

# Python Introduction Course: introduction

With emphasis on data-science problems

[Geant4 Course at the 16th Seminar on Software for Nuclear, Sub-nuclear and Applied Physics, Porto Conte, Alghero \(Italy\), 26-31 May 2019. \(https://agenda.infn.it/event/17240/\)](https://agenda.infn.it/event/17240/)

This course is available on [gitlab \(https://gitlab.com/andreadotti/pyalghero2019\)](https://gitlab.com/andreadotti/pyalghero2019)

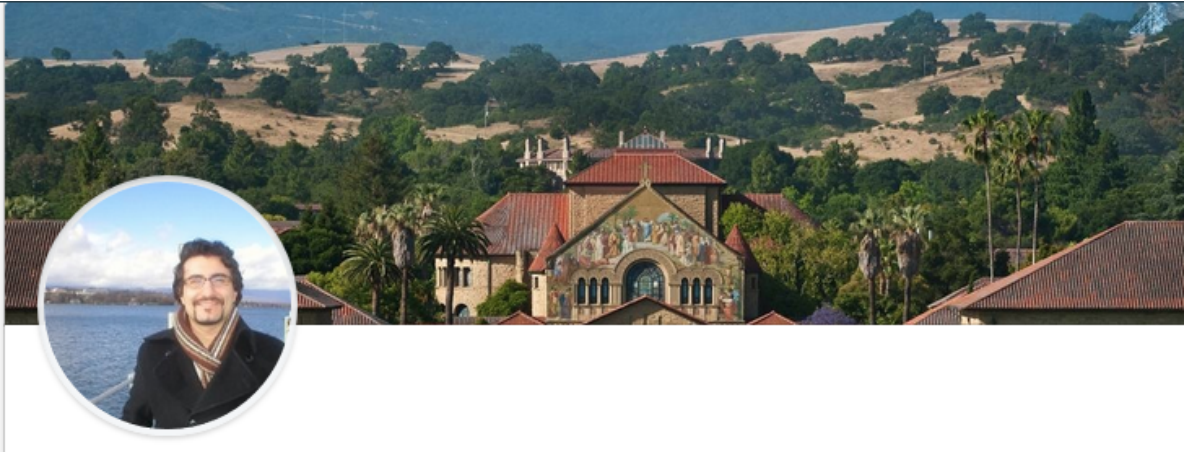
Contact me: (mailto:andrea.dotti@gmail.com)

# Introduction

# The who

1. I am a (HEP) physicist as background, I worked at CERN and SLAC/Stanford:
  - A. Geant4 Collaboration member: parallelization, HPC, physics validation
  - B. ATLAS Experiment member: simulations and data-analysis
  - C. Worked briefly at LCLS-II: off-line software design
2. I have recently moved to the private sector and I am working as a data scientist

I am very lucky to have had the opportunity to see *both sides* of data-analysis and large data-analytics. I program in C++ and Python with the latter used (mainly) for data-analysis.



# The Why

Python is one of the fastest growing programming language (among the [most populars](https://www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10?IR=T#4-php-7) (<https://www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10?IR=T#4-php-7>) in industry, the second most active on [github](https://octoverse.github.com/2017/) (<https://octoverse.github.com/2017/>), and number 4 on [stackoverflow](https://insights.stackoverflow.com/survey/2019) (<https://insights.stackoverflow.com/survey/2019>).

It is getting more and more traction for science and basic research problems (see [here](https://arxiv.org/abs/1807.04806) (<https://arxiv.org/abs/1807.04806>), [here](https://developer.ibm.com/dwblog/2018/use-python-for-scientific-research/) (<https://developer.ibm.com/dwblog/2018/use-python-for-scientific-research/>), [here](https://www.stat.washington.edu/~hoytak/blog/whypython.html) (<https://www.stat.washington.edu/~hoytak/blog/whypython.html>)), thus it is a good moment to learn it.

I am not an expert of Python, but I hope to be able to give you:

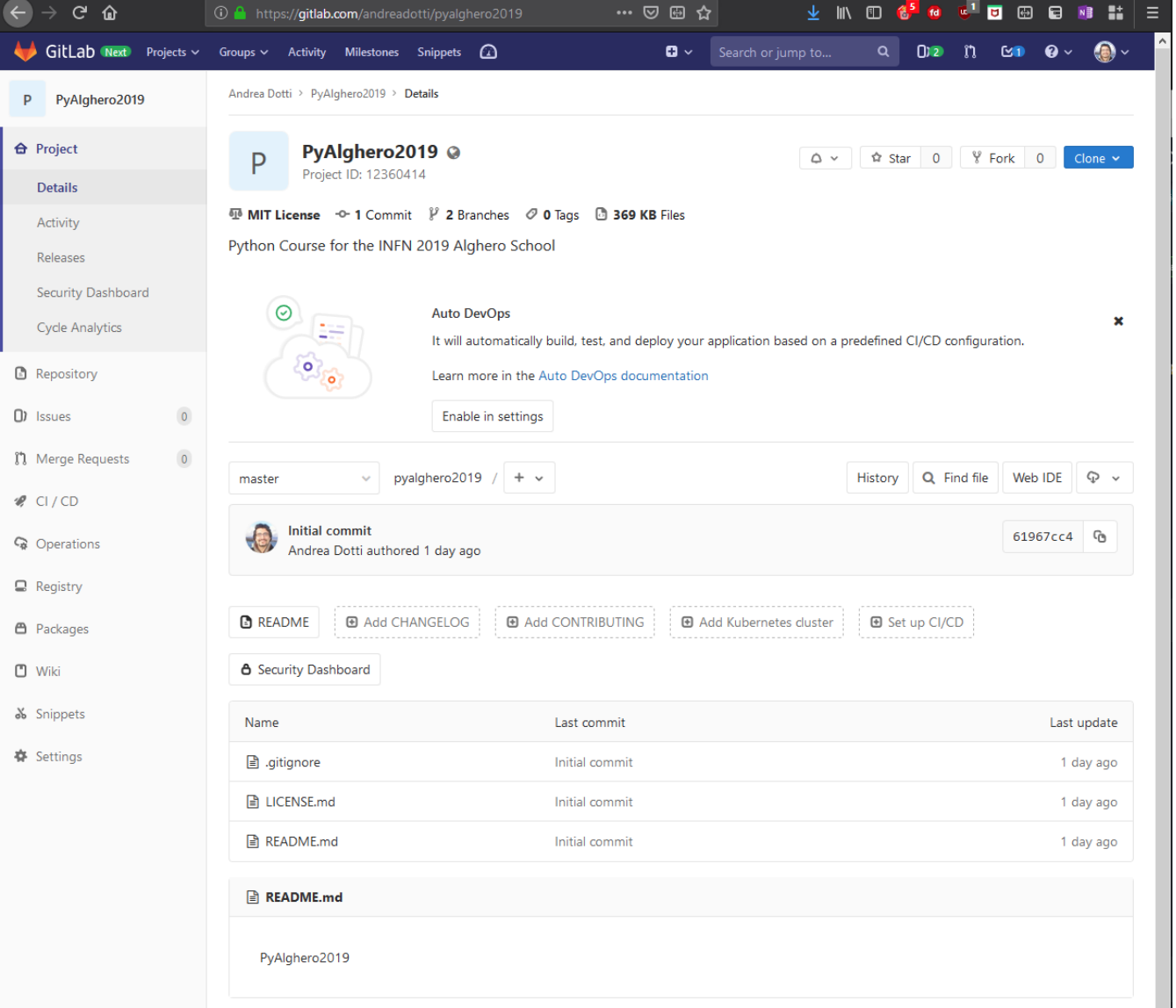
1. Some insights on the programming language itself
2. A feeling of what python is good for (and what is **not** good for)
3. Examples and applications to the problem of data-science

# The How

I will complement, especially for the data-science applications, the ML lectures. Refer to these lectures for specific questions on ML techniques. I will show some basic techniques used in the field of data science

The course is organized in lectures and hands-on.

All the material is available on [gitlab \(https://gitlab.com/andreadotti/pyalghero2019\)](https://gitlab.com/andreadotti/pyalghero2019).



The screenshot shows the GitLab web interface for the project 'PyAlghero2019'. The browser address bar shows the URL 'https://gitlab.com/andreadotti/pyalghero2019'. The page header includes the GitLab logo and navigation tabs: 'Projects', 'Groups', 'Activity', 'Milestones', and 'Snippets'. A search bar is present with the text 'Search or jump to...'. The left sidebar contains a navigation menu with options: 'Project', 'Details', 'Activity', 'Releases', 'Security Dashboard', 'Cycle Analytics', 'Repository', 'Issues' (0), 'Merge Requests' (0), 'CI / CD', 'Operations', 'Registry', 'Packages', 'Wiki', 'Snippets', and 'Settings'. The main content area shows the project details for 'PyAlghero2019' (Project ID: 12360414) by 'Andrea Dotti'. It features a 'Clone' button, 'Star' (0), and 'Fork' (0) buttons. The project description is 'Python Course for the INFN 2019 Alghero School'. Below this, there is an 'Auto DevOps' section with a checkmark icon and the text: 'It will automatically build, test, and deploy your application based on a predefined CI/CD configuration. Learn more in the Auto DevOps documentation. Enable in settings'. The 'Initial commit' section shows 'Andrea Dotti authored 1 day ago' with a commit hash '61967cc4'. Below the commit, there are buttons for 'README', 'Add CHANGELOG', 'Add CONTRIBUTING', 'Add Kubernetes cluster', and 'Set up CI/CD'. A 'Security Dashboard' button is also visible. A table lists the files in the repository:

Name	Last commit	Last update
.gitignore	Initial commit	1 day ago
LICENSE.md	Initial commit	1 day ago
README.md	Initial commit	1 day ago

Below the table, the 'README.md' content is displayed, showing the text 'PyAlghero2019'.



# Python Language



*Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aims to help programmers write clear, logical code for small and large-scale projects. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is often described as a "batteries included" language due to its comprehensive standard library.*

*[From Wikipedia \(https://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)\)](https://en.wikipedia.org/wiki/Python_(programming_language))*



# Interpreted language

The python interpreter reads the input (interactive or in a script) and executes each line of code sequentially. A python distribution comes with a *REPL* (Read Evaluate Print Loop) shell. E.g.:

```
# Technical notes, for this course, use the provided VM and in each  
# new terminal type:  
conda activate course  
python
```

Which will give you:

```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)  
[GCC 7.3.0] :: Anaconda, Inc. on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

the >>> sequence is the python prompt, type a command and see the result, for example:

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

**Hint:**Type `Ctrl+D` to exit, or type `quit()`.

# Interpreted language

Other (python) shells are available, for example to simplify/improve the user-experience, for example IPython ( `ipython` ), or GUIs (jupyter integration).

We will talk about these in the next slide deck.

# High-level

Python strongly abstracts the specific hardware details.

- This means that on one side it makes easier to program (e.g. no explicit memory handling, forget about new/delete)
- On the other side the interpreter must work *more* to translate user input to machine code, this fact together with the interpreted nature of it, makes the code slower compared to a *lower-level programming language* (e.g. C++).

Python is **not** a good language for performance critical applications. Use a lower level language instead.

It is a very good prototype language. E.g. do you want to experiment with a new project, especially one that requires a lot of data analytics? Probably it is a good idea to create a *proof of principle* and a *prototype* in python.

## Is python really slow?

Python usually provides a very rich set of libraries and it supports C-binding allowing for *offloading* computationally heavy parts of the code to optimized routines.

**Hint:** if you know that you have a computationally expensive routine, check if it is available in some libraries, it is probably well optimized (e.g. do **not** write your own linear algebra functions, use `scipy.linalg`). **Hint:** some popular libraries or extension even come with GPU support to speed up the calculations if you have access to the hardware (e.g. `tensorflow` vs `tensorflow-gpu`).

# General Purpose

Python can be used for a rich set of applications:

1. Web applications
1. Data gathering and manipulation
1. Scientific computation
1. Data science

Traditionally, python is considered a *glue* language, used to coordinate programs (possibly written in other languages) and to manipulate the input and output from one to the other (a *pipeline*). Consider it, for this aspect, as a `bash` on steroids.

However the growing number of specialized libraries (e.g. the scientific python stack), powerful visualization tools and rich I/O capabilities, has made it very popular among data scientist and for scientific computations.

# History

- Implementation started in 1989
- Python 2.0 released in October 2000
- Python 3.0 released in December 2008

Python 3.0 is not backward compatible: a program written for python 2 may not run in python 3 out of the box (and vice versa). python3 is getting more and more traction and ver. 2 will retire [soon \(https://pythonclock.org/\)](https://pythonclock.org/):

**Python 2.7 will retire in...**

<b>0</b> Years	<b>7</b> Months	<b>15</b> Days	<b>6</b> Hours	<b>42</b> Minutes	<b>53</b> Seconds
-------------------	--------------------	-------------------	-------------------	----------------------	----------------------

**What's all this, then?**

Python 2.7 [will not be maintained past 2020](#). Originally, there was no official date. Recently, that date has been updated to [January 1, 2020](#). This clock has been updated accordingly. My original idea was to throw a Python 2 Celebration of Life party at PyCon 2020, to celebrate everything Python 2 did for us. That idea still stands. (If this sounds interesting to you, email [pythonclockorg@gmail.com](mailto:pythonclockorg@gmail.com).)

Python 2, thank you for your years of faithful service.  
Python 3, your time is now.

**How do I get started?**

If the code you care about is still on Python 2, that's totally understandable. Most of PyPI's popular packages now [work on Python 2 and 3](#), and more are being added every day. Additionally, a number of critical Python projects have [pledged to stop supporting Python 2 soon](#). To ease the transition, the [official porting guide](#) has advice for running Python 2 code in Python 3.

**If you are starting now with python, go directly to version 3, if you are still at 2, start migration!**

- Some more conservative linux distributions tend to have older python interpreter installed (e.g. CentOS-7 includes python 2.7)
- We'll see how to avoid this problem and use a more recent python interpreter (hint: conda to the rescue)

# Getting Python



- The version number specifies the capabilities of the system (i.e. what the interpreter can understand) and the content of the *python standard library* (that comes with the interpreter)
- A *distribution* is a packaging of an interpreter and a selection of libraries. For example the [official CPython \(https://python.org\)](https://python.org) one, the [PyPy \(https://www.pypy.org/\)](https://www.pypy.org/) -optimized for performances- one, and specialized ones, like [Anaconda \(https://anaconda.org/\)](https://anaconda.org/), are all distributions



PYPY



ANACONDA  
CLOUD

# Python code organization

Code is written in *modules*: a file containing functions, global variables, classes. Differently from C++ and Geant4, usually one module contains more than one class/function all related to each other (it would be like if in Geant4 all classes related to EM Bremsstrahlung are in a single file). *Note*: in python there is no `.hh/ .cc` distinction (no forward declaration), in C++ terminology everything is inlined.

```
In [1]: #Import a single module and use a function in it  
import os  
print(os.uname())  
# IT is possible to import a single function from a module. And (optionally change its name)  
from os import uname as un  
print(un())
```

A *package* is a directory containing one or more modules (or sub-packages). The directory **must** contain a special file `__init__.py` that tells python that the directory is a package. The content of the file can tailor the package behavior (see [here \(https://docs.python.org/2/tutorial/modules.html#packages\)](https://docs.python.org/2/tutorial/modules.html#packages) for details).

```
In [2]: #Import a package  
import numpy  
#Import a module from a package  
import numpy.random as rnd  
print("Call 1:",rnd.binomial(10,0.5))  
#Import a function  
from numpy.random import binomial  
print("Call 2:",binomial(10,0.5))  
#Depending on how the __init__ file is written it is possible to:  
from numpy.random import *  
print("Call 3:",binomial(10,0.5))  
#I do not recommend import * since you may have name clashes...
```

Call 1: 6

Call 2: 6

Call 3: 6

## Since we are here...

Python has a built-in function `help(...)` that can be very useful:

```
In [3]: help(binomial)
```

Help on built-in function binomial:

```
binomial(...) method of mtrand.RandomState instance  
  binomial(n, p, size=None)
```

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters,  $n$  trials and  $p$  probability of success where  $n$  an integer  $\geq 0$  and  $p$  is in the interval  $[0,1]$ . ( $n$  may be input as a float, but it is truncated to an integer in use)

Parameters

-----

```
n : int or array_like of ints  
    Parameter of the distribution,  $\geq 0$ . Floats are also accepted,  
    but they will be truncated to integers.  
p : float or array_like of floats  
    Parameter of the distribution,  $\geq 0$  and  $\leq 1$ .  
size : int or tuple of ints, optional  
    Output shape. If the given shape is, e.g., ((m, n, k)), then  
    m * n * k samples are drawn. If size is None (default),  
    a single value is returned if n and p are both scalars.  
    Otherwise, np.broadcast(n, p).size samples are drawn.
```

Returns

-----

```
out : ndarray or scalar  
    Drawn samples from the parameterized binomial distribution, where  
    each sample is equal to the number of successes over the  $n$  trials.
```

See Also

-----

```
scipy.stats.binom : probability density function, distribution or  
    cumulative density function, etc.
```

Notes

-----

The probability density for the binomial distribution is

```
.. math:: P(N) = \binom{n}{N} p^N (1-p)^{n-N},
```

Documentation is written together with the code as comments. If you follow some specific rules (see [here \(https://docs.python-guide.org/writing/documentation/\)](https://docs.python-guide.org/writing/documentation/)) you get pretty nicely formatted documentation ([tools exist to create documentation from code \(https://wiki.python.org/moin/DocumentationTools\)](https://wiki.python.org/moin/DocumentationTools)):

```
In [1]: def foo():  
        '''  
        This is the documentation.  
  
        It is written as multi-line comment  
        '''  
        # This is a single line comment  
        return  
  
help(foo)
```

```
Help on function foo in module __main__:
```

```
foo()  
    This is the documentation.  
  
    It is written as multi-line comment
```

**Let's get started**

# Firing-up the python interpreter

- Type `python` to enter the interactive python interpreter. `quit()` (or `Ctrl+d`) to quit
- If you want to execute a script (a file containing some python code) type at the shell prompt:

```
python myscript.py
```

- You can execute python commands without entering the python interpreter:

```
python -c "print(3+2)"
```

- Modules may also be executed as scripts, in such a case:

```
python -m os
```

- But you can enter interactive mode after importing a module, or executing commands:

```
python -i -m os
```

The `-i` should come before `-m`. Whatever follows the name of the module is passed as arguments to it!



## A note on encoding and scripts

Scripts, by default, are encoded in UTF-8 (terminal and font must support special character if you use them). You can specify a different encoding adding this special comment line as the **first line** in your `.py` file:

```
# -*- coding: cp1252 -*-
```

If you use *UNIX shebangs* this line can be put as second, as in:

```
#!/usr/bin/env python  
# -*- coding: cp1252 -*-
```

# The first python session: variables, types and data structures

Let's start the python interpreter and define some variables:

```
In [5]: #This is a comment
a = 3    #This is an integer, this is a comment on the same line
b = 2.3  #This is a float (actually a double 64-bits in C++)
print('a is of type {}, b is of type {}'.format(type(a), type(b)))
print('a value is {}, while b's is {}'.format(a, b))
```

```
a is of type <class 'int'>, b is of type <class 'float'>
a value is 3, while b's is 2.3
```

- = is the assignment operator: assigns the rhs to the variable on the lhs (e.g. `n = 3`)
- Arithmetic operations are the usual ones: +, -, \*, /
- **Important:** Division / returns a float (e.g. `5 / 2` returns 2.5) in python3 but not always in python2 (e.g. `5 / 2` returns 2, while `5. / 2.` returns 2.5). *Remember this is you are porting code from 2 to 3!*
- In python3 // is the floor operation (e.g. `5 // 2` returns 2).
- % is the remainder operator
- \*\* is the exponent operator

## Talking of variables...

Python is an inferred, dynamically and strongly typed language. This means:

- Variable types are guessed by the interpreter (like using everywhere `auto` in C++11)
- The same variable name can be *re-used* for another type
  - IMPORTANT: the variable name is just a name, pointing to an underlying object, that has a specific type

```
In [6]: a = 1 #a is an int
print(type(a))
a = "abc" #Now a points to a string
print(type(a))
```

```
<class 'int'>
<class 'str'>
```

Memory is automatically managed, and you can think of everything like an instance of a class:

```
In [7]: a = 3
        #Print memory address id(..) in hexadecimal format hex(...)
        print(hex(id(a)))
```

0x10d4985e0

```
In [8]: a = 3.2
        print(hex(id(a)))
```

0x11142af18

Note the two addresses are different, even if the variable name is the same, python interpreter takes care of cleaning the memory, when not needed anymore.

**Side Note:** In an *interactive session*, you do not need to write `print` to show the return value of the last statement. E.g. writing `a` directly at the interpreter is equivalent to `print(a)`.

# String type

Strings are enclosed in '...' or "...". Multi-line literals are allowed using """...""" / '''...''' as in the following example:

```
In [9]: str1 = "A string"
str2 = 'Another string'
print(str1)
print(str2)
print("""First line
Second line
Third line
""")
```

```
A string
Another string
First line
Second line
Third line
```

String manipulation is supported with + (concatenation) and \* (repetition), as in:

```
In [10]: a = 'First string, '  
b = 'Second string'  
print(a+b)  
print(3*"abc")
```

First string, Second string

abcabcabc



Special character can be escaped with `\` (e.g. `\n` to produce a new line). Unless the string is a *raw string*, in such a case the `\` are interpreted as character:

In [11]:

```
str1 = 'Special \t character'
str2 = r'A raw string with special \n character' #Note the r'...'
print(str1)
print(str2)
```

```
Special      character
A raw string with special \n character
```

## Strings can be indexed and sliced:

```
In [12]: message = "A message"
         #First and third characters
         print( message[0] )
         print( message[2] )
         #Last and second to last characters
         print( message[-1] )
         print( message[-2] )
         #Substring from 4rd to 5th characters (0-indexed up to 6 excluded)
         print( message[3:6] )
         #Substring from beginning to 2nd character
         print( message[:3] )
         #Substring from 6th to the end
         print( message[5:] )
```

```
A
m
e
g
ess
A m
sage
```

## Side note: f-strings

Python3 has a special f(ormatted)-string construct:

```
In [13]: variable = 3
         fstring = f'An f-string: {variable}, look at me!'
         print(fstring)
```

```
An f-string: 3, look at me!
```

In a f-string, the `{...}` characters are replaced with the *value* of the variable name.

# Lists, tuples and dictionaries

There is a number of *containers* data structures:

- list are read/write containers of objects that can be indexed, sliced and much more

```
In [14]: alist = [ 1, 2, 3, 4]
list3 = [1, 3.3, "aaa", 3+3j ]
```

- tuples are read-only compounds objects, useful to write compact code (packing/unpacking)

```
In [15]: atuple = (1, 2, 3, "aaa") #() are actually not needed
(first, second, third, fourth) = atuple #also here () are not needed
```

- dictionaries are collections of *key/value* pairs

```
In [16]: dict1 = { 'value1' : 3.2 , 'value2' : "msg", 'value3' : [1,2,3] }
dict2 = dict(value1=3.2, value2="msg", value3=[1,2,3])
dict3 = { #python can accept constructs on multiple lines
    'value1' : 3.2,
    'value2' : "msg",
    'value3' : [1,2,3]
}
```

## More on lists

Lists are probably the most useful data structure in python, let's see few more details, starting from some common methods:

```
In [17]: a = [ 1,2,3 ]  
         a.append(4)  
         a
```

```
Out[17]: [1, 2, 3, 4]
```

```
In [18]: b = [5,6,7]  
         a.extend(b)  
         a
```

```
Out[18]: [1, 2, 3, 4, 5, 6, 7]
```

```
In [19]: last = a.pop()
         print(last)
         a
```

7

Out[19]: [1, 2, 3, 4, 5, 6]

```
In [20]: del a[3]
         a
```

Out[20]: [1, 2, 3, 5, 6]

```
In [21]: a.remove(5)
         a
```

Out[21]: [1, 2, 3, 6]

A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses:

```
In [22]: [ x**2 for x in range(10) ]
```

```
Out[22]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
In [23]: [ x**2 for x in range(10) if x%2 == 0 ]
```

```
Out[23]: [0, 4, 16, 36, 64]
```

I use often the following two:

```
In [24]: list(enumerate(["a","b","c"]))
```

```
Out[24]: [(0, 'a'), (1, 'b'), (2, 'c')]
```

```
In [25]: list(zip(["a","b","c"],[10,20,30]))
```

```
Out[25]: [('a', 10), ('b', 20), ('c', 30)]
```



Lists can be transversed with:

```
In [26]: alist = [1, 3, 5, 7]
         for e in alist:
           #do something
           pass
```

```
In [27]: alist = [0,1,2,3,4,5,6,7,8,9,10]
alist[2]
```

```
Out[27]: 2
```

```
In [28]: alist[-1]
```

```
Out[28]: 10
```

```
In [29]: alist[2:4]
```

```
Out[29]: [2, 3]
```

```
In [30]: alist[2:8:2]
```

```
Out[30]: [2, 4, 6]
```

# More on dictionaries

```
In [31]: d = { "a":3, "b":5 }  
         assert(d["a"]==3)  
         d.update({"c":6,"a":2})  
         d
```

```
Out[31]: {'a': 2, 'b': 5, 'c': 6}
```

```
In [32]: #This has changed between python2 and python3  
         for k,v in d.items():  
             print(k,"is",v)
```

```
a is 2  
b is 5  
c is 6
```

```
In [33]: d = { x: x**2 for x in range(5)}  
d
```

```
Out[33]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

```
In [34]: d = dict(a=1, b=2, c=3)  
d
```

```
Out[34]: {'a': 1, 'b': 2, 'c': 3}
```

Finally a useful data structure is `set`, a collection of unique values. I use this data structure only exclusively to get the list of the unique elements in a list:

```
In [35]: a = list(range(5))  
a.extend(range(5))  
a
```

```
Out[35]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

```
In [36]: s = set(a)  
s
```

```
Out[36]: {0, 1, 2, 3, 4}
```

# Control flow

# if statement

```
In [37]: x = 2.3
         if x > 10:
             print("x is large")
         elif x > 5:
             print("x is not so large")
         else:
             print("x is small")
```

```
x is small
```

## Since we are here

Note the indentation: In C++ you use curly brackets `{ }` to delimit code blocks. In python you use indentation (one of the basic [principles of python \(https://www.python.org/dev/peps/pep-0020/\)](https://www.python.org/dev/peps/pep-0020/) is code readability). Python interpreters and IDEs will help you with code indentation, but if you do not respect it, you will get errors or, worse, wrong behavior. A command line tool like `pylint` can help you check a module/script respects code standards. It is worth trying it out if you need to share the code with someone else.

# for statement

```
In [38]: alist = ["one", "two", "three"]
message = ""
for element in alist:
    message = message + "," + element
#Python is coincide, the same can be achieved using the method
#join of str object:
message = ",".join(alist)
```



break and continue have the same behavior as in C/C++, in addition for supports the else clause, as in the example:

```
In [2]: prime_numbers = list()
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            # n is not a prime number, we have found a factor
            break
    else:
        #Note here else is alligned with for, not the if!
        # loop fell through without finding a factor
        prime_numbers.append(n)

#Python is also functional (this is not a good idea, do you know why?):
prime_numbers = filter(
    lambda n: 0 not in map(lambda x: n%x , range(2,n)),
    range(2,10)
)
```

## pass statement

This statement *does nothing*: it is useful when the syntax requires something, but there is nothing to do, usually used to create an empty class or a stub for a method to be implemented in the future:

In [40]:

```
class MyClass:  
    #TODO: Stop procrastinating and get to work!  
    pass
```

# Input/Output

# Writing formatted strings

We have already seen f-strings, r-strings and normal strings:

```
In [41]: str1 = "A string"
str2 = r"A raw string, with special characters like \n"
somespecialval = 3
#Error if somespecialval is not already defined
str3 = f"An f-string {somespecialval}"
```

This is the old "C-printf" style, still valid to format a string:

```
In [42]: str4 = "Some string with a value inside %d"%3.2
str4
```

```
Out[42]: 'Some string with a value inside 3'
```

Strings can be formatted via the `.format` method (preferred):

```
In [3]: "A string with a first {} and a second {} value".format(3,4)
```

```
Out[3]: 'A string with a first 3 and a second 4 value'
```

```
In [44]: str1 = "Look here: {0}, {1}".format(1,2)
str2 = "Look here: {1}, {0}".format(1,2)
print(str1)
print(str2)
```

```
Look here: 1, 2
```

```
Look here: 2, 1
```

```
In [4]: from math import pi
"A formatted string {0:.3f}".format(pi)
```

```
Out[4]: 'A formatted string 3.142'
```

An interesting use case for `.format` is if you need to print some values from a dictionary:

```
In [46]: d = dict(first=1., second="a", third=3)
         "A use case where I print only part of the info {first} and {second}".format(**d)
```

```
Out[46]: 'A use case where I print only part of the info 1.0 and a'
```

# Reading files

Let's focus here on reading text files. If you need to do data manipulation there are other formats you may consider that usually come with a dedicated I/O module (e.g. Excel, to matlab, to HDF5).

```
In [47]: inf = open("afile.txt", "r")  
inf.readline()
```

```
Out[47]: 'A line\n'
```

```
In [48]: inf.readline()
```

```
Out[48]: 'Another line\n'
```

```
In [49]: inf.readline()
```

```
Out[49]: ''
```

```
In [50]: inf.close()
```

It is good practice to use the following construct to operate on files (because it is safer in case of errors), after the `with` block, the file is closed automatically:

```
In [51]: with open("afile.txt", "r") as f:
          for line in f:
              print(line)
```

A line

Another line



Often in physics we want to read a file containing numerical values (note that we will see more efficient ways to do this):

```
In [52]: v1s = list()
v2s = list()
with open("afile.csv","r") as f:
    for line in f:
        v1, v2 = line.split(",")
        v1s.append(float(v1))
        v2s.append(float(v2))
print(v1s)
print(v2s)
```

```
[10.0, 3.0]
```

```
[20.0, 5.2]
```

# Writing data to files

A file object, opened with 'w' option has the method `.write`:

```
In [53]: with open("outfile.txt", "w") as f:  
         f.write("Some text\n")
```

A python module called `pickle` can be used to store/read python data structures:

```
In [54]: import pickle
d = dict(key1="ABC", key2=3.2, key3=[1,2,3,4])
with open("outfile.pkl","wb") as f: #b is for binary, more efficient
    pickle.dump(d,f)
```

```
In [55]: # Read data back:
with open("outfile.pkl","rb") as f:
    d_read = pickle.load(f)
d_read
```

```
Out[55]: {'key1': 'ABC', 'key2': 3.2, 'key3': [1, 2, 3, 4]}
```

Finally JSON is an internet standard for data exchange. It is probably worth to note that it is well supported:

```
In [56]: d = dict( key1="ABC", key2=3.2, key3=[1,2,3,4])
import json
json.dumps(d) #the 's' here means "dump to string"
```

```
Out[56]: '{"key1": "ABC", "key2": 3.2, "key3": [1, 2, 3, 4]}'
```

```
In [57]: with open("outfile.json","w") as f:
        json.dump(d,f)
```

```
In [58]: with open("outfile.json","r") as f:
        d_read_j = json.load(f)
        assert(isinstance(d_read_j, dict))
        d_read_j
```

```
Out[58]: {'key1': 'ABC', 'key2': 3.2, 'key3': [1, 2, 3, 4]}
```

# Defining functions

A function is defined with the following syntax `def fun_name(arguments):`, the return value is implicitly determined by the `return` statement.

**WARNING** different code paths (e.g. `if - else` statements) could make a function return different data types

```
In [59]: def foo( arg1, arg2 ):
          """A simple function"""
          return arg1+arg2

#Note the return type is dynamic:
assert( isinstance(foo(3,2), int) )
assert( isinstance(foo("a","b"), str) )
```

Functions can have default arguments, that can be omitted when calling the function:

```
In [60]: #Note the special value of arg2, of type NoneType
def foo( arg1, arg2=None ):
    """A simple function"""
    if arg2:
        return arg1+arg2
    else:
        return arg1
```

Functions can be called with positional and keyword arguments:

```
In [61]: def foo( arg1, arg2='Value', arg3=None):
          pass

foo(100) # 1 positional argument, using 2 defaults
foo(100, "two", 3) # 3 positional arguments
foo(arg2="two", arg1=20) #2 keyword arguments, note now the order does not matter
```

A special signature of functions contains (one or both) parameters with a name preceded by \* or \*\*. Better explained with an example:

```
In [62]: def foo(arg1, *args, **kwargs):
          pass

foo(3, "two", "three", key1 = "value1", key2 = "value2")
# Foo's parameter args is the list ["two","three"] and kwargs is a dict { "key1":
value1", "key2":"value"}
```

This is very useful to write functions that accepts an arbitrary number of arguments. E.g:

```
In [63]: def do_something_complex(data, **conf):
          """conf is a 'configuration' dictionary"""
          if 'method' in conf and conf['method'] == 'linear':
              pass
```



# Lambda expressions

In python functions are first class citizens, they are objects that can be passed as argument to other functions:

```
In [5]: def foo(a, b):  
        return a+b  
  
        def bar(fun, c):  
            return fun(c,c)  
  
        bar(foo,2)
```

Out[5]: 4

Lambda expressions can be used to create anonymous functions, usually when these are pretty small:

```
In [65]: def bar(fun, c):  
         return fun(c,c)  
  
         #Let's make a variant:  
         fun2 = lambda a,b: a+b+2  
         print(type(fun2))  
         bar(fun2, 3)
```

```
<class 'function'>
```

```
Out[65]: 8
```

Lambdas are useful in combination with many built-in functions like `map`, `reduce` and `filter`. For example `map` accepts as arguments a function and an iterable. It applies the function to all elements, returning a new iterable. For example:

```
In [66]: power_of_two = lambda x: 2**x
inp = range(10) # An iterable of all numbers 0..9
out = map( power_of_two, inp)
out
```

```
Out[66]: <map at 0x11de6a2e8>
```

Note that `out` is not a list, but an instance of an object (an iterable), let's get the list of the values:

```
In [67]: list(out)
```

```
Out[67]: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

The extensive use of iterables in python3 is probably its most confusing feature, especially if you have experience with python2.

## On iterables

In python3 many function now returns an iterable instead of the collection. An iterable is more efficient than the collection itself, but once consumed, it becomes *empty*:

```
In [68]: out = map(power_of_two, inp)
         for e in out:
             print(e, end=",")
         print("\n once more:")
         for e in out:
             print(e, end=",") #Never comes here because out has been consumed
```

```
1,2,4,8,16,32,64,128,256,512,
once more:
```

```
In [69]: #If you need to use more than once an iterable you can
         # 1. convert it to the underlying data type:
         out = map(power_of_two, inp)
         out = list(out)
         assert( len(out) == len(inp) )
         # 2. Or "clone" the iterable itself:
         from itertools import tee
         out = map(power_of_two, inp)
         it1, it2 = tee(out)
```

# Annotating functions

In python3 it is possible to *document* functions specifying the *expected* type of arguments and return type. Also local variables can be annotated. However the interpreter will allow a call with *wrong* types, this is purely for documentation/readability:

```
In [70]: def some_complex_function( arg1: int, arg2: str) -> bool:
         """An annotated function

         The first argument should be an integer, and the second a string.
         The function returns a boolean.
         Annotations are used to actually avoid writing this documentation..."""
         result: bool = str(arg1)==arg2
         return result

         assert(some_complex_function(3, "3"))
         assert(some_complex_function("a", "a")) #Does not fail because str("3") == "3", but
         type(arg1)!=int
```

Classes can be annotated too.

# Classes

Python supports object oriented programming style. However there are few differences with respect to C++:

1. All data members are public
2. All methods are virtual

```
In [71]: class MyClass:
          """Class documentation"""
          i = 3 #This is a class data member.
          def foo(self): #Note the 'self' keyword
              """This is a class method"""
              print("Called method foo!")

          m = MyClass()
          m.foo()
```

Called method foo!

```
In [72]: m.i
```

```
Out[72]: 3
```

Class instances can be dynamically extended:

```
In [73]: m.another_val = 3
```

```
In [74]: m.another_val
```

```
Out[74]: 3
```



Constructors exist and is the `__init__` method, these are used to initialize instance data-field. There is (usually) no need for a destructor because memory is not managed explicitly (see [here \(https://docs.python.org/3.7/tutorial/classes.html\)](https://docs.python.org/3.7/tutorial/classes.html) for more).

```
In [75]: class MyClass:
          """Class documentation"""
          i = 3
          def __init__(self, val):
              """Consturctor with one parameter"""
              self.value = [val] #An instance data member

          m = MyClass(3.14)
          m.value
```

```
Out[75]: [3.14]
```

Inheritance exists, with the expected behavior:

```
In [76]: class Derived(MyClass):  
         """A derived class"""  
         def __init__(self):  
             MyClass.__init__(self,3.14)#Explicitly call the base class constructor  
  
         m = Derived()  
         m.value
```

```
Out[76]: [3.14]
```

Think twice before writing your own class: if you want just a container for your (heterogeneous) data, a `dict` is what you are looking for (maybe with few helper functions). At least this is 90% of the use-cases for a G4 user. E.g. instead of:

```
In [77]: class Electron:
          m = 511
          q = -1
          name = "electron"
          family = "lepton"
          def __init__(self,energy):
              self.energy = energy
```

Use:

```
In [78]: e1 = dict(m=511,q=-1,name="electron",family="lepton",energy=3)
          e2 = e1.copy()
          e2.update( dict(energy=2.2))
```

Private methods/data fields do not exist. However a convention exists, if the field name starts with an `_` character, this is considered *implementation details*:

```
In [79]: class MyClass:
          """A class"""
          i = 3
          _cache = None #Something internal, e.g. a cache
          def __init__(self, val):
              """Constructor with one parameter"""
              self.value = val
          def _update(self):
              #some heavy calculation and store an intermediate number
              _cache = 3.14
          def do_stuff(self):
              if not self._cache:
                  self._update()
              #Rest
```

Finally in python3 there is a useful Enum class to provide enumeration functionalities:

```
In [80]: from enum import Enum

class ParticleName(Enum):
    #By convention use all capital letters
    ELECTRON = "electron"
    PROTON = "proton"

class ProcessType(Enum):
    EM = 2
    HAD = 4

ParticleName.ELECTRON
```

```
Out[80]: <ParticleName.ELECTRON: 'electron'>
```