Functions and modules

1 Functions

https://docs.python.org/3/tutorial/controlflow.html#defining-functions

https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions

In order to define a function the programmer should use the **def** keywork:

```
>>> def funct_1():
...     print(12)
...
>>> funct_1()
12
>>>
```

The parameters are enclosed in parentheses and the instructions are inside a block.

1.1 Function calling

Functions are called as ….. regular functions

1.2 Arguments

Functions can receive arguments.

The programmer does not need to define the type of the arguments.

These type are checked in run-time.

```
>>> def adicao(a,b):
...         return a+b
...
>>> adicao(12, 13)
25
>>> print(adicao(12, 13))
25
>>> print(adicao("aaaa", "bbbb"))
aaaabbbb
>>> print(adicao("aaaa", 12))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 2, in adicao
TypeError: can only concatenate str (not "int") to str
```

1.3 Return Values

A function can return values of various types (numeric, string or sequence) and even tuples.

To return a value the programmer should use the keyword **return**.

To return a tuple, the programmer can separate the various values with a comma.

```
    >>> def integer_div(a, b):
...     return a // b, a %b
...
>>> integer_div(11,5)
(2, 1)
>>> d = integer_div(11,5)
>>> type(d)
<class 'tuple'>
>>> d[0]
2
>>> d[1]
1
>>>
```

1.4 Default parameters

Parameters can be assigned default values.

In the definition of the function the programmer should assign the default value.

```
>>> def simple_adition(a, b=0):
...     return a+b
...
>>> simple_adition(12)
12
>>>
```

They are overridden if a parameter is given for them

```
>>> simple_adition(12, 4)
```

```
16
>>>
```

The type of the default doesn't limit the type of a parameter

```
>>> simple_adition("aaa", "bbb")
"aaabbb"
>>>
```

the arguments with default values should not appear before the non-default arguments

```
>>> def f1(a = 0, b):
...     return a+b
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
>>>
```

1.5 Advanced functions

Functions are treated like any other variable in Python, the def statement simply assigns a function to a variable, and function names are like any variable

Functions are first class objects that can be assigned to variables.

```
>>> def ff(a, b):
...     return a+b
...
>>> function_variable = ff
>>> ff(2, 3)
5
>>> function_variable(2, 3)
5
>>>
```

Can be passed as a parameter.

```
>>> def strange_function(f):
...     print(f(4, 5))
...
>>> strange_function(ff)
9
```

```
>>>
```

1.6 High order-functions

Since functions can be sent to other functinos and called inside them, it is possible to create some high order functions that repeat a function (provided by the programmer) to any sequece.

1.6.1 Map

https://docs.python.org/3/library/functions.html#map

The map function applies a transformation function to every element of a sequence to create a new sequence.

map(func,seq)

- for all i, applies func(seq[i]) and returns the corresponding sequence of the calculated results.
- Equivalent to

```
for x in seq:
   new_seq.append(func(x))
```

```
>>> def double(x):
...     return 2*x
...
>>> lst = list(range(10))
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mp = map(double,lst)
>>> new_lst = list(mp)
>>> new_lst
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>
```

1.6.2 Filter

https://docs.python.org/3/library/functions.html#filter

The map function applies a condition function to every element of a sequence and creates a new with the elements to witch the function return true.

filter(func,seq)

- returns a sequence containing all those items in seq for which boolfunc is True.
- Equivalent to

```
for x in seq:
   if boolfunc(x):
      new_seq.append(x)
```

```
>>> def double(x):
...     return 2*x
...
>>> lst = list(range(10))
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> def even(x):
...     return(x%2) == 0
...
>>> flt = filter(even,lst)
>>> filtered_list = list(flt)
>>> filtered_list
[0, 2, 4, 6, 8]
>>>
```

1.7 lambda functions

https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions

Small anonymous functions can be created with the lambda keyword.

If not assigned to a variable these lambda functions can be sent as arguments to other functions.

```
>>> flt_odd = filter( lambda x: x%2 ==1 ,lst)
>>> lst_odd = list(flt_odd)
```

```
>>> lst_odd
[1, 3, 5, 7, 9]
>>>
```

if assigned to a variable can be used as regular functions.

```
>>> fl = lambda x, y: x+y
>>> fl(3, 5)
8
>>>
```

2 Modules

https://docs.python.org/3/tutorial/modules.html

Modules are objects that serves as an organizational unit of Python code.

Modules have a name-space containing arbitrary Python objects.

Modules are loaded into Python by the process of importing.

2.1 Importing modules

To use an existing module the programmer should use the **import** keyword

it is possible to import multiple modules at once:

- **import module1[, module2[,... moduleN]**

The basic **import** statement (just with the module name) is executed in two steps:

- find a module, loading and initializing it if necessary
- define a name or names in the local name-space for the scope where the import statement occurs.

If the module with the provided name does not exist an exception is raised

```
>>> import stdio
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'stdio'
>>>
```

if the **import** works correctly the objects inside such module can be used

```
>>> import datetime
>>> x = datetime.datetime.now()
>>> print(x)
2020-10-02 12:52:05.331894
>>> datetime.now()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: module 'datetime' has no attribute 'now'
>>>
```

When using the simple import it is always necessary to use the package name before any object class or function.

In the previous example to used that functions and classes inside the **datetime** module it is necessary to use the **datetime.** prefix.

If only a few names defined in a package are needed it is possible to define in the import expression those names:

- `from modname import name1 [, ... nameN]`

```
>>> from json import dumps, loads
>>> str = '[12, 13, [12, 13]]'
>>> o = loads(str)
>>> o
[12, 13, [12, 13]]
>>> type(o)
<class 'list'>
>>> str = '{12:"12"}'
>>> oo = loads(str)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.8/json/__init__.py", line 357, in loads
    return _default_decoder.decode(s)
  File "/usr/lib/python3.8/json/decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
    File  "/usr/lib/python3.8/json/decoder.py",  line  353,  in
raw_decode
    obj, end = self.scan_once(s, idx)
json.decoder.JSONDecodeError: Expecting property name enclosed in
double quotes: line 1 column 2 (char 1)
>>> str = '{"12":12}'
>>> oo = loads(str)
>>> oo
```

```
{'12': 12}
>>> type(oo)
<class 'dict'>
>>>
```

import mymodule	Brings all elements of **mymodule** in. To use the objects imported they must be refered as **mymodule.<elem>**
from mymodule import fun_x	Imports x from mymodule right into this namespace it is not necessary to use **mymodule.** before fun_x
from mymodule import *	Imports all elements of **mymodule** into this namespace it is not necessary to use **mymodule.** before the objects names

2.2 Module Creation

The simplest way to create a module is to create a python file and store it in the same folder as the other code.

A file named **module_x.py** can contain classes, variables and functions that can be accessed from other python file in the same directory if the programmer issues the **import Module_x expression**

3 EXERCISE 1

https://matplotlib.org/tutorials/introductory/pyplot.html

Create a module called **aux_functions.py** and define in that file the following functions:

- **count_letter** - function that receives as an argument a string and returns a dictionary with the count for every letter. Consider uppercase and lowercase letter as the same.
- **count_digit** – function that receives as an argument a string and returns a dictionary with the count for every digit

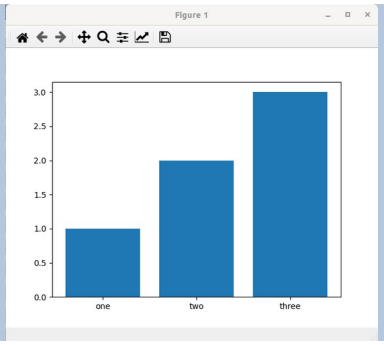
In another python file, implement a program that reads a string from the keyboard and presents a graph with two histograms with the number of letters and digits present in the file.

To show the histograms use the bar plot graphs from the **matplotlib**.

To show the plots you will need to

- install the matplotlib library
  - `pip3 install matplotlib`
- import the class matplotlib.pyplot from matplotlib module
  - `import matplotlib.pyplot as plt`
- create a bar plot with the x and y values
- show the plot

```
>>> import matplotlib.pyplot as plt
>>> x_legends = ['one', 'two', 'three']
>>> y_data = [1, 2, 3]
>>> plt.bar(x_legends, y_data)
<BarContainer object of 3 artists>
>>> plt.show()
>>>
```



Classes and Objects

4 Classes and objects

4.1 Objects

A unique instance of a data structure that's defined by its **class**.

An object comprises both data members (class variables and instance variables) and methods.

Each object has his set of instance variables that are different for each object.

The class variables have the same value for all the objects of a class

Objects are creates calling the name of the class followed by some arguments:

```
• r1 = range(10)
• l1 =  list((1, 2, 3))
• l2 = list()
```

or by means of other functions that return newly created objects:

```
• f = open(“filename.txt”, ‘r’)
• l2 = [1, 2, 3, 3, 4, 5]
• d2 = {1:'one', 2: 'two'}
```

these objects have a set of methods that can be called. These methods execute instructions (process the arguments, read/write data and return values) but also change the internal state of the object:

```
• f.readline()
• l1.pop(1)
• d2.values()
```

It is possible to know the type or class calling the **type** function:

```
>>> type(f)
<class '_io.TextIOWrapper'>
>>> type(l1)
<class 'list'>
>>> type(d1)
<class 'dict'>
>>>
```

4.2 Classes

https://docs.python.org/3/tutorial/classes.html

Classes are programmer defined prototypes for objects.

Classes defines a set of attributes that characterize any object of the class. These attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

The definition of a class follows a simple template:

```
1.  class EmployeeClass:
2.      empCount = 0
3.      def __init__(self, name, wage):
4.          self.name = name
5.          self.wage = wage
6.          EmployeeClass.empCount += 1
7.
8.      def print_total_empoyess(self):
9.          print ("Total Employee %d" % EmployeeClass.empCount)
10.
11.     def show_employee_info(self):
12.         print ("Name %s wage %d" % (self.name, self.wage))
13.
14.     def change_wage(self, new_wage):
15.         self.wage += new_wage
```

In the class definition the rules of the blocks and indentation also apply.

4.2.1 Class name

The first line starts the class definition and should include the new class name.

4.2.2 Class variables

Line 2 exemplify the declaration of a class variable. The value of this variable is the same on all this class objects.

In line 6 this attribute is changes for all the objects.

These variables are accessed using the class name followed by the attribute name:

- `EmployeeClass.empCount += 1`

4.2.3 Class constructors

The lines 3 to 6 present a constructor.

This special method receives the suitable arguments and is called when an object of this class is created.

One class can have multiple constructors, each with diferent number of arguments.

These constructors should follow the following structure:

- `def __init__(self,` arguments, ...`):`
- `    instructions`
- The argument cav vary from constructor to constructor

First argument of every constructor should be **self** (the object running the code)

4.2.4 Instance variables

Instance variables allow each object to have their own state.

These variables, allow each object to have their own “private, local variables”.

To manipulate these attributes it is necessary to use the **self** prefix:

- `self.name = name`

4.2.5 Methods

The lines 8 to 15 present methods that can be defined inside a class.

These methods are associated to objects and can only be called after creating an object

The declaration of these methods is similar to the declaration of functions but:

- should be done inside the class definition
- the first argument of these methods should be **self** (the object running the code)

Inside every method or constructor it is possible to access the class attributes (**EmployeeClass.empCount**), object attributes (**self.wage**) and method arguments (**new_wage** on line 14).

4.3 Use of classes

The creation of objects of user defined classes is similar to the creation of any other object as described in Section 4.1:

- `e1 = EmployeeClass('Joao', 100)`

Calling methods in our objects is also similar

- `e1.change_wage(12)`

4.3.1 Built-in attributes

Every object in python has the following attributes:

- __**dict**__ : Dictionary containing the class's name-space.
- __**doc**__ : Class documentation string or None if undefined.
- __**name**__ : Class name.
- __**module**__ : Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- __**bases**__ : A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

5 EXERCISE 2

Implement a class (named **rpnCalculator**) in a file called **rpnCalculator.py**: with the following characteristics:

- Object variables
  - stack of numbers
- Object methods
  - **pushValue**: pushes a value to the top of the stack
  - **popValue**: removes and returns the value on the top of the stack
  - **add**: removes two topmost values on the stack, adds them, and pushes the value to the top of the stack
  - **sub**: removes two topmost values on the stack, subtract them, and pushes the value to the top of the stack

Test the code on a simple example:

```
c = rpnCalculator()
c.pushValue(12)
c.pushValue(2)
c.add()
print(c.popValue()) #should print 14
```

Sockets

6 Python sockets

https://docs.python.org/3/library/socket.html

https://docs.python.org/3/howto/sockets.html

Socket is a technology that allows any process to communicate with another process (event in different computers).

The sockets concept includes an API (set of libraries that is implemented in various languages), the format of the communication endpoints identifiers and the protocols that allow the exchange of information between those processes.

6.1 Protocols

For communication over the internet the more commonly used protocols are **IP**, **TCP** and **UDP**.

6.1.1 IP

I**P** defines the way to address communication entities and the structure of the low level communication mechanisms.

For the sake of programming, **IP** defines the address structure of the communication entities. These addresses allow any entity to be able to establish communication with another entity as long at the address is known.

IP addresses are composed of:

- a 4 byte network address (127.0.0.1, 146.193.41.15, …)
- a 16 bit port number (port 80).

An entity that wants to receive messages or connections from a remote entity should define his own port and inform the OS of that value.

To receive and send messages or accept connections, a socket needs to be associated to the network address (previously configure in the operating system) and the port number provided by the programmer.

The **127.0.0.1** network address allows any two processes in the same machine to communicate using IP.

In order to communicate with a entity in another machine it is necessary to know the defined port and the network address of that entity.

6.1.2 UDP

The UDP protocol is built on top of the IP.

The UDP protocol allows the exchange of messages between two endpoints/sockets.

In every message the sender must explicitly state what is the IP address/port of the recipient.

Both entities should create their sockets and assign a port.

On reception of a message, in order to reply, first it is necessary to retrieve the address of the sender and use that address in the reply message.

When using this protocols, there is no guarantee that the sent messages are received by the recipient, nor it is possible to guarantee that the order of message reception is the same as the sending order.

6.1.3 TCP

The TCP protocol is also built on top of the IP and is alternative to the UDP.

With TCP communication two entities first establish a communication channel and from that moment on, when sending messages (on both ways) it is not necessary to explicitly state the address of the other entity.

Before the communication connection, the server (the entity that will receive the connection) should create a socket and assign it an address. After accepting the connection the server and client can exchange messages.

6.2 UDP-IP programming

For two entities to communicate using the UDP-IP protocols it is not necessary to establish a connection, but both entities should have a port assigned to their sockets. Afterward every message that is sent should be explicitly addressed.

It is necessary to import the socket module:

- **import socket** or **from socket import ***

The generic code is as follows:

	Server (receives connection)	**Client (makes connection)**
1	`from socket import *`	`from socket import *`
2	`s =socket(AF_INET,SOCK_DGRAM)`	`s =socket(AF_INET,SOCK_DGRAM)`
3	`host = '0.0.0.0'`	`host = '0.0.0.0'`
4	`port = 12350`	`port = 12351`
5	`s.bind((host, port))`	`s.bind((host, port))`
6		
7	`print(host, port)`	`s_addr = ('127.0.0.1', 12350)`
8	`while True:`	
9	`  data, addr = s.recvfrom(512)`	`s.sendto(b'OHOH', s_addr)`
10	`  print('recv', addr)`	
11	`  print(data)`	
12	`  s.sendto(b'EHEH', saddr)`	`data = s.recv(512)`
13		`print (data)`
14	`s.close()`	`s.close`

	Server (receives connection)	**Client (makes connection)**
1	Import of sockets library	Import of sockets library
2	**creation of a UDP-IP socket**	**creation of a UDP-IP socket**
3	definition of reception address	definition of reception address
4	definition of reception port	definition of reception port
5	**assignment of address/port to socket**	**assignment of address/port to socket**
6		
7		definitions of server address/port
8		
9	**reception of messages**	**send byte string to server addr**

10	print of client address	
11		
12	**send reply to client addr**	**reception of reply**
13		
14	**close of socket**	**close of socket**

6.3 TCP-IP programming

For two entities to communicate using the TCP-IP protocols it is necessary that one is considered server (that will receive the connection) and the other should be the client that established the connection.

It is necessary to import the socket module:

- **import socket** or **from socket import ***

The generic code is as follows:

	Server (receives connection)	**Client (makes connection)**
1	`from socket import *`	`from socket import *`
2	`s =socket(AF_INET,SOCK_STREAM)`	`s =socket(AF_INET,SOCK_STREAM)`
3		
4	`host = '0.0.0.0'`	`s_host = '127.0.0.1'`
5	`port = 12350`	`s_port = 12350`
6	`s.bind((host, port))`	
7	`print(host, port)`	
8	`s.listen(5)`	
9	`while True:`	
10	`  c, addr = s.accept()`	`s.connect((s_host, s_port))`
11	`  print('accpt', addr)`	
12	`  c.send(b'Hello')`	`l = s.recv(1024)`
13		`print (l)`
14	`  c.close()`	`s.close`

	Server (receives connection)	**Client (makes connection)**

1	Import of sockets library	Import of sockets library
2	**creation of a TCP-IP socket**	**creation of a TCP-IP socket**
3		
4	definition of reception address	definitions of server address
5	definition of reception port	definitions of server port
6	**assignment of address/port to socket**	
7		
8	t**ell OS to start accepting connections**	
9	while True:	
10	**reception of connection (creates new sockets**	**establishing of connection**
.		.
11	print('accpt', addr)	
12	**send byte string to client**	**reception of byte string from server**
13		
14	close of socket	s.close

7 EXERCISE 3

Implement a client-server system extending the **EXERCISE 1**.

The server will receive from the client a string and will reply to the client the results of applying the functions **count_letter** and **count_digit**.

The client will read a string from the keyboard, send it to the server, wait for the reply and print the received message in the screen.

8 Object serialization

https://www.json.org/json-en.html

https://docs.python.org/3/library/json.html

The previous exercises posed a complex problem: how to send the results to the client.

In the resolution it was necessary to convert the results of the functions into strings. If these strings are to be printed in the screen they can be formatted in the server.

If the results are to be processed it is necessary to convert these strings back to python objects.

One solution to this problem is to use object serialization format and libraries.

One of such format is **JSON**.

The **json** python library provides two functions that serializes python data types to a json string and another that de-serializes a json string back to python objects:

- **json.dumps** – function that receives a python object and converts it to a string that follows the json formation
- **json.loads** – function that receives a string and tries to convert it into a python object. If the string follows the json fomrt the conversion is successful and a python object is returned. If the string does not follow the json format an exception is raised.

```
>>> import json
>>> l = [1,2,"aaa", (1,2)]
>>> type(l)
<class 'list'>
>>> l
[1, 2, 'aaa', (1, 2)]
>>> json_s = json.dumps(l)
>>> type(json_s)
<class 'str'>
```

```
>>> import json
>>> d = {1: "one", 2: 'two'}
>>> type(d)
<class 'dict'>
>>> d
{1: 'one', 2: 'two'}
>>> json_s = json.dumps(d)
>>> type(json_s)
<class 'str'>
```

```
>>> json_s
'[1, 2, "aaa", [1, 2]]'
>>> new_l = json.loads(json_s)
>>> type(new_l)
<class 'list'>
>>> new_l
[1, 2, 'aaa', [1, 2]]
>>>
```

```
>>> json_s
'{"1": "one", "2": "two"}'
>>> new_d = json.loads(json_s)
>>> type(new_d)
<class 'dict'>
>>> new_d
{'1': 'one', '2': 'two'}
>>>
```

```
>>> import json
>>> int_json = "123"
>>> type(int_json)
<class 'str'>
>>> int_json
'123'
>>> o = json.loads(int_json)
>>> type(o)
<class 'int'>
>>> o
123
>>>
```

```
>>> import json
>>> str_json = '"abc"'
>>> type(str_json)
<class 'str'>
>>> str_json
'"abc"'
>>> o = json.loads(str_json)
>>> type(o)
<class 'str'>
>>> o
'abc'
>>>
```

```
>>> import json
>>> bool_json = 'true'
>>> type(bool_json)
<class 'str'>
>>> o = json.loads(bool_json)
>>> type(o)
<class 'bool'>
>>> o
True
>>>
```

If the programmer only applies the **json.loads** method to json strings created with the **json.dumps**, the function will always work correctly.

9 EXERCISE 4

Modify the previous exercise so that the string that is exchanged between the server and the client is in the json format.

In the client, after reception of the server reply, use the data to produce the two plots.

10 What to do next?

These short tutorial presented a brief introduction to python.

There are a lot of features, characteristics and libraries of the language that were note covered.

But now, the student have the minimum to read and understand most of the tutorials and documentation available.

Besides the www.python.org documentation, the next sites offer a lot of knowledge:

- https://www.learnpython.org/
- https://realpython.com/
- https://www.tutorialspoint.com/python/index.htm
- https://www.w3schools.com/python/
- https://pymotw.com/3/