Multimedia Communications - COMUL
&
Multimedia Systems - SISMUL

Lab 1.1 - Introduction to Python

Degree in Telecommunications and Informatics Engineering
&
Degree in Electrical And Computer Engineering

2022 / 23

Authors: Luís Vilaça (snv@isep.ipp.pt), Paula Viana (pmv@isep.ipp.pt)

---

1. Objectives

This lab script aims at introducing the student to the Python programming language, based on the assumption that the student has basic C programming and algorithmic skills. In addition, we introduce the most important Python packages (libraries) for computer vision/image processing and data science.

After reading, executing and understanding the exercises and demonstrations provided along this lab script, the student should develop minimum skills to read and develop software using Python. Obviously, this does not imply a profound knowledge on the topic, thus the student is encouraged to seek additional information and consult the official documentation.

2. Introduction

Python is a free, portable, dynamically-typed, object oriented scripting language [1]. It difers from the traditional system programming languages in the following:

System Programming Languages	Scripting Languages
Statically typed - variable type is known at compilation	Dynamically typed - variable type is associated at runtime
Compiled	Interpreted
Low-level - Represent a modest abstraction of the machine code	High-level languages - "Can be seen as a glue" that interconnects independent components into large-scale applications.

Python has a **block stucture terminated by the end of line and indicated by indentation**. Programs look like pseudo-code and have the advantage of avoiding usual syntax errors (e.g., semi-colons, bracketing). For example, the following Python code sums the numbers 0-9 and prints the result:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(sum(a))
```

In comparison, system languages typically look as follows, which is arguably less-intuitive:

```
#include <iostream>
#include <vector>
#include <numeric>

int main() {
    std::vector<int> a = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::cout << std::accumulate(a.begin(), a.end(), 0) << std::endl;
}
```

As such, Python is a language that is used to rapidly prototype and test code. Additionally, it is open-source, meaning anyone can participate in improving/maintaining the language. All these aspects made Python exceptionally popular among scientists, engineers and researchers.

2.1. Python interpreter

Python code is "executed" by a piece of software called the interpreter. An interpreter is any computer program that is capable of doing the following:

- Read in a text file containing your code (e.g. my_code.py).
- Parse the text and determine if it obeys the rules of the language (the interpreter will raise an error if the rules are not obeyed).
- Translate the parsed text into instructions, according to the Python language's specifications.
- Instruct the computer to carry out those tasks.

The first Python interpreter was written in C (programming language) and is known as CPython. Accordingly, it is the official interpreter of the Python language. Any new rules/features that are introduced to the Python language are guaranteed be implemented in the CPython code base. To compare the different computational paradigms, we illustrate in Fig. 1 and 2 the differences between a system language and an interpreted language.

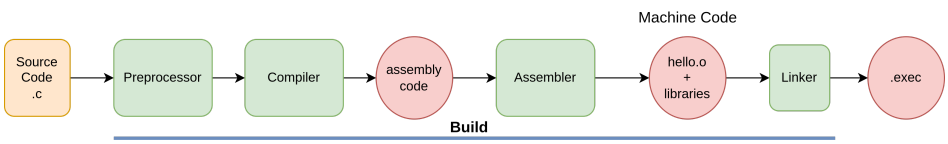

Fig.1 - C-code compilation

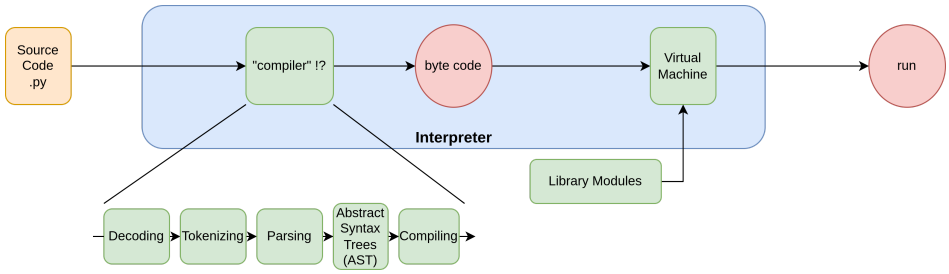

Fig.2 - Python-code interpretation

Clearly, we can see that Python is not a completely interpreted language since **it has a compilation step that converts raw code into a set of instructions called byte code**. Its comprised of nearly 101 instructions (Complex Instruction Set Computer - CISC). This step can be compared with the compilation step we commonly know from C, thus we can say that Python is a compiled and interpreted language [2,3]. Using the following Python code you can obtain the byte code corresponding to the function "def func (): x = y + 2".

```
import dis
# create expression
s = " def func (): x = y + 2"
# compile to bytecode
c = compile(s , "" , " exec " )
# obtain formated view
dis.dis(c)
```

```
>> 0 LOAD_GLOBAL              0 (y)
   2 LOAD_CONST               1 (2)
   4 BINARY_ADD
   6 STORE_FAST               0 (x)
   8 LOAD_CONST               0 (None)
   10 RETURN_VALUE
```

Installing Python on your computer is simply downloading an executable program that operates as an interpreter. This is the final result of the CPython's code. This software is bundled with a set of useful functions and tools that you can use in your code, which is known as the Python standard library. You can check its contents here.

Once you have a Python interpreter installed on your machine, using it to execute a Python script is quite simple. Assume for the sake of simplicity that the python interpreter program and my_script.py are in the same directory (a.k.a 'folder') in your computer. Then, in a terminal (cmd.exe for Windows) you can execute the following command:

```
python my_code.py     or     python3 my_code.py
```

Alternatively, you can use the interactive Python interpreter (IPython) by only typing `python` on the terminal. This will give you access to a line-by-line interpreter where you can quickly evaluate your code.

```
user@pc:~$ python
Python 3.10.8 (main, Nov 24 2022, 14:13:03) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2.2. Python memory management: how does the language deals with memory and variables?

A variable is a container that can store different values:

```
int a = 5;
int b = 4;
```

which can be allocated as follows: | Variable | Location | Value | | ---------- | ---------- | ---------- | | a | 0x3E8 | 101 | | b | 0x3E9 | 010 |

These values can be stored in "fixed-size buckets" which can be allocated in two different ways: **stack and heap**. In the following Table we summarise the most important key aspects between both memory allocation paradigms. For more information the student is advised to read [4] and [5].

Stack	Heap
Static allocation	Dynamic allocation (no enforced pattern of allocation)
Allocation in contiguous blocks of memory	Can be complex to keep track of which parts are allocated or freed
Allocation happens in the function call stack (attached to the thread)	Size is set when the application is started but can grow (allocator can request more from the operating system (OS)
Data stored can only be accessed by its corresponding thread	Data stored is visible to all threads
LIFO behaviour: most recently reserved block is always the first to be freed	Not very safe when comparing with stack-memory: the programmer needs to deal with memory deallocation to avoid memory leaks
Faster than heap memory allocation	

In C, stack allocation is attributed through the scope "{}" (local variables), while heap objects are attributes through C's dynamic allocation methods (malloc(), calloc(), etc).

In Python we have a uniform memory model where (almost) everything is an object. We do not use the concept of variable because variables are mere references (pointers) to objects allocated in the heap. Thus, Python is based on the following model: **names -> references -> objects**. We encorage the student to run the following snippet line-by-line:

```
# id() prints the memory location of the object
x = "test"
type (x)

y = x
id(x)
id(y)

y = x + '2'
id(x)
id(y)

z = "test"
```

```
id(z)
id(x)

z  = ["test", "b"]
type(z)
id(z)
```

Here, we can clearly see that the same memory location is assigned to different names/"variables". This is all automatically assigned and the programmer do not have to deal with memory management or allocation. Then, how is this managed? How can Python's inner works deal with possible memory leaks? How is memory freed?

Objects in Python store three different properties: type, value and reference count. Reference count is increased by assigning references to the same value (object reuse) or by assigning references to other references. These can be deleted by: 1) using `del var`; 2) assigning variables to None (`var = None`); 3) going out of scope (scope is defined by identation in Python).

When the objects' reference count reaches 0 it is marked by deletion (thrash) and the garbage collector takes place. This mechanism is used to automatically release memory when allocated objects are no longer in use. Garbage collection based on reference counting is the most basic mechanism. However, more complex methods can be used. We advise the student to read the following: [6] and [7]. In the following snipped we use the `sys.getrefcount()` function (that belongs to Python's standard library) to analyse the count of references as follows:

```
# sys.getrefcount() prints the object's reference count
# Its expected output is the actual reference count + 1
import sys

x = "test"
sys.getrefcount(x)

y = x
sys.getrefcount(y)

y = x + '2'
sys.getrefcount(x)
sys.getrefcount(y)

z = "test"
sys.getrefcount(z)

z  = ["test", "b"]
sys.getrefcount(z[0])
sys.getrefcount(x)

del(x)
sys.getrefcount(z[0])
```

This snippet is a small example of the reference counting mechanism used in Python. The student can observe how "variable assigning" is in fact assigning names to references and how changing those assignments modifies the objects' reference count.

2.3 Package installation and dependency management

In the previous snippet we've used the import statement to import the system package into our code (`import sys`). This allowed to access functions and objects that are provided in the sys package, which is part of Python's standard library. Python's import system is composed by modules and packages. **Modules are individual .py files** from which we can import functions and objects, and **packages are a collection of modules**.

To import both elements you can use the following types of imports:

1. Standard import (with rename): Python will run the `numpy.py` file and make its namespace available under `numpy` . In this case, the namespace `numpy` is renamed to `np` for simplicity.

```
# the module object for numpy is returned with the variable name `np`
>>> import numpy as np
>>> np.array([2., 3.])
array([2., 3.])
```

2. Partial import: Import some components. Python will run the `numpy.py` file and make array and Inf available in the common namespace.

```
>>> from numpy import array, Inf
>>> np.array([2., 3.])
array([2., 3.])
```

3. Global import: Import everything. Python will run the `numpy.py` file and make its namespace available in the common namespace (i.e., you can use array() without np.array()).

```
# import all of the contents of `my_module`
>>> from numpy import *
```

If you do:

```
import xxx.yyy
```

- `xxx` is the **directory** that you want
- `yyy.py` is the **file** you are importing

When importing from a directory, Python will run any `__init__.py` file, if such a file exists. We will see how this works with packages.

A package is composed by a directory that contains a file named `__init__.py` , modules and other subpackages (i.e., other directories with `__init__.py` files). The `__init__.py` file is used for indexation purposes and serves as an indicator to treat that directory as a package. Consider the following:

```
    -> Tree of directories:

    - your current Jupyter notebook / console session
    - package/
        |--__init__.py
```

```
        |-- utils.py
        |-- math/
            |-- __init__.py
            |-- calculus.py

    -> Contents of utils.py
    ---
    def convert_to_string(a):
        return string(a)
    ---

    -> Contents of calculus.py
    ---
    def divide_func(x, y):
        return x / y
    ---
```

Therefore, to access the contents of those modules the student will need to import the objects as follows:

```
# importing a function from the `utils` module
>>> from package.utils import convert_to_string

# importing the entire `math` module
>>> from package import math

# importing a function from the `math` subpackage
>>> from package.math.calculus import divide_func
```

What if the package we want to import was developed by other programmers? Do we need to have all the required packages by our code in our project folder (local)? How is it that we are able to import NumPy in any session or module without any knowledge of where that package is located?

When a package is installed it is placed where the `PYTHONPATH` environment variable points to. It points to the location where the interpreter is installed. The specific folder where the packages are installed by default is `site-packages` directory. Python provides package managers to automatically download and install packages from the Python Package Index (PyPI) [8]. The two most commonly used package managers are pip and conda. To install a package via pip and conda the student can execute `pip install <package_name>` and `conda install <package_name>` respectively. In this lab script we will not dwelve much into their details. However, the student is advised to read the official docs related to project packaging, anaconda, pip and distutils for module/package distribution.

2.4 Telling Python where the main() is located in a script

In all the snippets shown so far, Python runs the entire file. However, you might want to arrange your code as you usually do in C.

In this case, if you have a file that you want to run/import, but there is part of the code that you don't want to run, you can do:

```
if __name__ == '__main__':
    print('hey')
```

This will only print 'hey'. You can use it to develop your "main()". `__name__` is the name of the file, unless you are running the file explicitly, in which case its `"__main__"`.

Now that we shed some lights into Python's inner workings, the student is ready to learn its syntax and functionalities.

4. Basic Object Types

Object is anything that we can assign to a name/variable. This section entails all the basic object types that Python allows to use.

In [3]:
```
# Integer variables
x = 5
print(x, type(x))
```

```
5 <class 'int'>
```

In [5]:
```
# float variables
x = 5.0
print (x, type(x))
```

```
5.0 <class 'float'>
```

In [27]:
```
# String variables
x = "5"
print(x, type(x))
```

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

In [21]:
```
# String: formating strings
a = 12
b = 10
print(f"a = {a} and b = {b}")
```

```
a = 12 and b = 10
```

A string is sequence of characters. Thus, parts of strings can be taken using the slice operator (`[ ]` and `[:]`) with indexes starting from 0 at the beginning of the string and working their way up to index 'end – 1'. When you create a string you are essentially creating an object of the class string which contains several helper functions/methods:

- Strip(): it removes all whitespaces from the beginning/end of the string.
- Len(): returns the string length.
- Lower(): transforms the contents of the string to lowercase.
- Upper(): transforms the contents of the string to uppercase.
- Replace(): replaces the string with another string.
- Split(): splits the string into substrings if it finds separators (separator: comma).

In [8]:
```
# boolean variables
x = True
print (x, type(x))
```

```
True <class 'bool'>
```

Logic operators with booleans

Logic Operator	Symbollic Operator
`and`	`&`
`or`	`|`

In [9]:
```
# We need to pay attention to the type of variables we are using.
# Common mistakes when performing operations

# with int variables
a = 5
b = 3
print(a + b)
```

```
8
```

In [12]:
```
# with string variables
a = "5"
b = "3"
print(a + b)
# This happens because we are concatenating strings using the '+' operation.
```

```
53
```

Number types and available operations:

Operation	Description
`x + y`	Sum of two numbers
`x - y`	Difference of two numbers
`x * y`	Product of two numbers
`x / y`	Quotient of two numbers
`x // y`	Quotient of two numbers, returned as an integer
`x % y`	`x` "modulo": `y` : The remainder of `x / y` for positive `x` , `y`
`x ** y`	`x` raised to the power `y`
`-x`	A negated number
`abs(x)`	The absolute value of a number
`x == y`	Check if two numbers have the same value
`x != y`	Check if two numbers have different values
`x > y`	Check if `x` is greater than `y`
`x >= y`	Check if `x` is greater than or equal to `y`
`x < y`	Check if `x` is less than `y`
`x <= y`	Check if `x` is less than or equal to `y`

For more complex operations the student should use the math module available in the standard library.

In [11]:
```
# Casting: Python's standard library provides functions
# to convert one variable type into another.
# It Works with all the basic object types

x = 1
y = float(x)
print(type(x), type(y))
```

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

5. Sequence Types

5.1. Lists

List are defined through comma separated values enclosed by square brackers. Lists are mutable objects, which implies that the elements within can be changed and reordered. It can be comprised of many different types of variable objects due to the nature of the language, i.e., a list is a collection of pointers to objects. Similar to strings, lists contain large number of methods, thus we recommend consulting the official documentation.

In [89]:
```
l = [1, 2, 3]
print(l)
```

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

In [78]:
```
# Print the second element of the string
print(l[1])  # second element!
```

```
2
```

In [90]:
```
# Adding to a list

# use `append` to add a single object to the end of a list
l.append(7)

# use `extend` to add a sequence of items to the end of a list
l.extend([True, None])


print(l, len(l)) # Size of list

# Sidenote: to remove from a list you can consult
# the official documentation for the methods pop() and remove()
```

```
[1, 2, 3, 7, True, None] 6
```

In [91]:
```
# change the second element of the list to the string "bye"
l[1] = "bye"
print(l)
```

```
[1, 'bye', 3, 7, True, None]
```

In [92]:
```
# replace a subsequence of `l`: using slicing

# start:2, stop:4, step:1
l[2:4:1] = [-3, -4]
print(l)
```

```
[1, 'bye', -3, -4, True, None]
```

In [93]:
```
l.append(8)  # add a single item
l += [8]  # add one or more items
print(l)
```

```
[1, 'bye', -3, -4, True, None, 8, 8]
```

In [94]:
```
# Operations with lists
y = [2.4, "world"]
z = l + y
print(z)
```

```
[1, 'bye', -3, -4, True, None, 8, 8, 2.4, 'world']
```

In [100…
```
# List comprehension: use loops to create lists in succint code

# example: using a typical for loop
x = [1, 2, 3, 4, 5]
y = []
for item in x:
    if item > 2:
        y.append(item)
print(f"y (for loop) = {y}")

# using list comprehension
y = [item for item in x if item > 2]
print(f"y (l. comprehension) = {y}")
```

```
y (for loop) = [3, 4, 5]
y (l. comprehension) = [3, 4, 5]
```

QUESTIONS: lists

In [3]:
```
# 1. Create a list with only one entry which is the None object.
```

```
[None]
```

In [2]:
```
# 2. Assign the variable k to a list that contains an integer,
#    a boolean, and a string, in that order. Then, add two more
#    entries to the end of the list - a float and a complex number
# Note: use the complex() function - check docs
```

```
[4, False, 'moo', 3.14, (9-2j)]
```

In [5]:
```
# 3. Alphabetize the following list of names:
```

```
['Adam', 'Bob', 'Jack', 'Jackenzie', 'Jane', 'Ryan', 'Zordon']
```

Tuples

Tuples are collections of ordered values but are immutable (unchangeable). Tuples are used to store values that will never be changed.

In [101…
```
# Creating a tuple
x = (3.0, "hello") # tuples start and end with ()
print (x)
```

```
(3.0, 'hello')
```

In [102…
```
# Adding values to a tuple
x = x + (5.6, 4)
```

```
print (x)
```

```
(3.0, 'hello', 5.6, 4)
```

In [103…
```
# Try to change (it won't work and we get an error)
x[0] = 1.2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[103], line 2
      1 # Try to change (it won't work and we get an error)
----> 2 x[0] = 1.2

TypeError: 'tuple' object does not support item assignment
```

5.2. Sets

Sets are mutable collections of unordered values. However, every item in a set must be unique (no duplicates). These object types are mutable. However, the student can use the 'frozenset' type, which is a immutable version of sets (frozenset(x)).

In [112…
```
# Sets
text = "Learn Python"
print(set(text))
print(set(text.split(" ")))
```

```
{'n', ' ', 'L', 'P', 'a', 'e', 't', 'h', 'o', 'r', 'y'}
{'Learn', 'Python'}
```

In [111…
```
# initializing a set containing various immutable objects
{1, 3.4, "apple", False, (1, 2, 3)}
```

Out[111]:
```
{(1, 2, 3), 1, 3.4, False, 'apple'}
```

In [110…
```
# initialization via set-comprehension
{i**2 for i in range(5) if i != 3}
```

Out[110]:
```
{0, 1, 4, 16}
```

In [109…
```
# creating an empty set
set()  # specifying `{}` would create an empty *dictionary*
```

Out[109]:
```
set()
```

In [113…
```
# filter repeat-items from a collection by feeding it into a set
x = [1, 2, 1, 2, 1, "moo", "moo"]
set(x)
```

Out[113]:
```
{1, 2, 'moo'}
```

In [118…
```
# add a single member to `x`
x.add("dog")
print(x)
```

```
{'dog', 'b', 'a', 'c', 'd'}
```

In [122…
```
# update `x` by adding members of an iterable
x.update([1, 2, 3])
```

```
print(x)
```

```
{1, 2, 3, 'dog', 'b', 'c', 'd'}
```

In [119…

```
# remove a member of `x`
x.remove("a")
print(x)
```

```
{'dog', 'b', 'c', 'd'}
```

In [116…

```
# Set operations
# demonstrating set-comparison operations
x = {"a", "b", "c", "d"}
y = {"a", "b", "e"}

# union: items in x or y, or both
set_op1 = x | y  # or x.union(y)
print(f"union: {set_op1}")

# intersection: items in both x and y
set_op2 = x & y  # or x.intersection(y)
print(f"intersection: {set_op2}")

# difference: items in x but not in y
set_op3 = x - y  # or x.difference(y)
print(f"difference: {set_op3}")

# symmetric difference: in x or y, but not in both
set_op4 = x ^ y  # or x.symmetric_difference
print(f"symmetric difference: {set_op4}")

# check if set_1 is a superset of set_2
set_op5 = {1, 2, 3, 4} >= {1, 2}
print(f"superset: {set_op5}")

# check if set_1 and set_2 are equivalent sets
set_op6 = {1, 2, 3, 4} == {1, 2}
print(f"equivalent set: {set_op6}")
```

```
union: {'b', 'a', 'e', 'c', 'd'}
intersection: {'b', 'a'}
difference: {'d', 'c'}
symmetric difference: {'d', 'e', 'c'}
superset: True
equivalent set: False
```

QUESTIONS: sets

In [124…

```
# 1. Use a set to find all of the unique letters in the string
#    "The cat in the hat". Ignore all non-letter characters and
#    lowercase all letters.

# Sidenote: Use the built-in string functions isalpha() and lower()
# to filter out non-letter characters, and to lowercase the letters.
```

Out[124]:

```
{'a', 'c', 'e', 'h', 'i', 'n', 't'}
```

5.3. Dictionaries

Dictionaries are an unordered, mutable and indexed collection of key-value pairs. You can retrieve values based on the key and a dictionary cannot have two of the same keys.

In [126…
```
# Creating a dictionary
person = {"name": "Goku", "eye_color": "brown"}

print(person)
print(person["name"])
print(person["eye_color"])
```

```
{'name': 'Goku', 'eye_color': 'brown'}
Goku
brown
```

In [127…
```
# Changing the value for a key
person["eye_color"] = "green"
print (person)
```

```
{'name': 'Goku', 'eye_color': 'green'}
```

In [128…
```
# Adding new key-value pairs
person["age"] = 24
print (person)
```

```
{'name': 'Goku', 'eye_color': 'green', 'age': 24}
```

In [ ]:
```
# Length of a dictionary
print (len(person))
```

QUESTIONS: dictionaries

In [133…
```
# Given the tuple of student names (Ashley, David, Edward, Zoe),
# and their corresponding exam grades (0.92, 0.72, 0.88, 0.77):
#    1) Create a dictionary that maps: name -> grade.
#    2) Update Zoe's grade to .79
#    3) Add a new student, Ryan, whose grade is 0.34.
```

```
{'Ashley': 0.92, 'David': 0.72, 'Edward': 0.88, 'Zoe': 0.79, 'Ryan': 0.34}
```

6. Control Flow

6.1. Conditional statements (if)

In [1]:
```
x = 4
if x < 1:
    score = "low"
elif x <= 4: # elif = else if
    score = "medium"
else:
    score = "high"
print(score)

# if statement with a boolean
x = True
if x:
    print("it worked")
```

```
medium
it worked
```

6.2. For-loops

In [142…
```
veggies = ["carrots", "broccoli", "beans"]
for veggie in veggies:
    print(veggie)
```

```
carrots
broccoli
beans
```

In [140…
```
# `break` from a for loop = stop
veggies = ["carrots", "broccoli", "beans"]
for veggie in veggies:
    if veggie == "broccoli":
        break
    print(veggie)
```

```
carrots
```

In [141…
```
# `continue` to the next iteration = skip
veggies = ["carrots", "broccoli", "beans"]
for veggie in veggies:
    if veggie == "broccoli":
        continue
    print(veggie)
```

```
carrots
beans
```

6.3. While-loops

In [145…
```
# While loop
x = 3
while x > 0:
    x -= 1 # same as x = x - 1
    print(x)
```

```
2
1
0
```

6.4. Generators

Generators allows to generate arbitrarily-many items in a series, without having to store them all in memory at once. You can use them to iterate through a fixed sequence of values.

In [159…
```
# the range() operator/generator

# start: 2  (included)
#  stop: 7  (excluded)
#  step: 1  (default)
for i in range(2, 7):
    print(i)
```

```
2
3
4
5
6
```

In [163…
```python
# the enumerate() operator - used to create an index key

for idx, i in enumerate(range(2, 7)):
    print(idx, i)
```

```
0 2
1 3
2 4
3 5
4 6
```

In [165…
```python
# to create your own generator you can define the following

def generator():
    for i in range(10):
        if bool(i % 2):
            # indicates where a value is sent back to the caller,
            # but unlike return, you don't exit the function afterwards
            yield i

for item in generator():
    print(item)
```

```
1
3
5
7
9
```

7. Functions

To define a function, Python uses the keyword def. Then, to execute this function, just call it by its name:

In [158…
```python
# Define the function

def add_two(x = 1):
    """Increase x by 2.""" # explains what this function will do
    x += 2
    return x

print(add_two(3))
print(add_two()) # it will use the default values
```

```
5
3
```

In [147…
```python
# Function with multiple inputs
def join_name(first_name, last_name):
    """Combine first name and last name."""
    joined_name = first_name + " " + last_name
    return joined_name

# Use the function
```

```
first_name = "Multimedia"
last_name = "Systems"
joined_name = join_name(first_name=first_name, last_name=last_name)
print(joined_name)
```

```
Multimedia Systems
```

It's good practice to always use keyword argument when using a function so that it's very clear what input variable belongs to what function input parameter. Alternatively, you can use the terms ***args** and ****kwargs** which stand for arguments and keyword arguments. You can extract them when they are passed into a function. The symbol * establishes that any number of arguments and keyword arguments can be passed into the function.

In [149…

```
def f(*args, **kwargs):
    x = args[0]
    y = kwargs.get("y")
    print (f"x: {x}, y: {y}")

f(5, y=2)
```

```
x: 5, y: 2
```

QUESTIONS: control flow and functions

Function to see if a given number is a prime number or not:

```
function ret = is_prime(n)
    ret = true;
    for m = 2:sqrt(n)
        if mod(n, m) == 0
            ret = false;
            break
        end
    end
end
```

Convert this MATLAB code into Python.

Remember: A number is prime if it is not divisible by any number other than 1 and itself.

In [262…

```
# Write your answer here:
from math import sqrt



# TEST:
for n in range(1, 10):
    print(is_prime(n))
```

```
True
True
True
False
True
False
True
False
False
```

Expected Output: True, True, True, False, True, False, True, False, False

8. Object Oriented Programming

So far we only introduced the basic object types without explaining what is an object. Objects are the basic data representation within Python, they are an instance of a class. Their type is defined using a class, which is a blueprint to create objects, usually composed by a set of functions that define the different attributes and methods (i.e., functions bound to objects).

- Attributes: variable fields that characterize and shape a class/object
- Methods: functions bound to objects/classes

Sidenote: programmers usually use the word "type" to refer to Python's built-in types (e.g. int and str - basic object types) and "class" to refer to user-defined types.

8.1. Classes

Sidenote: methods defined using __(something)__ are Python's magic methods. They're always surrounded by double underscores (e.g. __init__). For more information, check this tutorial by Rafe Kettler on Python's magic methods. For more information on the definition of class objects check Python's official tutorial.

In [2]:
```
# class definition

import math

class Rectangle:

    # function used when an instance of the class is initialized (constructor)
    # once executed it produces the class object Rectangle, which encapsulates
    # the above definition and can be used to create objects that are instances
    # of this class
    def __init__(self, width, height, center=(0.0, 0.0), shape="Rectangle"):
        # The self operator refers to the object itself.
        # It is used to access and assign values to the
        # variables of this object.
        self.width = width
        self.height = height
        self.center = center
        self.shape = shape

    # - function used format the output message when printing
    # an object of this class
    # - "self" as the argument is meant to indicate that
    # the instance object is passing
    # - itself as the first argument of the method.
    def __str__(self):
        # return message when printing the object
        return f"{self.shape}(width={self.width}, height={self.height}, center={

    # function used when an instance of
    # the class is deleted (destructor)
    def __del__(self):
```

```
        print(f"{self.shape}(width={self.width}, height={self.height}, center={s

    # Instance methods
    # - when we call an instance method from an object, Python automatically
    # passes that instance object as the first argument and the other
    # arguments that were defined
    # - instance methods can modify the class and object instance states
    def compute_area(self):
        return self.width * self.height

    def rect_instance_method(self):
        """ A class method defined to simply
            return `self` unchanged"""
        return "instance method called", self

    # Class methods
    # - when we call a class method, the class object is automatically passed
    # as the first argument. There is a distinction between class instance
    # and class object. Class instance is the "class Rectangle" definition
    # and the object an initialized element of that class.
    # - class methods cannot modify the class state
    @classmethod
    def rect_class_method(cls):
        """ A class method defined to simply
            return `cls` unchanged"""
        return "class method called", cls

    # Static methods
    # - static methods do not receive any automatic arguments by Python
    # and can be called from an uninstantiated class object
    @staticmethod
    def euclidean_distance(a, b):
        # used to compute distance between two points
        (ax, ay) = a
        (bx, by) = b
        return math.sqrt(math.pow(ax-bx, 2) + math.pow(ay-by,2))

# create a class object named rect1
rect1 = Rectangle(10, 10, (0, 0.3))
print(f"{rect1}: area = {rect1.compute_area()}")

# check difference between class and instance method
print(Rectangle.rect_class_method, rect1.rect_instance_method)

# use the static method
d1 = rect1.euclidean_distance((1, 3), (3, 1))
# or
d1 = Rectangle.euclidean_distance((1, 3), (3, 1))
print(f"d1 = {d1}")

# delete the object rect1
del rect1
```

```
Rectangle(width=10, height=10, center=(0, 0.3)): area = 100
<bound method Rectangle.rect_class_method of <class '__main__.Rectangle'>> <bou
nd method Rectangle.rect_instance_method of <__main__.Rectangle object at 0x7f0
b586178b0>>
d1 = 2.8284271247461903
Rectangle(width=10, height=10, center=(0, 0.3)) was deleted
```

8.2. Inheritance

The inheritance mechanism allows a class to inherit and build off of the attributes of another class. The newly created class is called the derived or child class, and the one from which it inherits is called the base or parent class.

In [244…

```
# Creating Square, a subclass of Rectangle
class Square(Rectangle):
    def __init__(self, side, center=(0, 0)):
        # equivalent to `Rectangle.__init__(self, side, side, center)`

        # we pass in that single side length as both the width and height
        # to Rectangle.__init__. super always refers to the "super class"
        # or "parent class" of a given class, thus super is Rectangle here.
        super().__init__(side, side, center, shape="Square")

    # Only the derived/child class instance and objects
    # have access to this method
    def print_perimeter(self):
         return self.width * 2 + self.height*2

square1 = Square(10)
print(square1)

# print perimeter using the attributes
# from the parent class
perim = square1.print_perimeter()
print(f"Perimeter: {perim}")

del square1
```

```
Square(width=10, height=10, center=(0, 0))
Perimeter: 40
Square(width=10, height=10, center=(0, 0)) was deleted
```

QUESTIONS: object oriented programming

In [3]:

```
# 1.  Write a Python class named "Rectangle",
#     which accepts a length and a width as
#     parameters.
```

In [ ]:

```
# 2. Write a method dubbed "print_rectangle_surface()"
#    in this class, which calculates and prints out
#    the area of the rectangle.
```

In [ ]:

```
# 3.  Write a method called "print_rectangle_perimeter()"
#     in this class, which calculates and prints out the
#     perimeter of the created rectangle.
```

In [5]:

```
# 4. Write a method "rectangle_equalSides()" in this class,
#    which checks whether or not the sides of the rectangle
#    are equal and prints out the result.
```

In [7]:

```
# 5. Now, create an equal and an unequal rectangle object and
#    apply all methods on both rectangles.
```

```
120
44
False
---
36
24
True
```

9. Command-line arguments (argv)

The command-line arguments can be accessed in Python through `sys.argv`. The following snippet illustrates how to use it.

```
import sys

print("This is the name of the program:", sys.argv[0])

print("Argument List:", str(sys.argv))
```

However, this does not provide any built-in functionality to do argument parsing or checking. The argparse package is a brilliant way to help you automatize the creation of user-friendly command-line interfaces. Argparse is part of Python's standard library, thus you do not need to install it through pip or conda.

The following snippet gives an example of how to use the package for your projects.

```
import argparse

parser = argparse.ArgumentParser()

# Positional argument (string)
parser.add_argument('dataset')

# Non-positional argument, but required
parser.add_argument('--model', required=True)

# Required integers
parser.add_argument('--epochs', required=True, type=int)

# Optional strings
parser.add_argument('--optimizer', default='adam')

# Optional flag
parser.add_argument('--no-validation', action='store_true')


# To run the argument parser
args = parser.parse_args()
```

Possible arguments:

- `required=True` : give an error if this argument does not exist
- `type=xxx` : for types different than str
- `default=xxx` : if not required, you can provide a default
- `action=store_true` : set this argument to True/False if used or unused
- `choices['a', 'b', 'c']` : to restrict the subset of possible options

10. Ctypes: using funtion libraries from C

`ctypes` is a Python module used to load foreign function libraries into your code. It provides C compatible data types, and allows calling functions in DLLs or shared libraries.

In other words, if you compile your C or C++ code with `gcc -shared`, then you can import it into Python using ctypes. In the following example we illustrate an example of a function written in C which we can load into Python and use it freely.

Consider the following function developed in C:

```
% clibrary.c
int fn(int a, int b, int c) {
    return a+b+c;
}
```

Afterwards, your code can be compiled into a shared library using:

```
gcc -shared -o clibrary.so clibrary.c
```

And can be used in Python as follows:

```
import ctypes
lib = ctypes.cdll.LoadLibrary('./clibrary.so')
print(lib.fn(1, 2, 3))
```

11. Final Challenge

Develop the QUESTION from the object oriented programming section in your local development environment. You must follow these rules:

1. Directory structure:

```
- main.py (console session, where you run your "main")
- geo/
    |--__init__.py
    |-- rectangle.py
    |-- calc/
        |-- __init__.py
        |-- aux.c
        |-- aux.so
        |-- utils.py
```

Follow this directory structure. Your code in `main.py` you should import from the package geo the Rectangle class and it should contain the steps from question 5. The file `rectangle.py` should contain only the definition of the class Rectangle and import the helper functions from the subpackage `math`. Thereafter, the script utils.py should import the functions developed in C from `aux.so`.

2. Compile the helper functions (C) using gcc
3. Import the helper functions (C) in utils.py using `ctypes`
4. By running `main.py` you should be able to produce the same output as the notebook

Sidenote: Take special attention into the file paths that you are using. If it fails to load the *.so file due to an invalid file path you can use the `os` package. In this case, you can use

a combination of the following function and special variable: `os.path.dirname(__file__)`. **file** is a variable that contains the path to the module that is currently being imported. os.path.dirname() is a function used to obtain the directory name from a `path` object (__file__).

Sidenote 2:: Since we are using C functions that return floats we should modify the inputs and output in the ctypes lib object:

```
import ctypes
lib = ctypes.cdll.LoadLibrary('./clibrary.so')

# Cast the input variables into a ctype float which
# is compatible with primitive C data types
# In this case, c_float -> float
width = ctypes.c_float(width)
length = ctypes.c_float(length)

# define the type of output
lib.rectangle_surface.restype = ctypes.c_float
lib.rectangle_perimeter.restype = ctypes.c_float
```

- Utils functions in C:

```
% aux.c
#include <stdio.h>

float rectangle_surface(float width, float length) {
    return width * length;
}

float rectangle_perimeter(float width, float length) {
    return 2 * (length + width);
}
```

References

[1] K. Ousterhout, "Scripting: higher level programming for the 21st Century," in Computer, vol. 31, no. 3, pp. 23-30, March 1998, doi: 10.1109/2.660187.

[2] PyCon 2013: From Source to Code: How CPython's Compiler Works - Brett Cannon

[3] Awesome links for more in-depth information: Python behind the scenes; Official docs: Design of CPython;

[4] Geeks for Geeks: Stack vs Heap Memory Allocation

[5] Stack Overflow: What and where are the stack and heap?

[6] Nina Zakharenko talk

[7] CPython Official Documentation

[8] Python Package Index (PyPI)

References (development of this lab script)

This lab script was inspired from the tutorials/documentation from:

[a] Python Like You Mean It by Ryan Soklaski

[b] Made with ML by Goku Mohandas, Github link

[c] Scripts from workshops presented in previous editions of the CTM OpenDay at INESC TEC

[d] Introduction to Python3 from RealPython.com

[e] Ramalho, Luciano. Fluent python. " O'Reilly Media, Inc.", 2022.