list, target)
print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))
```

```
  #Large list
  large_list = create_list(1000000)

  target = large_list[0]
  print("4-Searching first in a large list...")
  t = time.time()
  binary_search(large_list, target)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))

  target = large_list[len(large_list)-1]
  print("5-Searching last in a large list...")
  t = time.time()
  binary_search(large_list, target)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))

  target = large_list[len(large_list)//2]
  print("6-Searching middle in a large list...")
  t = time.time()
  binary_search(large_list, target)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))
```

3. **Implement the binary search algorithm with a recursive function.**

- Input: a list `[1,2,3,4,5]` and a target value: 3
- Expected output: `True`

```
  def binary_search(alist, first, last, target):
      if not first < last:
            return False
        mid = (first + last)//2
        if alist[mid] < target:
            return binary_search(alist, mid + 1, last
, target)
        elif alist[mid] > target:
            return binary_search(alist, first, mid, t
arget)
        else:
            return True


    values = [1,2,3,4,5]
    target = 3
    print(binary_search(values,0,len(values)-
1,target))
```

4. **Implement the bubble algorithm and sort a list of 1000 random integer values. Repeat the previous experiment but now with a sorted list. Compare execution times.**

- Expected output:

```
1-Sorting a random list...
    Time Taken: 0.07449 sec
2-Sorting a sorted list...
    Time Taken: 0.04275 sec
```

- Can you explain time difference?

```
    import time
    from random import randrange


    def bubble_sort(values):
        n = len(values)
        for i in range(n):
            for j in range(n-i-1):
                if values[j] > values[j+1]:
                  #Swap
                  values[j], values[j+1] = values[j+1
  ], values[j]

    def create_list(n):
     return [randrange(n) for i in range(n)]

    values = create_list(1000)

    print("1-Sorting a random list...")
    t = time.time()
    bubble_sort(values)
    print("\n\tTime Taken: %.5f sec\n" % (time.time()
  -t))

    print("2-Sorting a sorted list...")
    t = time.time()
    bubble_sort(values)
    print("\n\tTime Taken: %.5f sec\n" % (time.time()
  -t))
```

5. **Create a random list of 100 Person (name and age) and implement the next operations:**

- Sort (ascending) by name.
- Sort (descending) by age.

```
  class Person:
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)
    def __repr__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  def bubble_sort_name(values):
      n = len(values)
      for i in range(n):
          for j in range(n-i-1):
              if values[j].name < values[j+1].name:
                values[j], values[j+1] = values[j+1
], values[j]

  def bubble_sort_age(values):
      n = len(values)
      for i in range(n):
          for j in range(n-i-1):
              if values[j].age > values[j+1].age:
                values[j], values[j+1] = values[j+1
], values[j]

  def create_list(n):
    people = []
    for i in range(n):
      rand = randrange(n)
      people.append(Person("Name "+str(rand),rand))
    return people

  people = create_list(5)
  print(people)
  bubble_sort_name(people)
  print(people)
  bubble_sort_age(people)
  print(people)
```

6. **Reuse the people list and sort using the Python function `sorted`.**

```
  import operator

  class Person:
    def __init__(self, name, age):
      self.name = name
      self.age = age
    def __str__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)
    def __repr__(self):
      return "Person with name {} and age {}".forma
t(self.name, self.age)

  def create_list(n):
    people = []
    for i in range(n):
      rand = randrange(n)
```

```
        people.append(Person("Name "+str(rand),rand))
    return people

people = create_list(5)
print(sorted(people,key=operator.attrgetter("name
"), reverse=True))
print(sorted(people,key=operator.attrgetter("age"
), reverse=False))
```

7. **Sort a list of string values by length in ascending order.**

- Input: names = ['Smith', 'Simpson', 'Flanders']
- Expected output:

```
['Flanders', 'Simpson', 'Smith']
```

```
def criteria(s):
    return len(s)

names = ['Smith', 'Simpson', 'Flanders']

print(names)
names.sort(reverse=True, key=criteria)
print(names)
```

8. **Given a word (string), an anagram is another word produced by swapping two characters. Example: python and typhon are anagrams. Make a program to detect whether two words are anagrams.**

- Input: "python" and "typhon"
- Expected output: True

```
def is_anagram(w1, w2):
    if w1 and w2 and len(w1) == len(w2):
     return sorted(w1) == sorted(w2)


 print(is_anagram("python","typhon"))
```

9. **Take the implementation of the selection and insertion sort algorithms and trace how the list is changing in each iteration.**

- Input: [5, 8, 12, 1, 3, 4]
- Expected output:

```
Selection sort...
[5, 8, 12, 1, 3, 4]
     [1, 8, 12, 5, 3, 4]
     [1, 3, 12, 5, 8, 4]
     [1, 3, 4, 5, 8, 12]
     [1, 3, 4, 5, 8, 12]
     [1, 3, 4, 5, 8, 12]
     [1, 3, 4, 5, 8, 12]


Insertion sort...
[5, 8, 12, 1, 3, 4]
     [5, 8, 12, 1, 3, 4]
     [5, 8, 12, 1, 3, 4]
     [1, 5, 8, 12, 3, 4]
     [1, 3, 5, 8, 12, 4]
     [1, 3, 4, 5, 8, 12]
```

```
 def selection_sort(values):
   n = len(values)
   for i in range(n):
     min_idx = i
     for j in range(i+1, n):
         if values[min_idx] > values[j]:
             min_idx = j
     #Swap with the minimum value position
     values[i], values[min_idx] = values[min_idx],
values[i]
```

```
    print ("\t",values)

def insertion_sort(values):
  n = len(values)
  for i in range(1, n):
    key = values[i]
    j = i-1
    while j >=0 and key < values[j] :
      values[j+1] = values[j]
      j -= 1
    values[j+1] = key
    print ("\t",values)


values = [5, 8, 12, 1, 3, 4]
print("Selection sort...")
print(values)
selection_sort(values)

print("Insertion sort...")
values = [5, 8, 12, 1, 3, 4]
print(values)
insertion_sort(values)
```

10. Make a program to find out a number. Create a sorted list of random numbers and ask the user to find out a number performing a binary search.

- Input: [20, 15, 34, 17, 22, 8]
- Expected output:

```
   Introduce a value...19
...try again...
Introduce a value...22
```

```
def binary_search(values, target):
  first = 0
  last = len(values)-1
  found = False
  while first <= last and not found:
    mid = (first + last)//2
    if values[mid] == target:
      found = True
    else: #Discard half of the problem
      if target < values[mid]:
        last = mid - 1
      else:
        first = mid + 1
  return found

values = [20, 15, 34, 17, 22, 8]
#unique_values = list(dict.fromkeys(values))
values.sort()
print(values)
found = False
while not found:
  value = int(input("Introduce a value..."))
  found = binary_search(values, value)
  if not found:
    print("...try again...")
```

19.2 Relevant resources

- Interesting exercises (section "Web Exercises"): https://introcs.cs.princeton.edu/java/42sort/

20 LAB 9-RESOURCE MANAGEMENT

In this chapter, we propose some exploratory exercises about resource management in Python. The goal is to learn good programming practices more than solving concrete programming problems.

20.1 List of exercises

1. **Let's explore the information about our processes in Python. To do so, the module `psutil` will be used to gather the process information.**

- Learn more about this module in the following link: https://psutil.readthedocs.io/en/latest/

```
from psutil import Process
from datetime import datetime
from os import getpid

#Getting information from the current process
```

```
  p = Process(getpid())
  print(p)
  print("Name: " + p.name())
  print("ID: {}".format(p.pid))
  print("ID parent: {}".format(p.ppid))
  print("Memory info: {}".format(p.memory_info))
  print("User: {}".format(p.username))
  d = p.__dict__
  for k, v in d.items():
    print(k," :", v)
```

```
  import psutil
  #Getting some information from the CPU
  psutil.cpu_stats()
```

```
  import psutil
#Getting some information about the virtual memory
#Learn more: https://psutil.readthedocs.io/en/lates
t/#psutil.virtual_memory
  mem = psutil.virtual_memory()
  mem
```

```
  import psutil
  #Getting some information from the disk
  #Learn more: https://psutil.readthedocs.io/en/lat
est/#psutil.disk_partitions
  psutil.disk_partitions()
```

```
  import psutil
  #Getting some information from the network
  #Learn more: https://psutil.readthedocs.io/en/lat
est/#psutil.net_io_counters
  psutil.net_io_counters()
```

```
  psutil.net_connections()
```

```
  import psutil
  #Getting some information from the network
  #Learn more: https://psutil.readthedocs.io/en/lat
est/#psutil.sensors_temperatures
  psutil.sensors_temperatures()
```

2. **Gathering information of the system and processes.**

```
  import psutil, datetime
  psutil.boot_time()
```

```
  import psutil
  #Processes ids
  psutil.pids()
```

```
  import psutil
  #Process table
  for proc in psutil.process_iter(['pid', 'name', '
username']):
    print(proc.info)
```

3. **In this exploratory exercise, the low-level code of a Python program is presented. To do so, the dis module will be used.**

Learn more: https://docs.python.org/3/library/dis.html

To proceed, let's follow the next steps:

- Create a simple program to add two numbers.

- Save the file.
- Run the following command: `python -m dis dis_example.py`

```
#Program version 1
if __name__=="__main__":
    a = 2
    b = 3
    c = a + b
    print(c)
```

This is the output, our Python code in assembly code (34 lines):

- Three variables are created: `a, b and c`
- `a` and `b` are initialized with constant values
- `c` is assigned with the value of `a+b`.

```
8              0 LOAD_NAME                0 (__name__)
               2 LOAD_CONST               0 ('__main__')
               4 COMPARE_OP               2 (==)
               6 POP_JUMP_IF_FALSE       32

 10            8 LOAD_CONST               1 (2)
              10 STORE_NAME               1 (a)

 11           12 LOAD_CONST               2 (3)
              14 STORE_NAME               2 (b)

 12           16 LOAD_NAME                1 (a)
              18 LOAD_NAME                2 (b)
              20 BINARY_ADD
              22 STORE_NAME               3 (c)

 13           24 LOAD_NAME                4 (print)
              26 LOAD_NAME                3 (c)
              28 CALL_FUNCTION            1
              30 POP_TOP
         >>   32 LOAD_CONST               3 (None)
              34 RETURN_VALUE
```

```
#Program version 2
if __name__=="__main__":
    a = 2
    b = 3
    print(a+b)
```

Here, the output is quite similar, but we have saved 4 instructions.

```
  8           0 LOAD_NAME                0 (__name__)
              2 LOAD_CONST               0 ('__main__')
              4 COMPARE_OP               2 (==)
              6 POP_JUMP_IF_FALSE       28

 10           8 LOAD_CONST               1 (2)
             10 STORE_NAME               1 (a)

 11          12 LOAD_CONST               2 (3)
             14 STORE_NAME               2 (b)

 12          16 LOAD_NAME                3 (print)
             18 LOAD_NAME                1 (a)
             20 LOAD_NAME                2 (b)
             22 BINARY_ADD
             24 CALL_FUNCTION            1
             26 POP_TOP
        >>   28 LOAD_CONST               3 (None)
             30 RETURN_VALUE
```

- There are some code optimizations that can be done with techniques called "partial evaluation". These are advanced techniques that can save a lot of time by analyzing which parts of the code can be improved. For instance, taking a look to the "peval" module, it is focused in the following optimizations:
    - constant propagation
    - constant folding
    - unreachable code elimination
    - function in lining

Take a look to these introductory slides:

- http://www.cs.utexas.edu/~wcook/presentations/2011-PartialEval-simple.pdf

4. **Let's take a look now to the time that is spent in the different parts of our code. To do so, the cProfile module of the CPython interpreter will be used.**

Learn more: https://docs.python.org/3/library/profile.html

To proceed, let's follow the next steps:

- Create a simple program to invoke a function and wait five seconds.
- Save the file.
- Run the following command: `python -m cProfile profiling_example.py`

```
import time


def wait():
    time.sleep(5)


if __name__=="__main__":
    wait()
```

Here, we can see the calls and the time to execute each one. Our function is taking 5 seconds.

```
5 function calls in 5.006 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:l
ineno(function)
        1    0.000    0.000    5.006    5.006 profiling_
example.py:7(<module>)
        1    0.000    0.000    5.006    5.006 profiling_
example.py:9(wait)
        1    0.000    0.000    5.006    5.006 {built-
in method builtins.exec}
        1    5.006    5.006    5.006    5.006 {built-
in method time.sleep}
        1    0.000    0.000    0.000    0.000 {method 'd
isable' of '_lsprof.Profiler' objects}
```

There is an interesting library for getting information about the memory: Guppy.

5. **Let's explore now the number of object references. This is the information managed by the garbage collector. It is convenient recall here how references are increased:**

    - Assign it to another variable.
    - Pass the object as an argument.
    - Include the object in a list.

Notice that the call to the `refcount` increases the number of references.

```
import sys
def f(param):
  print(sys.getrefcount(a))

a = []
b = a
#f(a)
print(sys.getrefcount(a))
```

6. **Let's explore in detail the information of the garbage collector.**

```
import gc

gc.set_debug(gc.DEBUG_SAVEALL)

print(gc.get_count())
lst = []
lst.append(lst)
list_id = id(lst)
del lst
gc.collect()
for item in gc.garbage:
    print(item)
```

```
      print(list_id == id(item))
```

7. **One of the key points to improve the performance of our programs relies on the use of the Python standard library. To do so, let's compare the execution time between some operations with lists.**

```
  import time


  def create_list(n):
    alist = []
    i = 0
    while i<n:
      alist.append(i)
      i = i + 1


  def create_list_2(n):
    return [i for i in range(n)]



  n = 10000000
  print("Manual list creation")
  t = time.time()
  create_list(n)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))

  print("Comprehension list creation")
  t = time.time()
  create_list_2(n)
  print("\n\tTime Taken: %.5f sec\n" % (time.time()
-t))
```

8. **Let's iterate over a list. In this case, there is no difference. However, the use of a range is associated with a better memory management, since it will only keep a reference to the next value.**

```
import time

def iterate_values(alist):
  for v in alist:
    pass

def iterate_range(alist, n):
  for i in range(n):
    pass
def create_list_2(n):
  return [i for i in range(n)]

n = 1000000
alist = create_list_2(n)
print("Iterate using values")
t = time.time()
iterate_values(alist)
print("\n\tTime Taken: %.5f sec" % (time.time()-
t))

print("Iterate using range")
t = time.time()
iterate_range(alist, n)
print("\n\tTime Taken: %.5f sec" % (time.time()-
t))
```

9. **In this example, we are going to show how a list of uppercase strings can be created in different manners having impact in the execution time. Again, it is important to remark the use of built-in functions.**

```
  n = 1000000
  alist=[str(i) for i in range(n)]
  upper_list = []
  print("Manual conversion to uppercase")
  t = time.time()
  for word in alist:
        upper_list.append(word.upper())
  print("\n\tTime Taken: %.5f sec" % (time.time()-
t))


  print("Using builtins to uppercase")
  t = time.time()
  upper_list_2=map(str.upper,alist)
  print("\n\tTime Taken: %.5f sec" % (time.time()-
t))
```

10. **In this example, we will show that we should extract (if possible) from a loop those operations that are repetitive and time consuming.**

```
  import time


  s = "Hello"
  n = 1000000


  print("N times evaluation")
  t = time.time()
  for i in range(n):
    a = (len(s)+i)
  print("\n\tTime Taken: %.5f sec" % (time.time()-
t))


  print("N times NO evaluation")
  t = time.time()
  l = len(s)
  for i in range(n):
    a = (l+i)
  print("\n\tTime Taken: %.5f sec" % (time.time()-
t))
```

11. **In this example, we will explore the different ways of reversing a Python string.**

```
  import time


  def reverse_1(s):
    reverse = ""
    for i in range(len(s)-1,0,-1):
      reverse = reverse + s[i]
    return reverse


  def reverse_2(s):
    reverse = ""
    for i in range(len(s)):
      reverse = s[i] + reverse
    return reverse


  def reverse_slicing(s):
    return s[::-1]
  def reverse_list(s):
    sl = list(s)
    sl.reverse()
    return ''.join(sl)

  def reverse_builtin(s):
    return ''.join(reversed(s))

  s = "Hello"
  print("Backward")
  t = time.time()
  reverse_1(s)
  print("\n\tTime Taken: %.8f sec" % (time.time()-
t))

  print("Forward")
  t = time.time()
  reverse_2(s)
  print("\n\tTime Taken: %.8f sec" % (time.time()-
t))

  print("Slicing")
  t = time.time()
  reverse_slicing(s)
  print("\n\tTime Taken: %.8f sec" % (time.time()-
t))

  print("Reverse list")
  t = time.time()
  reverse_list(s)
  print("\n\tTime Taken: %.8f sec" % (time.time()-
t))

  print("Reverse builtin")
  t = time.time()
  reverse_builtin(s)
  print("\n\tTime Taken: %.8f sec" % (time.time()-
t))
```

```
  #To run from the command line
  if __name__ == '__main__':
      test_to_code = '''
  reverse_1('abcdefghijklmnopqrstuvwxyz')
      '''
      import timeit
      print(timeit.repeat(stmt=test_to_code, setup=
"from __main__ import reverse_1", repeat=5))
```

12. **String concatenation. A common discussion in the Python community is which is the best manner of concatenating strings since they are immutable and, in each operation, a new string will be generated consuming memory.**

Take a look to the official documentation:

- https://docs.python.org/3/faq/programming.html#what-is-the-most-efficient-way-to-concatenate-many-strings-together

Other options are:

- The use of StringIO.
- The use of f-string.

```
  msg = "value 1\n"
  msg +="value 2\n"
  my_msg=["line1","line2"]
  msg2 = "\n".join(my_msg)
  # Because strings are immutable, every time you a
dd an element to a string, Python creates a new str
ing and a new address.
```

```
  import time

  def concat_1(s1, s2):
      return s1 + s2

  def concat_2(s1, s2):
      return "%s%s" % (s1, s2)

  def concat_3(s1, s2):
      return "{0}{1}".format(s1, s2)



  s1 = "123"
  s2 = "abc"

  print("Short strings...")

  print("Concat +")
  t = time.time()
  concat_1(s1, s2)
  print("\n\tTime Taken: %.8f sec\n" % (time.time()
-t))
```

```
  print("Concat %")
  t = time.time()
  concat_2(s1, s2)
  print("\n\tTime Taken: %.8f sec\n" % (time.time()
-t))



  print("Concat format")
  t = time.time()
  concat_3(s1, s2)
  print("\n\tTime Taken: %.8f sec\n" % (time.time()
-t))

  s1 = "123" * 100000
  s2 = "abc" * 100000

  print("Large strings...")

  print("Concat +")
  t = time.time()
  concat_1(s1, s2)
  print("\n\tTime Taken: %.8f sec\n" % (time.time()
-t))

  print("Concat %")
  t = time.time()
  concat_2(s1, s2)
  print("\n\tTime Taken: %.8f sec\n" % (time.time()
-t))
```

```
  print("Concat format")
  t = time.time()
  concat_3(s1, s2)
  print("\n\tTime Taken: %.8f sec\n" % (time.time()
-t))
```

In general, some tips for improvement our Python programs are:

- Use built-in functions.
- Try to refactor large calculations in loops.
- Use `while 1` instead of `while True` in infinite loops.
- Use generators like `range`.
- Try to use constants when possible.
- Be careful with operations in immutable types like strings.
- Use an up-to-date Python version.
- ... Optimize what can be optimized not more.

20.2 Relevant resources

- https://wiki.python.org/moin/PythonSpeed/PerformanceTips/

21 LAB 10-INPUT/OUTPUT: FILES

In this chapter, we propose exercises to explore the file object in Python. It is important to remark that the operations presented should be protected with exceptions (out of scope in this course). Furthermore, there are many ways of reading and writing files, e.g. use of the with statement in Python.

IMPORTANT: to run these exercises it is preferred to use a local installation instead of a Jupyter Notebook in Google Colab (to avoid issues saving and loading files).

21.1 List of exercises

1. **Write a program to list directory contents: files and folders. Given an initial path, the program shall output the name of all files inside.**

Make use of the following file functions in the modules `os` and `os.path`:

- isfile: returns `True` if the path corresponds to a file.
- isdir: returns `True` if the path corresponds to a directory.

```
from os import listdir
```

```
  from os.path import isfile, isdir, join


  if __name__ == "__main__":
      path = "../"
      for f in listdir(path):
          if isfile(join(path, f)):
              print(f)
          elif isdir(join(path, f)):
              print(f)
```

2. **Write a program to list only the files ending with some extension (".pdf"). Given an input directory, the program shall list only the PDF files.**

```
  from os import listdir
  from os.path import isfile, isdir, join


  if __name__ == "__main__":
      path = "../"
      ext = ".pdf"
      for f in listdir(path):
          if isfile(join(path, f)) and f.endswith(e
xt):
              print(f)
```

3. **Write a program that given an input directory, displays the tree (each level of the tree shall display the file name including first as many tabs as the depth of the directory) of directories and files.**

```
  from os import listdir
  from os.path import isfile, isdir, join
```

```
def recursive_list(path, level):
    for f in listdir(path):
        if isfile(join(path, f)):
            print("\t"*level, f)
        elif isdir(join(path, f)):
            print("\t"*level, f)
            recursive_list(join(path, f), level+1
)


if __name__ == "__main__":
    path = "../"
    recursive_list(path, 0)
```

4. **Write a program to serialize a list of strings in a text file ("messages.txt"). Each list item shall be written in a new line.**

- Input: ["Hello", "Mary", "How are you?"]
- Output: a new file "message.txt" with the following contents,

```
Hello
Mary
How are you?
```

```
if __name__ == "__main__":
    msgs = ["Hello", "Mary", "How are you?"]
    path = "messages.txt"
    file = open(path,"w")
    for msg in msgs:
        file.write(msg+"\n")
    file.close()
```

5. **Refactor the program in the previous exercise to make use of the file method `writelines`.**

```
if __name__ == "__main__":
    msgs = ["Hello", "Mary", "How are you?"]
    path = "messages.txt"
    file = open(path,"w")
    #Add new line to each string in the list
    file.writelines([m+"\n" for m in msgs])
    file.close()
```

6. **Refactor the exercise 4, to write all elements in a list with just one statement and making use of the file method write.**

```
if __name__ == "__main__":
    msgs = ["Hello", "Mary", "How are you?"]
    path = "messages.txt"
    file = open(path,"w")
    #Add new line to each string in the list
    file.write('\n'.join(msgs))
    file.close()
```

7. **Write a program to read the file created in exercise 4, "messages.txt", loading each line in a list.**

- Input: the filename "messages.txt"
- Output: `['Hello', 'Mary', 'How are you?']`

```
if __name__ == "__main__":
    msgs = []
    path = "messages.txt"
    file = open(path,"r")
    for line in file:
        msgs.append(line.strip())
    file.close()
    print(msgs)
```

8. **Write a program to print only the first line of a given file.**

- Input: the filename "messages.txt"
- Output: "Hello"

```
if __name__ == "__main__":
    path = "messages.txt"
    file = open(path,"r")
    print(file.readline())#only the first line
    file.close()
```

9. **Write a program to read the file created in exercise 4, "messages.txt", loading all lines at once.**

- Input: the filename "messages.txt"
- Output: `['Hello', 'Mary', 'How are you?']`

```
if __name__ == "__main__":
    msgs = []
    path = "messages.txt"
    file = open(path,"r")
    msgs = file.readlines()
    file.close()
    print([m.strip() for m in msgs])
```

10. **Write a program to read the file created in exercise 4, "messages.txt" char by char.**

- Input: the filename "messages.txt"
- Output: `HelloMaryHow are you?`

```
if __name__ == "__main__":
    msgs = []
    path = "messages.txt"
    file = open(path,"r")
    for line in file:
        for ch in line.strip():
            print (ch,end="")
    file.close()
```

11. Refactor the exercise 4 to use the Python with statement.

- Learn more: https://docs.python.org/3/reference/compound_stmts.html#grammar-token-with-stmt

```
if __name__ == "__main__":
    msgs = []
    path = "messages.txt"
    with open(path,"r") as file:
        msgs = file.readlines()
    file.close()
    print([m.strip() for m in msgs])
```

12. Write a program to read and write the information of a list of people as a CSV ("Comma Separated Values") file.

  - The program shall serialize a list of people.
  - The first line shall contain the person field names: name, age and location.
  - Each line shall contain the information of a person.
  - Each field shall be separated by a comma, ",".
- Input:

```
person = {"name":"Mary", "age":20, "location":"Madrid"}
people = [person]
```

- Output: "people.txt"

```
name,age,location,
Mary,20,Madrid,
```

```
from os import listdir
from os.path import isfile, isdir, join

def save_people(people, filename):
    if people and filename:
        file = open(filename,"w")
        header = False
        for person in people:
            if not header:
                for k in person.keys():
                  file.write(k+",")
                file.write("\n")
                header = True
            for v in person.values():
                file.write(str(v)+",")
            file.write("\n")
        file.close()

def load_people(filename):
    people = []
    if filename and isfile(filename):
        first = True
        file = open(filename,"r")
        for line in file:
            person = {}
            if first:
                keys = line.split(",")
                first = False
```

```
              else:
                  values = line.split(",")
                  for i in range(len(keys)):
                      if i < len(values):
                          if len(keys[i].strip())>0
 and len(values[i].strip())>0:
                              person[keys[i]] = val
ues[i]
                  people.append(person)
      else:
          print("File does not exist: ",filename)

      return people

  if __name__ == "__main__":
      filename = "people.txt"
      person = {"name":"Mary", "age":20, "location"
:"Madrid"}
      people = [person]
      print(people)
      save_people(people,filename)
      loaded_people = load_people(filename)
      print(loaded_people)
```

13. Refactor exercise 12 to make use of the specific CSV file objects.

- Learn more about
  the `csv` module: https://docs.python.org/3/library/csv.html
- `csv.DictWriter`
- `csv.DictReader`

```
from os import listdir
from os.path import isfile, isdir, join
import csv

def save_people(people, filename):
    if people and len(people) > 0 and filename:
        keys = people[0].keys()
        csv_file = open(filename,"w")
        dict_writer = csv.DictWriter(csv_file, ke
ys)
        dict_writer.writeheader()
        dict_writer.writerows(people)
        csv_file.close()

def load_people(filename):
    people = []
    if filename and isfile(filename):
      csv_file = open(filename, encoding="utf-
8")
      reader = csv.DictReader(csv_file)
      for row in reader:
            people.append(row) #Ordered dict
      csv_file.close()
    else:
        print("File does not exist: ",filename)

    return people

if __name__ == "__main__":
    filename = "people.txt"
```

```
      person = {"name":"Mary", "age":20, "location"
:"Madrid"}
      people = [person]
      print(people)
      save_people(people,filename)
      loaded_people = load_people(filename)
      print(loaded_people)
```

14. **Refactor exercise 12 to make use of the specific Pickle file objects (to serialize Python objects as binary files).**

- Learn more about the `pickle` module: https://docs.python.org/3/library/pickle.html
- `pickle.dump`
- `pickle.load`

```
  from os import listdir
  from os.path import isfile, isdir, join
  import pickle

  def save_people(people, filename):
      if people and filename:
          file = open(filename,"wb")
          pickle.dump(people, file)
          file.close()

  def load_people(filename):
      people = []
      if isfile(filename):
        file = open(filename,"rb")
        people = pickle.load(file)
        file.close()
      else:
          print("File does not exist: ",filename)
      return people


  if __name__ == "__main__":
      filename = "people.bin"
      person = {"name":"Mary", "age":20, "location"
  :"Madrid"}
      people = [person]
      print(people)
      save_people(people,filename)
      loaded_people = load_people(filename)
      print(loaded_people)
```

15. Refactor exercise 12 to make use of the specific JSON ("JavaScript Object Notation) file objects (to serialize Python objects as JSON files).

- Learn more about the `json` module: https://docs.python.org/3/library/json.html
- `json.dump`
- `json.load`

```
from os import listdir
from os.path import isfile, isdir, join
import json


def save_people(people, filename):
    if people and filename:
        file = open(filename,"w")
        json.dump(people, file)
        file.close()
```

```
  def load_people(filename):
      people = []
      if isfile(filename):
        file = open(filename,"r")
        people = json.load(file)
        file.close()
      else:
          print("File does not exist: ",filename)
      return people


  if __name__ == "__main__":
      filename = "people.json"
      person = {"name":"Mary", "age":20, "location"
:"Madrid"}
      people = [person]
      print(people)
      save_people(people,filename)
      loaded_people = load_people(filename)
      print(loaded_people)
```

16. Write a program to generate (read/write) and to play a quizz from a file.

- The program shall accept questions in a simplified version of the AIKEN format.
- The AIKEN format for multiple-choice questions follows the next syntax:
    - Each question has:
        - A statement question: a text line.
        - A response: a letter followed by . and the text option.
        - A valid response: "ANSWER:" and the correct response letter.

As an example of question:

```
How many different countries are in the wine review data
set which name length is 5?
A.    3
B.    4
C.    9
D.    12
ANSWER: B
```

- After loading the questions, the program shall display the questions to the user and ask for an option. If the option is valid, a new point is added.
- After finishing the quiz, the program shall display the grade.
- Input: the file "q1-aiken"
- Output: a file "q1-aiken-saved" and the quiz

```
from os import listdir
from os.path import isfile, isdir, join
import json

def save_aiken(questions, filename):
    if questions and filename:
        file = open(filename,"w")
        for question in questions:
            file.write(question["statement"]+"\n"
)
            for option in question["choices"]:
                file.write(option[0]+"."+"\t"+opt
ion[1]+"\n")
            file.write("ANSWER:"+question["correc
t_answer"]+"\n")
        file.close()

def load_aiken(filename):
    questions = []
```

```
      if filename and isfile(filename):
        question = None
        file = open(filename,"r")
        for line in file:
            if len(line.strip()) > 0:
                if line.startswith("ANSWER:"):
                    question_answer = line.split("A
NSWER:")[1]
                    question["correct_answer"] = qu
estion_answer.strip()
                elif len(line.split("\t"))>0 and li
ne.split("\t")[0][1]==".":
                    tokens = line.split("\t")
                    question_choice = tokens[0][0].
strip() #Remove .
                    question_choice_text = tokens[1
].strip()
                    question["choices"].append((que
stion_choice, question_choice_text))
                else:
                    if question :
                        questions.append(question)
                    question = {"statement":line.st
rip(), "choices":[]}

        if question:
            questions.append(question)
        file.close()
      else:
          print("File does not exist: ",filename)
      return questions
```

```
def show_question(question):
    print(question["statement"]+"\n")
    #Sort options and print
    sorted(question["choices"], key=lambda choice
: choice[0], reverse=True)
    for option in question["choices"]:
        print(option[0]+"."+"\t"+option[1]+"\n")

def play(questions):
    points = 0
    if questions:
        for question in questions:
            show_question(question)
            option = input("Select your choice: "
).upper()
            if option == question["correct_answer
"]:
                points += 1
    return points

if __name__ == "__main__":
    filename = "q1-aiken"
    questions = load_aiken(filename)
    print("\n\nLoaded original questions..."+str
(len(questions)))
    print(questions)
    save_aiken(questions,filename+"-saved")
    questions = load_aiken(filename+"-saved")
    print("\n\nLoaded saved questions..."+str(le
n(questions)))
```

```
        print(questions)
        print("\n\nPlaying game...")
        points = play(questions)
        print("You have got: "+str(points)+" points.
")
```

21.2 Relevant resources

- https://docs.python.org/3/library/filesys.html
- https://www.python-course.eu/python3_file_management.php
- https://realpython.com/working-with-files-in-python/

22 BIBLIOGRAPHY

[1] J. Catsoulis, Designing embedded hardware, 2nd ed. Sebastopol, CA: O'Reilly, 2005.

[2] D. E. Knuth, The art of computer programming, 3rd ed. Reading, Mass: Addison-Wesley, 1997.

[3] M. Summerfield, Programming in Python 3: a complete introduction to the Python language. Upper Saddle River, NJ: Addison-Wesley, 2009.

[4] L. Ramalho, Fluent Python, First edition. Sebastopol, CA: O'Reilly, 2015.

[5] D. M. Beazley and B. K. Jones, Python cookbook, Third edition. Sebastopol, CA: O'Reilly, 2013.

[6] P. Barry, Head first Python, 1. Aufl. Sebastopol, Calif.: O'Reilly, 2011.

[7] K. Beck, Test-driven development: by example. Boston: Addison-Wesley, 2003.

[8] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," Computer, vol. 38, no. 9, pp. 43–50, Sep. 2005, doi: 10.1109/MC.2005.314.

[9] G. J. Myers, C. Sandler, and T. Badgett, The art of software testing, 3rd ed. Hoboken, N.J: John Wiley & Sons, 2012.

[10] R. C. Martin, Ed., Clean code: a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall, 2009.

[11] D. Thomas and A. Hunt, The pragmatic programmer, 20th anniversary edition: journey to mastery, Second edition. Boston: Addison-Wesley, 2019.

[12] L. Bass, I. M. Weber, and L. Zhu, DevOps: a software architect's perspective. New York: Addison-Wesley Professional, 2015.

[13] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," IEEE Softw., vol. 33, no. 3, pp. 94–100, May 2016, doi: 10.1109/MS.2016.68.

[14] J. Smeds, K. Nybom, and I. Porres, "DevOps: A Definition and Perceived Adoption Impediments," in Agile Processes in Software Engineering and Extreme Programming, vol. 212, C. Lassenius, T. Dingsøyr, and M. Paasivaara, Eds. Cham: Springer International Publishing, 2015, pp. 166–177.

[15] S. Nelson-Smith, Test-driven infrastructure with Chef, Second edition. Beijing ; Sebastopol, CA: O'Reilly, 2013.

[16] F. Faber, "Testing in DevOps," in The Future of Software Quality Assurance, S. Goericke, Ed. Cham: Springer International Publishing, 2020, pp. 27–38.

ABOUT THE AUTHOR

Jose María Alvarez Rodríguez holds a PhD from the University of Oviedo (Spain) in the field of linked data and web services applied to e-Procurement. He also held as Marie-Curie postdoc within the RELATE-ITN network in the field of quality and cloud computing. Currently, he is Associate Professor within the Computer Science and Engineering Department of the Carlos III University of Madrid (Spain). His main research interests are semantic-based technologies, linked (open) data, service-oriented computing, knowledge engineering, systems and software engineering, design patterns, large scale architectures and graph analysis techniques applied to fields such as e-Procurement, Systems Engineering or Open Science. He is member of INCOSE, ProSTEP, OMG and OSLC.