3 Laboratories

Course of Computational Linear Algebra for Large Scale Problems

Francesco Della Santa

Politecnico di Torino

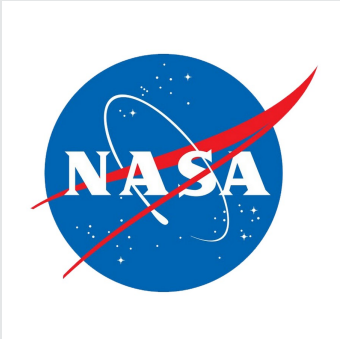
Contents i

Introduction

What is Pyhton

Python[1] is an interpreted programming language that in the last years has obtained very good results, especially for its versatility (scientific computing, games, 3D graphic design, etc.).

Who uses python



[1] Official website: `https://www.python.org/` .

Advantages of Python

- **Free** (no copyright restrictions);
- **Multi-paradigm** (both procedural and object oriented);
- **Multi-platform** (the Python interpreter is available for quite all the operating systems);
- **Performing** (functions and structures are internally implemented in C; interpretation $\Rightarrow$ bytecode compilation $\Rightarrow$ execution)
- **Automatic memory management** (garbage collection);
- **Integrable with other languages** (through other interpreters);
- More than 200 standard **modules** e thousands additional ones;
- **Syntax** simple but powerful;

Documentation

Online, for Python, many guides are available, as also courses and other kind of documentations:

- **Official documentation:**
  `https://www.python.org/doc/versions/`
- **Some free exercises:**
  `https://www.w3resource.com/python-exercises/`
- `...`

Installing Python

In many OS (operating systems) Python has been already installed by default, even if usually it is the version 2.7.

For installing **Python 3** on your computer, we suggest to **follow** the Section "Python Setup and Usage" of the **official documentation**[2] (with respect to your own OS).

Python can be downloaded and installed from `https://www.python.org/downloads/`.

For the laboratories, Python 3.7 or greater are **suggested** (but avoid the most recent one).

[2] `https://www.python.org/doc/versions/` and/or `https://docs.python.org/3/`.

Installing Python on Windows

If you are using Windows, read carefully all the steps of the installation and install what your OS is asking and suggesting.

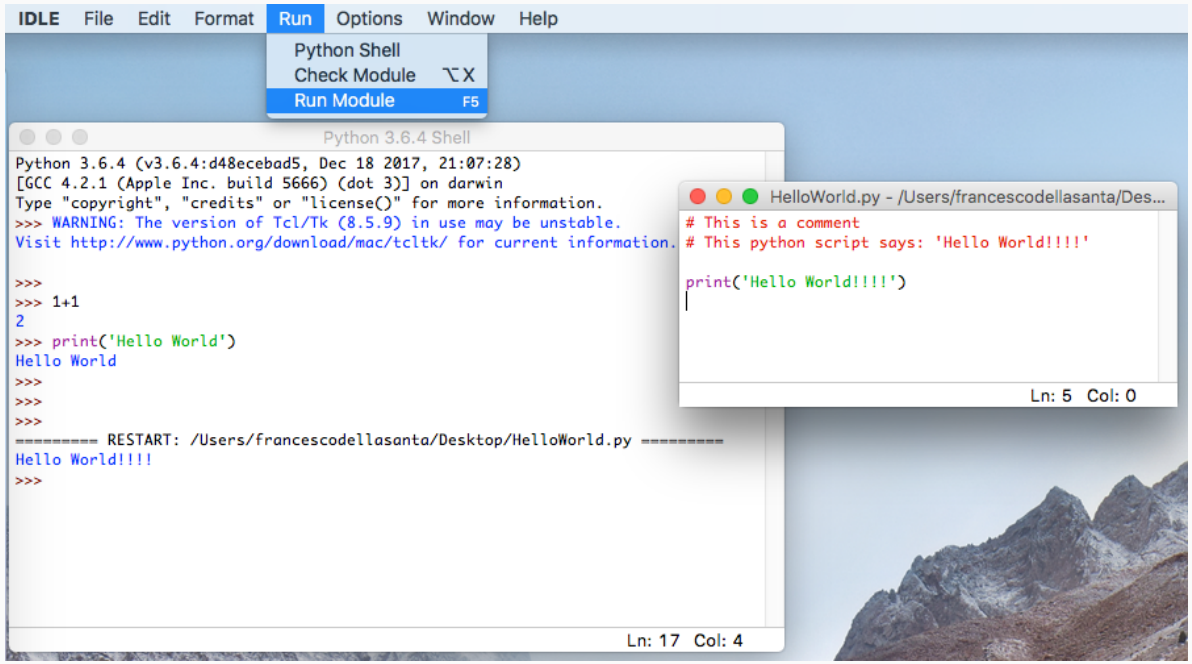
- Most of Windows PCs do not have problems, but other ones may return (**later**) errors installing the **scipy** package, due to the lack of some background tools/softwares in the OS.
- In the previous cases, the OS asks for additional installations that should fix the problem. If they do not work, we suggest to read carefully the following guide and try to understand if the python installation was correct: `https://docs.python.org/3/using/windows.html`
- If the problem persists, try to re-install python (or change the version) or consider to install **alternative bundles**[3] such as Anaconda (**suggested**) or WinPython.

[3] https://docs.python.org/3/using/windows.html#alternative-bundles

Using IDLE

The **default visual environment**[4] of python is IDLE (Integrated Development and Learning Enviroment).

This environment consists mainly in an interactive shell, but it lets also **create, open and execute** python programs ("*.py*" files).



[4] usually installed together with the interpreter.

Using other IDEs

IDLE is not the unique visual environment for working with python; many others IDEs exist (also free).

IDE

the acronym IDE means (Integrated Development Enviroment) and it is used to describe the visual development environments (like IDLE).

Some of the more famous free **IDE**s for python are:

- PyCharm (strongly **recommended**);
- Spyder (default with the alternative bundle Anaconda)
- ...

Using other IDEs

IDLE is not the unique visual environment for working with python; many others IDEs exist (also free).

IDE

the acronym IDE means (Integrated Development Enviroment) and it is used to describe the visual development environments (like IDLE).

Some of the more famous free **IDE**s for python are:

- PyCharm (strongly **recommended**);
- Spyder (default with the alternative bundle Anaconda)
- ...

Running Python Programs and Package/Module Installation

Almost all the IDEs are able to run python programs and allow package/module installations without using **command lines from terminal**.

Indentation in Python

A big difference between **Python** and other programming languages is given by a particular way to distinguish code blocks: the indentation.

Most of the other languages use **parentheses** or **restricted keywords** (e.g. begin/end) as delimiters for code blocks, using indentation only to read better the code.

PROs:

- Better legibility of the code;
- The structure of the code is **always** the same that you read from the indentation.

CONs:

- Indentation must be always the same for each block;
- Indentation inaccuracies bring to errors.

Indentation in Python - Example

```
>>> printme = True
>>> if printme:
        print('Print me!')


Print me!
>>>
```

```
>>> printme = True
>>> if printme:
                print('Print me!')
        print('Print me again!')

SyntaxError: unindent does not match any
outer indentation level
```

Figure 1: good indentation (**left**) and wrong indentation that returns an error (**right**).

Variables in Python i

In many programming languages (e.g. C), **variables** correspond to specific memory locations $\Rightarrow$ w.r.t. their type, variables have **fixed** dimension and location in the memory.

Python variables are instead a sort of "labels" (more precisely pointers) referred to **objects** of a given type characterized by fixed dimension and location in memory.

```c
// Codice C
int x;
x = 10;
x = 20;
// Cosa NON potrei fare
// x = "una_stringa"
```

```python
# Codice Python
x = 10
x = 'una_stringa'
x = [1, 2, 3]
```

Variables in Python ii

C

Codice	Memoria
int x;	x → vuoto (interi)
x = 10;	x → 10
x = 20;	x → ~~10~~ 20

PYTHON

Codice	Memoria
x = 10	x → 10
x = 'una_stringa'	x → 10, 'una_stringa'
x = [1, 2, 3]	x → 10, 'una_stringa', [1, 2, 3]

Figure 2: Left: In C the variable x must be declared and for it a space in memory for integer numbers is allocated. The first two codes ($x = 10$ and $x = 20$) don't create problems, but assigning a string to x will return an error. **Right:** In Python, x is just a pointer to objects in memory. All the codes illustrated correspond to the creation of three different objects in memory, each time pointed by x.

Variables in Python iii

Since Python variables are pointers to objects, we must **distinguish** between variables that point to the same object and variables that point to two different but equal objects.

```
>>> a = [1, 2]
>>> b = a
>>> c = [1, 2]
>>> b.append(3)
>>> print(a, b, c)
[1, 2, 3] [1, 2, 3] [1, 2]
>>>
```

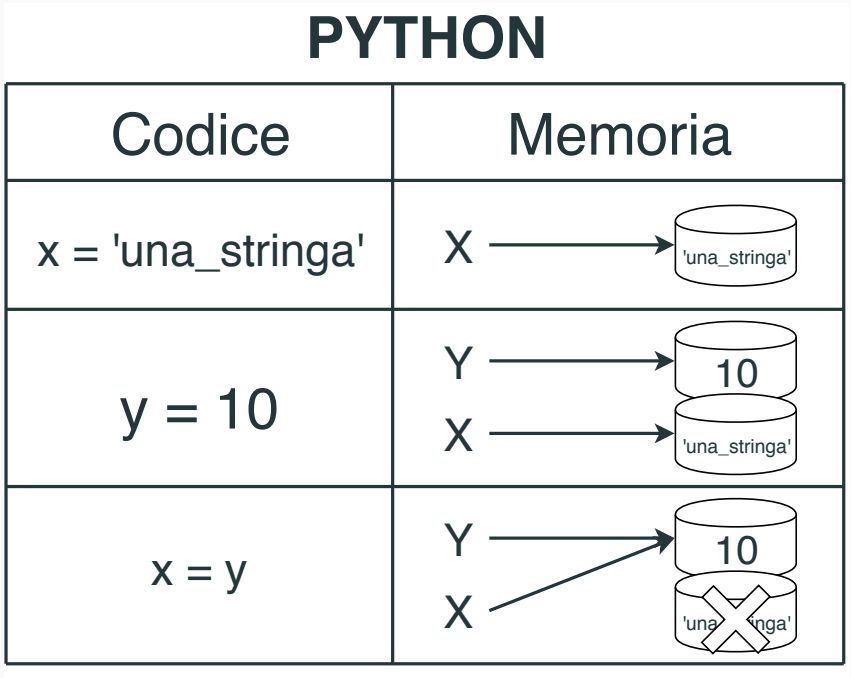
Figure 3: Once defined the variable a, the variables b and c are defined in the following way: **b points to the same object of a** ("b = a"), **c points to a new object** equal to the one pointed by a and b. Since a and b point to the same object, is indifferent which variable we will use to call the object.

Automatic Memory Management

The characteristics of variables in **Python**, let the language to be characterized by an automatic method for the memory management.

Automatic Management of the Memory in Python ("briefly")

Using a "**counter of pointers**" and a "**garbage collector**", Python delete automatically from the memory all those objects that are not pointed by any variables.



Data Types

With respect to the **type of datum** pointed by a variable, Python dedicates a specific amount of space in the memory for the information. The main data types in Python are:

Type	Command	Description	Examples
Integers	int	Integer number	-13, 0, 99999, ...
Floats	float	Floating point number	3.14, 1.0, 1.0E10, ...
Complexes	complex	Complex number	(13 + 4j), (2.0 + 1.1j), (0 + 3j), ...
Booleans	bool	True/False	True, False
Strings	str	immutable sequence of characters	'', 'Hello World!', ...
Lists	list	mutable sequence of objects	[], [1, 2, 3], [0, 'Hello', 1.5], ...
Tuples	tuple	immutable sequence of objects	(), (1, 2, 3), (0, 'Hello', 1.5), ...
Sets	set	Set of unique objects	{}, {1, 2, 3}, {0, 'Hello', 1.5}, ...
Dictionaries	dict	Structure that associates keywords and objects	{}, {'name': 'Francesco', 'course': 'CLA'}, ...

Focus on Linear Algebra

Since the focus of this lessons is how to **implement linear algebra with Python**, the details of the data types in the table will not be discussed, unless if required for numerical exercises. However, all information needed to understand the properties of these data types are reported in the official documentation: `https://www.python.org/doc/versions/`

Conversion of Data Types

In Python we can create objects of different type but representing the same "concept" (e.g. each $n \in \mathbb{Z}$ can be described as an int, a float or a str).

Therefore, in many (but not all) situations we can create new representations of a datum **converting** the old representations stored in memory. Here some examples:

Code	**Output**
int(13.9)	13
float(13)	13.0
complex(13), complex(13.9)	(13 + 0j), (13.9 + 0j)
str(13), str(13.9)	'13', '13.9'
int('13')	13
float('13'), float('13.9')	13.0, 13.9

Python Projects

We temporary stop the description of the Python properties to introduce briefly the **creation** of a Python project.

"Standard" project: it is the **simplest** method. We create a project folder that will **contain** all the programs of the project (".py" file), then we run Python from the folder;

Virtual Environments: more "**sophisticated**" method. After the creation of the "standard" project folder containing all the programs, we don't run directly Python but a "**copy**" of it, called Virtual Environment.

Use Virtual Environments!

Virtual Environments are strongly **recommended**. Indeed they let to specify and "freeze " the version of Python for the project, as also the different modules and packages.

In this way the **original installation** of Python on the computer is safe and protected from any possible corruption, while the **project** is protected from undesired updates.

Virtual Environments are not Portable! (usually)

Attention!

Virtual Environments (usually) are **not portable**! Then, a change in the path of the installation folder returns errors.

I you need to "move" a virtual environment to another computer (or share it with other users), particular methods exist but not moving/sending the installation folder.

Let's Create a Virtual Environment!

Let's create, using the terminal, a **virtual environment** (v.env.) based on **Python3.X**, through the **venv module** installed with Python (more info here):

1. open the **terminal**[5];
2. digit:

OS	command
Mac/Linux	`python3.x -m venv VEnvFolder_Path/myenv_name`
Windows	`PyInstall_Path\python -m venv VEnvFolder_Path\myenv_name`

where:

- *VEnvFolder_Path* is the path to the directory that will contain the v.env.;
- *PyInstall_Path* is the Python3.X installation folder on **your** Windows OS (e.g., something like `C:\Python3x`).

[5] in Windows, it is the command prompt or the PowerShell (PS)

Recalling Basic Terminal Commands

We can distinguish three main types of **terminal**:

1. **bash** (Mac/Linux);
2. **command prompt** (Windows). Denoted by **cmd**;
3. **PowerShell** (Windows). Denoted by **PS**.

The "Change Directory" Command (cd)

The cd command lets you **move** through the computer folders. For example:

OS	command example
Mac/Linux	`cd Directory/SubDirectory/SubSubDirectory`
Windows	`cd Directory\SubDirectory\SubSubDirectory`

Special uses:

- **Go back to Sup.Dir.:** `cd ..`
- **Indicate the Current Dir.:** `cd .`

Basic Commands: cd Example and Folder Content

Let us consider the following system of directories

- **Directory**
  - *DirectoryA*
    - DirectoryA1
  - *DirectoryB*
    - DirectoryB1

If you are in "DirectoryB1" and you want to move to "DirectoryA1" you digit (in a bash terminal):

```
cd ../../DirectoryA/DirectoryA1
```

Content of a Directory

If you want to see the content of a directory, use these commands (from the inside of the directory).

- **bash/PS:** `ls`
- **cmd/PS:** `dir`

Arithmetic Operators i

In previous slides we introduced some of the many different data types and, in particular, we showed **4 main numerical data types**: integers (`int`), floats (`float`), complexes (`complex`) and booleans (`True`, `False`[6]).

The basic **arithmetic operators** of python are:

Operator	Name	Operator	Name
+	sum	*	multiplication
-	subtraction	/	division
//	integer division	**	power
%	int. div. reminder		

[6] w.r.t. arithmetic operations, "True" and "False" are considered as 1 and 0, respectively.

Arithmetic Operators ii

All the arithmetic operators return an **output** of the "dominant" numerical data type among the inputs' types and the theoretical output's type (`bool` $<$ `int` $<$ `float` $<$ `complex`).

Arithmetic Operators ii

All the arithmetic operators return an **output** of the "dominant" numerical data type among the inputs' types and the theoretical output's type (`bool` $<$ `int` $<$ `float` $<$ `complex`).

The only exception is the **division** operator that returns always at least a float number. For example:

$$\texttt{10 / 5} \mapsto 2.0$$

$$\texttt{5 / True} \mapsto \texttt{5.0}$$

$$\texttt{(5 + 5j) / 5} \mapsto \texttt{(1 + 1j)}$$

Combination of Arithmetic and Assignment Operators

In Python special **assignment operators** exist to combine together the assignment operator and an arithmetic one. The role of such operators is to update the values of a given variable.

Here some examples:

Special Operator	Equivalent Assignment	Special Operator	Equivalent Assignment
x += y	x = x + y	x *= y	x = x * y
x -= y	x = x - y	x /= y	x = x / y
x //= y	x = x // y	x **= y	x = x ** y
x %= y	x = x % y		

Comparison Operators and Booleans

For numerical data, Python has also some **comparison operators** (obviously characterized by boolean output, `True` o `False`).

Operator	Name	Operator	Name
==	equal to	!=	not equal to
<	less than	<=	less than or equal to
>	grater than	>=	greater than or equal to

For boolean data, Python has particular built-in **boolean arithmetic operators** (or **logic operators**).

Operator	Name	Operator	Name
and	conjunction	or	disjunction
not	negation		

Logic Operators - Recap about Tables of Truth

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

	True	False
not	False	True

Identity Operators

In Python also identity operators exist and they are used to compare the "congruency" between two objects stored in memory.

Operator	Name
is	congruency between objects
is not	not congruency between objects

"is/is not" are different from "==/!="

"is/is not" tell if the two variables point to exactly the same object in memory. **"=/!="** instead tell if the variables point to two objects having same type and value (they are not interested if they are the same object or not). For example:

```
a = [1, 2],  b = a,  c = [1, 2]
```

`a is b` $\mapsto$ `True`, `a is c` $\mapsto$ `False`, `a == c` $\mapsto$ `True`

Comments

Python, as all other programming languages, lets the user to add **comments** in the code, through the symbol "#".

Every command or symbol written after #, will not be read (and then translated) by the python interpreter.

```
>>> # This is a comment
>>> a = 10
>>> b = 5 # + a
>>> print(b)
5
>>>
```

Hints of Object Oriented Programming

What is the Object-Oriented Programming

The **Object-Oriented Programming** (OOP) differs from the procedural one (functions & routines) because it is characterized by the usage of "**organized units**" (the so-called objects) that **store** inside themselves both data and functions.

Focus on Linear Algebra (Again)

Since the focus of this lessons is how to **implement linear algebra with Python**, only for what concerns the main concepts of the **OOP** will be discussed, avoiding the many other details (unless if they are required for numerical exercises). However, all information needed to understand the OOP in Python are reported in the official documentation:
`https://www.python.org/doc/versions/`

Differences between OOP and Procedural Programming

Action	Procedural Prog.	OOP
Data and Functions	managed separately	stored together in objects
Execution of Functions	all inputs are external, must be verified	exist "inner" inputs that don't need verifications
Code Updates/Corrections	more complicated	less complicated

We should use Procedural Prog. for:
representations of operations and/or data.

We should use OOP for:
representations of "entities" characterized by parameters and "actions".

Procedural Programming for Linear Algebra

From these observations we deduce that for Linear Algebra implementations, **we need a procedural approach**; however, these approach will be used with the support of OOP objects.

OOP's Terminology

Classes and Objects: **classes** are constructs used for the characterization of the "abstract" properties of the **objects** (also called instances) belonging to them.

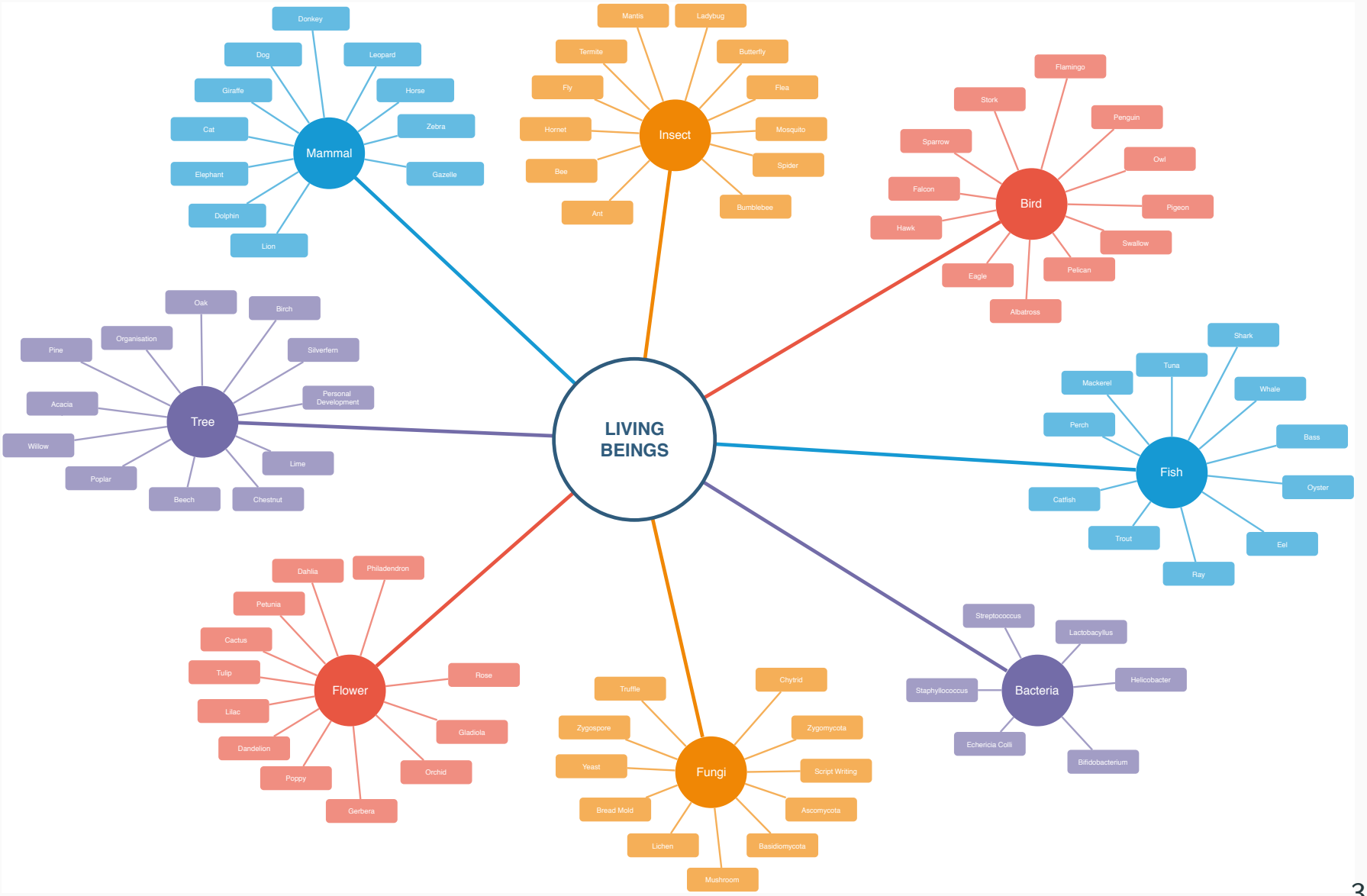
Methods: they are actually **functions** but defined inside a class. Methods can be executed easily typing[7]: `object.method_name(args)` .

Attributes: they are actually "**personal variables**" of the class' instances that point to other objects characterizing them. They are divided in **class att.** (constant) and **instance att.** (different from object to object). For calling the attribute of an object, type: `object.attribute_name` .

Inheritance: in the OOP, some classes can be defined as sub-class of an another one, **inheriting** all the methods and attributes defined in the so-called super-class.

[7] alternatively and less used: `class.method_name(object, args)` .

Super-classes and Sub-classes Visual Example



Differences Between Functions and Methods

In this slide we focus on the (already described) differences between **functions** and **methods**.

Functions: they take one, more or none input arguments. They can be used with **objects of different types** (classes); e.g., the print function. For the execution of a function the command is something like

```
function_name(args)
```

Methods: actually, they are **functions** at all but bounded to a given class, since they are defined directly inside the class code. For the execution of a method for a given object, type:

```
object.method_name(args)
```

Example - the Function abs and the Method is_integer

```
>>> # Creation of variable pointing to two float type objects
>>> x = 5.5
>>> y = -3.0
>>> # Function for absolute value computation
>>> abs(x)
5.5
>>> abs(y)
3.0
>>> # Method to check if the float object describes an integer number
>>> x.is_integer()
False
>>> y.is_integer()
True
>>>
```

The help and dir Functions

Two **important** built-in functions of Python are the function **help** and the function **dir**.

help: this function print on screen a short guide of the object received as input argument (`help(arg)`). If the function does not receive any input (`help()`), it opens in the shell an interactive "help-shell" where the user can type the names of functions, classes, etc., that wants to understand (`quit` to exit from the help-shell).

dir: this function returns a list of all the methods and attributes of the class of the object given as input argument (`dir(object)` or also directly `dir(class)`).

help and dir - Examples

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.

>>> dir(float)
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__dir__', '__divmo
d__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__', '_
_getattribute__', '__getformat__', '__getnewargs__', '__gt__', '__hash__', '__init
__', '__init_subclass__', '__int__', '__le__', '__lt__', '__mod__', '__mul__', '__
ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__', '__r
educe__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '_
_round__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__setformat__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__'
, 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
>>>
>>>
>>> help(float.is_integer)
Help on method_descriptor:

is_integer(...)
    Return True if the float is an integer.
```

Figure 4: Example of the function **help** called for the abs function (**top**) and the is_integer method (**bottom**). In the **middle**, example of the **dir** function called for the float type.

Conditional and Cycle Blocks

Introduction to Conditional and Cycle Blocks

In this section we introduce the Python syntax used for conditional blocks (if-elif-else) and cycle blocks (for and while).

We recall the important role of indentation, already introduced in slide 9. All the blocks are characterized and delimited by the **indentation** and not by restricted keywords or parentheses.

if-elif-else Block i

The **if-elif-else** blocks let to execute instructions under precise circumstances. The syntax of this blocks is the following:

```
if main_condition:
    if_instructions
elif otherwise_condition_1:
    elif_intructions_1
elif ... :
    ...
elif otherwise_condition_n:
    elif_intructions_n
else:
    else_instructions
```

Observations about the if-elif-else Blocks

Attention!

Don't forget the colons (":") after all the conditions and pay attention to **indentation**!

conditions' type: all the conditions (next to if and elif) must be **booleans**;

if-instructions: they are executed only if `main_condition` is `True`;

i^{th} **elif-instructions:** they are executed only if `otherwise_condition_i` is `True` and **all the previous conditions** are `False`;

else-instructions: they are executed only if **all the previous conditions** are `False`;

elif blocks quantity: theoretically there is no limit and, also, they are not necessaries;

else blocks quantity: maximum one.

Python Iterables

In Python some objects are classified as "**iterables**". Without going into the details, an iterable object in Python is any "container" object able to **return all its elements** (other objects) one after the other.

For example, are iterables: lists, tuples, strings, ndarrays.

range

Special iterable objects are the **ranges**. They describe sequences of numbers using a **starting number** (included, default is 0), a **last number** (exluded) and a **step length** (default is 1). This objects are created with the homonym function range[8]: **range(i0,iFin,steplength)**

Observation: the elements of a sequence described using a **range** are not visible unless the range is **converted** into (e.g.) a list or a tuple.

[8] underlined variables, if not inserted, are substituted by default values

range - Examples

```
>>> r = range(10)
>>> r
range(0, 10)
>>> list(r)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> r1 = range(3,12,3)
>>> r1
range(3, 12, 3)
>>> list(r1)
[3, 6, 9]
>>>
```

for Cycles

The **for Cycles** are code blocks that (in Python) let iterate some instructions **for each one of the elements** belonging to a given iterable object.

The **syntax** is the following (pay attention to indentation and to the colon):

```
for element_obj in iterable_obj:
    iteration_instructions
```

Attention!

`element_obj` is a variable that **points to the current object** in `iterable_obj` at each iteration. Then, at the end of the for cycle, `element_obj` **will point to the last object** of `iterable_obj`.

Other details about the for cycles (continue & break commands, for-else block) will not be discussed.

while Cycles

The **while cycles** are code blocks that iterate instructions if and until a certain **condition** is `True` (a sort of mixture between the if and the for blocks).

The **syntax** is the following (pay attention to indentation and to the colon):

```
while condition:
    iteration_instructions
```

Loops

Sometimes, while coding, happens to make the **mistake** of inserting a condition for a while cycle that is always `True`. In this cases an **infinite loop starts.** To break the loop, remember that Ctrl + C **stops** running programs.

Other details about the while cycles (continue & break commands, while-else block) will not be discussed.

Functions

Functions

Obviously, also in Python **functions** can be defined. Functions, in a programming language, are used to run always the same operations, with the possibility of **varying some parameters** (input arguments of the function).

The **syntax** is the following (pay attention to indentation and to the colon):

```
def func_name(arg_1, ..., arg_n):
    func_instructions
    return output_1, ..., output_m
```

To **execute** a function (and save its outputs), just type:

```
output_1, ..., output_m = func_name(arg_1, ..., arg_n)
```

Basic Informations about Functions

Variables: The **variables** defined inside the function during the execution are local (not memorized after the execution). Also global variables **can be used** inside a function.

Return: this command **stops** immediately the execution of the **function**, giving back as outputs the **variables written after it** (on the same line). The return command, is not mandatory if the function has no outputs.

Outputs Supression: if we **don't want** to save the **outputs** of a function, we just type `func_name(arg_1, ..., arg_n)`. Otherwise, we can write an underscore symbol for those specific outputs we want to suppress, e.g.:

```
out_1, _, out_3 = eg_func(arg)
```

Positional Arguments

In Python, the **input arguments** of a function can be distinguished in three main categories. The most simple ones (used in the syntax of the previous slide) are the **positional arguments**.

This kind of arguments, **must be all furnished as inputs** to the function when executed, in the same order written in the function's definition.

If one of the arguments is **missing**, the function will return an error; if the **order is not respected**, the function will return an error too or, worse, return (silently) wrong outputs.

Other kind of input arguments and function details

The other two categories of input arguments for functions (`*args` and `**kwargs`), **will not be discussed here** and they will be introduced in the course only if needed during the python laboratories. The same reasoning applies to all the other details about functions.

Modules and Packages

What a Module is

In Python, the **modules** are someway equivalent to the libraries of the other programming languages.

More precisely, **modules** are ".py" **files** where constants, functions, classes, etc. are defined and they have the purpose of **collecting such material** for better organizing the projects. **Packages** are folders[9] containing modules or sub-packages.

Python furnishes lots of bult-in modules/packages, but many others can be **created** by users or are distributed **online**.

In particular an **official repository** of modules/packages exists and it is called **Python Package Index**[10] (PyPI); the most famous modules and packages for Python are stored in this repository and can be easily downloaded and installed.

[9] with particular characteristics not specified here.
[10] `https://pypi.org/`

How to Use Modules and Packages

Before using a module/package, we must **check** that it has been installed (third party ones) or that its path belongs to the working path of the interpreter (user-created ones); then, the module/package can be "**imported**" (in the shell or in a script) typing:

```
import modorpack_name
```

In this way, Python creates a **variable** `modorpack_name` that points to the **module/package**.

Once the module/package has been imported, with the **help** function (`help(modorpack_name)`) it is possible to read the **description** of it and all its **contents**.

Modules are Scripts and Vice-versa

For Python, there is **no differences between scripts and modules**; both of them are ".py" **file** and therefore can be **executed** or **imported** equivalently.

Working with Imported Modules/Packages

Once we imported a **module**, we can work with the objects defined **inside** it (variables, classes, functions, etc.), with the following syntax:

`mod_name.var_name` or `mod_name.func_name` or `...`

If we have imported a **package**, the syntax changes in:

`pack_name.mod_name.var_name` or `...`

Other Ways to Import

- **import my_module as mymod** :
  imports my_module as a variable mymod.
- **from my_module import obj_1, . . ., obj_n** :
  Iimports the objects $obj\_1, \ldots, obj\_n$ defined in my_module as variables with same names. All the other objects in the module will not be imported.
- **from my_module import *** :
  imports all the objects in my_module as variables. This technique of importation is highly deprecated since it can overwrite some variables imported from other modules.
- **from my_module import obj_1, . . ., obj_n as o_1, . . ., o_n** :
  imports the objects $obj\_1, \ldots, obj\_n$ defined in my_module as variables named $o\_1, \ldots, o\_n$, respectively. All the other objects in the module will not be imported.

Other Ways to Import - Examples

```
>>> import newtonmod as nw
>>> from newtonmod import G, Mt, Mm, Rt, Rm
>>> from newtonmod import newton_to_kgf as weightfunc
>>> kg_man = 80
>>> man_earth_force = nw.newton_law(kg_man, Mt, Rt)
>>> weight_man = weightfunc(man_earth_force)
>>> man_mars_force = nw.newton_law(kg_man, Mm, Rm)
>>> marsweight_man = weightfunc(man_mars_force)
>>> print('Peso sulla Terra:', weight_man)
Peso sulla Terra: 80.05580503261903
>>> print('Peso su Marte:', marsweight_man)
Peso su Marte: 30.254480632782517
>>>
```

How to Install Modules & Packages from PyPI - Terminal

For **installing a third party module/package** from the PyPI repository, we need the built-in module **pip** (keep pip updated!).

Modules and Virtual Environments

ATTENTION: it is always better to install modules/packages in v.env.s and not in the original Python installation!

How to install a module/package in a v.env.:

1. **V.Env. Activation** (more info here):

Shell	command
bash	`source VEnvFolder_Path/myenv/bin/activate`
cmd	`VEnvFolder_Path\myenv\Scripts\activate.bat`
PS	`VEnvFolder_Path\myenv\Scripts\Activate.ps1`

2. **Module/Package Installation:** `pip install modorpack_name`

Terminal, V.Env.s, and Packages - Further Information

N.B.: when a v.env. is activated you can **see** its name on the left of the terminal's command line (in our example, we see "`(myenv_name)`").

Other commands[11]**:**

- **Deactivate the V.Env.:** `deactivate`
- **Update Mod./Pack. to last version:**
  `pip install --upgrade modorpack_name`
- **Install Mod./Pack. of a specific version:**
  `pip install modorpack_name==x.xx.x`
- **Install a list of modules/packages written in a "list.txt" file:**
  `pip install -r list.txt`

[11] assuming an activated v.env.

Jupyter Lab e Jupyter Notebooks

Jupyter Lab

Jupyter Lab is a browser-based editor and user interface of the jupyter project used to work (mainly) with python. To work with jupyter-lab, you first need to install the `jupyterlab` module; then, to launch it, just digit the command `jupyter-lab`.

PROs and CONs of Using Jupyter Lab

+ jupyter **notebooks** (for teaching and/or tech reports);
+ user-friendly interface for **remote-server connections**;
- not good for **code development** (other IDEs are better, e.g., Pycharm);
- not good for **long and heavy computations** (that requires scripts and not notebooks);

Jupyter Notebooks

The **Jupyter Notebooks** are special files (`.ipynb` extension). They are documents where text and interactive code cohexist. Specifically, the notebooks integrate the interactive shell of the `ipython` module inside a text (markdown) editor.

These type of file is very useful when you need to show well described code examples, step-by-step operations, and/or results, such as:

- **Tech reports**;
- **Visualizations**;
- **Teaching**.

ATTENTION: for "real" scientific computing works, do not use notebooks, use scripts and modules (i.e., `.py` files)!

Jupyter Notebooks - Keyboard Shortcuts

N.B.: keys CTRL and ALT are for Linux/Windows; they change into CMD and OPTION, respectively, in Mac OS.

- **SHIFT + Enter:** run current cell and select next cell;
- **CTRL + Enter:** run selected cells;
- **ALT + Enter:** run current cell and, then, insert a new cell;
- **Esc:** exit from "insert mode";
- **Enter:** enter into "insert mode";
- **DD:** delete selected cells (out of "insert mode");
- **Z:** cancel delete-cell operation (out of "insert mode");
- **C:** copy selected cell (out of "insert mode");
- **V:** paste copied cells (out of "insert mode");
- **M:** Change cell-type, from "code" to "markdown text" (out of "insert mode");
- **Y:** Change cell-type, from "markdown text" to "code" (out of "insert mode").