Programming notes

Maria Dolors Ayala Vallespí
Grup d'Informàtica a l'Enginyeria
E.T.S.E.I.B.
Universitat Politècnica de Catalunya

Fall 2021

.

Índex

1 Introduction

1.1 Programming and programming languages

A computer does two things only but it does them extremely well. A computer performs calculations at a very high speed and is able to deal with large amounts of information.

On the other hand, humans want to take advantage of these features by solving problems with computer programs. This is a wide research field called computational thinking or programming.

For more than 70 years there has been a large amount of research in this area and there are basically two programming paradigms: declarative and imperative.

The declarative paradigm is based on statements of fact. For example these mathematical formulas: $diameter = 2radius$ and $circle\_area = \pi radius^2$ are statements of fact but they don't tell us how to compute the diameter of a circle given its area.

On the other hand the imperative paradigm gives recipes describing how to compute something. In the previous example, the recipe will be:

1. compute the radius using the expression: $radius = \sqrt{circle\_area/\pi}$
2. compute the diameter as the double of the radius

This is a very simple recipe but following this paradigm you will be able to use known methods or to devise new ones to solve a given problem.

In this course we will follow the **imperative paradigm**.

A method or algorithm is a finite list of instructions that when executed on a provided set of inputs will proceed through a set of well-defined states and eventually produce an output.

When an algorithm is aimed to a computer it is called a program or a function. So in this course we will learn to write programs or, more specifically, functions.

Programming languages

Earlier computing machines were fixed-program computers which means that they were allowed to execute only one specific method, as for example solving systems of linear equations, but nothing more. Nowadays we have still some very simple computers that use this approach as a four-function calculator and many devices in the mechanical industry, just to put some examples.

On the other hand, current computers as personal computers are stored-program computers: they can store and execute many different programs. This approach was first formulated by Alan Turing in the beginning of the last century.

A programming language is a formal language for writing programs. Languages as Python, C, Java, etc. are said to be Turing-complete which means that can be used to write programs to be executed in stored-program computers.

An important thing about programming languages is that humans write programs and computers execute them. Actually a computer will execute exactly what you tell it to do. Sometimes this isn't what you wanted it to do: and this is the worst error that you could make.

We can classify programming languages using several dimensions:

1. Low-level versus high-level. It refers to whether the instructions are at machine level or they are more abstract operations that have been provided by the language designer and that are more familiar to human beings.
2. General versus targeted to an application domain. A general purpose language allows to write programs that solve several kinds of problems whereas there are languages aimed at a particular application: for example to design web pages (Adobe Flash) or to deal with data bases (Excel).
3. Interpreted versus compiled. An interpreted language executes directly the program instructions and a compiled language first translates the source code into the machine code (by a program called a compiler) and then executes it. Interpreted languages are generally easier to debug and therefore are a good choice for beginners. On the other hand compiled languages are generally more efficient concerning execution time and space.

Python is a **high-level**, **general purpose** and **interpreted** language. It is relatively easy to learn and there are a large number of free libraries that interface to it which provide extended functionality.

A programming language has a set of basic elements together with a syntax and a semantics. A Python program (script) is a sequence of statements, instructions or commands that are interpreted and executed by the Python interpreter, using basic elements as values, types, variables, ...

1.2 Values and types

Value

Values (also called objects) are the core elements that a Python program manipulates.

Type

Any value falls into a class or type. The type defines the range of values and the operations that we can do with objects of this type.

Types are either scalar or non-scalar. Scalar values are indivisible while non-scalar values have an internal structure.

Python has three scalar basic types: `int` (integer), `float` (real) and `bool` (Boolean).

The `type` function shows the type of a value.

Examples:

```
>>> type (3)
<class 'int'>
>>> type(3.0)
<class 'float'>
>>> type(True)
<class 'bool'>
```

1.3 Type `int`

Although integers are infinite when we work with a computer we have limitations. The type `int` allows to represent a subset of the integers between a MIN negative number and a MAX positive number. For example, an `int` of 4 bytes can represent the following range of values: $[-2^{31}, 2^{31} - 1]$.

Literals of type `int` are represented in the same way as integers: with a sequence of digits and the character - for negative values. Examples: `3, -24, 6789, 0` ....

The type `int` operators are: +, -, *, /, //, % and **. Note that the negative (opposite) operation and the subtract operation share the same symbol: -. Operators that act on two operands are called **binary operators** while those that act on one operand are called **unary operators**.

The result of an addition, subtraction and product of two type `int` values is an `int` value.

Concerning division, there are two division operators. The first one is denoted by the symbol / and the result is a `float` regardless of whether the operands are `int` or `float`. Examples:

```
>>> 6/4
1.5
>>> 6.0/4
1.5
>>> 6.0/4.0
1.5
>>> 4/2
2.0
```

The other division operator, the so called **floor division** operator, //, obtains the integer part of the result. In this case, the result is an `int` if both operands are integers and a `float` otherwise.

Examples:

```
>>> 6//4
1
>>> 6.0//4
1.0
>>> 5//-2.0
-3.0
```

The **floor division** and the **modulo** operator, % allow to perform integer division. Let D and d be the dividend and divisor, respectively, in an integer division, then the corresponding quotient and remainder are obtained using these operators:

$$D = q * d + r$$

$$q = D//d$$

$$r = D\%d$$

`D%d` means D modulo d or D mod d for short.

```
>>> 6%4
2
>>> 18%5
3
```

Integer division applied to negative operands gives a result that might be surprising. Analyze the following examples and observe that the behavior with negative operands is the same that with positive ones.

```
>>> 7//2
3
>>> 7%2
1
>>> 7//-2
-4
>>> 7%-2
-1
```

The exponentiation operator uses the symbol double star (asterisk): **. Example: the mathematical expression 2^4 is written as `2**4` in Python.

1.4 Type `float`

Type `float` is used to represent a subset of the real numbers. Type `float` in Python has a wider range of values than type `int`. However, `float` values have limitations not only in their magnitude but also in their precision.

`float` values are represented by a sequence of digits, the dot character (mandatory) representing the decimal point and the - character for negative values. They can also be represented in scientific notation. Examples: `3., -24.45, 6789.3, 0.0`. In scientific notation `2.3E-5` stands for 2.3 10^{-5}.

The name `float` comes from the way values of this type are stored in the computer memory, similar to scientific notation and called floating point representation.

`float` operators are: +, -, *, /, //, **.

You can operate `int` objects together with `float` objects. The rules that govern the type of the result in these operations, depending on the type of the operands, are shown in the following table:

op	`int`	`float`
`int`	`int`	`float`
`float`	`float`	`float`

1.5 Boolean: type (`bool`)

Boolean values are represented by the `bool` type. The two unique Boolean values are: `False` and `True`.

1.5.1 Comparison operators

Comparison operators are external: they apply to objects of the same type and give a Boolean result. These operators are: `==` (equals), `!=` (not equal to), `<` (less than), `<=` (less than or equal to) (at most), `>` (greater than), `>=` (greater than or equal to) (at least).

Examples:

```
>>> 3 > 4
False
>>> 2.5 < 3
True
```

1.5.2 Boolean operations

There are the following logic or Boolean operators: `and`, `or` and `not`, which are defined by the truth tables:

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

not	True	False
	False	True

Using comparison operators together with Boolean operators we can build more complex Boolean expressions. The expression `a <= b and b <= c` means that both `a` is less than or equal to `b` and `b` is less than or equal to `c`.

Comparisons can be chained arbitrarily: `a <= b <= c < d > e` is equivalent to `a <= b and b <= c and c < d and d >e`. Example:

```
>>> 1 <= 6 <= 10
True
>> 1 <= 6 and 6 <= 10
True
```

1.6 String type (`str`)

The `str` (string) type allows to represent string characters, i.e., character sequences as names, codes, phone numbers, etc. String values are represented in either single quotes or double quotes. Examples: `'abc', 'Tomorrow,' ''Hanna's bag'', 'aa@aaa.aa', '44444444H'`.

The type `str` is a non-scalar type, it is a sequence type as other types we will see in the following sections. In this section we will see operations for strings considered as a block (as if they were a scalar type). Later on you will see how to access and operate with each character of the sequence individually. These block operations are concatenation (+) and replication (*). These operators are said to be overloaded: they have different meanings deending upon the types of the objects to which it is applied. For instance operator + means addition when applied to two numbers and concatenation when applied to two strings.

In addition, the `len` function of the standard library gives the length of a string (its number of characters). Examples:

```
>>> 'abc' + 'def'
'abcdef'
>>> len ('measures')
8
>>> 'abc'*3
'abcabcabc'
```

Operators + and * are said to be overloaded: they have different meanings depending on the types of objects to which they are applied. Nonetheless, they have equivalent meanings when applied to `int` (or `float`) and to `str`: `2+3` can be expressed as `1+1+1+1+1` and `'you'+ 'me'` as `'y'+'o'+'u'+'m'+'e'`. Concerning replication (*), expression `2*3` is equivalent to expression `3+3` and expression `2*'you'` is equivalent to expression `'you'+'you'`.

Characters are encoded internally in the computer following the codification shown in the ASCII table. This codification is arbitrary except for three groups of characters: digits, uppercase letters and lowercase letters. These groups in the ASCII table follow the numeric and alphabetical order respectively so that the following intervals exist:

- $['0','9']$
- $['A','Z']$
- $['a','z']$

String comparison is based on the so-called lexicographical order, which is a generalization of the alphabetical order. Examples:

```
>>> 'a' < 'v'
True
>>>'7' > '9'
False
>>> 'leg' < 'wing'
False
>>> 'abracadabra' > 'amber'
False
>>>'100' < '200'
True
>>> '100' < '2'
True
```

1.7 Expressions

An expression is a combination of variables, values, operators and calls to functions, which follows some syntactic rules. Expressions are **evaluated** by the Python interpreter giving a result. The values or variables involved are the **operands**. Expressions in programming languages are written in line format: no subscripts, superscripts, horizontal line (fraction bar) are allowed. These are some examples of mathematical expressions translated to Python syntax:

mathematics	Python
2^4	`2**4`
$\frac{2+x}{3}$	`(2+x)/3`
$\frac{6}{1+x}$	`6/(1+x)`
$\frac{8}{2x}$	`8/(2*x)`
$\frac{2x}{8}$	`2*x/8`

Rules of precedence:

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. The arithmetic operators have the usual precedence. The following table shows the rules of preference:

()	parenthesis have the highest preference
**	right-associative (evaluated from right to left)
-	opposite operator
*, /, //, %	left-associative
+, -	left-associative
==, !=, ...	
not	
and	
or	

The order of evaluation can be modified using parentheses. Operators with the same precedence are evaluated from left to right (left-associative) except for the exponentiation operator that, like in the mathematical context, is right-associative. Example: `2**3**2` is the Python expression corresponding to $2^{(3^2)} = 2^9 = 512$ and `(2**3)**2` is the Python expression corresponding to $(2^3)^2 = 2^6 = 64$.

1.8 Variables

A variable is a name or identifier that refers (is associated) to a value.

The **identifier or name** of a variable can be arbitrarily long. Variable names can contain both letters and digits and the underscore character (_). They must begin with a letter or

an underscore character. Variable names are case sensitive: `volume` and `Volume` are different variable names.

Variable names cannot be **reserved words** or **keywords**. Keywords, as `int`, `def`, `if`, `False`, ... are used in the syntax of the language and cannot be used as variable names.

It is advisable to choose names that are meaningful in order for the programs to be readable and self-explanatory. Variables may have compound names. In this case, you cannot use the space character to separate the corresponding parts. Instead, you may use the underscore character: `sphere volume` is an incorrect name whereas `sphere_volume` is a correct one. Another possible notation is to capitalize the parts name: `sphereVolume`.

1.9 Assignment statement

A statement is an instruction that can be interpreted and executed by the Python interpreter. In this section the assignment statement is introduced. Other kinds of statements as `if`, `for` or `while` statements will be introduced in the following sections.

The assignment statement associates a value to a variable (name). The syntax of the assignment statement is:

```
var_name = expression
```

This syntax and its semantics (meaning) is described next.

- The assignment operator is represented by the symbol `=` (equal). It should not be confused with the equality comparison operator represented by the `==` (double equal) symbol.
- When reading this code we must say:
  - **var_name is assigned expression**, or
  - **var_name gets the value of expression**
  - DON'T say *var_name equals expression.*
- The assignment operator, `=`, is not commutative:
  - On the left-hand side of the assignment statement there is always the variable name, **var_name**.
  - On the right-hand side of the assignment symbol there is always an expression.
- The behavior of the assignment statement is as follows:
  1. the right-hand side expression is evaluated producing an object
  2. the variable on the left-hand side is assigned this value

Examples:

```
>>> area = 5
>>> Area = area * 2
>>> area, Area
(5, 10)
>>> Area = Area + 10.5
>>> result = Area == 30
>>> area, Area, result
(5, 20.5, False)
>>> type(area)
<class 'int'>
>>> type(Area)
<class 'float'>
>>> type(result)
<class 'bool'>
```

A value can have one, more than one or no name associated with it. On the other hand, a name can be associated to a unique value at a time.

1.10 Library of built-in functions

Python has a set of built-in functions (the **builtins** library). See the Python documentation for a complete list and documentation of these functions. For a short documentation, the Python shell can also be used with functions `dir` and `help`:

list	`dir(__builtins__)`
documentation	`print(__builtins__.functionname.__doc__)` `help (functionname)`, `q` to quit

Some of these functions are cast functions (functions that convert from a type to another): `int`, `float`, `str`. Examples:

```
>>> str (42.85)
'42.85'
>>> str (3*4)
'12'
>>> int (3*4.2)
12
>>> float (3*4)
12.0
>>> int('100')
100
>>> int('100 miles')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '100 miles'
```

The `bool` function always returns `True`, except in the following cases:

```
>>> bool(0.0)
False
>>> bool(0)
False
>>> bool('')
False
```

Therefore, the behavior of the `bool` function applied to the two Boolean values is not as expected:

```
>>> bool('True')
True
>>> bool('False')
True
```

There is the `eval` function, also in the `builtins` library, that evaluates any expression represented as a string and that can be used to convert a Boolean value into a string. Examples:

```
>>> eval('True')
True
>>> eval('False')
False
>>> eval ('3*4')
12
>>> a = 5
>>> eval  ('a*2+6 < 20 and a < 10')
True
```

Next a list of the more common functions of the `builtins` library is shown:

- `print`: shows the value of a variable or expression
- `type`: shows the type of a variable or value
- `len`: returns the number of elements in a sequence
- `abs`: computes the absolute value
- `round`: rounds a real value
- `max, min`: maximum and minimum of a set of values

```
>>> n = 2
>>> print (2*n)
>>> round(45.987623, 4)
45.9876
>>> max (4, 6, 5, 3)
6
>>> min (4, 6, 5, 3, 8)
3
```

1.11 Library of mathematical functions

The mathematical library includes common mathematical functions as `exp(x)` (e^x), `log(x)`, `sqrt(x)` ($\sqrt{(x)}$), `factorial(x)`, ..., trigonometric functions as `sin(x), cos(x), tan(x)`, ..., conversion functions between degrees (sexagesimal) and radians (`degrees(x), radians(x)`) and constants as `PI, e,` ...

Trigonometric functions require the angle to be expressed in radians.

For libraries other than the `builtins` library it is necessary to import the corresponding modules. You can use the following alternative syntaxes:

import library	function call
`import math`	`math.sin(x)`
`from math import *`	`sin(x)`

```
>>> import math
>>> math.exp(4)
54.598150033144236
>>> math.log(54.598150033144236)
4.0
>>> math.sqrt(81)
9.0
>>> rad = math.radians(30)
>>> rad, math.sin(rad)
(0.5235987755982988, 0.49999999999999994)
```

See the Python documentation for a complete list and documentation of these functions. For a short documentation, the Python shell can also be used:

list	`dir(math)`
documentation	`print(math.functionname.__doc__)` `help (math.functionname)`

1.12 Input

The `input` built-in function gets input from the program user:

```
>>> x = input ('Enter the value of x: ')
Enter the value of x: 25
>>> x
'25'
>>> x = float(input ('Enter the value of x: '))
Enter the value of x: 25
>>> x
25.0
```

1.13 Composition

A program is a composition of statements.

The sequential composition is the more simple one. It can be represented as:

```
statement 1
statement 2
........
statement n
```

which means that statement $i+1$ is executed after statement i. A program with this composition executes one statement after another in the order in which they appear and stops when it runs out of statements.

Example 1. Value exchange between two variables: version 1

```
>>> a = 34      # line 1
>>> b = 58      # line 2
>>> aux = a     # line 3
>>> a = b       # line 4
>>> b = aux     # line 5
>>> a, b
(58, 34)
```

Example 2. Value exchange between two variables: version 2

```
>>> a = 34      # line 1
>>> b = 58      # line 2
>>> a = a+b     # line 3
>>> b = a-b     # line 4
>>> a = a-b     # line 5
>>> a, b
(58, 34)
```

1.14 Comments

Programs are executed by computers but they are also read by humans. For this reason, the code must be clear also for humans. Some suggestions to enhance the code readability are using meaningful variable names that reflect their content and including explanatory comments, aimed solely at humans and that will not be executed by the computer. These comments are preceded by the `#`, number (also named sharp), sign and are not interpreted by Python.

1.15 Program tracing

Tracing a program means hand-simulating the code: tracing the value of the variables along the program statements when the program is executed. It is a rudimentary but still useful

technique to understand some results of a program. There exist some applications that perform automatic program tracing as the Python tutor. We can also do a manual tracing. In this case, we construct a table where the columns correspond to variables and rows to statements.

In this example the traces of the programs in the previous section are shown.

Version 1			
line	a	b	aux
0	?	?	?
1	34	?	?
2	34	58	?
3	34	58	34
4	58	58	34
5	58	34	34

Version 2		
line	a	b
0	?	?
1	34	?
2	34	58
3	92	58
4	92	34
5	58	34

1.16 Input/Output

Nowadays computer interfaces are very sophisticated, they must be user friendly and enjoyable. They must guide and help users to perform tasks safely, effectively, and efficiently. Actually the code for the interface is about 80% of the total code of an application.

Programming interfaces is out of the scope of this course as we will concentrate on the application core and we will design black-box functions, as presented in the next chapter.

Therefore, in this section, only the rudimentary functions to input and output data are presented.

The `input` built-in function gets input from the program user. The `print` built-in function shows values, variables and expressions. Examples:

```
>>> x = input ('Enter the value of x: ')
Enter the value of x: 25
>>> x
'25'
>>> x = float(input ('Enter the value of x: '))
Enter the value of x: 25
>>> x
25.0
>>> import math
>>> print('The square root of', x, 'is:', math.sqrt(x))
The square root of 25.0 is: 5.0
```

The `input` function always returns a string. To obtain a value of a different type, we must use the corresponding conversion functions.

The `print` call in the example prints 4 values separated by commas, correspo to two literal strings, a variable , `x`, and the result of an expression `math.sqrt(x)`.

1.17 Exercises

1. Write in the Python shell the group of sentences they take to perform the following steps: assign an integer value to the variable `kgtot` which indicates an amount expressed in kg and then obtain the following variables corresponding to its decomposition in metric tonnes (t), metric quints (q), myriagrams (mag) and kilograms (kg).

```
>>> kgtot = 45391
>>> t = kgtot // 1000
>>> remainder = kgtot % 1000
>>> q = remainder //100
>>> remainder = remainder%100
>>> mag = remainder //10
>>> kg = remainder%10
>>> print(t,'tonnes,',q,'quints,',mag,'mags and',kg,'kg')
45 tones, 3 quints, 9 mags and 1 kg
```

2. A body moves with an uniformly accelerated rectilinear motion with an acceleration a expressed in m/s^2 (m: meters, s: seconds), an initial velocity v_0 expressed in m/s and an initial position x_0 in m. Write in the Python shell the group of sentences than perform the following steps: define names and assign initial values to variables a, v_0, x_0 and to variable x indicating a determined final position of the body and expressed in Km (Kilometers). Then compute the value corresponding to the velocity of the body when it reaches the mentioned final position in km/h (h: hour). You can apply the following expression: $v^2 - v_0^2 = 2a(x - x_0)$

```
>>> import math
>>> x0 = 134
>>> v0 = 11
>>> a = 2
>>> x0 = 134.0
>>> v0 = 11.2
>>> a = 2.3
>>> x = 4.4
>>> v = math.sqrt(v0**2 + 2*a*(x*1000 - x0))
>>> v = v/1000*3600
>>> print ('Expected velocity is: ', round(v, 2), 'km/h')
Expected velocity is:  505.91 km/h
```

2 Functions

2.1 Definition

A function is a sequence of statements that perform a specific task and that is identified by a name. Their primary purpose is to help us organize programs into pieces corresponding to the set of statements that solve a specific problem.

We have to take into account that the more code a program contains, the more chance there is to make errors and the harder the code is to maintain. Therefore, splitting our code into meaningful pieces will reduce time for debugging and maintaining. Moreover, functions allow to generalize and reuse code.

The syntax for a function definition is:

```
def function_name (formal parameters list):   # header
    statements                                # body
    return return_values
```

The Python definition of a function uses two keywords: `def` and `return`. The `def` keyword stands for *define*. Function names follow the same rules shown for variable names. Following the function name there is a list of parameters enclosed in parenthesis. This list can be empty or contain one or more parameters separated from one another by commas. Parenthesis are always required.

Function definition is the first **compound statement** we will see, all of which have the same pattern:

- A **header** line which begins with a keyword and ends with the character `:` (a colon).
- A **body** of one or more **indented** statements. The Python style guide recommends a 4 spaces indentation.

A function receives input data through its **parameters**, and computes and returns results as **return values**. Parameters that appear in the header are called **formal parameters** because the function is defined formally using them. These parameters are variables represented by names following the well-known rules. They cannot neither be values nor expressions. They specify the data that the function needs in order to be correctly executed.

The last statement executed by a function is the `return` statement. It begins with this keyword followed by a list of returning expressions. This list can be empty or contain one or more expressions. Parenthesis are not required. Python evaluates the return expressions and returns the corresponding values. When there is no return expression, the keyword `return` can be omitted. In this case the function returns a special value called `None`[1]. The return statement is the last statement executed by a function and therefore if there is code after this statement, it will never be executed (dead or unreachable code). However it is not necessarily the bottom-most statement)

A file that contains the code for one or more functions is called a **module**.

[1] `None` is the unique value of the non scalar type `NoneType`

2.2 Calling a function

A function is executed when it is called (or invoked). The **call statement** or **function call** has the following syntax:

```
variables = function_name (actual parameters list)
```

So far, we have already called functions of the built-ins or math library. Parameters in a function call are referred as **actual parameters**. Actual parameters are the specific values to which the function is applied. Inside the function, the values that are passed (actual parameters) get assigned to the formal parameters. Actual parameters can be any legal expression.

Each **formal parameter** is assigned the value of an **actual parameter**. Therefore, there must be the same number of formal and actual parameters and the correspondence of each formal-actual parameter pair is usually **positional** which means that the first formal parameter is assigned the value of the first actual parameter, the second formal parameter is assigned the value of the second actual parameter, and so on. The same positional rule applies for the return values in a function and the variables that get these values in the function call.

There is another way to bound that formal parameters get bound to actual parameters. Using the so called **Keyword arguments**, formal parameters are bound to actual parameters by using the name of the formal parameter.

2.3 Local variables and scoping

Any variable used within a function that is not a formal parameter is called a **local variable**.

Each function defines a new **namespace** or **scope**. All the formal parameters and local variables of a function exist only within the scope of the definition of this function.

2.4 Examples

Write the function `sphere_volume` that takes the radius of a sphere and returns its volume.

```
import math

def sphere_volume (radius):
    return 4/3 * math.pi * radius ** 3
```

This function has one parameter (`radius`) and one return value, the volume of the corresponding sphere, obtained from the expression `4/3 * math.pi * radius ** 3`

Example of use:

```
>>> sphere_volume (2.4) #volume of a sphere with radius = 2.4
57.90583579096705
>>> round(sphere_volume (2.4), 2) #rounded to 2 decimals
57.91
```

```
# volume of three spheres with radius = 2.4
# function call as part of an expression
>>> three_spheres = 3 * sphere_volume(2.4)
>>> round(three_spheres, 2)
173.72
```

Let's now write another function named `diventer` that, from two integers corresponding to the dividend and divisor, returns the quotient and the remainder.

```
def diventer(dividend, divisor):
   quotient = dividend//divisor
   remainder = dividend%divisor
   return quotient, remainder
```

This is a function with two parameters (`dividend`, `divisor` and two return values obtained from the expressions `dividend//divisor`, which corresponds to the quotient, and `dividend%divisor`, which corresponds to the remainder.

We save this function into the file (module) with the name `diventer.py`.

Example of use:

```
>>> mydividend = 25    # assign values to the actual parameters
>>> mydivisor = 4
# call to diventer function
>>> myquotient, myremainder = diventer (mydividend, mydivisor)
# shows variable values that get the return values
>>> myquotient, myremainder
(6, 1)  # Python always shows them enclosed into parenthesis
# call to diventer function: actual parameters are values
>>> diventer(67, 7)
(9, 4)                  # return values
>>> a = 64
>>> b = 27
# call to diventer function:
# actual parameters are general expressions
>>> diventer (a+5, b//3)
(7, 6)                  # return values
>>> diventer(divisor = 4, dividend = 25)  # using keyword
   arguments
(6, 1)
```

In the Python built-in library there is a function named `divmod` that performs this task.

2.5 Flow of control

Program instructions can be executed sequentially but sometimes execution may jump to a specified instruction. This is called flow of control.

A general application consists of a large number of functions that can call one another. It can be represented by a directed graph in which an edge going from function A to function B means that function A calls function B. Therefore, a function can call other functions and can be called by other functions.

As we might expect, we have to define a function before we call it. Moreover, the statements inside a function are not executed until the function is called.

Example: Write the following three functions:

- function `area_square` that returns the area of a square given its side length
- function `area_circle` that returns the area of a circle given its radius
- function `square_diff` that takes a float value corresponding to the side of a square, and returns the area difference between the square and its inscribed circle. This function must call the previous ones

Functions code:

```
import math
def area_square (side):
    return side **2
def area_circle(radius):
    return math.pi * radius **2
def square_diff(lside):
    asquare = area_square (lside)
    acircle = area_circle (lside/2)
    return asquare - acircle
```

Example of use:

```
>>> round(square_diff(10.0), 2)
21.46
```

`square_diff` first evaluates the expression in the first assignment statement and this implies a change of the flow: the control moves to the `area_square` function and its formal parameter `side` is assigned the value of the actual parameter `lside`. Then the unique statement in function `area_square` is executed: the expression `side*2` is evaluated and the corresponding value returned. Flow then moves backwards to the main function `square_diff` and its first assignment statement is fully accomplished, variable `asquare` now has the value returned by function `area_square`. Then the following assignment statement of function `square_diff` is executed but first the expression `lside/2` (actual parameter) is evaluated so that its value could be assigned to the formal parameter `radius` at the same time that the control moves to the `area_circle` function, etc.

At this point it will be clarifying to run this example with the Python tutor application. Include here the first statement to be executed: `asqr = square_diff(10)` at the end of the code. First, Python gets acquainted of all the elements that intervene in this module: the math library, and all three functions. See how the flow control jumps or moves from a point to another when the function calls and the return statements are executed.

2.6 Exercises

1. Write the function `unities` that takes an amount expressed in kg (`int`) and returns the values corresponding to its decomposition in metric tonnes (t), metric quints (q), myriagrams (mag) and kilograms (kg). Save this function into file `unities.py`.

```
def unities(kgtot):
    t = kgtot //1000
    remainder = kgtot%1000
    q = remainder //100
    remainder = remainder%100
    mag = remainder //10
    kg = remainder%10
    return t, q, mag, kg
```

2. A body moves with an uniformly accelerated rectilinear movement with an acceleration a expressed in m/s^2 (m: meters, s: seconds), an initial speed v_0 expressed in m/s and an initial position x_0 in m. Write function `mrua` that takes these parameters, a, v_0, x_0, and another parameter, x, indicating a determined final position of the body and expressed in Km (Kilometers) and returns the value corresponding to the speed of the body when it reaches the mentioned final position in km/h (h: hour). You can apply the following expression: $v^2 - v_0^2 = 2a(x - x_0)$. Save this function into file `velmrua.py`.

```
import math
def mrua(x0, v0, a, x):
    v = math.sqrt(v0**2 + 2*a*(x*1000 - x0))
    v = v/1000*3600
    return v
```

3. Write function `multiple(x, y)` that takes two integers, x and y and returns `True` if x is multiple of y or `False` otherwise. Save this function into file `multiple.py`.

```
def multiple (x, y):
    result = x % y == 0
    return result
```

   This function applies the modulus operator to determine if `x` is multiple of `y`: the Boolean expression `x%y==0` is equivalent to saying that x is is multiple of y. Note that this function can be simplified by doing without variable `result`.

```
def multiple (x, y):
    return x % y == 0
```

   Example of use:

```
>>> multiple (4, 5)
False
>>> multiple (10, 2)
True
```

```
>>> multiple (10, 5)
True
```

4. Write the following functions and save them into the module `character.py`

   - Function `uppercase` that takes a string with a unique character and returns `True` if this character is an uppercase letter or `False` otherwise.
   - Function `lowercase` that takes a string with a unique character and returns `True` if this character is a lowercase letter or `False` otherwise.
   - Function `digit` that takes a string with a unique character and returns `True` if this character is a digit and `False` otherwise.
   - Function `others` that takes a string with a unique character and returns `True` this character is neither an uppercase nor a lowercase nor a digit or `False` otherwise. This function must call the three previous ones.

```
def uppercase (c):
    return  'A' <= c <= 'Z'

def lowercase (c):
    return  'a' <= c <= 'z'

def digit (c):
    return '0' <= c <= '9'

def others (c):
    return not (uppercase (c) or lowercase(c) or digit(c))
```

   These functions are based on the fact that in the ASCII code, there exist the intervals for uppercase letters (['A','Z']), lowercase letters (['a','z']) and digits (['0','9']).

   Examples of use:

```
>>> uppercase ('A')
True
>>> uppercase ('e')
False
>>> lowercase ('B')
False
>>> lowercase ('t')
True
>>> digit ('8')
True
>>> digit ('p')
False
>>> others ('k')
False
>>> others (':')
True
```

3 Conditional statement

3.1 Definition

The conditional statement (`if` statement) allows to check a condition and modify the program flow accordingly.

A condition is represented by a Boolean expression and therefore checking a condition means that the corresponding Boolean expression will be evaluated and depending on the resulting value, `True` or `False`, the program will execute one or another set of statements or code blocks.

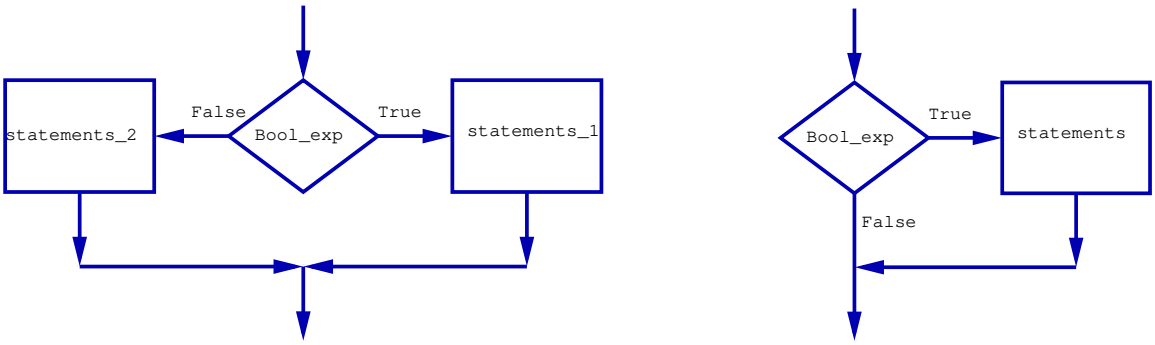
3.2 Syntax: simple cases

There are the following simple cases (left and right syntaxes):

```
if Bool_exp:                          if Bool_exp:
    statements1                           statements
else:
    statements2
```

The conditional statement is a compound statement with the already described pattern. The left-hand side syntax includes two lines which begin with two keywords, `if` and `else`, respectively, and end with a colon. They include a block of indented statements and can be called **branches**.

The following graphical schemes show the corresponding flowcharts:



For the left-hand side syntax, the following steps are performed:

- the Boolean expression is evaluated
- if it evaluates to `True` the code block `statements1` is executed
- if it evaluates to `False` the code block `statements2` is executed

In both cases, after the conditional statement, execution resumes at the code following this statement.

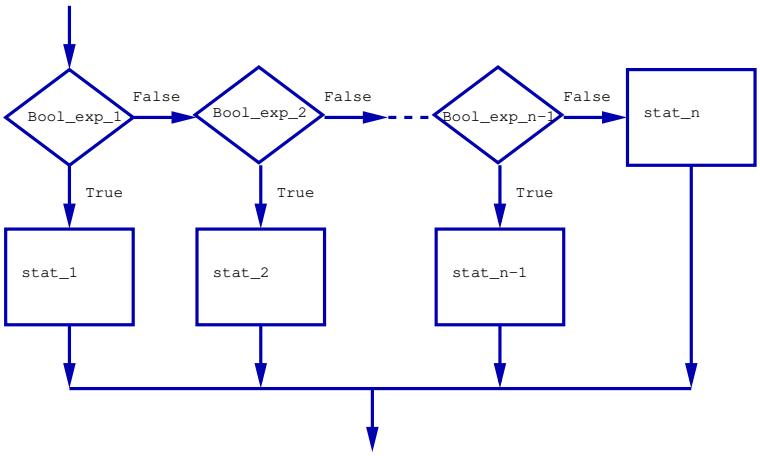
The case shown on the right is a particular case of the previous one in which when the Boolean expression is `False` no statements are executed. This is equivalent to using the statement `pass`, which is the statement that does nothing:

```
if Bool_exp:
    statements
else:
    pass
```

3.3 Syntax: general case

The syntax for the general case with more than one condition and the corresponding flowchart are shown below:

```
if Bool_exp1:
    stat1
elif Bool_exp2:
    stat2
.........
else:
    statn
```



In this case, the Boolean expression `Bool_exp_1` is evaluated. If the result is `True` the block of statements `stat_1` is executed and the conditional statement finishes. If the result is `False`, then the next Boolean expression `Bool_exp_2` is evaluated. If it is `True` the block of statements `stat_2` is executed and the conditional statement finishes. And so on. Finally, the last Boolean expression `Bool_exp_n-1` is evaluated and depending on its result, `True` or `False`, the block of statements `stat_n-1` or `stat_n` is executed and then the conditional statement finishes

Exercise 1

A sports center offers swimming training. The center only trains boys and girls of ages 11 to 19 (both included). The official category names for these ages is the following: ages of 11 and 12: *BEGINNER*, ages 13 and 14: *UNDER-15*, ages 15 and 16: *UNDER-17* and ages 17, 18 and 19 *UNDER-20*. Write a function that given the age (integer) of one candidate to enroll in the sports center, returns a string with the corresponding category. If the age of the candidate doesn't allow him/her to enroll in this sports center then the function must return the string `'NO SERVICE'`.

Design

Next a first version of this function is shown:

```
def swim1 (age):
    if age < 11:
        s = 'NO SERVICE'
    elif age <= 12:
        s = 'BEGINNER'
    elif age <= 14:
        s = 'UNDER-15'
    elif age <= 16:
        s = 'UNDER-17'
    elif age <= 19:
        s = 'UNDER-20'
    else:
        s = 'NO SERVICE'
    return s
```

This function works because we have carefully placed the Boolean expressions in the conditional statement in increasing age order.

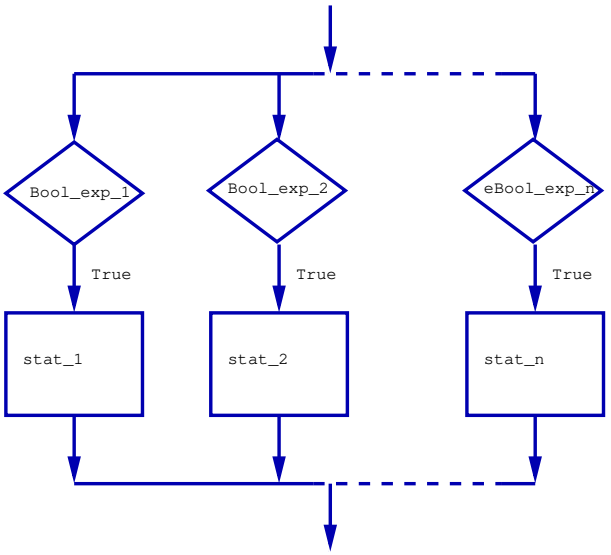
Let's now rewrite the previous function but interchanging the branches corresponding to the categories *BEGINNER* and *UNDER-15*:

```
def swim2 (age):
    if age < 11:
        s = 'NO SERVICE'
    elif age <= 14:
        s = 'UNDER-15'
    elif age <= 12:
        s = 'BEGINNER'
    elif age <= 16:
        s = 'UNDER-17'
    elif age <= 19:
        s = 'UNDER-20'
    else:
        s = 'NO SERVICE'
    return s
```

This new version doesn't work because it classifies ages 11 to 14 as *UNDER-15* and doesn't classify any age as *BEGINNER*. Therefore, using the syntax for the general case as strictly defined in Python may be error-prone.

3.4 General case: theoretical scheme

The following diagram shows the theoretical flowchart for the general conditional statement:



In this case, the set of conditions are a partition of unity, i.e., one and only one condition can be true at a time.

Next, a third version of the previous function is rewritten again using the theoretical scheme:

```
def swim3 (age):
    if age < 11:
        s= 'NO SERVICE'
    elif 13 <= age <= 14:
        s= 'UNDER-15'
    elif 11 <= age <= 12:
        s= 'BEGINNER'
    elif 15 <= age <= 16:
        s = 'UNDER-17'
    elif 17 <= age <= 19:
        s = 'UNDER-20'
    else:
        s = 'NO SERVICE'
    return s
```

In this function branches are not in increasing age order as in the previous version. However, the set of Boolean expressions is a partition of unity and therefore the function gives always the correct result.

As a conclusion, this theoretical scheme is highly recommended.

3.5 Nested conditional statements

The block statements within the branches of a conditional statement composition may include other conditional statements. We call them **nested** conditional statements.

Another version of the previous function is shown using nested conditional statements:

```
def swim4 (age):
    if 11 <= age <= 19:
        if 13 <= age <= 14:
            s= 'UNDER-15'
        elif 11 <= age <= 12:
            s= 'BEGINNER'
        elif 15 <= age <= 16:
            s = 'UNDER-17'
        else:
            s = 'UNDER-20'
    else:
        s = 'NO SERVICE'
    return s
```

In this case there is a main conditional statement that classifies ages between those for which the center offers training and those for which there is no training. In the first block there is another conditional statement that classifies accepted candidates accordingly to their ages.

As an exercise write down the Boolean expressions that are implicit in both `else` clauses of the previous design.

3.6 Functions with several `return` statements

In all the previous designs there is only one `return` statement. The result in the different blocks is assigned to one variable that is returned at the end of the conditional statement.

You can avoid this variable and return the values directly in each block, as shown in the next design. However, you must note that although there are more than one `return` statement only one of them will be executed. If you apply this kind of design, you must include a `return` statement in each block.

```
def swim5 (age):
    if 11 <= age <= 19:
        if 13 <= age <= 14:
            return 'UNDER-15'
        elif 11 <= age <= 12:
            return 'BEGINNER'
        elif 15 <= age <= 16:
            return 'UNDER-17'
        else:
            return 'UNDER-20'
    else:
        return 'NO SERVICE'
```

3.7 Boolean functions

A **Boolean function** is a function that returns a Boolean value.

Exercise 2

We are given a closed interval of ages `[e1, e2]` and a single age `e3`. Write a function that given the values of `e1, e2` and `e3` returns `True` if the age `e3` falls in the interval and `False` otherwise.

Design

```
def interval1 (e1, e2, e3):
    if e1 <= e3 <= e2:
        result = True
    else:
        result = False
    return result
```

In this design we have translated almost literally the exercise definition to Python. The function works but the conditional statement can be simplified with an assignment statement. The `result` Boolean variable can be assigned the Boolean expression `e1 <= e3 <= e2`:

```
def interval2 (e1, e2, e3):
    result = e1 <= e3 <= e2
    return result
```

The function can be further simplified by avoiding the `result` variable:

```
def interval3 (e1, e2, e3):
    return e1 <= e3 <= e2
```

3.8 Exercises

1. Write the function `minimum` that takes two integers, `x` i `y`, and returns its minimum.

   ```
   def minimum (x, y):
       if x < y:
           return x
       else:
           return y
   ```

2. A store offers the following promotion to its clients: for the second product unit, it is discounted by 30% (only this second unit) and if three units or more of a product are purchased, the entire purchase is reduced by 25%. In addition, if the price of the product is €50 or more there is an additional 5% reduction on the resulting amount in both cases. If you only buy a unit, no type of discount applies. Design the `promo` function that given the price and quantity of units of a product, returns the amount to pay without and with discount.

   ```
   def promo(price, units):
       if units == 1:
           return price, price
       else:
   ```

```
        if units == 2:
            amount = price * 1.7
        else:
            amount = price *units * 0.75
        if price >= 50:
            amount = amount * 0.95
        return price*units, amount
```

3. A hotel offers an trip that includes a boat and a train ride. The client can choose between starting with the boat and continuing on a train or vice versa and also between morning or afternoon. The trip has a price at which a general discount of a given percent applies. Moreover, if you start the trip by boat an extra 5% is added to the general discount, and if you choose doing the trip in the afternoon, an additional 10% is added to the general discount. Write function `trip` that takes two strings: one indicating how the client starts the trip, 'train' or 'boat', and the other one indicating when, 'morning' or 'afternoon', and two real values: the base price and the general discount (a percentage) and returns the final price of the trip.

```
def trip(how, when, price, d):
    # discounts are cumulative
    if how == 'boat':
        d = d + 5
    if when == 'afternoon':
        d = d + 10
    return price * (1-d/100)
```

4 Strings

4.1 Definition

Non-scalar types that comprise smaller pieces are also called **compound data types**. Strings, lists, tuples and dictionaries are compound data types.

Specifically, strings (type `str`) are ordered collections of characters, and are also referred as **sequence types**. It is a homogeneous data type because all of its elements are of the same type. In the first section we have dealt with strings as a block. Now we will learn how to access to their characters individually or in groups.

Strings are represented in single (or double) quotes: $'c_0c_1 \dots c'_n$ ("$c_0c_1 \dots c_n$") where c_i is a one-character string.

There are some one-character strings worth to highlight. There is the NULL character which is represented as `''` (there is nothing enclosed in single quotes) and corresponds to a void character. Beware not to mix up this character with the space character `' '` (now there is a space in single quotes) There are also control characters or sequence escapes that represent actions instead of graphical signs. In a string, the character `'\'` is an escape character used to indicate that the next character should be treated in a special way. For example, the string `'\n'` indicates a new line character. Beware that although the representation of this character involves two characters, it is a single one.

Examples:

```
>>> a = 'Today is a very nice day'
>>> b = "That's all folks"
>>> b
"That's all folks"
>>> c = 'this computer costs 422.22€'
>>> c
'this computer costs 422.22€'
>>> nothing = ''
>>> nothing
''
>>> d = 'order: 200 €\nvat(10%): 20 €\ntotal amount: 220 €'
>>> print(d)
order: 200 €
vat(10%): 20 €
total amount: 220 €
```

Observe the representation and interpretation of the **line feed** character.

4.2 Operations

So far we have seen operations with strings taken as a unique block: concatenation (+) and replication (*). We have also seen that we can operate strings with the comparison operations:

`==`, `!=`, `<`, `>`, `<=`, `>=` and that these comparisons are based on the lexicographical order.

Moreover, there is the built-in function `len` that gives the length of a string (its number of characters).

Examples:

```
>>> a = "That's all " + 'folks'
>>> a
"That's all folks"
>>> len (a)
16    # all characters are counted
>>> 'no '*3
'no no no ' # the space is also replicated thrice
>>> len('') # this is the NULL character (it's length is 0)
0
>>> len('\n') # this is the line feed character
1          # representation uses 2 symbols but length is 1
>>> len('anaconda')
8
>>> len('anatomy')
7
>>> len('anaconda') < len('anatomy') # comparing lengths
False
>>> 'anaconda' < 'anatomy'   # lexicographical order
True
>>> '23' < '123000'          # lexicographical order
False
```

4.3 Indexing

Besides the `len` function, sequence types share **indexing** and **slicing** operations.

You can access to any single character substring of a string by means of the indexing operator. The Python syntax uses square brackets, `[]`, enclosing an integer index. The index indicates the position of any character of a string.

Let `a` be a string and `index` an integer expression, the expression `a[index]` refers to the character substring of `a` in the `index` position.

`index` can be an (integer) variable, value or expression. Let `a` be a string and `n = len(a)`. Indices are zero-based, they fulfill the inequality $0 \leq index < len(a)$. This means that Python indices are such that the first character has index `0` and the last one has index `n-1`, giving a total of `n` characters. A typical beginners' error is to write the expression `a[n]` which gets the error `IndexError: string index out of range`, because the last element of `a` is `a[n-1]`.

Indices can be also negative: $-len(a) \leq index < 0$. They count backwards from the end to the beginning of the string.

For instance, let's consider the string `'strawberry'` with its positive and negative indices:

```
characters:            s  t  r  a  w  b  e  r  r  y
positive indices:      0  1  2  3  4  5  6  7  8  9
negative indices:    -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

Examples:

```
>>> a = 'lemon juice'
>>> len(a)
11
>>> a[0]
'l'
>>> a[-1]   # yields the last character (recommended syntax)
'e'
>>> a[-2]   # yields the second to last character
'c'
>>> a[5]    # this is a space
' '
>>> a[8]
'i'
>>> a[-8]
'o'
>>> list(enumerate(a))        # pairs (index, element)
[(0, 'l'), (1, 'e'), (2, 'm'), (3, 'o'), (4, 'n'), (5, ' '),
(6, 'j'), (7, 'u'), (8, 'i'), (9, 'c'), (10, 'e')]
```

4.4 Slicing

A slice is a substring of a string. Let `a` be a string, the Python syntax for the slice operator is: `a[start:stop]`, $0 \leq start < stop < len(a)$, considering positive indices. It refers to the substring of `a` from the *nth* character to the *mth* character, including the first but excluding the last one. It results that `len(a[start:stop])` is `stop-start`.

If you omit the first index (before the colon), the slice starts at the beginning of the string (index is supposed to be `0`). If you omit the second index, the slice extends to the end of the string (index is supposed to be `len(a)`). Moreover, if you provide a value for the second index that is bigger than `len(a)`, the slice will take all the values up to the end (it won't give an "out of range" error as the indexing operation does).

This syntax can be extended with a third element, `a[start:stop:step]`, $0 \leq start < stop < len(a)$. This expression refers to a slice of `a` from `start` to `stop` with a step of `step`. The default value for `step` is 1.

`start`, `stop` and `step` are integer expressions and can be positive or negative.

Examples:

```
>>> a = 'lemon juice'
>>> a[0:5]  # from the 0th (first) char to the 5th (excluded)
```

```
'lemon'
>>> a[:5]
'lemon'
>>> a[6:]
'juice'
>>> a[6:200] # stop can be greater or equal to len(a)
'juice'
>>> a[::2]
'lmnjie'
>>> a[::-1]  # recommended syntax to reverse a string
'eciuj nomel'
>>> a[-1:-100:-3]
'eune'
>>> b = 'lalalalalalala'
>>> b[::2]
'lllllll'
>>> b[1::2]
'aaaaaaa'
>>> c = 'lemons'  # syntax to select all but the last one
>>> c[:-1]
'lemon'
```

4.5 Membership operators

Membership operators are: `in` and `not in`. They allow to ask whether or not a string is a substring of another string. Both operators have two strings operands and give a Boolean result. The syntax is:

```
substring in string     --> Boolean
substring not in string    --> Boolean
```

Note that any string is a substring of itself, and that the empty string is a substring of any other string

Examples:

```
>>> a = 'strawberry'
>>> 'berry' in a
True
>>> 'strawberry' in a
True
>>> '' in a
True
>>> 'lemon' in a
False
>>> 'orange' not in a
True
```

```
>>> 'str' not in a
False
```

So far we have seen that the ASCII table has three intervals for lowercase and uppercase letters as well as for digits. Therefore, to check whether a character, `c`, is an uppercase letter or not we can use the Boolean expression `'A' <= c <= 'Z'`. However there are other characteristics that cannot be checked in this way as, for instance, to check whether a character is a vowel or not.

Membership operations allow to use another strategy. We can define any meaningful set of characters (vowels, DNA nucleotide, etc.) and then ask whether or not a single element belongs to this set. This strategy can be applied to any character set and it also can be extended to types list and tuple that will be introduced in next sections.

Examples:

```
>>> vowels = 'aeiouAEIOU'
>>> 'e' in vowels
True
>>> 'd' in vowels
False
>>> 'U' in vowels
True
>>> 'B' in vowels
False
>>> dna = 'ACGT'
>>> 'A' in dna
True
>>> 'E' in dna
False
>>> punct_marks = '.,;:'  # period, comma, semicolon, colon
>>> ':' in punct_marks
True
>>> '9' in punct_marks
False
>>> 'z' not in punct_marks
True
```

4.6 Immutability

Strings are immutable which means you cannot modify an existing string.

If we intend to modify a character in a string, Python will produce a run-time error:

```
>>> a ='employee'
>>> a[-1] = 'r'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If we want to get a string as a modification of an existing one, we have to create a new one:

```
>>> a = 'employee'
>>> a = a[:-1] + 'r'  # 'new' a is constructed from 'old' a
>>> a
'employer'
```

4.7 Built-in functions

Some built-in functions that apply to strings are: `len`, `min` and `max`. Examples:

```
>>> max('Programming')
'r'
>>> min('Programming')
'a'
>>> max ('ship', 'car', 'bike', 'motorcycle')
'ship'
>>> min ('ship', 'car', 'bike', 'motorcycle')
'bike'
```

Remember that strings are ordered by the lexicographical order and that the maximum string in not the longest one but the last considering the lexicographical order (alphabetical, in these examples).

4.8 Type `str`: methods

Python provides several data types with the corresponding methods. One of them is type `str` (strings). A method is a function that applies to an object of a determined data type. A method is also called (or invoked) but the syntax is slightly different. It uses the so called **dot notation** in which the parameter to which the method is applied (main parameter) comes before the name of the method and they are separated by a dot. The remaining parameters (secondary parameters came after the name of the method and between parenthesis, as well as in functions:

`str_variable_name.str_method_name(actual_parameters)`

- `str_variable_name` is the name of the type `str` variable to which the method is applied
- `str_method_name` is the name of the method
- `actual_parameters` is the list of the secondary parameters

We will see now some of the most used methods for strings. It is recommended to look up to the documentation to learn more about them. You can also use the shell for a short documentation, using functions `dir` and `help` of the standard library. Next, the short documentation shown in the shell is included for some methods together with some examples.

Function `dir` lists all the methods of the `str` type:

```
>>> dir (str)
```

There are basic methods as `count`, `find` and `replace`. Methods to classify strings as `isalnum`, `isalpha`, `isdigit`, `islower`, `isspace`, `isupper`. Other methods allow to change the case of alphabetical strings: `lower`, `upper`, `swapcase`, `capitalize`, `title`. Finally, other useful methods are `ljust`, `center`, `rjust`, `zfill`, `endswith`, `startswith`.

Next some of these methods are described.

4.8.1 Method `count`

```
>>> help(str.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.
```

Examples:

```
>>> a = 'strawberry'
>>> a.count('r')
3
>>> b = 'coconut'
>>> b.count('co')
2
>>> dice ='1266666654323332666'
>>> dice.count('66')
4
>>> dice = '166266366'
>>> dice.count('66')
3
```

4.8.2 Method `find`

The `find` performs an inverse process to the indexing operator. This operator gives the element corresponding to a given index and method `find` returns the index corresponding to a character.

```
>>> help(str.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.
```

Examples:

```
>>> c = 'Hello'
>>> c.find('lo')
3
>>> c[3]
'l'
>>> c.find('bye')
-1
>>> c.find('l')   # index of the first 'l'
2
```

4.8.3 Method `replace`

```
>>> help(str.replace)
S.replace(old, new[, count]) -> string

Return a copy of string S with all occurrences of substring
old replaced by new. If the optional argument count is given,
only the first count occurrences are replaced.
```

Examples:

```
>>> d = "I'm an employee"
>>> e = d.replace('e', 'r')
>>> e
"I'm an rmployrr"
>>> f = d.replace('ee', 'er')
>>> f
"I'm an employer"
>>> d
"I'm an employee"
```

In the previous example, variable `d` remains unchanged. Strings are immutable and therefore variables of type `str` cannot be modified and the `replace` method works accordingly: it does not modify variable `d`, it creates a copy with the requested replacements instead. The `replace` method is applied twice and, in both times, variable `d` keeps its initial value.

4.8.4 Methods concerning character type and case

Next the short documentation shown in the shell is included for some of these methods together with some examples.

```
>>> help(str.isalnum)
S.isalnum() -> bool

Return True if all characters in S are alphanumeric and
there is at least one character in S, False otherwise.
```

```
>>> help(str.swapcase)
S.swapcase() -> string

Return a copy of the string S with uppercase characters
converted to lowercase and vice versa.
```

Examples:

```
>>> a = 'How are you'
>>> b = 'Programming'
>>> c = 'USA'
>>> d = 'a'
>>> e = '4'
>>> f = '?'
>>> a.isalnum()
False                  # there are space characters
>>> b.isalnum()
True
>>> f.isalnum()
False
>>> f.isdigit()
False
>>> e.isdigit()
True
>>> a.swapcase()
'hOW ARE YOU'
>>> c.lower()
'usa'
>>> b.upper()
'PROGRAMMING'
>>> a = 'albert'
>>> a.capitalize()
'Albert'
>>> b = 'dr. albert fox'
>>> b.title()
'Dr. Albert Fox'
```

4.8.5 More methods

Finally, other string methods that can be useful are shown with several examples. These methods are: `ljust, center, rjust, endswith, startswith, zfill`

```
>>> help(str.ljust)
  S.ljust(width[, fillchar]) -> str

  Return S left-justified in a Unicode string of length
```

```
  width. Padding is done using the specified fill character
  (default is a space).

>>> help(str.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.

>>> help(str.zfill)
    S.zfill(width) -> str

    Pad a numeric string S with zeros on the left, to fill
    a field of the specified width.
    The string S is never truncated.
```

Examples:

```
>>> a = 'Hellow !!!'
>>> a.ljust(20, ' ')
'Hellow !!!          '
>>> a.center(20, ' ')
'     Hellow !!!     '
>>> a.rjust(20, ' ')
'          Hellow !!!'
>>> a.ljust(20, '-')
'Hellow !!!----------'

>>> b = 'telecommunication'
>>> prefix = 'tele'
>>> sufix = 'tion'
>>> b.startswith(prefix) # equivalent to next expression
True
>>> b[:len(prefix)] == prefix
True
>>> b.endswith(sufix) # equivalent to next expression
True
>>> b[-len(sufix):] == sufix
True
>>> '987'.zfill(6)
'000987'
```

In order to know the specification of all of these methods you must look up at the documentation.

4.9 Exercises

1. In 2000 a new car registration system was implemented in Spain, which is based on a 3-letter and 4-digit encoding. The letters can only be the following 20: `B, C, D, F, G, H, J, K, L, M, N, P, R, S, T, V, W, X, Y, Z`. In this codification, the group of three letters is in base-20 and has more weight than the group of 4 digits (in base-10)

   (a) Design function `license_val` that takes a string representing a license plate and returns its absolute position from the start of the new car registration system.

   (b) Write function `license` that takes two strings representing two car license plates and returns the number of cars between them. We can assume that the first given license plate is always previous to the second one. This function must call function `license_val`.

   Examples:

```
>>> license_val ('0000-BBB')
0
>>> license_val ('6589-BBB')
6589
>>> license_val ('9999-BBB')
9999
>>> license_val ('0000-BBC')
10000
>>> license_val ('9999-ZZZ')
79999999
>>> license ('0000-BBB', '9999-ZZZ')
79999999
>>> license ('0000-BBB', '9999-BBB')
9999
>>> license ('0000-BBB', '0000-BBC')
10000
>>> license ('9999-KLZ', '0000-KMB')
1
```

   Solution:

```
def license_val (lic):
    letters = 'BCDFGHJKLMNPRSTVWXYZ'
    p1 = letters.find(lic[5])
    p2 = letters.find(lic[6])
    p3 = letters.find(lic[7])
    return (p1*20**2 + p2*20 + p3)*10000 + int(lic[:4])

def license (lic1, lic2):
    return license_val (lic2) - license_val (lic1)
```

2. Write function `pool` that takes a string with the result of a football match with the pattern `'htg-vtg'` (home team goals, a hyphen and visiting team goals) and returns the character `'1'` if the home team has won, `'2'` if the visiting team has won and `'X'` if there is a tie. Examples:

```
>>> pool('5-3')
'1'
>>> pool('2-12')
'2'
>>> pool('0-0')
'X'
>>> pool('12-15')
'2'
>>> pool('10-10')
'X'
>>> pool('1-0')
'1'
```

Solution:

```
def pool(result):
    hyphen_pos = result.find('-')
    home_goals = int(result[:hyphen_pos])
    visiting_goals = int(result[hyphen_pos+1:])
    if home_goals > visiting_goals:
        return '1'
    elif home_goals < visiting_goals:
        return '2'
    elif home_goals == visiting_goals:
        return 'X'
```

5 Iteration

5.1 Statement `for`

Computers perform calculations at a very high speed and they do extremely well repetitive tasks without making errors. Repeated execution of a set of statements is called **iteration**.

There are two main statements to do iteration: `for` and `while`. In this section we will focus on the `for` statement and on the traversal of data structures as strings.

Notation:

```
for character in a:
    process character
```

This `for` statement is another compound statement with the already described pattern. The first line includes two new keywords (`for` and `in`) and ends with a colon, and there is a block of indented statements.

Variable `character` is assigned, successively and from the first to the last, the values of all characters of string `a`. This process is called **traversal**. Instead of a complete string, a slice can be used. This statement repeats the same process for each character of the string. The application of this process (block of statements) to each individual character is called **iteration** or **loop**.

Most problems involving a traversal of a data structure can be classified according to the following typologies: **reduce**, **map** and **filter**.

The reduce typology obtains a scalar result from a compound data type as a string. Next, some examples of problems that fall into this typology are formulated:

- count the number of occurrences of string `'if'` in a given string
- count the number of uppercase letters in a given string
- compute the percentage of vowels with respect to the total letters in a given string

The `count` method seen in the previous section falls into this typology. However, this method has limitations and we would only be able to solve the first problem using it. We cannot count neither uppercase letters nor vowels using the `count` method .

Functions `max` and `min` of the standard library apply also a reduce process.

In this section the reduce process is introduced. Map and filter processes will be introduced in the following section together with the data structure `list`.

5.2 String traversal: reduce process

In this section we deal with problems that from a given string a scalar result is computed.

Most of these processes are **counting** or **summation** processes. In these processes there is a **counter** or **summation** variable. This variable is initialized to a starting `0` value before the string traversal and is updated properly at each iteration.

Exercise 1

Write the function `uppercase_number` that takes a non empty string and returns its number of uppercase letters. Example:

```
>>> s = 'Robin Hood and Little John walking through the Forest'
>>> uppercase_number(s)
5
```

Design

```
def uppercase_number (s):
    number = 0
    for c in s:
        if c.isupper():
            number = number + 1
    return number
```

Variable `number` is a counter. We have to set (initialize) this counter to `0` before starting (before the `for` statement) and to update it in each iteration. The `str` method `isupper` is used to check if the character is an uppercase letter.

To update a counter or a summation variable the **increment** and **decrement** operations are used. The Python syntax for these operations is showed in the following examples:

```
>>> counter = counter + 1 # counter is incremented by one unit
>>> counter = counter - 1 # counter is decremented by one unit
>>> summ = summ + amount # summ is incremented by amount
>>> summ = summ - amount # summ is decremented by amount
```

Python as well as other languages has a specific syntax for these operations:

```
>>> counter += 1
>>> counter -= 1
>>> summ += amount
>>> summ -= amount
```

Exercise 2

Write the function `vowel_percent` that takes a non-empty string and returns the percentage of vowels with respect to the total number of letters in it. If the string does not contain any letter, the function must return `0.0`. Example:

```
>>> s = 'Robin Hood and Little John walking through the Forest'
>>> round(vowel_percent(s), 2)
33.33
```

Design

```
def vowel_percent (s):
    vowels = 'aeiouAEIOU'
    total_letters = 0
    total_vowels = 0
    for c in s:
        if c.isalpha():
            total_letters = total_letters + 1
            if c in vowels:
                total_vowels = total_vowels + 1
    if total_letters != 0:
        return total_vowels/total_letters*100
    else:
        return 0.0
```

This function applies two counting processes to count the number of total letters and the number of uppercase letters. Note that both counter processes can be performed at once with a single `for` statement. The `str` method `isalpha` is used to check if the character is an uppercase letter while variable `vowels`, set to an string with all upper and lowercase vowels, is used together with the membership operator `in` to check if the character is vowel.

5.3 Traversal through indices

Some processes require addressing to string elements by their indices. For example, if we want to analyze pairs of consecutive characters, we have to be able to access a character and the following one at a time, and therefore we have to address them by their positions, i.e., indices.

The `for` statement iterates through string indices using the built-in function `range`.

Function `range` generates an integer sequence which can be used as a sequence of indices. It can be called in three different ways:

1. `range (stop)`: generates the integer sequence from `0` to `stop-1` (both included).
2. `range (start, stop)`: generates the integer sequence from `start` to `stop-1` (both included).
3. `range (start, stop, step)`: generates the integer sequence:

   `start`, `start+step`, `start+2*step`, ..., `end`, with `end< stop`

   This is the general case.

Default values for parameters `start` and `step` are `0` and `1` respectively.

Note the syntax similarity between function `range` and the slices' operator.

Examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range (2, 10, 2))
[2, 4, 6, 8]
```

To traverse a string through its indices, we will use he following notation:

```
for i in range (len(a)):
    process a[i]
```

In this notation the `for` statement traverses the string `a` indices from `0` to `len(a)-1`, which are all possible indices. To access the string element we use the indexing operator: `a[i]`.

The previous examples can also be written using indices instead of elements. However, it is a good practice not to use indices if they are not strictly necessary.

Next function `vowel_percent_indices` is a version of the previous one using indices:

```
def vowel_percent_indices (s):
    vowels = 'aeiouAEIOU'
    total_letters = 0
    total_vowels = 0
    for i in range(len(s)):
        if s[i].isalpha():
            total_letters = total_letters + 1
            if s[i] in vowels:
                total_vowels = total_vowels + 1
    if total_letters != 0:
        return total_vowels/total_letters*100
    else:
        return 0.0
```

Exercises that require the use of indices

Exercise 3

In a game two dice are thrown several times and we want to know the number of times that two six have come out. We represent in a string a sequence of these two dice roll (digits between `1` and `6` both included), i.e., the first and second characters represent the first two dice roll, the third and fourth represent the second two dice roll and so on. Write function `sixs` that takes a string like this and returns the number of times that two six have come out in one game. Example:

```
>>> sixs('12654323332666')
1
>>> sixs('3466526612')
2
```

Design

```
def sixs (dice):
    n = 0
    for i in range(0, len(dice), 2):
        if dice[i] == '6' and dice[i+1] == '6':
            n = n + 1
    return n
```

In this exercise, function `range` generates even indices (0, 2, 4, ...) and we have to visit the following pairs of characters $(c_0, c_1), (c_2, c_3), \ldots$. The statement `for` traverses these indices with variable `i`, and accesses to the corresponding string item `dice[i]` and to the next one `dice[i+1]`.

We can express the Boolean expression: `dice[i] == '6' and dice[i+1] == '6'`
in this other way: `dice[i:i+2] == '66'`

Analyze whether this exercise could be solved with method `count` or not.

Exercise 4

A string contains a text with several sentences. Each sentence has its ending period and the next sentence begins after this period (there is not a separation space). Write the function `phrases` that takes a string as described and returns the number of sentences beginning with a consonant letter. Example:

```
>>> s1 = 'Tomorrow we are going out.You coming?.Yes.Great!.'
>>> s2 = 'Emma coming?.Dunno.Where are we going?.To the movies.
   '
>>> s = s1 + s2 + 'And dinner?.Maybe.'
>>> phrases(s)
8
```

Design

```
def phrases (s):
    vowels = 'aeiouAEIOU'
    if s[0] in vowels:
        n = 0
    else:
        n = 1
    for i in range(len(s)-1):
        if s[i] == '.' and (s[i+1].isalpha() and s[i+1] not in
           vowels):
            n = n +1
    return n
```

The counter is initialized to 1 or 0 depending on whether or not the first letter is a consonant. Then it visits the overlapping pairs of characters: $(c_0, c_1), (c_1, c_2), (c_2, c_3) \ldots (c_{len(s)-2}, c_{len(s)-1})$ in order to identify pairs with the pattern `('.', consonant)`.

5.4 Searching

In all the previous examples the string traversal is always complete. There are other problems in which it is not always necessary to perform a complete traversal which are called **searching** processes. If we are looking for a string element that meets a certain condition, when the first element that meets it is found the search is completed and the traversal can be stopped. In a searching process the traversal will be complete only in two particular cases: when no element fulfills the searching condition or when only the last fulfills it.

The `str` method `find` applies a searching process. Examples:

```
>>> 'First check if it can be simplified'.find ('if')
12
>>> 'First check if it can be simplified'.find ('of')
-1
```

In the first example, once you have found the first occurrence of string `'if'` in position 12, the traversal of the given string stops. In the second example, the traversal of the given string must be complete in order to conclude that string `'of'` has not been found into the given string.

All `str` methods that ask for the type of a string (`isupper`, `islower`, etc.) also apply a search process. Examples:

```
>>> '3 2 1 I will vote YES !!!'.isupper()
False
>>> '3 2 1 I WILL VOTE YES !!!'.isupper()
True
```

In the first example, once the first lowercase letter was found, `'w'`, the method returns `False` because it can already assert that not all cased characters in the given string are uppercase. In the second example, the given string must be completely traversed in order to assert that all cased characters in the given string are uppercase.

To reflect this behavior when writing a new Python function, we need to be able to exit from the `for` statement before it has finished all the iterations. The `break` statement is used to exit from the `for` body so that, in this case, the next statement to be executed will be the first one after the `for` body.

The proposed searching scheme is:

```
found = False
for c in s:
    found = search_condition (c)
    if found:
        break
if found:
    process_found(c)
else:
    process_not_found(c)
```

In this scheme a state variable `found` is used. It indicates whether or not the current element meets the condition. At the beginning, variable `found` is initialized to `False`. At each `for` iteration, the search condition is evaluated for the current element and the variable `found` is updated accordingly. When the variable `found` is `True`, the `break` statement is executed and the flow control leaves the `for` body. Execution resumes at the code following the `for` statement by considering the two possible cases of whether or not an element fulfilling the search condition has been found.

Exercise 5

Consider the same game described in Exercise 3. Write the function `equals` that takes a string as described and checks for the first two dice roll such that both dice have the same value. This function returns the string `'affirmative'` if there exists such a pair and the string `'negative'` otherwise. Example:

```
>>> equals('12654321332666')
'affirmative'
>>> equals('34652112')
'negative'
```

In the first example the traversal finishes when the pair `'33'` is found. In the second example the traversal is complete and the result is `'negative'`: there is a pair `'11'` but does not correspond to a roll.

Design

```
def equals (dice):
    found = False
    for i in range(0, len(dice)-1):
        found =  dice[i] == dice[i+1]
        if found:
            break
    if found:
        return 'affirmative'
    else:
        return 'negative'
```

Write now another version of the previous exercise that instead of the strings `'affirmative'` or `'negative'`, returns a Boolean: `True` or `False`. Below are three equivalent versions:

```
def equals1 (dice):
    found = False
    for i in range(0, len(dice)-1):
        found =  dice[i] == dice[i+1]
        if found:
            break
    if found:
        return True
    else:
        return False
```

```
def equals2 (dice):
    found = False
    for i in range(0, len(dice)-1):
        found =  dice[i] == dice[i+1]
        if found:
            break
    return found

def equals3 (dice):
    for i in range(0, len(dice)-1):
        if dice[i] == dice[i+1]:
            return True
    return False
```

Function `equals1` applies the schema directly. Function `equals2` simplifies the last `if` statement as a Boolean expression. Finally, function `equals3` does not use the state variable `found` and substitutes the `break` statement by the `return` statement which exits not only from the `for` statement but also from the function returning the corresponding result. Although this third version seems simpler, keep in mind that this simplification can only be done for very simple problems like this one and the recommendation is to use always the initial complete scheme.

It is important to understand the meaning of indentation in the previous exercises. Recall that indentation represents graphically the execution flow of Python functions.

5.5 Exercises

1. The morse code is difficult to remember but there are mnemonic rules that help to remember the codes. In Wikipedia we can find one that consists in remembering a keyword for each letter. Then, from this word the morse code of the letter can be obtained following the following rules:

   (a) The initial letter of the keyword is the corresponding letter

   (b) The number of vowels in the keyword indicates the length of the morse code of this letter

   (c) If the vowel is an 'O' it is replaced by a line (-).

   (d) If it is any other vowel it is replaced by a point (·)

   Write function `morse` that takes a keyword and returns the corresponding letter and the number of points and lines in its morse code. Example:

```
>>> morse('Coca-cola')
('C', 2, 2)
>>> morse('Himalaya')
('H', 4, 0)
>>> morse('Motor')
('M', 0, 2)
```

   Design

```
def morse (keyword):
    vowels = 'aeiouAEIOU'
    letter = keyword[0]
    nlines = keyword.count('o') +  keyword.count('O')
    nvow = 0
    for c in keyword:
        if c in vowels:
            nvow = nvow + 1
    npoints = nvow - nlines
    return letter, npoints, nlines
```

   In this exercise the number of lines is computed using the method `count` that computes the number of characters `'o'` and `'O'`. Then the number of points will be the total number of vowels minus the number of lines.

2. The intelligence service of a country asks us to design the `substring` function which takes two strings containing secret codes and returns the first substring with the same characters and in the same positions shared by the two given strings. If there is no such substring, the function must return the string null. Examples:

```
>>> s1 = 'aeioubcde'
>>> s2 = 'dfgthbghtf'
>>> substring(s1, s2)
'b'
>>> s2 = 'aeiuoj'
>>> substring(s1, s2)
'aei'
>>> s2 = 'AE g-9cde'
>>> substring(s1, s2)
'cde'
>>> s2 = '..aeiou--!!!+hg'
>>> substring(s1, s2)
''
```

```
def substring (s1, s2):
    found = False
    lmin = min (len(s1), len(s2))
    for i in range(lmin):
        found = s1[i] == s2[i]
        if found:
            start = i
            break
    if not found:
        return ''
    else:
        found = False
        for i in range(start+1, lmin):
            found = s1[i] != s2[i]
            if found:
                stop = i
                break
        if found:
            return s1[start:stop]
        else:
            return s1[start:lmin]
```

In this function, the searching scheme is applied twice. Firstly, to find the position of the first matching character and, secondly, to find the position of the last matching character.

Since we have to traverse two strings in parallel, we will use indices and the extension of this traversal will be given by the length of the shortest string.

6 Lists

Lists are sequence data types. Unlike strings, list elements may be of any type. There are homogeneous lists with all its elements of the same type and heterogeneous lists with elements that can be of different type.

A list is represented in square brackets (`[]`) and with its elements separated by commas: $[e_0, e_1, \ldots, e_n]$. An empty list is represented as `[]`. Examples:

```
# homogeneous list of integers
>>> a1 = [3, 8, 5, 30, 23]
# homogeneous list of strings
>>> a2 = ['car', 'boat', 'motorcycle']
# heterogeneous list
>>> a3 = ['home', True, 45.78, 32, 'hello']
>>> empty = []
```

6.1 Operations

Concatenation (+) and replication (*) operations also apply to lists. Comparison operators, `==`, `!=`, `<`, `>`, `<=`, `>=`, are also applicable and based on the lexicographical order. When two lists are compared, their elements are compared in the order in which they appear in each list. When comparing list elements, we must take into account that they must be of the same type.

The `len` function of the standard library gives the length of a list, i.e., its number of elements.

Membership operators `in` and `not in` are also applicable to lists:

```
element in list
element not in list
```

Examples:

```
>>> a1 = [3, 4, 5, 6]
>>> a2 = [10, 20, 30]
>>> a1 + a2
[3, 4, 5, 6, 10, 20, 30]
>>> a1 * 3
[3, 4, 5, 6, 3, 4, 5, 6, 3, 4, 5, 6]
>>> ['a', 'b'] * 3
['a', 'b', 'a', 'b', 'a', 'b']
>>> 5 in a1
True
>>> 10 in a1
False
>>> 20 in a1 + a2
True
>>> a3 = ['home', 23, True, 45.78, 32, 'hello']
>>> 'home' in a3
```

```
True
>>> ['home', 23] in a3
False
>>> a4 = ['home', 'car', True, 45.78, 32, 'hello']
>>> a3 == a4
False
>>> a3 <a4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: '<' not supported between instances
                                of 'int' and 'str'
```

The expression `['home', 23] in a3` is `False` because list `['home', 23]` is not an element of list `a3`.

On the other hand, lists `a3` and `a4` do not have all their corresponding elements of the same type (the second element of `a3` is an integer while the second element of `a4` is a string) and, therefore, they can also be compared with operators `==` and `!=`. Expression `a3 == a4` evaluates to `False` but expression `a3 < a4` cannot be evaluated: their first elements can be compared because they are both strings but their second elements cannot be compared because they are of different type; therefore, Python raises an error.

6.2 Indexing and slices

Indexing and slicing mechanisms applied to lists have the same behavior than with strings.

Elements of a list can be accessed through an index that is an integer expression. If `a` is a list, the syntax to access an element is `a[index]`, $0 \leq index < len(a)$. Indexes can also be negative: $-len(a) \leq index < 0$

A slice is a subset of a list. If `a` is a list, the syntax for a slice is `a[start:stop:step]`, $0 \leq start < stop < len(a)$, where `start`, `stop` and `step` are integer expressions.

Examples:

```
>>> a1 = [10, 20, 30, 40, 50, 60]
>>> a1[0]
10
>>> a1[3]
40
>>> a1[2:4]
[30, 40]
>>> a1[::2]
[10, 30, 50]
```

6.3 Nested lists

List elements can be of any type and, in particular, they can also be lists. Lists with other lists as elements are called **nested lists**. To refer to the hierarchy in these structures we can use the terms **lists** and **sublists**. Examples:

```
>>> a = [1, 2, 3, [4, 5]]
>>> a[3]
[4, 5]
>>> a[3][0]
4
```

The last element in list `a`, `a[3]`, is another list, `[4, 5]`. To access to the elements of this sublist we must use another index: `a[3][0]` accesses the first element of list `a[3]`.

6.4 String methods dealing with lists

Two `str` methods have lists as secondary parameters: `split` and `join`. Remember that to invoke methods we have to use the dot notation.

Below, the short documentation given by the `help` function is shown:

```
S.split ([sep [, maxsplit]]): list of strings

Return a list of the words in the string 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.
```

```
S.join (iterable): string

Return a string which is the concatenation of the strings
in the iterable. The separator between elements is S.
```

In the `split` method documentation it appears the term whitespace string. In computer programming, **whitespace** is any character or sequence of characters that represent horizontal or vertical space in typography: space bar, tab and CR.

The `split` method has two different behaviors: general and specific. In the general behavior, the string is split using the given separator which is removed. The specific behavior is aimed at a particular case that occurs very often: a text where words are separated by whitespace characters. Calling the `split` method without specifying the `sep` parameter has the following behavior: any whitespace character is a separator and therefore they are all removed and, in addition, all empty strings created are also removed. Examples:

```
>>> a = '00-11-22-33 '
# general behavior
>>> a.split ('-')
```

```
['00', '11', '22', '33']
>>> a = '00--11--22--33 '
# general behavior: empty strings are created
>>> a.split ('-')
['00', '', '11', '', '22', '', '33 ']
>>> b = 'Hi    John ?\nHow do  you    do ?'
# specific behavior: empty strings are removed
>>> b.split ()
['Hi', 'John', '?', 'How', 'do', 'you', 'do', '?']
```

In expression `a.split ('-')` method `split` has the general behavior. In the second example, empty strings are created between each pair of consecutive hyphens. In expression `b.split()` there is no separator and method `split` has the specific behavior: whitespace characters are used as separator and empty strings created are removed.

Method `join` concatenates all the elements in the given list and inserts the given separator (joint) between each consecutive pair.

Method `join` and function `list` of the standard library are somewhat related: they perform the inverse operation of each other when the joint string is the empty string. function `list` converts any sequence type object (as a string) into a list.

See the following examples and observe the result of applying function `str` to a list:

```
>>> c = ['34', '93', '432', '7698']
>>> '-'.join (c)
'34-93-432-7698'
>>> d = ["Let's", 'add', 'words', 'and', 'make', 'sentences']
>>> ' '.join (d)
"Let's add words and make sentences"
>>> list ('abcd')
['a', 'b', 'c', 'd']
>>> ''.join (['a', 'b', 'c', 'd'])
'abcd'
>>> str(['a', 'b', 'c', 'd']) # function str applied to a list
"['a', 'b', 'c', 'd']"
```

The next two exercises show the use of these methods:

Exercise 1

Given a string with a group of consonants and a Boolean, write function `syllable` that returns a string with all the syllables formed by these group of consonants and each of the vowels in alphabetical order. If the Boolean is `True` the consonant group comes before the vowels and if it is `False` it comes after the vowels. To simplify, we will consider only lowercase letters. Example:

```
>>> syllable('m', True)
'mamemimomu'
>>> syllable('rf', False)
'arferfirforfurf'
```

Design

```
def syllable(cons, before):
    vowels = 'aeiou'
    if before:
        return cons + cons.join(vowels)
    else:
        return cons.join(vowels) + cons
```

Exercise 2

In a string there are several integer amounts separated by a hyphen. Design the function `summ` that from such a string returns the sum of the quantities represented in the string. Example:

```
>>> summ('1-2-3-4-5')
15
>>> summ('34-89-77-654-1')
855
```

Design

```
def summ (s):
    summ = 0
    lvalues = s.split('-')
    for c in lvalues:
        summ = summ + int(c)
    return summ
```

6.5 List traversal

As seen in the previous chapter for strings, we can also traverse a list and search into a list using the `for` statement both directly through the list elements and indirectly using the indexes.

The mentioned problem cases of reduce, map and filter can be applied to lists. We begin with a reduce example:

Exercise 3

Given a list with real values corresponding to the heights of a sample of the female population aged 18, write the function `average` that returns the average height of the sample. Example:

```
>>> lf = [1.7, 1.73, 1.65, 1.59, 1.61, 1.78, 1.81, 1.66, 1.55]
>>> round(average(lf), 2)
1.68
```

Design

```
def average_1 (lheights):
    summ = 0
    for height in lheights:
        summ = summ + height
```

```
    return summ/len(lheights)

def average_2 (lheights):
    return sum(lheights)/len(lheights)

average = average_1
#average = average_2
```

The first version of function `average` computes the sum of all elements in the list using a `for` statement and a summation variable that is initialized to `0` before the `for` statement. The second version uses the standard library function `sum` instead.

Exercise 4

An interval is represented by a list of two elements that represent their lower and upper extremes (`float`). Given a list of intervals, write the function `largest` that returns the length of the longest interval (`float`). Example:

```
>>> lint = [[-3, 3], [1.2, 4.8], [-56, 24.6], [-20, -10]]
>>> largest(lint)
80.6
```

In this exercise we show another example of synthesis: the process of finding the maximum or the minimum of a data structure. In this process, we use a variable as the reference value and that will met the invariant condition of being the maximum or minimum of the visited elements. This variable must be initialized to a value *small enough* in the case of a maximum (and to a value *large enough* in the case of a minimum). If the typology of the data does not allow these values to be determined, the first element can be used as the reference value.

Design

```
def largest (lintervals):
    larg = 0
    for interval in lintervals:
        length = interval[1] - interval[0]
        if length > larg:
            larg = length
    return larg
```

We initialize the reference variable `larg` to a value *small enough* (we want to get a maximum value). At each iteration we compare the variable `larg` with the length of the current interval and we update it if necessary. Note that if there are two intervals with the same maximum length, the function returns the first one it finds. Also note that in this process of "search"of the maximum value, it is necessary to apply a traversal scheme, since it is necessary to visit all the elements in order to obtain its maximum.

Exercise 5: searching process

Write first the `pentavowel` function that given a word (string) returns a Boolean that indicates if the word contains all the vowels.

Now, given a list of words, write the `firstpenta` function that returns the first word in the list that contains all the vowels. If there is no such word, the function must return an empty string. This function must use function `pentavowel` function. Example:

```
>>> pentavowel('euphoria')
True
>>> pentavowel('mathematics')
False
>>> pentavowel('equation')
True
>>> pentavowel('function')
False
>>> l1 =['mathematics', 'variable', 'equation',
...   'formula', 'function']
>>> firstpenta(l1)
'equation'
>>> l2 = ['me', 'you', 'we', 'they']
>>> firstpenta(l2)
''
>>> l3 = ['euphoria', 'equation', 'capability', 'pentavowel'
...  'previous']
>>> firstpenta(l3)
'euphoria'
```

Design

```
def pentavowel(word):
    vowels = 'aeiou'
    all_vowels = True
    for vowel in vowels:
        all_vowels = vowel in word
        if not all_vowels:
            break
    return all_vowels
def firstpenta (lwords):
    found = False
    for word in lwords:
        found = pentavowel(word)
        if found:
            break
    if found:
        return word
    else:
        return ''
```

6.6 List mutability

Unlike strings, lists are mutable, i.e., its elements can be modified. Examples:

```
>>> a = ['hello', 2.0, 5, [10, 20]]
>>> a[1] = 'bye' # replacement of an item
>>> a
['hello', 'bye', 5, [10, 20]]
>>> a = ['hello', 2.0, 5, [10, 20]]
>>> a [1] = [1.0, 3.0] # replacement of an item by a list
>>> a
['hello', [1.0, 3.0], 5, [10, 20]]
>>> a = ['hello', 2.0, 5, [10, 20]]
>>> a [1:3] = [1, 2, 3, 4] # replacement of several elements
>>> a
['hello', 1, 2, 3, 4, [10, 20]]
>>> s = [22, 33, 44, 55, 66, 77, 88]
>>> t = ['a', 'b', 'c', 'd']
>>> s [0:7:2] = t # alternate substitution
>>> s
['a', 33, 'b', 55, 'c', 77, 'd']
```

In the previous examples we verify that the list has been modified by showing it before and after the modification

Aside from directly modifying their individual components, lists can be modified by removing or adding elements.

Statement `del` removes one or more items from a list. Examples:

```
>>> aa = [3, 5, 8, 6, 2, 4, 1, 9]
>>> del aa[2]
>>> aa
[3, 5, 6, 2, 4, 1, 9]
>>> del aa[1:3]
>>> la
[3, 2, 4, 1, 9]
>>> lb = ['a', 'e', 'i', 'o', 'u']
>>> del lb[::2]
>>> lb
['e', 'o']
```

Method `append` adds an element at the end of a list. Section 6.9 describes this one and other list methods.

6.7 Objects, identifiers, alias and cloning

In the first section we have introduced the concepts of value and variable. Values (objects) are the core elements that a Python program manipulates. A variable is a name that refers to a

value. Now we introduce the concept of object *identity*. This is an integer which is unique and constant for this object during its lifetime and which is the memory address of this object. Two objects with non-overlapping lifetimes may have the same identity. The `id` function of the built-ins library returns the identity of an object.

For immutable objects, such as strings, each value has a single identity and, therefore, two or more variables having the same value, will have the same identity; they are different names for the same object. This fact is not a problem because strings are immutable. Examples:

```
>>> x = 5
>>> y = 3 + 2
>>> id (x)
10105952
>>> id (y)
10105952            # x and y refer to the same object
>>> a = 'aeiou'
>>> b = 'aeiou'
>>> c = a
>>> d = 'ae' + 'iou'
>>> id (a)
139889255176600
>>> id (b)
139889255176600
>>> id (c)
139889255176600
>>> id (d)
139889255176600     # a, b, c, d refer to the same object
```

For mutable types as lists, each time a list object is referenced, a new object is created. Therefore, there can be more than one object sharing the same value. Examples:

```
>>> alist = [1, 2, 3, 4, 5, 6]
>>> blist = [1, 2, 3, 4, 5, 6]
>>> id (alist)
139889255147336
>>> id (blist)
139889255187464 #
>>> alist == blist
True
>>> id(alist) == id(blist)
False
>>> alist[2] = 34
>>> blist[1] = 56
>>> alist
[1, 2, 34, 4, 5, 6]
>>> blist
[1, 56, 3, 4, 5, 6]
```

Lists `alist` and `blist` have the same value but they are not the same object (their identities are different). List mutability is based on this behavior. If both lists `alist` and `blist` referred the same object, i.e., had the same identity, modifying a list would change the other one.

With mutable objects as lists we can have the two following situations. In the first situation, which is shown in the previous example, lists `alist` and `blist` have different identities. Therefore, modifying one of them has no effect in the other one.

In the other situation, several lists may refer to the same object, i, e, may have the same identity: they are **alias** and when we modify one of them, the others are modified in the same way. The syntax to define a list `xlist` as an alias of another existing list `alist` is:

```
xlist = alist
```

Examples:

```
>>> alist = [1, 2, 3, 4, 5, 6]
>>> blist = [1, 2, 3, 4, 5, 6]
>>> clist = alist[:3] + alist[3:]
>>> dlist = alist[:]
>>> id (alist)
139889255187144
>>> id (blist)
139889255187656
>>> id (clist)
139889255188168
>>> id (dlist)
139889255187528
>>> xlist = alist      # alist and xlist are alias
>>> id (xlist)
139889255187144
>>> alist [2] = 456
>>> alist
[1, 2, 456, 4, 5, 6]
>>> blist
[1, 2, 3, 4, 5, 6]
>>> clist
[1, 2, 3, 4, 5, 6]
>>> dlist
[1, 2, 3, 4, 5, 6]
>>> xlist
[1, 2, 456, 4, 5, 6]
>>> xlist [3] = 1234
>>> alist
[1, 2, 456, 1234, 5, 6]
>>> xlist
[1, 2, 456, 1234, 5, 6]
```

Lists `alist, blist, clist` and `dlist` have different identity. There is no alias effect.

In contrast, lists `xlist` and `alist` have the same identity. Therefore, when an element of list `alist` is modified, lists `blist, clist` and `dlist` remain unchanged. Conversely, as lists `xlist` and `alist` are alias, when list `alist` is modified, list `xlist` is also modified in the same way . And if `xlist` is modified, the change also affects `alist`.

If we want to have a copy of a list avoiding the alias effect, **cloning**, we need to create a different list object with the same value. The more suitable syntax to do it is:

```
dlist = alist[:]
```

List `dlist` is a copy of list `alist`, but they are not the same object, i.e., they have not the same identity.

6.8 Functions of the standard library

Functions of the standard library `max`, `min` and `sum`, already seen, and function `sorted` are applicable to homogeneous lists. The `sorted` function receives a list and returns another one with the same elements in ascending order. Examples:

```
>>> am = [4,6,5,1,2,3]
>>> max (am)
6
>>> min (am)
1
>>> sum (am)
21
>>> sorted (am)
[1, 2, 3, 4, 5, 6]
>>> am
[4, 6, 5, 1, 2, 3]
```

6.9 Type list: methods

In this section, we show some type `list` methods. These methods can modify the list or not. There are methods that have return values and others that return nothing. Actually any function or method returns always a value. When it returns *nothing* it actually returns the value `None`.

Functions and methods that modify a parameter are called **modifiers** and the changes they make are called **side effects**. This concepts are described in mode detail in Section 6.12.

The following table shows a selection of type `list` methods classified according to these characteristics:

	NOT modifier	modifier
return value	`count, index`	`pop`

returns `None`		`append`
		`sort, reverse`
		`insert, remove`

Methods are invoked using the dot notation:

`list_variable_name.method_name (actual_parameters)`

Next, the short documentation given by the `help` function is shown for these methods.

```
count(...)
    L.count(value) -> integer
    return number of occurrences of value

index(...)
    L.index(value, [start, [stop]]) -> integer
    return first index of value.
    Raises ValueError if the value is not present.

pop(...)
    L.pop([index]) -> item
    remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range
       .

append(...)
    L.append(object) -> None
    append object to end

sort(...)
    L.sort(key=None, reverse=False) -> None
    stable sort *IN PLACE*

reverse(...)
    L.reverse() -- reverse *IN PLACE*

insert(...)
    L.insert(index, object)
    insert object before index

remove(...)
    L.remove(value) -> None
    remove first occurrence of value.
    Raises ValueError if the value is not present.
```

Methods `count` and `index` are equivalent to methods `count` and `find` of `str` type. They are not modifiers. Method `index` raises an error when the value is not found.

Method `append` is a modifier: it adds an element at the end of a list. This method returns `None`.

Method `sort` is a modifier that sorts the given list and returns `None`.

Examples:

```
>>> lm = [4, 6, 5, 1, 2, 3]
>>> lm.append (10)
>>> lm
[4, 6, 5, 1, 2, 3, 10]
>>> lp = ['t', 'e', 'g', 'h', 'a']
>>> lps = sorted (lp)
>>> lp
['t', 'e', 'g', 'h', 'a']
>>> lps
['a', 'e', 'g', 'h', 't']
>>> lq = [6, 7, 2, 1, 5, 4]
>>> lq
[6, 7, 2, 1, 5, 4]
>>> lq.sort ()
>>> lq
[1, 2, 4, 5, 6, 7]
```

Note the difference between function `sorted` and method `sort`.

Method `reverse` is a modifier that reverses the given list and returns `None`.

Syntax `[::-1]` also applies to lists: it is evaluated to a new list and the given list remains unchanged.

Example:

```
>>> a1 = ['1', '2', '3', '4', '5']
>>> a2 = ['1', '2', '3', '4', '5']
>>> a1.reverse ()
>>> a1
['5', '4', '3', '2', '1']  # list a1 has changed
>>> rtm = a2 [::-1]
>>> rtm
['5', '4', '3', '2', '1']
>>> a2                      # list a2 has not changed
['1', '2', '3', '4', '5']
```

6.10 List traversal: map and filter

Typologies **map** and **filter** generally involve the creation of a new list. A map process applies a procedure to each element in the initial list in order to get the elements of the new list. In

a filter process, a condition is evaluated for each item in the initial list and only those items that meet the condition are added to the new list. Typologies reduce, map and filter can be combined.

Creating a list requires the initialization of an empty list `[]`. Then, to add a new element at the end of this new list the `append` method is used.

Exercise 6

An interval is represented by a list of two `float` elements which are its endpoints. Write a function `center` that takes a list of intervals, and returns another list with the values (`float`) of each interval midpoint, in the same order as in the initial list. Example:

```
>>> lint = [[-3, 3], [1.2, 4.8], [-56, 24.6], [-20, -10]]
>>> center (lint)
[0.0, 3.0, -15.7, -15.0]
```

Design

```
def centre (lintervals):
    lcentres = []
    for interval in lintervals:
        c = (interval[0] + interval[1])/2
        lcentres.append(c)
    return lcentres
```

In this example we are given a nested list of lists, `lint`. This is a map problem: for each interval of the given list, the corresponding midpoint is computed and added to the new list `lcentres`.

Exercise 7

Write function `nouns` that takes a string with a text, and returns an alphabetically sorted list with all the nouns in the given text. To detect nouns we will apply only the following simple rule: a noun is always preceded by either the word 'a' or 'the' and these two words only come before a noun. We can assume that in the text there are no punctuation marks and that all letters are lowercase. Words are separated by one or more spaces. Example:

```
>>> text = 'a book is on the table and
... a notebook has fallen under the stool'
>>> nouns (text)
['book', 'notebook', 'stool', 'table']
```

Design

```
def nouns(text):
    lwords = text.split()
    lnouns = []
    for i in range(len(lwords)-1):
        if lwords[i]=='a' or lwords[i]=='the':
            lnouns.append(lwords[i+1])
    lnouns.sort()
    return lnouns
```

In this example, we obtain a list with all the words in the given string using the `split` method in its specific form with whitespace characters as separators. Then we generate a set of pairs of consecutive words in order to detect pairs following the pattern `('a', noun)` or `('the', noun)`. To deal with pairs of consecutive words the list traversal will be performed through the index and, then, a filter process is applied. The new list obtained is finally sorted with method `sort`.

Map and filter processes to create new strings

We can also apply map and filter processes to create new strings. A first strategy constructs a list of characters instead of a string so that the method `append` could be applied, and then converts the constructed list into a string using the `join` method.

Exercise 8

Write the function `extract_vowels` that takes a string containing a text and returns another string with all and only the vowels of the given string and in the same order. Example:

```
>>> extract_vowels('Hello. Are you OK ?')
'eoAeouO'
>>> extract_vowels("The cat could very well be man's best
...         friend but would never stoop to admitting it.")
'eaoueeeaeieuoueeoooaiii'
```

Design

```
def isvowel (c):
    vowels = 'aeiouAEIOU'
    return c in vowels

def extract_vowels (s):
    newlist = []
    for c in s:
        if isvowel(c):
            newlist.append(c)
    return ''.join(newlist)
```

This is a filter example: the condition that must to be met by characters in order to be in the returned string is that they have to be a vowel.

Exercise 9

Write the function `apply_vowel` that takes a string containing a text and another string with a vowel, and returns another string in which all the vowels have been substituted for the given one. Example:

```
>>> extract_vowels('Hello. Are you OK ?', 'o')
'Hollo. oro yoo oK ?'
>>> extract_vowels("The cat could very well be man's best
...    friend but would never stoop to admitting it.", 'a')
Tha cat caald vary wall ba man's bast fraand
bat waald navar staap ta admattang at."
```

Design

```
def isvowel (c):
    vowels = 'aeiouAEIOU'
    return c in vowels

def apply_vowel (s, v):
    newlist = []
    for c in s:
        if isvowel(c):
            newlist.append(v)
        else:
            newlist.append(c)
    return ''.join(newlist)
```

In this example the processes of map and filter are combined.

6.11 Append vs. concatenation

Python has another mechanism to add elements at the end of a structure: concatenation. For strings, at each iteration we concatenate the new string and the current character. Next another version of exercise 8 using this strategy is shown:

Design

```
def isvowel (c):
    vowels = 'aeiouAEIOU'
    return c in vowels

def extract_vowels (s):
    newstr = ''
    for c in s:
        if isvowel(c):
            newstr = newstr + c
    return newstr
```

Now, we do not use any auxiliary list. A new string `newstr` is initialized as an empty string and at each iteration the current character `c` is concatenated to `newstr`. Although this version works correctly, we have to pay attention to the assignment statement `newstr = newstr + c`. It actually creates a new object at each iteration and, therefore, the memory management with this strategy could be less efficient.

6.12 Modifiers

A **modifier** is a function that modifies some of its parameters: it produces a **side effect**. To modify a parameter it has to be of a mutable type, as lists or dictionaries (see Section 9. Therefore, functions without any mutable parameter cannot be modifiers while functions with

mutable parameters can be modifiers or not. Functions that are not modifiers can be referred to as **pure functions** in order to better distinguish them.

We have seen that when a list is assigned to another one, they become alias. Modifiers rely on this fact because when a function with a mutable type parameter is called, both the corresponding formal parameter and actual parameter become alias. Therefore if the the formal parameter is modified by the function the corresponding actual parameter is also modified accordingly.

Exercise 10

Write a modifier (function) `double_modifier` that takes a list and modifies it so that the new elements are the previous ones multiplied by two. Example:

```
>>> ln = [3, 5.2, 'hello', 25]
>>> rtn = double_modifier (ln)
>>> ln
[6, 10.4, 'hellohello', 50]
>>> print (rtn)
None
```

Design

```
def double_modifier(lm):
    for i in range(len(lm)):
        lm[i] = 2* lm[i]
    return
```

Observations:

- the list may have elements of whatsoever type provided it has the `*` operator
- this function needs to modify the list elements and this operation requires the list index. Therefore the list traversal is done through indexes.
- this function does not create a new list. Instead it modifies the given list parameter. The applied tests show list `ln` values before and after calling the function. The modifier function `double_modifier` modifies the list formal parameter and, as a side effect, it also modifies the actual parameter, as they both are alias.
- this function does not return any list. It returns the value `None`.

If we do not want to modify the parameter, we could write a pure function that creates a local variable which will be returned. The following exercise solves a problem similar to the previous one with a pure function.

Exercise 11

Write the function `double_pure` that takes a list, and returns another list such that its elements are equal to those of the given list multiplied by two. Example:

```
>>> ln = [3, 5.2, 'hello', 25]
>>> rtn = double_pure (ln)
```

```
>>> rtn
[6, 10.4, 'hellohello', 50]
>>> ln
[3, 5.2, 'hello', 25]
```

Design

```
def double_pure (lm):
    newlist = []
    for e in lm:
        newlist.append(2*e)
    return newlist
```

Observations:

- this function can perform the list traversal directly without indexes
- this function creates a new list and does not modify the list given as a parameter. The given tests show that list `ln` has not been modified after calling function `double_pure`
- this function returns the newly created list. In the given tests, the returned list is assigned to variable `rtn`, and then its content is shown

It is recommended to compare exercise definitions, functions and doctest of the two previous exercises, for a better understanding of the concepts of mutability and modifiers.

In the previous sections we have seen several functions of the standard library that apply to lists as well as several type `list` methods. Functions `max`, `min`, `sum` and `sorted` are not modifiers. Concerning methods, a table in Section 6.9 shows which methods are modifiers and which are not.

In the following example, function `sorted` (pure function) and method `sort` (modifier) are compared.

```
>>> alist = [34, 25, 88, 76, 23, 14]
>>> blist = sorted (alist)
>>> alist                  # alist unchanged
[34, 25, 88, 76, 23, 14]
>>> blist                  # list returned by function sorted
[14, 23, 25, 34, 76, 88]
>>> rtn = alist.sort()
>>> alist                  # alist has been modified
[14, 23, 25, 34, 76, 88]
>>> print(rtn)             # method sort return None
None
```

Function `sorted` is a pure function that does not modify the list parameter `alist` and returns a different list with the same elements as the given list in ascending order. Conversely, method `sort` is a modifier: it modifies the given list `alist` and returns `None`.

Compare these observations with those given for previous function examples `double_modifier` and `double_pure`.

6.13 Optional parameters

Both function `sorted` and list method `sort` have two optional parameters: `reverse` and `key`.

Parameter `reverse` is a Boolean. By default is `False` and the sorting is performed in ascending order. If set to `True`, then the sorting is performed in descending order. It has the same effect as first sorting and then reversing (method `reverse`. Examples:

```
#### examples with function sorted
>>> a = [3, 5, 1, 7, 8]
>>> sorted(a)
[1, 3, 5, 7, 8]
>>> sorted(a, reverse=True)
[8, 7, 5, 3, 1]

#### same examples with method sort
>>> a = [3, 5, 1, 7, 8]
>>> a.sort()
>>> a
[1, 3, 5, 7, 8]
>>> a = [3, 5, 1, 7, 8]
>>> a.sort(reverse = True)
>>> a
[8, 7, 5, 3, 1]
```

Parameter `key` is a function of one parameter. The sort order can be customized with this key function. The default value is `None`.

The key function can be an already available function or method (see examples below with function `len` and method `lower`). It can also be a specifically designed function (see example below with function `vowels_number`) or an anonymous function.

Anonymous (`lambda`) functions can be created with the `lambda` keyword. Lambda functions are syntactically restricted to a single expression. Examples:

`lambda a, b: a+b` returns the sum of its two arguments
`lambda n: 2**n` returns the nth power of 2

Below, there are examples of the `sorted` function and `sort` method using the `key` parameter in several ways.

```
>>> from vowels import vowels_number

#### examples with function sorted: key parameter

# alphabetical order
# first the text is converted to lowercase letters
```

```
>>> la = 'this is a test string from Andrew'.split()
>>> sorted(la, key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'this']

# sort by length
>>> sorted(la, key=len)
['a', 'is', 'this', 'test', 'from', 'string', 'Andrew']

# sort by number of 't' characters
>>> sorted(la, key=lambda x: x.count('t'))
['is', 'a', 'from', 'Andrew', 'this', 'string', 'test']

# sort by number of vowels
>>> lb ='these are some test strings from Paul'.split()
>>> sorted(lb, key= vowels_number)
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']
>>> sorted(lb, key= lambda x: vowels_number(x))
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']


#### same examples with method sort: key parameter

# alphabetical order
# first the text is converted to lowercase letters
>>> la = 'this is a test string from Andrew'.split()
>>> la.sort(key=str.lower)
>>> la
['a', 'Andrew', 'from', 'is', 'string', 'test', 'this']

# sort by length
>>> la = 'this is a test string from Andrew'.split()
>>> la.sort(key=len)
>>> la
['a', 'is', 'this', 'test', 'from', 'string', 'Andrew']

# sort by number of 't' characters
>>> la = 'this is a test string from Andrew'.split()
>>> la.sort(key=lambda x: x.count('t'))
>>> la
['is', 'a', 'from', 'Andrew', 'this', 'string', 'test']

# sort by number of vowels
lb = 'these are some test strings from Paul'.split()
>>> lb.sort(key= vowels_number)
>>> lb
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']
```

```
>>> lb.sort(key= lambda x: vowels_number(x))
>>> lb
['test', 'strings', 'from', 'these', 'are', 'some', 'Paul']
```

and function `vowels_number` is defined as:

```
def vowels_number(s):
    vow = 'aeiouAEIOU'
    nv = 0
    for c in s:
       if c in vow:
          nv = nv +1
    return nv
```

Sorting nested lists is performed by default in lexicographical order. A `key` function can also be used to change this default ordering. See the examples bellow.

```
#### nested lists ordering: function sorted
# example: list of tuples of type (name, grade, age)
>>> students = [('john', 'A', 15), ('jane', 'A', 12),
...             ('dave', 'B', 10), ('ed', 'B', 12)]

# sort by name (default: first element in tuples)
>>> sorted(students)
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# sort by age
>>> sorted (students, key=lambda student: student[2])
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]

# sort first by age and then by name
>>> sorted (students,
...    key=lambda student: (student[2], student[0]))
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# sort first by age and then by grade
>>> sorted (students,
...    key=lambda student: (student[2], student[1]))
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]


#### nested lists ordering: method sort
# example: list of tuples of type (name, grade, age)
>>> students = [('john', 'A', 15), ('jane', 'A', 12),
```

```
...              ('dave', 'B', 10), ('ed', 'B', 12)]

# sort by name (default: first element in tuples)
>>> students.sort()
>>> students
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# sort by age
>>> students = [('john', 'A', 15), ('jane', 'A', 12),
...              ('dave', 'B', 10), ('ed', 'B', 12)]
>>> students.sort(key=lambda student: student[2])
>>> students
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]

# sort first by age and then by name
>>> students = [('john', 'A', 15), ('jane', 'A', 12),
...              ('dave', 'B', 10), ('ed', 'B', 12)]
>>> students.sort(key=lambda student: (student[2], student[0]))
>>> students
[('dave', 'B', 10), ('ed', 'B', 12),
 ('jane', 'A', 12), ('john', 'A', 15)]

# sort first by age and then by grade
>>> students = [('john', 'A', 15), ('jane', 'A', 12),
...              ('dave', 'B', 10), ('ed', 'B', 12)]
>>> students.sort(key=lambda student: (student[2], student[1]))
>>> students
[('dave', 'B', 10), ('jane', 'A', 12),
 ('ed', 'B', 12), ('john', 'A', 15)]
```

There are other functions also based on the lexicographical order with the optional `key` parameter, as `max` and `min`.

7 Tuples

Tuples are a sequence data type. As lists, tuples can also be homogeneous or heterogeneous. A tuple is represented in parentheses, (), and with its elements separated by commas: $(e_0, e_1, \ldots, e_n)$. Actually parentheses are not compulsory but its use is recommended and Python always represents tuples between parentheses. An empty tuple is represented as `()`. Examples:

```
>>> t1 = (3, 8, 5, 30, 23)  # homogeneous tuple
>>> t2 = ('home', 23, True, 45.78, 'bye') # heterogeneous tuple
```

A tuple with a single element `e` is represented as `(e,)`. The comma is necessary to distinguish it from a scalar object. The operator comma has less priority than other operators and if we want to evaluate it first we will need to use parentheses.

Concatenation (+) and replication (*) operations also apply to tuples. Comparison operators, `==, !=, <, >, <=, >=`, are also applicable and based on the lexicographical order. As with lists, when two tuples are compared, their elements are compared in the order in which they appear in each tuple. When comparing tuple elements, we must take into account that they must be of the same type. The `len` function of the standard library gives the length of a tuple, i.e., its number of elements.

Membership operators `in` and `not in` as well as indexing and slicing operators are also applicable to tuples. Let `t` be a tuple, the syntax to access an element is `a[index]`, $0 \leq index < len(t)$.

Examples:

```
>>> (5)
5
>>> (5)  # an integer value
5
>>> (5,)  # a tuple with one element
(5,)
>>> a = 3
>>> a + 4,  # operator + has more priority than operator ,
(7,)
>>> a = (5, 6, 7)
>>> a + 4,  # operator + has more priority than operator ,
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
>>> b = a + (4,)  # tuple concatenation
>>> b
(5, 6, 7, 4)
>>> b[1]
6
>>> b[1:]
(6, 7, 4)
>>> 7 in b
True
```

In expression `(5, 6, 7) + 4`, operator `+` has more priority than operator comma (`,`). Then Python is asked to concatenate an integer to a tuple and therefore raises an error as this operation is not allowed. To concatenate a tuple with 3 elements and another tuple with an element, both tuples have to be represented in parentheses.

The fundamental difference between tuples and lists is that tuples are immutable while lists are mutable.

7.1 Tuples and multiple assignment

Tuple assignment can be interpreted as multiple variable assignment. Examples:

```
>>> m, n = 5, 6     # this is a tuple assignment
>>> m, n
(5, 6)
>>> m
5
>>> n
6

>>> a = 3, 4    # tuple assignment: pack
>>> a
(3, 4)
>>> x, y = a    # tuple assignment: unpack
>>> x, y
(3, 4)
>>> x
3
>>> y
4
```

Packing and unpacking can be applied to strings and lists as well:

```
>>> la = ['bye', 'hello']
>>> s1, s2 = la
>>> s1
'bye'
>>> s2
'hello'
>>> sa = 'xyz'
>>> sa1, sa2, sa3 = sa
>>> sa1
'x'
>>> sa2
'y'
>>> sa3
'z'
```

7.2 Tuples and functions with several return values

A Python function always returns a single value. So far, we have written functions with several return values, but actually they returned a single value: a tuple that Python shows between parentheses.

Let's analyze function `example` that takes two numeric values and returns their addition and product:

```
def example(a, b):
    return a+b, a*b
```

This function actually returns a single tuple object with two elements. Examples:

```
>>> sum, prod = example(4, 5)    # unpacked tuple assignment
>>> sum, prod
(9, 20)
>>> t1 = example(3, 8)  # tuple assignment
>>> t1
(11, 24)
```

8 Exercises: lists and tuples

1. An interval is represented by a list of two elements that represent their lower and upper extremes (`float`). Given a list of intervals, write the function `length` that returns another list with the length of those intervals with a lower extreme negative, in the same order as in the given list. Example:

```
>>> lintervals = [(-3, 3), (1.2, 4.8), (-56, 24.6),
...  (-20, -10)]
>>> length (lintervals)
[6, 80.6, 10]
```

```
def length (lintervals):
    llengths = []
    for interval in lintervals:
        ei, es = interval
        if ei < 0:
            a = es - ei
            llengths.append(a)
    return llengths
```

   In this exercise a tuple assignment (unpack) is performed. Moreover a filter together with a map process is applied.

2. An array is represented as a list of sublists where each sublist represents a row in the array. Design a pure function `product` that from an array and a numerical value, returns another array with the result of multiplying the array given by the value. Example:

```
>>> array = [[2, 3, 4], [7, 5, 3], [8, 0, 9]]
>>> product(array, 2)
[[4, 6, 8], [14, 10, 6], [16, 0, 18]]
>>> array
[[2, 3, 4], [7, 5, 3], [8, 0, 9]]
```

```
def product(arr, v):
    arrnew = []
    for row in arr:
        rownew = []
        for element in row:
            rownew.append(element * v)
        arrnew.append(rownew)
    return arrnew
```

3. Design a modifier function `product` that from an array and a numerical value, modifies the given array by multiplying it by the given value. Example:

```
>>> array = [[2, 3, 4], [7, 5, 3], [8, 0, 9]]
>>> productmod(array, 2)
```

```
>>> array
[[4, 6, 8], [14, 10, 6], [16, 0, 18]]
```

```
def productemod(mat, esc):
    nfil = len(mat)
    ncol = len(mat[0])
    for i in range(nfil):
        for j in range (ncol):
            mat[i][j] = mat[i][j] * esc
```

Compare the two previous exercises, code and examples, in order to understand the concepts of mutability and modifier functions. Note that while function `product` does NOT modify the given array, function `productmod` does modify it.

4. A company represents its stock with a list of tuples where each tuple represents a product with its code (string) and units (integer). Write the function `maxunits` that takes a list of tuples as indicated, and returns the tuple corresponding to the product with the maximum number of units. We can assume that the list is not empty.

```
>>> ll =[('AR35', 100), ('FT78',89), ('YH98', 34),
...  ('JH65', 120), ('UH56', 77), ('WE34', 76)]
>>> maxunits(ll)
('JH65', 120)
```

```
def maxunits (lstock):
    maxp = -1
    for product in lstock:
        code, units = product
        if units > maxp:
            maxp = units
            codemax = code
    return codemax, maxp

def maxunits_1 (lstock):
    return max(lstock, key=lambda x: (x[1], x[0]))
```

In this exercise we apply a synthesis process: obtaining the maximum. The second version of this function uses the `key` parameter of function `max`.

9 Dictionaries

9.1 Definition

So far, we have seen compound data types, strings, lists, and tuples, which are sequence types and are an ordered aggregation of elements. Dictionaries (type `dict`) are also a compound type but they are a **mapping** type that represents a mapping between two sets: the **keys** set and the **values** set. Keys must be of any type provided it is an immutable type while values can be of any type (immutable and mutable). The key must be unique.

A dictionary is a set of pairs **key:value** and is represented in braces ({}) and with its elements separated by commas. The pair key:value is separated by a colon:

$$\{k_A : v_A, k_B : v_B, \ldots, k_Z : v_Z\}$$

An empty dictionary is represented as {}. Examples:

```
>>> d1 = {'apples': 2.45, 'pears':2.45, 'cherries':3.5 }
>>> d2 = {(3, 4): 'blue', (3, 5): 'green', (4, 5):'green'}
>>> d3 = {'A': ['Amelia', 'Ann'], 'B': ['Bertha', 'Blanche'],
...       'C': ['Claire', 'Chloe']}
```

Dictionary `d1` represents a mapping between product names (`str`) and prices (`float`). Dictionary `d2` represents a mapping between pixels (2-tuples) and colors (`str`). Finally, `d3` is a dictionary in which the key is a letter (`str`) and the value a list of female names beginning with this letter (list of strings).

In a dictionary, the key must be unique but this restriction does not apply to values. In the previous examples the value `2.45` is the same for for `'apples'` and `'pears'` and the value `'green'` is the color of pixels `(3, 5)` and `(4, 5)`.

9.2 Indexing and other operations

Dictionaries are not sequence types and, therefore, we cannot access to its components by an integer index. To access to a dictionary element we have to use the key, and the syntax is:

`dictionary_name[key]`

This expression gives the value corresponding to the given key. Examples:

```
>>> d1['apples']      # acces to an element
2.45
>>> d2[(3, 5)]
'green'
>>> d3['B']           # d3['B'] is a list
['Bertha', 'Blanche']
>>> d3['B'][1]        # second element of list d3['B']
'Blanche'
```

Pairs key:value in a dictionary are stored in an arbitrary order determined by Python which is inaccessible by the programmer.

Comparison operators are applicable to dictionaries but only for equality (`==`) or inequality (`!=`). The remainder comparison operators are not supported for dictionaries.

The `len` function of the standard library gives the length of a dictionary, i.e., its number of pairs key:value.

Membership operators `in` and `not in` are also applicable to dictionaries and they use the key to ask for a pair key:value to belong to a dictionary. The syntax is: `key in dictionary`.

Dictionaries do NOT support neither concatenation nor replication and as they do not use integer indexes, slicing is not supported either.

Dictionaries are a mutable type as lists. Therefore, we are allowed to insert, modify and delete elements. To insert a new element as well as to modify the value of an existing pair, the syntax is the same: `dict_name[key] = value`. If there is no pair with the given `key` in the dictionary, this assignment actually inserts the pair `key:value` and if this `key` is already in the dictionary then its value is updated by the new given `value`.

To remove an item (a pair `key:value`) from a dictionary, `d`, we can use the statement `del` with the syntax `del d[key]`.

Examples:

```
>>> d1 = {'apples': 2.45, 'pears':2.45, 'cherries':3.5 }
>>> d2 = {(3, 4): 'blue', (3, 5): 'green', (4, 5):'green'}
>>> d3 = {'A': ['Amelia', 'Ann'], 'B': ['Bertha', 'Blanche'],
...       'C': ['Claire', 'Chloe']}
>>> d1 == d3
False
>>> len(d1)
3
>>> len(d3)
3
>>> 'A' in d3
True
>>> 'm' in d3
False
>>> (10, 10) in d2
False
>>> # insert a new element 'oranges':1.98
>>> d1 ['oranges'] = 1.98
>>> d1 == {'apples': 2.45, 'cherries': 3.5, 'pears': 2.45,
...  'oranges': 1.98}
True
>>> d2[(4, 4)] = 'green'       # insert a new element
>>> d2 == {(4, 5): 'green', (4, 4): 'green', (3, 4): 'blue',
...    (3, 5): 'green'}
True
```

```
>>> d3['D'] = ['Dolly']     # insert a new element
>>> d3 == {'D': ['Dolly'], 'A': ['Amelia', 'Ann'],
...  'C': ['Claire', 'Chloe'], 'B': ['Bertha', 'Blanche']}
True
>>> d3['A'] = []  # new value for item with key 'A'
>>> d3 == {'D': ['Dolly'], 'A': [], 'C': ['Claire', 'Chloe'],
...        'B': ['Bertha', 'Blanche']}
True
>>> del d3['C']              # item removal
>>> d3 == {'D': ['Dolly'], 'A': [], 'B': ['Bertha', 'Blanche']}
True
```

9.3 Dictionaries traversal and search

Traversal and search schemes can be applied to dictionaries with the statement `for` and dictionary pairs are addressed through their keys. Python traverses a dictionary following the internal order in which Python stores its elements. Syntax:

```
for key in dictionary:
    process key, dictionary[key]
```

9.4 Type `dict`: methods

Type (`dict`) has several methods. To invoke them, the dot notation is used:

`dict_variable_name.dict_method_name(actual_parameters)`

Methods `keys`, `values` and `items` return an *iterator* object that can be converted to a list (with function list). They return an iterator with the dictionary `keys`, `values` and 2-tuples `(key, value)`, respectively. Examples:

```
>>> d1 = {'apples': 2.45, 'pears':2.45, 'cherries':3.5 }
>>> list(d1.keys())
['cherries', 'pears', 'apples']
>>> list(d1.values())
[3.5, 2.45, 2.45]
>>> list(d1.items())
[('cherries', 3.5), ('pears', 2.45), ('apples', 2.45)]
```

Method `get` has the following specification:

`get(k[,d]):  D[k] if k in D, else d.  d defaults to None.`

which is equivalent to the following function:

```
def get_equivalent(D, k, d):
    if k in D:
        return D[k]
    else:
        return d
```

As said before, dictionaries are mutable and, therefore the concepts of alias and cloning are also applicable. The syntax to define a dictionary `dicty` as an alias of another existing dictionary `dictx` is (the same as with lists):

```
dicty = dictx
```

If we want to have a copy of a dictionary avoiding the alias effect (cloning), we have to use method `copy` of type `dict`. Examples:

```
>>> d1 = {'apples': 2.45, 'pears':3.25, 'cherries':3.5 }
>>> d2 = d1      # d1 and d2 are alias
>>> d3 = d1.copy()  # cloning: d3 and d1 are NOT alias
>>> # a change in d1 is also produced in d2 but not in d3
>>> d1['apples'] = 1.98
>>> d1 == {'cherries': 3.5, 'pears': 3.25, 'apples': 1.98}
True
>>> d2 == {'cherries': 3.5, 'pears': 3.25, 'apples': 1.98}
True
>>> d3 == {'apples': 2.45, 'pears': 3.25, 'cherries': 3.5}
True
>>> # a change in d2 is also produced in d1 but not in d3
>>> del d2['pears']
>>> d1 == {'cherries': 3.5, 'apples': 1.98}
True
>>> d2 == {'cherries': 3.5, 'apples': 1.98}
True
>>> d3 == {'apples': 2.45, 'pears': 3.25, 'cherries': 3.5}
True
>>> d3['pears'] = 2.15  # changing d3 only affects d3
>>> d1 == {'cherries': 3.5, 'apples': 1.98}
True
>>> d2 == {'cherries': 3.5, 'apples': 1.98}
True
>>> d3 == {'apples': 2.45, 'pears': 2.15, 'cherries': 3.5}
True
```

9.5 Dictionaries and tests

In the previous tests we can see that we actually check whether the obtained dictionary is equal or not to the expected one: this is a Boolean expression involving the equals sign (`==`) that gives a result of `True` or `False`:

```
>>> d3 == {'apples': 2.45, 'pears': 2.15, 'cherries': 3.5}
True
```

If we did the test in the same way as for other type objects:

```
>>> d3
{'apples': 2.45, 'pears': 2.15, 'cherries': 3.5}
```

we could obtain a dictionary `d3` in the following form:

`d3 = {'pears': 2.15, 'cherries': 3.5, 'apples': 2.45}`.

Obviously these dictionaries, although not shown in the same order, are equal and the following expression

```
{'apples': 2.45, 'pears': 2.15, 'cherries': 3.5} ==
{'pears': 2.15, 'cherries': 3.5, 'apples': 2.45}
```

evaluates to `True`.

However the doctest facility compares them for literal equality, i.e., actually it compares two strings:

`'{'apples': 2.45, 'pears': 2.15, 'cherries': 3.5}'`

and

`'{'pears': 2.15, 'cherries': 3.5, 'apples': 2.45}'`

and then the result is `False`. Therefore, we have to write these tests in the mentioned Boolean expression form.

We can design a more explicit test:

```
>>> if d3 != {'apples': 2.45, 'pears': 2.15, 'cherries': 3.5}:
...     print(d3)
```

Using this test, if the obtained dictionary is different than the expected one, the test will print the obtained dictionary so that the programmer will be able to compare it with the expected one

9.6 Exercises

1. A text contains several numeric quantities and we want to modify it so that if these quantities are less than 10, they will appear with the corresponding word instead of the digit. We can assume that the letters of the text are all lowercase and that there are no punctuation marks. There is also a dictionary with a pair for each digit, where the key is a string with a digit and the value is another string with the corresponding word.

   Write function `changenums` that takes a text and a dictionary as described and returns the text modified as shown in the previous paragraph. Example:

   ```
   >>> dic = {'0': 'zero', '1': 'one', '2': 'two',
   ... '3': 'three', '4': 'four', '5': 'five',
   ```

```
... '6': 'six', '7': 'seven', '8': 'eight', '9': 'nine'}
>>> t1 = 'there are 5 tables and 3 chairs; only 1 wardrobe'
>>> changenums (t1, dic)
'there are five tables and three chairs; only one wardrobe'
```

In this exercise a dictionary is used as a look-up tool that allows to apply changes to another data structure.

```
def changenums (text, dic):
    lwords = text.split()
    newl = []
    for word in lwords:
        if word in dic:
            newl.append(dic[word])
        else:
            newl.append(word)
    return ' '.join(newl)
```

2. There is a list of strings with Christmas lottery numbers that have received first prizes in previous years. Write a function that returns a dictionary where the key is the ending digit of a number (string) and the value is the number of times that there have been first prizes ending with that digit. We have to write two different functions.

   The first function, named `endings1`, must return a complete dictionary, i.e., with the 10 possible elements. Example:

```
>>> lot = ['12345', '54321', '98541', '76781', '98765',
...        '65432', '98762', '98654', '01235']
>>> d = endings1(lot)
>>> d == {'8': 0, '9': 0, '0': 0, '6': 0, '3': 0, '7': 0,
...       '2': 2, '5': 3, '4': 1, '1': 3}
True
```

```
def endings1 (lottery):
    digits = '0123456789'
    dendings = {}
    for d in digits:
        dendings[d] = 0
    for num in lottery:
        digit = num[-1]
        dendings[digit] = dendings[digit] + 1
    return dendings
```

In this example a dictionary is created that represents a frequency table, which is one of the dictionaries applications. In this case, the dictionary must be initialized with all the possible keys since the problem statement requires a complete dictionary. In addition, since the value is a counter, all must be initialized to `0`.

The second function, named `endings2`, must return a dictionary that does not need to be complete: only the keys corresponding to ending digits of the numbers in the given list must appear in the dictionary. Example:

```
>>> lot = ['12345', '54321', '98541', '76781', '98765',
...        '65432', '98762', '98654', '01235']
>>> d = endings2(lot)
>>> d == {'2': 2, '5': 3, '4': 1, '1': 3}
True
```

```
def endings2 (lottery):
    dendings = {}
    for num in lottery:
        digit = num[-1]
        if digit in dendings:
            dendings[digit] = dendings[digit] + 1
        else:
            dendings[digit] = 1
    return dendings
```

In this case, we do not have to initialize the dictionary with all the possible keys since the dictionary does not need to be complete. The pairs of the dictionary are created on the fly: if the key is not in the dictionary, it is inserted with a value initialized to `1`. Conversely, if the key is already in the dictionary, the corresponding value is incremented in one unit.

3. There is a list of strings with Christmas lottery numbers that have received first prizes in previous years. Write the function `awarded` that takes a list as described and returns a dictionary where the key is the ending digit of a number (string) and the value is a list with all the numbers ending with that digit. Numbers in these lists must appear in the same order in which they are in the given list. The dictionary does not need to be complete. Example:

```
>>> lot = ['12345', '54321', '98541', '76781', '98765',
...        '65432', '98762', '98654', '01235']
>>> awarded (lot)
{'2': ['65432', '98762'], '5': ['12345', '98765', '01235'],
 '4': ['98654'], '1': ['54321', '98541', '76781']}
```

```
def premiats (loteria):
    premi = {}
    for num in loteria:
        acab = num[-1]
        if acab in premi:
            premi[acab].append(num)
        else:
            premi[acab] = [num]
    return premi
```

In this example a dictionary is created in which the values are lists. As we are no asked to create a complete dictionary, we do not have to initialize it with all the possible keys. Dictionary pairs are created on the fly: if the key is not in the dictionary, it is inserted and the corresponding value is a list with a single element: the current number. Conversely, if the key is already in the dictionary, the number is added at the end of the corresponding list using method `append`.

4. The expenses of a traveler are represented in a dictionary where the key is an integer indicating the day and the value is a list with the expenses of this day.

   (a) Write function `total_expenses` that takes a dictionary as indicated and returns the total sum of the expenses. Example:

```
>>> d = {12:[2,3,4,5], 13:[6,7,6,8], 14:[50,10,4,5],
...      15:[4,6,10,12]}
>>> total_expenses(d)
142
```

   In this exercise we have to traverse the dictionary.

```
def despesa_total (despeses):
    total = 0
    for dia in despeses:
        total = total + sum(despeses[dia])
    return total
```

   (b) Write the function `special_occasion` that takes a dictionary as indicated and a certain amount (all in the same currency), and returns an integer corresponding to the first day found in the dictionary in which the expense has exceeded that amount. If there is no such amount, the function must return the value `0`. Example:

```
>>> d = {12:[2,3,4,5], 13:[6,7,6,8], 14:[50,10,4,5],
...      15:[4,6,10,12]}
>>> special_occasion(d, 60)
14
>>> special_occasion(d, 100)
0
```

   In this exercise we have to perform a search through the dictionary.

```
def special_occasion (expenses, quant):
    found = False
    for day in expenses:
        found = sum(expenses[day]) > quant
        if found:
            theday = day
            break
    if found:
        return theday
    else:
        return 0
```

10 Files

10.1 Sequential text files (STF)

A computer file is a computer resource for recording data in a computer storage device. While a program is running, its data (variables, values, etc.) is stored in random access memory (RAM). RAM is fast but it is volatile, which means that when the program ends this data disappears. Data can also be stored in a non-volatile storage device, such as a hard disk or a usb memory.

Computer applications might need to work with data stored in files so that data is available each time the program is executed. Specifically, we need programs and functions to be able to get (read) data from and to write data to a file. By reading and writing files, programs can save information between program runs.

Files are basically a sequence of bytes. There is a wide typology of computer files, but in this course we will deal with an specific type: **sequential text files** (STF). An STF is a file that consists of a sequence of characters.

A **sequential** file is a file that has to be traversed from the beginning to the end and we cannot neither go backwards nor jump to an specific character position: this restriction implies that we cannot read from and write to a file at the same time. In contrast, direct access files allow to access to specific positions.

A **text** file is a file composed by bytes which correspond to printable characters and therefore they can be opened and edited with a plain text editor. In contrast, binary files are general files in which bytes can have other meanings than the ASCII codification for printable characters. Therefore binary files cannot be edited. For instance, a file with extension `.py` that contains Python code is a text file (Python source code). On the other hand, a file with extension `.pyc` is a binary file and it cannot be readable with a text editor: bytes in this file correspond to Python instructions translated to the machine language. Binary files with extension `.pyc` are built automatically by Python when we import a module. This makes importing it again later faster.

STF are generally organized in **lines** (also called **records**). This means that there are some line feed characters (`'\n'`) that stand for **end of line** (and that a text editor will interpret as a line feed). Lines can be of fixed or variable length. Files can be structured as tables used in data bases, i. e., all lines having the same data items or **fields**. For instance, a file with the data of the products sold by a company in which each file corresponds to a product and the fields are the data items stored for each product as its code, price, number of stored unities, etc.

An STF file with data has a name as any other file. An extension is not strictly required but we suggest to use the extension `.txt` (standing for plain text file). In this beginner's programming course we also highly recommend that an STF file be in the same directory than the Python file containing the function that will deal with this STF file. Obviously this is not strictly necessary, but if these files are not in the same directory, you will need to precede the name of the STF file by the corresponding path. These data files can be edited with any text editor as the `idle` or `emacs` editors.

10.2 STF and Python

To work with files, Python offers a suitable type. The actual name of this type in Python is `io.TextIOWrapper` but we will refer to it as `file` in this section. Any file used in a Python function will be represented by a variable of this type. This variable is known as **internal file** while the name with which we refer to the file in the storage device is **external file**.

A Python program or function must establish a communication **channel** with a file before reading from or writing to it. Function `open` of the standard library creates a channel or **link** between the external file and the internal file. When we finish working with a file we have to close this channel. However if we use the `with` statement the file is closed implicitly.

Next, the syntax of the `open` function together with the `with` statement is shown:

```
with open(external_file_name, mode) as internal_file_name
```

- `external_file_name` is a string with the name of the external file
- `mode` is also a string which specifies the opening mode. We will consider the following opening modes: reading ('r') or writing ('w'). When we open a file in reading mode, the file must exist. When we open a file in writing mode, the file must not exist: this mode creates a new file with the given name. If we open an existing file with 'w' mode, its content will be lost.
- `internal_file_name` is the name of the type file variable linked to the corresponding external file

Example:

```
with open ('first.txt', 'r') as f1:
    with open ('second.txt', 'w') as f2:
```

This code opens a file with name `first.txt` in reading mode and links it to the file type variable `f1`. If file `first.txt` doesn't exist in the working directory, this function raises an error. Then another file, `second.txt`, is opened in writing mode and linked to the file type variable `f2`. If file `second.txt` exists in the working directory, its content will be lost while if it doesn't exist, a new file with name `second.txt` will be created.

The previous statement can be also written as :

```
with open('fisrt.txt', 'r') as f1, open('second.txt', 'w') as
   f2:
```

Type file has the following methods:

Reading methods:

```
f.read(size)
    reads at most size characters from file f and
    returns a string with these characters
```

```
    If size is negative or omitted, read until EOF


f.readline(size)
    reads characters from the following line of file f
    (until line feed or end of file (EOF) is reached)
    returns a string with these characters

f.readlines()
    reads all file f
    returns a list of strings, each string corresponding to a line
```

Writing method:

```
f.write(text)
    writes string text to file f
    returns an integer (the length of text)
```

Closing method:

```
f.close()
    Closes file f. Returns nothing.
```

10.3 File traversal

Sequential text files consists of one or more lines ending with the line feed character (`'\n'`) and Python treats such files as a sequence of lines. Therefore, the `for` statement gives a natural way to traverse all lines of a file, as well as to apply a search process:

```
with open ('my_file_with_data.txt', 'r') as f:
    for line in f:
        process line
```

In this scheme we open a file named `my_file_with_data.txt` in reading mode and it is linked to the file type variable `f`. The `for` statement automatically traverses all file lines in such a way that, at each iteration, variable `line` is assigned the value of the string corresponding to the next line of file `f` (remember that STF contain characters): at the first iteration, variable `line` is a string with the first line, at the second iteration, variable `line` is a string with the second line, and so on. Variable `line` includes also the ending line feed character. The `for` statement stops automatically when reaches the end of file.

Processing a line involves a **decoding** process in order to obtain the correct data from it. Let's show this decoding process with an example. Let's suppose that file `my_file_with_data.txt` stores the different products of a company, one in each line of this file with the following data: product code (`str`), price (`float`) and number of stored unities (`int`); these three data items are separated by a hyphen. For a product with code 'DR45', a price of 56.75 and a number of unities of 200, the `str` variable `line` will have the following value:

```
'DR45-56.75-200\n'
```

Decoding this string means obtaining the corresponding data items in their corresponding types: product code (string), price (float) and number of stored unities (int). This process of decoding can be decomposed in three main steps:

- *cleaning*: extra characters at the beginning or at the end, as the line feed character are removed. This can be done in several ways:
  - using the `str` method `strip`. This method returns a copy of the string with the specified leading and trailing characters removed. If characters are not specified, it removes whitespace characters.
  - `line = line[:-1]`: removes the last character
  - `line = line.replace('\n','')`: removes all line feed characters
- *splitting*: the string is split into its different data items by using the `str` method `split`. If the data items are separated by a character different from a whitespace, then this character is used to split the given string (using the general behavior of method `split`). Conversely, if the data items are separated by one or more whitespace characters, method `split` is used without any explicitly given separator and it splits the string using the whitespace character and besides it removes the ending line feed.
- *decoding*: data items are converted to suitable types. Functions `int`, `float` and `eval` will convert the strings resulting from the previous steps into integer, float or Boolean, respectively.

Let's show again the value of variable `line`:

```
'DR45-56.75-200\n'
```

and the application of the three previous steps:

```
>>> line = line.strip()        # cleaning
>>> line
'DR45-56.75-200'
>>> linfo = line.split('-')    # splitting
>>> linfo
['DR45', '56.75', '200']
>>> code = linfo[0]            # strings needn't to be decoded
>>> price = float(linfo[1])    # decoding
>>> unities = int(linfo[2])    # decoding
>>> code, price, unities
('DR45', 56.75, 200)
```

If the data items in variable `line` were separated by whitespace characters instead of hyphen characters, `'DR45 56.75 200\n'`, the steps would be:

```
>>> linfo = line.split()      # cleaning and splitting
>>> codi = linfo[0]
>>> price = float(linfo[1])   # decoding
>>> unities = int(linfo[2])   # decoding
```

On the other hand, when we write lines into a file we will have to do the inverse process of **encoding** all the information into a string and adding a line feed character at the end of each line. In this case function `str` will be useful to convert from integer, float or Boolean to a string.

Exercise

Write the function `storage_amount` that takes two strings with the name of two files. The first name corresponds to an existing file that contains data for the different products of a company, one in each line of this file, with the following data items: product code (`str`), price (euros, `float`) and number of stored unities (`int`); these three data items are separated by a hyphen. The second name corresponds to a new file that has to be created by this function and that will contain in each line two data items, separated by a colon, for each product in the first file and in the same order: product code (string) and amount stored (euros, `float`): price multiplied by the number of unities. Moreover this function will return the total amount of the store in euros (`float`). An example of the first existing file, is a file named `'stock2018.txt'` with the following content:

```
FG89-123.5-25
HJ96-205.0-15
DR45-56.75-200
XZ65-312.5-14
TY76-124.0-89
TH89-89.5-250
HJ56-199.0-85
AH76-412.5-20
MV45-389.0-24
TY89-72.25-123
```

The following call:

```
storage_amount('stock2018.txt', 'stock2018total.txt')
```

returns a value of `98686.25` and creates a new file with name `'stock2018total.txt'` and with the following content:

```
FG89:3087.5
HJ96:3075.0
DR45:11350.0
XZ65:4375.0
TY76:11036.0
TH89:22375.0
HJ56:16915.0
```

```
AH76:8250.0
MV45:9336.0
TY89:8886.75
```

Design

```
def storage_amount (filename1, filename2):
    total = 0
    with open(filename1, 'r') as f1, open(filename2, 'w') as f2
       :
        for line in f1:
            line = line.strip()
            linfo = line.split('-')
            code = linfo[0]
            price = float(linfo[1])
            unities = int(linfo[2])
            total_product = price * unities
            total = total + total_product
            f2.write(code + ':' + str(total_product)+ '\n')
    return total
```

11 Iteration: statement `while`

11.1 Iteration: statements `for` and `while`

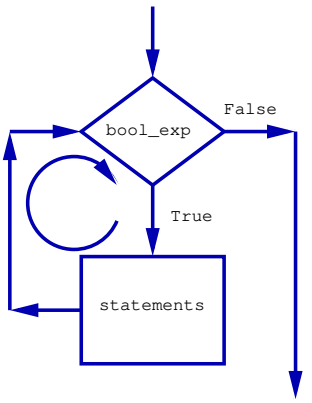
So far we have used iteration to traverse data structures as strings, lists, tuples and dictionaries as well as files. We have used the `for` statement, specifically designed for these cases. However not all problems requiring iteration can be solved with this statement.

The `for` statement does iteration in a controlled and secure way: it traverses all the structure elements, from the first to the last one:

```
for element in structure:
    process element
```

On the other hand, the `while` statement is the general statement for iteration allowing to design any kind of iteration. Below, the syntax for the `while` statement and its flowchart are shown:

```
while bool_exp:
   statements
```



First the Boolean expression is evaluated. If it evaluates to `True` the code block `statements1` is executed and then the Boolean expression is evaluated again and so on. When the Boolean expression evaluates to `False`, then the iteration finishes and execution resumes at the code following the `while` statement.

11.2 Iteration design using the `while` statement

The `while` statement is the general statement for iteration but it is not as secure and controlled as the `for` statement. When using the `while` statement we must watch for the following errors:

- do not process all the elements that have to be processed
- the Boolean condition evaluates always to `True`: **endless loop**

Example:

```
a = 5
while a < 10:
   a = a - 1
```

This is an example of an endless loop. The program flow enters in the loop and never comes out of it.

To debug `while` statements the trial and error technique is insufficient and we need a design methodology to guarantee code correctness.

Since data managed by iteration has a sequence structure, the design methodology consists in identifying and characterizing this sequence and then applying a traversal or searching scheme.

11.3 Sequence identifying and characterizing

A sequence is a data set of any type arranged linearly. More formally, a sequence is a data set characterized by the following three properties:

- there is always the **first element** of the sequence
- given a sequence element, we can obtain the **next element**
- a sequence is finite. There exists a property met by all the sequence elements and not met by any element that does not belong to the sequence: **ending property**

Identifying a sequence consists in determining the type of its elements and characterizing a sequence consists in finding its first element, its next element and ending property.

Examples:

- the sequence of multiples of `3` positive and inferiors to `100` is a sequence of integers. Characterization: the first element of this sequence is `3`, the next element is obtained by adding `3` to the previous one and all the elements must be less than `100`.
- the sequence of powers of `2` inferiors to `2000`. Characterization: the first element of this sequence is `1`, the next one is obtained multiplying the previous one by `2` and all the elements have to be inferior to `2000`.
- the sequence of points of the mathematical function $y = f(x) = x^2 + 4x + 2$ in the interval $[-5, 5]$ and with one unit abscissa increments is a sequence of points that can be represented by two integers (x, y). The first element is $(x, y) = (-5, f(-5))$; to obtain the next element, we must add `1` to the previous abscissa, x, and then compute $y = f(x)$. The ending property is met when the abscissa x is greater than 5

11.4 `while` iteration and sequences

Any problem that requires an iteration has an associated sequence and in the iteration design intervene three elements: the statements before the `while` statement, the while block statements and the Boolean expression that controls the `while` statement. These elements are related with the associated sequence characterization.

```
statements before the while statement
while bool_exp:
    while block statements
```

Then, the design methodology is based on the following points:

- the statements before the while statement are related with the first element of the sequence
- the while block statements are related with the next property
- the Boolean expression that controls the while statement is related with the ending property

Therefore to design a correct while iteration, we recommend to follow the next steps:

1. Identify the associated sequence
2. Characterize the associated sequence
3. Identify scheme: traversal or searching
4. Iteration design

11.5 Traversal scheme

The traversal scheme with the `while` statement is the following:

```
get first element
while not ending property:
    process element
    get next element
```

The statements before the `while` statement include getting the first element of the sequence. The last statement in the while block obtains the next element of the sequence. The ending property is a Boolean expression which evaluates to `False` for all elements of the sequence and `True` otherwise.

11.6 Searching scheme

The searching scheme with the `while` statement is the following:

```
found = False
get first element
while not ending property:
    found = search_condition (current_element)
    if found:
        break
    get next element
if found:
    statements block for case found
else:
    statements block for case not found
```

11.7 Structure traversal and searching with `while`

All the data structures seen so far (strings, lists, tuples dictionaries and files) have characterization mechanisms that the `for` statement uses automatically.

In general, it is not necessary to use the statement `while` to traverse these structures but, except dictionaries, it can also be used.

All sequence data structures, strings, lists and tuples, can be traversed with the `while` statement using the indexes sequence as the associated sequence. See exercise 4.

As dictionaries do not have an index, they can only be traversed with the `for` statement.

Files can also be traversed with the `while` statement and the associated sequence is the sequence of the file lines. The `readline` method gets the first line as well as any other line and the ending condition is met when an empty line (string) is found:

```
line = f.readline()
while line != '':
    process line
    line = f.readline()
```

11.8 Iterations counter

When certain sequences that often correspond to mathematical sequences are characterized, determining the first and next element can be obtained directly from the mathematical sequence definition. However, in some cases it is difficult to determine the ending property if, for example, this mathematical sequence has a cyclical behavior. In order to prevent the function from remaining in an endless loop, we can use the **iteration counter** technique. This technique consists in counting the iterations and leaving the loop either when the desired condition is met or when a certain number of iterations is exceeded. Example:

Let a and b be two integers, we define the following mathematical sequence:

$x_1 = a$

$$x_{i+1} = \begin{cases} x_i/2, & \text{if } x_i \text{ is even} \\ bx_i + 1 + 1, & \text{if } x_i \text{ is odd} \end{cases}$$

When $a = 4$ and $b = 4$ the elements of this mathematical sequence are:

$4, 2, 1, 5, 21, 85, 341, 1365, 5461, 21845, \ldots$

This mathematical sequence decreases first up to element 5 and then it is monotonically increasing.

However, when $a = 1$ and $b = 3$ this mathematical sequence is cyclic:

$1, 4, 2, 1, 4, 2, 1 \ldots$

In the first case an ending property can be defined with a maximum value: all the generated elements have to be less than this value.

In the second case we need an iteration counter. See exercise 5.

11.9 Exercises

1. Write the function `sum_mult` that takes an integer `n`, and returns the sum of all multiples of `3` positive and less than `n`. Example:

```
>>> sum_mult(10)
18
>>> sum_mult(30)
135
>>> sum_mult(100)
1683
```

```
def sum_mult (n):
    summ = 0
    mult = 3
    while mult < n:
        summ = summ + mult
        mult = mult + 3
    return summ
```

We have seen the characterization of the corresponding sequence in Section 11.3. We apply a traversal scheme with synthesis (summation).

Alternatively, we can use the `for` statement with function `range`:

```
def sum_mult (n):
    summ = 0
    for mult in range(3, n, 3):
        summ = summ + mult
```

```
    return summ
```

2. Write the function `pow_two` that takes an integer `x` less than 10 and another integer `n`, and returns the number of powers of `2` less than `n` which include the digit `x`. Example:

```
>>> pow_two(2, 1000)
5
>>> pow_two(3, 1000)
1
>>> pow_two(7, 10000)
0
>>> pow_two(7, 100000)
1
```

```
def pow_two (x, n):
    counter = 0
    pow2 = 1
    while pow2 < n:
        if str(x) in str(pow2):
            counter = counter + 1
        pow2 = pow2 * 2
    return counter
```

We have seen the characterization of the corresponding sequence in Section 11.3. We apply a traversal scheme with synthesis (counter).

We cannot use the `for` statement to design this function because the next element is obtained by a geometric progression. Note that the range function can only build arithmetic progressions.

3. We have sampled a sequence of points of the mathematical function $y = f(x) = x^2+4x+2$ in the interval $[a, b]$ and with d abscissa increments. Write the function `posneg` that takes the values `a, b, d` (integers), and returns two integers corresponding, respectively, to the number of sampled points with positive (or zero) and negative ordinate. Example:

```
>>> posneg(-5, 5, 1)
(8, 3)
>>> posneg(-5, 5, 2)
(4, 2)
>>> posneg(-4, 4, 2)
(4, 1)
```

We have seen the characterization of the corresponding sequence in Section 11.3. We apply a traversal scheme with synthesis (counter).

```
def posneg (a, b, d):
    npos = 0
    nneg = 0
    # primer element de la seqüència
    x = a
    while x <= b:
        y = x**2 + 4*x + 2
        if y >= 0:
            npos = npos + 1
        else:
            nneg = nneg + 1
        # següent element de la seqüència
        x = x + d
    return npos, nneg
```

4. We represent a closed interval with a tuple with two elements representing its end points. Write the function `total` that takes a list of intervals, and returns the sum of widths of all intervals. Write three version of this function: one with the `for` statement traversing the intervals directly; another with the `for` statement traversing the intervals indirectly through indexes, and the last one using the `while` statement. Example:

```
>>> lintervals = [(-3, 3), (1.2, 4.8), (-56, 24),
...  (-20, -10)]
>>> total (lintervals)
99.6
```

```
def total_1 (lintervals):
    tot = 0
    for int in lintervals:
        tot = tot + (int[1]-int[0])
    return tot

def total_2 (lintervals):
    tot = 0
    for i in range(len(lintervals)):
        tot = tot + (lintervals[i][1]-lintervals[i][0])
    return tot

def total_3 (lintervals):
    tot = 0
    i = 0
    while i < len(lintervals):
        tot = tot + (lintervals[i][1]-lintervals[i][0])
        i = i +1
    return tot
```

5. Given the two integers a and b, we define the following mathematical sequence:

   $x_1 = a$

   $$x_{i+1} = \begin{cases} x_i/2, & \text{if } x_i \text{ is even} \\ bx_i + 1 + 1, & \text{if } x_i \text{ is odd} \end{cases}$$

   Write the function `math_seq` that takes four positive integers `a, b, end` and `ni`, and computes the average of the terms of the previous sequence that are less than `end`. Parameter `ni` stands for the maximum number of allowed iterations. Function `math_seq` returns the computed average and the number of computed terms. Example:

```
>>> (aver, ni) = math_seq(4, 4, 50, 10000)
>>> round(aver, 2), ni
(6.6, 5)
>>> (aver, ni) = math_seq(2, 2, 50, 10000)
>>> round(aver, 2), ni
(9.83, 6)
>>> (aver, ni) = math_seq(3, 1, 50, 10000)
>>> round(aver, 2), ni
(1.5, 10000)
>>> (aver, ni) = math_seq(1, 3, 50, 10000)
>>> round(aver, 2), ni
(2.33, 10000)
>>> (aver, ni) = math_seq(64, 5, 100, 10000)
>>> round(aver, 2), ni
(5.72, 10000)
```

```
def math_seq (a, b, end, ni):
    summ = 0
    n = 0
    x = a
    while  x < end and n < ni:
        summ = summ + x
        n = n + 1
        if x%2 == 0:
            x = x//2
        else:
            x = b*x + 1
    return summ/n, n
```

   Variable `n` is a counter both for the number of terms and iterations.

12 pandas library

12.1 Introduction

Previous note: This chapter is an introduction to the possibilities of the pandas library. It is essential to always consult the documentation of the pandas library before using a method: there we will find the complete and updated specification, as well as other methods that are not introduced in this chapter.

pandas is an open source library providing data structures and data analysis tools for the Python programming language.

To use pandas we must download the library from:

https://pandas.pydata.org/

and import it:

```
import pandas as pd
```

pandas offers the following data structures (types, classes):

- `Series`: 1D indexed table
- `DataFrame`: 2D indexed table

pandas classes are implemented over classes in module `numpy`, in particular class `ndarray` which represents a multi-dimensional table.

Classes `Series` and `DataFrame` are related and share both attributes and methods. Moreover both rows and columns of a `DataFrame` are type `Series` objects and some `DataFrame` methods return `Series`. A `DataFrame` is a container of `Series` and a `Series` is a container of scalar values. Both data structures allow to look up, update, insert and remove items with a syntax similar to `dictionaries`.

`pandas` objects are mutable: we can modify their values. There are modifier and non-modifier methods and several methods can act in both ways by defining the Boolean optional parameter `inplace`: when `inplace = True` the method acts as a modifier and modifies the parameter `Series` or `DataFrame`, and when `inplace = False` it doesn't modify the parameter `Series` or `DataFrame` and it returns a new `Series` or `DataFrame` instead.

Real datasets often have the problem of missing data and databases must deal with it. `pandas` offers a standard marker for this specific data, `NaN`, which means *not a number*.

12.2 The `Series` class

The `Series` class represents 1D labeled tables that are containers of data of any other types: integer, float, string, ...

To define a `Series` we must create an object of this type with the following method:

```
class pandas.Series(data=None, index=None, dtype=None,
                    name=None, copy=False, fastpath=False)

data: array-like, Iterable, dict, or scalar value
      Contains data stored in Series.

index: array-like or Index (1d)

dtype: str, numpy.dtype, or ExtensionDtype, optional

name: str, optional
      The name to give to the Series.

copy: bool, default False
      Copy input data.
```

In the following example, we create a `Series` from a list:

```
>>> s1 = pd.Series([28, 34, 59, 91, 3])
>>> s1
0    28
1    34
2    59
3    91
4     3
dtype: int64
```

The class `Series` has the attributes `index` and `values`. It also has attribute `size` which corresponds to the number of items of the `Series`. Example:

```
>>> s1.axes
[RangeIndex(start=0, stop=5, step=1)]
>>> s1.index
RangeIndex(start=0, stop=5, step=1)
>>> s1.values
array([28, 34, 59, 91,  3])
>>> s1.size    # len(s1)
5
```

The `index` attribute can be redefined:

```
>>> s1.index = ['Joe', 'Liu', 'Pol', 'Jan', 'Iu']
>>> s1
Joe    28
Liu    34
Pol    59
Jan    91
Iu      3
dtype: int64
```

`Series` objects are 1D tables and, therefore, they can be converted to other 1D structures as lists, tuples and dictionaries. Examples:

```
>>> list(s1)
[28, 34, 59, 91, 3]
>>> tuple(s1)
(28, 34, 59, 91, 3)
>>> dict(s1)
{'Joe': 28, 'Liu': 34, 'Pol': 59, 'Jan': 91, 'Iu': 3}
```

We can apply indexing and slicing operators to `Series` using both the index or an integer with the position, with attributes `loc[]` and `iloc[]`. Example:

```
>>> s1['Joe']       # indexing by index
28
>>> s1.loc['Joe']   # indexing by index
28
>>> s1.iloc[3]      # indexing by position
91
>>> s1['Liu':'Iu'] # slicing
Liu    34
Pol    59
Jan    91
Iu      3
dtype: int64
```

Syntax to update, insert or delete an item is similar to dictionaries. Example:

```
>>> s1['Pol'] = 19   # update
>>> s1['Jim'] = 25   # insert
>>> del s1['Liu']    # delete
>>> s1
Joe    28
Pol    19
Jan    91
Iu      3
Jim    25
dtype: int64
```

We can create another `Series` by a selection process using a Boolean expression. It can be simple or compound (with Boolean operators). In this second case each simple Boolean expression must be between parenthesis and the Boolean operations are represented with the following symbols: & (`and`), | (`or`) and $\sim$ (`not`). Example:

```
>>> s1[s1<20]              # simple Boolean expression
Pol    19
Iu      3
dtype: int64
```

```
>>> s1[(s1>18) & (s1<50)] # compound Boolean expression
Joe     28
Pol     19
Jim     25
dtype: int64
```

Basic operators (+, -, *, /, >, >=, <, <=, ==, !=) are defined for `Series`. In order to operate two `Series` they must have the same index. Example:

```
>>> s1+10
Joe     38
Pol     29
Jan    101
Iu      13
Jim     35
dtype: int64

>>> s1+s1
Joe     56
Pol     38
Jan    182
Iu       6
Jim     50
dtype: int64

>>> s1>=18
Joe     True
Pol     True
Jan     True
Iu     False
Jim     True
dtype: bool
```

`Series` have many methods and most of them are shared by `DataFrame` class:

- `head` (`tail`): shows the first (last) n items
- `sum`, `mean`, `std` computes the summation, arithmetic mean and standard deviation, respectively
- `max` and `min` computes the maximum and minimum value respectively
- `idxmax` and `idxmin` computes the index for maximum and minimum value respectively

Examples (the example `Series s1` is included for readiness purposes):

```
>>> s1
Joe     28
```

```
Pol     19
Jan     91
Iu       3
Jim     25
dtype: int64
```

```
>>> s1.head(2)
Joe     28
Pol     19
dtype: int64

>>> s1.tail(2)
Iu       3
Jim     25
dtype: int64

>>> s1.sum()
166
>>> s1.max()
91
>>> s1.min()
3
>>> s1.idxmax()
'Jan'
>>> s1.mean()
33.2
>>> s1.std()
33.72239611889997
```

Besides these `Series` methods, `pandas` also support functions `sum`, `max`, `min`. However it is recommended to use always the methods as they are specifically defined by `Series` (and `DataFrames`) and they can deal with missing data.

`Series` are iterable:

```
>>> for e in s1:
...     print(e, end='-')
28-19-91-3-25-
```

In the following example we define a `Series` from a dictionary and then we redefine the index. We can observe that the `Series` is incomplete: there are missing data and `pandas` represents these data with the value `NaN` (not a number).

As we will see in Section 12.5, `pandas` can deal with missing data: methods `sum`, `min` and `max` deal with missing data. But functions `sum`, `min` and `max` of the standard library do not.

```
>>> s1 = pd.Series({'a': 0.1, 'b': 1.0, 'c': 2.4}, index=['b',
   'c', 'd', 'a'])
>>> s1
```

```
b    1.0
c    2.4
d    NaN
a    0.1
dtype: float64
>>> s1.sum()
3.5
>>> sum(s1)
nan
```

12.3 The `DataFrame` class

Class `DataFrame` represents 2D indexed and labeled tables which are containers of `Series` which are containers of data of any other types: integer, float, string, ...

To define a `DataFrame` we must create an object of this type with the following method:

```
class pandas.DataFrame(data=None, index=None, columns=None,
                       dtype=None, copy=False)

data : numpy ndarray (structured or homogeneous), dict, or
       DataFrame. Dict can contain Series, arrays, constants,
       or list-like objects

index : Index or array-like.
        Index for resulting frame, default: np.arange(n)

columns : Index or array-like.
          Column labels to use for resulting frame.
          Default: np.arange(n) if no column labels are
             provided

dtype : dtype, default None.
        Data type to force. Only a single dtype is allowed.

copy : boolean, default False. Copy data from inputs.
```

In the following example we create a `DataFrame` by defining the `Series` corresponding to each of its columns:

```
>>> df = pd.DataFrame()
>>> df['Name'] = ['Joe', 'Ann', 'Pol', 'Mar', 'Iu']
>>> df['Age'] = [18, 19, 21, 18, 19]
>>> df['Mark1'] = [4.1, 8.3, 5.9, 9.2, 7.4]
>>> df['Mark2'] = [5.1, 3.3, 6.9, 9.8, 8.4]
>>> df
  Name  Age  Mark1  Mark2
```

```
0   Joe    18     4.1     5.1
1   Ann    19     8.3     3.3
2   Pol    21     5.9     6.9
3   Mar    18     9.2     9.8
4    Iu    19     7.4     8.4
```

These are the basic attributes of class `DataFrame` and the result that returns method `info`:

```
>>> df.axes
[RangeIndex(start=0, stop=5, step=1), Index(['Name', 'Age', '
   Mark1', 'Mark2'], dtype='object')]
>>> df.index
RangeIndex(start=0, stop=5, step=1)
>>> df.columns
Index(['Name', 'Age', 'Mark1', 'Mark2'], dtype='object')
>>> len(df)      # len(df.index)
5
>>> len(df.columns)
4
>>> df.shape
(5, 4)
>>> df.size
20
>>> df.values
array([['Joe', 18, 4.1, 5.1],
       ['Ann', 19, 8.3, 3.3],
       ['Pol', 21, 5.9, 6.9],
       ['Mar', 18, 9.2, 9.8],
       ['Iu', 19, 7.4, 8.4]], dtype=object)

>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    5 non-null      object
 1   Age     5 non-null      int64
 2   Mark1   5 non-null      float64
 3   Mark2   5 non-null      float64
dtypes: float64(2), int64(1), object(1)
memory usage: 288.0+ bytes
```

Method `set_index` allows to define a column as the index. This method has the optional parameter `inplace` that indicates whether the method acts as a modifier (`True`) o not (`False`, default value).

```
>>> df.set_index('Name', inplace = True)
```

```
>>> df
      Age  Mark1  Mark2
Name
Joe    18    4.1    5.1
Ann    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8
Iu     19    7.4    8.4
```

In this example method `set_index` modifies the `DataFrame` `df` and returns `None`. If the statement was `dfnew = df.set_index('Name')` then the method would not modify `df` and would return a new `DataFrame`, that would be assigned to variable `dfnew`.

12.3.1 `DataFrame`: indexing

To select a row, a column or an item from a `DataFrame`, we can use attributes `loc` and `iloc`. Attribute `loc` uses labels, `df.loc[row, column]`, while attribute `iloc` uses an integer index corresponding to the position, `df.iloc[idx_row, idx_column]`.

To select a whole row, only the row index must be used: `df.loc[row]`, `df.iloc[idx_row]`. In this case this attribute gives the corresponding row which is a `Series` object.

Examples:

```
>>> df.loc['Mar']  # row selection with label: a Series
Age      18.0
Mark1      9.2
Mark2      9.8
Name: Mar, dtype: float64

>>> df.loc['Ann', 'Mark1']   # item selection with label
8.3
>>> df.iloc[2]
Age      21.0
Mark1      5.9
Mark2      6.9
Name: Pol, dtype: float64

>>> df.iloc[2, 2]            # item selection with position
6.9
```

To select a column, the `DataFrame` can be indexed directly, `df[column]` and this expression corresponds to a `Series` object. Example:

```
>>> df['Age']       # column selection: a Series
Name
Joe    18
Ann    19
```

```
Pol     21
Mar     18
Iu      19
Name: Age, dtype: int64
```

We can use this syntax to select an item:

```
>>> df['Age']['Mar']
18
```

However, it is less efficient than using attributes `loc` and `iloc` because with this syntax the selection is performed in 2 steps: first the column is selected giving a `Series` and then the item is selected from it. Using attributes `loc` and `iloc` item selection is performed directly in one step.

Rows and columns are `Series` and therefore can be converted to lineal data structures.

```
>>> tuple(df.loc['Mar'])
(18.0, 9.2, 9.8)
>>> dict(df.iloc[2])
{'Age': 21.0, 'Mark1': 5.9, 'Mark2': 6.9}
>>> list(df['Age'])
[18, 19, 21, 18, 19]
>>> dict(df['Age'])
{'Joe': 18, 'Ann': 19, 'Pol': 21, 'Mar': 18, 'Iu': 19}
```

12.3.2 `DataFrame`: subdataframes

A subdataframe is a subset of a `DataFrame`, i.e., a set of several rows and columns. The same attributes `loc` and `iloc` and direct column indexing are also used to define subdataframes. We can use lists and the slicing mechanism both for rows and for columns.

Using lists: `df.loc[rows list, columns list]` (same syntax for attribute `iloc`). Examples:

```
>>> df.loc[['Joe', 'Pol'], ['Age', 'Mark2']]
      Age  Mark2
Name
Joe    18    5.1
Pol    21    6.9

>>> df.iloc[[0, 2], [0, 2]]
      Age  Mark2
Name
Joe    18    5.1
Pol    21    6.9
```

Using slicing: `df.iloc[idx_row_min:idx_row_max, idx_col_min:idx_col_max]`. Examples:

```
>>> df.loc['Ann':'Mar']      # rows between 'Ann' and 'Mar'
```

```
      Age  Mark1  Mark2
Name
Ann    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8

>>> df.iloc[::2]            # rows in even position
      Age  Mark1  Mark2
Name
Joe    18    4.1    5.1
Pol    21    5.9    6.9
Iu     19    7.4    8.4

>>> df.loc['Ann':'Mar', 'Age':'Mark2']
      Age  Mark1  Mark2
Name
Ann    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8

# row selection by slicing and column selection by list
>>> df.iloc[1:4, [0, 2]]
      Age  Mark2
Name
Ann    19    3.3
Pol    21    6.9
Mar    18    9.8
```

`DataFrame` direct indexing only admits column selection using a list (slicing cannot be used). Example:

```
>>> df[['Age', 'Mark2']]['Ann':'Mar']
      Age  Mark2
Name
Ann    19    3.3
Pol    21    6.9
Mar    18    9.8
```

12.3.3 `DataFrame`: update, insertion and removal

The previous syntax used to index a `DataFrame` is also used to update it.

```
>>> df.loc['Iu', 'Age'] = 20  # item update
>>> df['Mark1']['Pol'] = 6.5  # item update
>>> df
      Age  Mark1  Mark2
```

```
Name
Joe     18     4.1     5.1
Ann     19     8.3     3.3
Pol     21     6.5     6.9
Mar     18     9.2     9.8
Iu      20     7.4     8.4

>>> df.loc['Iu'] = [21, 8.4, 7.5]  # update row
>>> df.loc['Jim'] = [20, 5.7, 8.2] # insert row for Jim
>>> df =df.astype({'Age':'int64'}) # restore 'Age' to int

# insert column
>>> df['Mark3'] = [4.4, 6.8, 9.2, 8.7, 6.5, 7.2]
>>> df.head(2)
      Age   Mark1   Mark2   Mark3
Name
Joe     18     4.1     5.1     4.4
Ann     19     8.3     3.3     6.8
```

When we insert the row corresponding to Jim, the value in column 'Age' changes to a float. In order to restore it at an integer, we use function `astype`.

Moreover the `del` statement already used for type `list` and `dict` objects can also be applied to `DataFrame` columns. Examples:

```
>>> del df['Mark2']   # removes column Mark2
>>> df.head(2)
      Age   Mark1   Mark3
Name
Joe     18     4.1     4.4
Ann     19     8.3     6.8
```

Method `drop` allows to remove in a more general way. It has the optional parameter `inplace`, already seen. It also has the optional parameter `axis`: `axis = 0` (default value) stands for rows while `axis = 1` refers to columns. In the following example, we insert again column `Mark2` and then we remove it with method `drop`:

```
>>> df['Mark2'] = [5.1, 3.3, 6.9, 9.8, 8.4, 8.2]
>>> df.drop('Mark2', axis = 1, inplace = True) # column removal
>>> df.drop('Mar', inplace =True)              # row removal
>>> df
      Age   Mark1   Mark3
Name
Joe     18     4.1     4.4
Ann     19     8.3     6.8
Pol     21     6.5     9.2
Iu      21     8.4     6.5
Jim     20     5.7     7.2
```

Method `insert` inserts a column in a given position. It is always a modifier method:

```
>>> df.insert(2, 'Mark2', [5.1, 3.3, 6.9, 8.4, 8.2])
>>> df.head(2)
      Age   Mark1   Mark2   Mark3
Name
Joe    18     4.1     5.1     4.4
Ann    19     8.3     3.3     6.8
```

Method `replace` behaves as expected (as with strings). Example:

```
>>> df.replace(18, 20)
      Age   Mark1   Mark2   Mark3
Name
Joe    20     4.1     5.1     4.4
Ann    19     8.3     3.3     6.8
Pol    21     6.5     6.9     9.2
Iu     21     8.4     8.4     6.5
Jim    20     5.7     8.2     7.2
```

As we have seen, most basic operators (+, -, *, /, >, >=, <, <=, ==, !=) are defined for `Series` objects and, therefore, they can be applied to `DataFrame` rows and columns. In the next example, we restore the initial dataframe by deleting column `Mark3` and we apply some of these basic operations:

```
>>> del df['Mark3']           # remove column 'Mark3'
>>> df['Mark'] = df['Mark1']*0.2 + df['Mark2']*0.8 # new column
>>> df
      Age   Mark1   Mark2   Mark
Name
Joe    18     4.1     5.1   4.90
Ann    19     8.3     3.3   4.30
Pol    21     6.5     6.9   6.82
Iu     21     8.4     8.4   8.40
Jim    20     5.7     8.2   7.70

>>> df.loc['Pau'] = df.loc['Pol'] + df.loc['Jim']*2  # new row
>>> df
       Age   Mark1   Mark2    Mark
Name
Joe   18.0     4.1     5.1    4.90
Ann   19.0     8.3     3.3    4.30
Pol   21.0     6.5     6.9    6.82
Iu    21.0     8.4     8.4    8.40
Jim   20.0     5.7     8.2    7.70
Pau   61.0    17.9    23.3   22.22
```

12.3.4 `DataFrame`: simple and advanced selection

As with `Series` we can use a simple or compound Boolean expression to perform a selection. Boolean operators are expressed by symbols: & (`and`), | (`or`), ∼ (`not`). Remember also that when we have a compound Boolean expression, its simple Boolean expressions must be parenthesized. Examples (first, we restore the initial dataframe):

```
>>> del df['Mark']
>>> df.drop('Pau', inplace =True)
>>> df =df.astype({'Age':'int64'})

### simple selection
>>> df[df['Age'] <= 19]
      Age  Mark1  Mark2
Name
Joe    18    4.1    5.1
Ann    19    8.3    3.3

####  advanced selection
>>> df[(df['Mark1'] > 7.5) | (df['Mark2'] > 7.5)]  # or
      Age  Mark1  Mark2
Name
Ann    19    8.3    3.3
Iu     21    8.4    8.4
Jim    20    5.7    8.2

>>> df[(df['Mark1'] > 7.5) & (df['Mark2'] > 7.5)]  # and
      Age  Mark1  Mark2
Name
Iu     21    8.4    8.4
```

12.3.5 `DataFrame`: basic methods

We copy here the initial `DataFrame` example:

```
      Age  Mark1  Mark2
Name
Joe    18    4.1    5.1
Ann    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8
Iu     19    7.4    8.4
```

Methods `head` and `tail` return a `DataFrame` with the first and last rows of the given `DataFrame`, respectively:

```
>>> df.head(2)
```

```
      Age   Mark1   Mark2
Name
Joe    18     4.1     5.1
Ann    19     8.3     3.3

>>> df.tail(2)
      Age   Mark1   Mark2
Name
Mar    18     9.2     9.8
Iu     19     7.4     8.4
```

Methods `sum`, `mean`, `std`, `max`, `min`, `idxmax` and `idxmin` have the same behavior as those with the same name for `Series`. However, they apply to all the `DataFrame` columns and then they return a `Series` object.

```
>>> df.sum()
Age       95.0
Mark1     34.9
Mark2     33.5
dtype: float64

>>> df.idxmax()
Age       Pol
Mark1     Mar
Mark2     Mar
dtype: object

>>> df.mean()
Age       19.00
Mark1      6.98
Mark2      6.70
dtype: float64
```

Method `describe` returns a `DataFrame` with several statistic parameters. Example:

```
>>> df.describe()
              Age      Mark1      Mark2
count    5.000000   5.000000   5.000000
mean    19.000000   6.980000   6.700000
std      1.224745   2.019158   2.581666
min     18.000000   4.100000   3.300000
25%     18.000000   5.900000   5.100000
50%     19.000000   7.400000   6.900000
75%     19.000000   8.300000   8.400000
max     21.000000   9.200000   9.800000
```

Method `reset_index` converts the index into a column and redefines it as an integer range. It has the optional parameter `inplace`. Example:

```
>>> df.reset_index()           # inplace = False (default)
  Name   Age   Mark1   Mark2
0  Joe    18     4.1     5.1
1  Ann    19     8.3     3.3
2  Pol    21     5.9     6.9
3  Mar    18     9.2     9.8
4   Iu    19     7.4     8.4
```

We can name the columns as well as the index with attributes `columns.name` and `index.name`. Method `rename` allows to change some column names and some index items names. Example:

```
### index and columns name
>>> df.columns.name = 'data'
>>> df.index.name = 'idx'

### rename columns and index
>>> df.rename(columns = {'Mark1': 'Midterm', 'Mark2': 'Final'},
    inplace = True)
>>> df.rename(index = {'Pol': 'Paul', 'Ann': 'Hanna'}, inplace
   = True)
>>> df
data    Age   Midterm   Final
idx
Joe      18       4.1     5.1
Hanna    19       8.3     3.3
Paul     21       5.9     6.9
Mar      18       9.2     9.8
Iu       19       7.4     8.4
```

Method `sort_values` is for sorting. Parameter `by` establishes the column used to sort. It has also the optional parameters `ascending` and `inplace`. Example:

```
### remove columns name and restore rows name
>>> df.columns.name = ''
>>> df.index.name = 'Name'

### method sort_values
>>> df.sort_values(by = 'Final')
        Age   Midterm   Final
Name
Hanna    19       8.3     3.3
Joe      18       4.1     5.1
Paul     21       5.9     6.9
Iu       19       7.4     8.4
Mar      18       9.2     9.8
```

Method `value_counts` returns a frequency table for all the values. Example:

```
>>> df['Age'].value_counts()
19    2
18    2
21    1
Name: Age, dtype: int64
```

Method `append` appends a `DataFrame` to another one. Example:

```
>>> dfa = pd.DataFrame()
>>> dfa['Name'] = ['Ona', 'Sira']
>>> dfa['Age'] = [21, 18]
>>> dfa['Midterm'] = [5.1, 3.3]
>>> dfa['Final'] = [4.1, 8.3]
>>> dfa.set_index('Name', inplace = True)
>>> dff = df.append(dfa)
>>> dff
       Age  Midterm  Final
Name
Joe     18      4.1    5.1
Hanna   19      8.3    3.3
Paul    21      5.9    6.9
Mar     18      9.2    9.8
Iu      19      7.4    8.4
Ona     21      5.1    4.1
Sira    18      3.3    8.3
```

Method `apply` allows to apply a function to a whole `DataFrame` or to a column. The applied function can be from a library (for example, `round`), a new defined function in the usual way (as function `new_mark` in the next example), or as an anonymous function (`lambda`). Examples:

```
>>> df.apply(round).head(3)
       Age  Midterm  Final
Name
Joe     18      4.0    5.0
Hanna   19      8.0    3.0
Paul    21      6.0    7.0

>>> df['Final'] = df['Final'].apply(round)
>>> def new_mark(m):
...     if m >= 5 and m <= 9.5:
...        m = m + 0.5
...     return m
>>> df['Midterm'] = df['Midterm'].apply(new_mark)
>>> df['Mark'] = df['Midterm']*0.5+df['Final']*0.5
>>> df['Mark'] = df['Mark'].apply(lambda x: round(x*10, 2))
>>> df.head(3)
       Age  Midterm  Final  Mark
Name
```

```
Joe      18       4.1       5   45.5
Hanna    19       8.8       3   59.0
Paul     21       6.4       7   67.0

>>> df.loc['Mar'] = df.loc['Mar'].apply(round)    # apply to a
   row
>>> df.tail(3)
      Age   Midterm   Final   Mark
Name
Paul    21       6.4       7   67.0
Mar     18      10.0      10   98.0
Iu      19       7.9       8   79.5
```

12.3.6 `DataFrame`: creation

We have already seen how to create a `DataFrame` by defining each column as a `Series`. There are other ways to create a `DataFrame` and two of the most usual are from a nested list and from a dictionary. Examples:

```
>>> ld = [['Joe', 18, 4.1, 5.1],
...       ['Ann', 19, 8.3, 3.3],
...       ['Pol', 21, 5.9, 6.9],
...       ['Mar', 18, 9.2, 9.8],
...       ['Iu', 19, 7.4, 8.4]]  # from a nested list
>>> idx = ['001', '002', '003', '004', '005']
>>> cols = ['Name', 'Age', 'Mark1', 'Mark2']
>>> df = pd.DataFrame(ld, index = idx, columns = cols)
>>> df
    Name  Age  Mark1  Mark2
001  Joe   18    4.1    5.1
002  Ann   19    8.3    3.3
003  Pol   21    5.9    6.9
004  Mar   18    9.2    9.8
005   Iu   19    7.4    8.4

# from a dictionary
>>> dd = {'Name': ['Joe', 'Ann', 'Pol', 'Mar', 'Iu'],
...       'Age': [18, 19, 21, 18, 19],
...       'Mark1': [4.1, 8.3, 5.9, 9.2, 7.4],
...       'Mark2': [5.1, 3.3, 6.9, 9.8, 8.4]}
>>> cols = ['Name', 'Age', 'Mark1', 'Mark2']
>>> df = pd.DataFrame(dd, columns = cols)
>>> df
  Name  Age  Mark1  Mark2
0  Joe   18    4.1    5.1
1  Ann   19    8.3    3.3
```

```
2  Pol   21    5.9    6.9
3  Mar   18    9.2    9.8
4   Iu   19    7.4    8.4
```

12.4 Input/output

pandas offers functions to read from and write to a file using several file formats. In this document only the file format `csv` (comma separated values) is considered.

Read function:

`df = pd.read_csv(filename, sep=';', usecols = ...)`

`sep` is an optional parameter whose default value is `','` (comma). Parameter `filename` is a string with the name of the file with the data we want to read. It can also be a url. Parameter `usecols` by default means that we take all the given columns, but we can perform a specific column selection using a list. Example:

```
>>> url='https://perso.telecom-paristech.fr/eagan/class/igr204/
   data/cereal.csv'
>>> df = pd.read_csv(url, sep=';', usecols=['name', 'calories',
    'sugars'])
```

Write function:

`df.to_csv(filename, sep = ',', index = False)`

Optional parameter `index` is `False` by default. When it is set to `True`, then it writes the index values.

12.5 Missing data

Databases are often incomplete and have missing data in some cells. It can occur for data of any type: integer, real, string, ... pandas can detect and deal with this deficiency. Missing data is referred to as NaN (not a number) or NA (not available).

A `Series` or `DataFrame` incomplete has cells with the NaN value:

```
  Name   Age  Mark1  Mark2
0  Joe  18.0    4.1    5.1
1  Ann  19.0    8.3    3.3
2  Pol  21.0    NaN    6.9
3  Mar   NaN    9.2    9.8
4   Iu  19.0    7.4    8.4
```

pandas methods can deal with missing data. Method `count` counts only those cells different from NaN and methods `sum`, `min`, etc., also ignore these cells. Example:

```
>>> dfn.count()
Name      5
```

```
Age       4
Mark1     4
Mark2     5
dtype: int64

>>> dfn['Age'].max()
21.0
>>> dfn['Mark1'].mean()
7.25
```

Method `isna` (or `isnull`) detects missing data: for each cell in a `DataFrame` it writes `True` if it is a NaN or `False` otherwise. Method `notna` does exactly the inverse process. Example:

```
>>> dfn.isna()
    Name    Age   Mark1   Mark2
0  False  False  False  False
1  False  False  False  False
2  False  False   True  False
3  False   True  False  False
4  False  False  False  False
```

Besides detecting missing data, pandas can also perform some sort of correction processes. Method `fillna` changes NaN values by a given value. Method `dropna` removes rows and columns with NaN values: when parameter `axis=0` (by default) it deletes rows and when `axis=1` deletes columns. Both methods have the parameter `inplace`. Example:

```
>>> dfn['Mark1'].fillna(0.0, inplace = True)
>>> dfn = dfn.dropna() # drop all rows with NaN
>>> dfn
  Name   Age  Mark1  Mark2
0  Joe  18.0    4.1    5.1
1  Ann  19.0    8.3    3.3
2  Pol  21.0    0.0    6.9
4   Iu  19.0    7.4    8.4
```

The `DataFrame` used in these examples has been created with the following statements:

```
>>> import numpy as np
>>> import pandas as pd
>>> dfn = pd.DataFrame()
>>> dfn['Name'] = ['Joe', 'Ann', 'Pol', 'Mar', 'Iu']
>>> dfn['Age'] = [18, 19, 21, np.nan, 19]
>>> dfn['Mark1'] = [4.1, 8.3, np.nan, 9.2, 7.4]
>>> dfn['Mark2'] = [5.1, 3.3, 6.9, 9.8, 8.4]
```

We observe two points. To define a value NaN we use the constant `np.nan` from the library `numpy` that we must import. Besides, column `Age` values must be integer, but as there is a NaN value, they are represented as float. In order to represent them with integers we can use method `astype`. Example:

```
>>> dfn = dfn.astype({'Age':'int64'})
>>> dfn
  Name  Age  Mark1  Mark2
0  Joe   18    4.1    5.1
1  Ann   19    8.3    3.3
2  Pol   21    0.0    6.9
4   Iu   19    7.4    8.4

>>> dfn['Age'].sum()
77
```

Another missing data feature of pandas is that when a `DataFrame` with missing data is created, pandas fills them automatically with NaN values.

12.6 `DataFrame`: groupby

The process referred as groupby for a database involves several steps:

- split data into groups using some criterion
- apply a function to each group
- combine the results in a new data structure

Example: in the next `DataFrame`, column `classG` is the class group for each student:

```
    Name  Age classG  Mark
0    Pol   23      A   5.6
1  Peter   22      A   3.5
2    Pau   23      B   2.1
3   John   22      A   6.7
4  Julie   22      C   8.7
5   Maia   21      A   4.5
6   Rosa   21      B   6.7
7  Alice   22      C   9.1
```

If we use the class group, `classG`, as the groupby criterion we obtain 3 groups, one for each class group. Then we can apply to each group an operation as, for example, to compute the average mark, and the result will be a `Series` with the average mark of each class group.

Method `groupby` performs a database distribution, grouping by one or more columns (`key`) and returns a `groupby` type object. To perform multiple groupings we must define the corresponding list of keys.

A `groupby` object is a tuple collection with format `(key, DataFrame)` and it is iterable.

We use the previous example and perform a grouping process using column `classG`:

```
>>> gb1 = dfstu.groupby('classG')
>>> type(gb1)
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
>>> len(gb1)
3
```

Observe that `gb1` is a `groupby` object with 3 groups. Next, these three groups are shown by iterating the tuples collection `(key, DataFrame)`:

```
>>> for key, df in gb1:
...     print(key)
...     print(df)
...
A
    Name  Age classG  Mark
0    Pol   23      A   5.6
1  Peter   22      A   3.5
3   John   22      A   6.7
5   Maia   21      A   4.5
B
   Name  Age classG  Mark
2   Pau   23      B   2.1
6  Rosa   21      B   6.7
C
    Name  Age classG  Mark
4  Julie   22      C   8.7
7  Alice   22      C   9.1
```

We can see on the upper-left corner the key corresponding to the class group ('A', 'B' or 'C') and below the corresponding `DataFrame`.

To access to a determined group we must use method `get_group` that given a key ('A', 'B' or 'C', in this example) returns its corresponding `DataFrame`. In the example below, we obtain the class group 'A' (a `DataFrame`), then we select column `Mark` and apply function `round` to it.

```
>>> gb1.get_group('A') # returns DataFrame of class group 'A'
    Name  Age classG  Mark
0    Pol   23      A   5.6
1  Peter   22      A   3.5
3   John   22      A   6.7
5   Maia   21      A   4.5

>>> gb1.get_group('A')['Mark'].apply(round)
0    6
1    4
3    7
5    4
Name: Mark, dtype: int64
```

After the subdivision step, next steps apply and combine can be performed. In the following example, we apply method `size` to each `DataFrame` of the `groupby` object and then we combine the results and get a `Series` with the size of each `DataFrame`:

```
>>> gb1.size()
classG
A    4
B    2
C    2
dtype: int64
```

We could do the same steps using methods `first` (the 1st row of each `DataFrame`), `last` (the last row), `nth` (the nth row), as well as the already seen methods `sum`, `min`, `idxmin`, `count`, `mean`, etc.

The next example applies method `first` to the previous `groupby` object and the result is a `DataFrame` with the 1st row of each `DataFrame` of the `groupby` object. When we apply a method as `sum`, only numeric columns are considered:

```
>>> gb1.first()                # equivalent to gb1.nth(0)
         Name  Age  Mark
classG
A         Pol   23   5.6
B         Pau   23   2.1
C       Julie   22   8.7

>>> gb1.sum()
        Age  Mark
classG
A        88  20.3
B        44   8.8
C        44  17.8
```

We can select a column of a `groupby` object and apply a method to this column. In the next example we compute the average mark of each class group:

```
>>> gb1['Mark'].mean()
classG
A    5.075
B    4.400
C    8.900
Name: Mark, dtype: float64
```

Expression `gb1.mean()['Mark']` is equivalent to expression (`gb1['Mark'].mean()`) but is less efficient because it first computes a `DataFrame` with the average of all the numeric columns and then selects column `Mark`.

Let's show another example. Given the `DataFrame` `dfstu` used in this section, subdivide by age and obtain for each age group the student with maximum mark.

We apply the following steps. First we use method `set_index` in order to define column `Name` as the index of the given `DataFrame`; then we subdivide using the age criterion; then we select column `Mark`, and finally we apply method `idxmax`. We have changed the index to the `Name` column in the first step in order that method `idxmax` gives us the student name instead of the initial numeric index.

```
>>> df1 = dfstu.set_index('Name')
>>> gb2 = df1.groupby('Age')
>>> for key, df in gb2:
...     print(key)
...     print(df)
...
21
      Age classG  Mark
Name
Maia   21      A   4.5
Rosa   21      B   6.7
22
       Age classG  Mark
Name
Peter   22      A   3.5
John    22      A   6.7
Julie   22      C   8.7
Alice   22      C   9.1
23
      Age classG  Mark
Name
Pol    23      A   5.6
Pau    23      B   2.1

>>> gb2['Mark'].idxmax()
Age
21     Rosa
22    Alice
23      Pol
Name: Mark, dtype: object
```

`groupby` objects have an attribute named `groups`. It is a dictionary in which keys are the `groupby` keys and values are each group index. Example:

```
>>> gb2.groups
{21: ['Maia', 'Rosa'], 22: ['Peter', 'John', 'Julie', 'Alice'],
    23: ['Pol', 'Pau']}
>>> gb2.groups.keys()
dict_keys([21, 22, 23])

>>> gb2.groups[22]
```

```
Index(['Peter', 'John', 'Julie', 'Alice'], dtype='object', name
   ='Name')
```

12.6.1 `DataFrame`: hierarchic groupby

We can use several subdivision criteria. In this case method `groupby` receives a list of keys as, for example, `groupby([key1, key2, ...])` and the multiple subdivision is performed first by `key1`, then `key2`, ...

In the next example we take the `DataFrame dstu` used in this section and perform a hierarchic subdivision first by class group and then by age. As there are 3 class groups and 3 age groups we would have potentially 9 groups, but only 6 of them are not null. For instance, the group corresponding to class group 'B' and age 22 is empty.

In this hierarchic grouping process, the `groupby` object is a collection of tuples with format `(multiplekey, DataFrame)` where `multiplekey` is the multiple key represented by a tuple with the used keys.

Example:

```
>>> for (key1, key2), df in gb3:
...     print(key1, key2)
...     print(df)
...
A 21
   Name  Age classG  Mark
5  Maia   21      A   4.5
A 22
    Name  Age classG  Mark
1  Peter   22      A   3.5
3   John   22      A   6.7
A 23
  Name  Age classG  Mark
0  Pol   23      A   5.6
B 21
   Name  Age classG  Mark
6  Rosa   21      B   6.7
B 23
  Name  Age classG  Mark
2  Pau   23      B   2.1
C 22
    Name  Age classG  Mark
4  Julie   22      C   8.7
7  Alice   22      C   9.1

>>> gb3.size()
classG  Age
```

```
A         21      1
          22      2
          23      1
B         21      1
          23      1
C         22      2
dtype: int64
```

Observe that `gb3.size()` is a multi-indexed `Series`