

Python Lecture 1 – Introduction

★ Bibliography:

<https://www.python.org/doc/>

<http://docs.python.it/>

and much more available in internet

★ Learning Materials:

[https://github.com/gtaffoni/Learn-Python/tree/master/
Lectures](https://github.com/gtaffoni/Learn-Python/tree/master/Lectures)

https://github.com/bertocco/abilita_info_units_1920

The python language



- ★ Python is an interpreted language.
 - **old-style interpreted languages** (like bash): the code is saved in the same format that you entered
 - **new-style interpreted languages** (like python): the code is pre-processed to produce a bytecode (similar to machine language) and then executed by the interpreter (virtual machine).
- ★ **Code portability**: means run on hardware/software platforms different from which used to develop the code.
 - Python is portable if the interpreter is available on the target platform

Work Environment Setup



- ★ Pipenv & Virtual Environments

- ★ The next step is to install Pipenv, so you can install dependencies and manage virtual environments.

- ★ A Virtual Environment is a tool to keep the dependencies required by different projects in separate places, by creating virtual Python environments for them. It solves the “Project X depends on version 1.x but, Project Y needs 4.x” dilemma, and keeps your global site-packages directory clean and manageable.

- ★ For example, you can work on a project which requires Django 1.10 while also maintaining a project which requires Django 1.8.

Work Environment Setup in Ubuntu



- ★ Verify if python is already installed
- ★ Install python3, numpy, scipy, matplotlib (next lessons)

In this course we will use python3

Install Python3 virtualenv on Ubuntu (1)



Step 1: Update your repositories

```
sudo apt-get update https://naysan.ca/2019/08/05/install-python-3-virtualenv-on-ubuntu/
```

Step 2: Install pip for Python3

```
sudo apt-get install build-essential libssl-dev libffi-dev python-dev  
sudo apt install python3-pip
```

Step 3: Use pip to install virtualenv

```
sudo pip3 install virtualenv
```

Step 4: Launch your Python3 virtual environment, here the name of my virtual environment will be env3

```
virtualenv -p python3 env3
```

Install Python3 virtualenv on Ubuntu (2)



Step 5: Activate your new Python3 environment. Two ways to do this
. env3/bin/activate # or source env3/bin/activate does exactly the same

you can make sure you are now working with Python3

python3 – version

this command will show you what is going on: the python executable
you are using is now located inside your virtualenv repository

which python

Step 6: code your stuff

Step 7: done? leave the virtual environment

deactivate

<https://naysan.ca/2019/08/05/install-python-3-virtualenv-on-ubuntu/>

Anaconda (<https://www.anaconda.com/>) is a free-open source distribution of the Python programming language for scientific computing, which aims to simplify package management and deployment. The Anaconda distribution includes data-science packages suitable for Windows, Linux, and macOS.

- I. The default installation of Anaconda2 includes Python2.7 and Anaconda3 includes Python3.7;
- II. Anaconda Navigator is a desktop GUI included in Anaconda distribution that allows users to manage python packages and to launch applications.
 - The following applications, among others, are available by default in Anaconda Navigator:
 - Jupyter Notebook -- a web-based interactive computational environment for creating Jupyter notebook documents;
 - Spyder -- open source cross-platform Integrated Development Environment (IDE) for scientific programming in Python language.

The python interpreter



- ★ Python is an interpreted language.

The python interpreter can be used:

- Interactively to interpret a single command or little sets of commands;
- Interactively to interpret set of commands collected in a file *.py

In this case, the interpreter produces files (*.pyc), as intermediate product, and interpret row by row the command present in the file.

All similarly to bash except for the bytecode production.

- ★ To fire up the Python interpreter, open up your terminal/console application, and type python3.

You should see something like this:

```
[bertocco@firiell ~]$ python3
```

```
Python 3.7.5 (default, Nov 20 2019, 09:21:52)
```

```
[GCC 9.2.1 20191008] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

- ★ Use quit() or Ctrl-D (i.e. EOF) to exit

Define a variable



A Python variable is a reserved memory location to store values.
The variable must be defined assigning it a value:

```
>>> a=3 #works
```

```
>>> b = 3 #works
```

```
>>> a
```

```
3
```

```
>>> c # does not work
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'c' is not defined

Rules for Python variable names: A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume):

- a variable name must start with a letter or the underscore character;
- a variable name cannot start with a number;
- a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _);
- white spaces and signs with special meanings, as "+" and "-" are not allowed;
- variable names are case-sensitive (age, Age and AGE are three different variables).

Examples on variable names



- a variable name must start with a letter or the underscore character

```
>>> alfa = 1
```

```
>>> alfa
```

```
1
```

```
>>> beta =10
```

```
>>> _pippo = 1
```

```
>>> _pippo
```

```
1
```

- a variable name cannot start with a number

```
>>> 1cane=3
```

```
1cane=3
```

```
^
```

```
SyntaxError: invalid syntax
```

```
^
```

```
SyntaxError: invalid syntax
```

- a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)

```
>>> urca! = 10
```

```
File "<stdin>", line 1
```

```
urca! = 10
```

```
^
```

```
SyntaxError: invalid syntax
```

- white spaces and signs with special meanings, as "+" and "-" are not allowed

```
>>> a-b=0
```

```
File "<stdin>", line 1
```

```
SyntaxError: can't assign to operator
```

```
>>> a$b='sara'
```

```
File "<stdin>", line 1
```

```
a$b='sara'
```

```
^
```

```
SyntaxError: invalid syntax
```

Variable types



Each variable in python has a type.

The variable type is not pre-defined, it is resolved at run-time.

In C programs variable declaration is:

```
int a = 10;  
float b = 3.4;  
char str[] = "pippo";
```

In python variable declaration is:

```
a = 10  
b = 3.4  
str = "pippo"
```

In python you can do (referring to the previous example):

```
a = b    # because type is dynamically resolved, i.e. at run-time
```

In python you cannot do (referring to the previous example):

```
new_val = a + str    # because each variable has a type  
                     # (the language is strongly typed)
```

List of some different variable types



```
x = 123                # integer
x = 3.14               # float
x = "hello"           # string
x = [0,1,2]            # list
x = (0,1,2)            # tuple
x = open('hello.py', 'r') # file
x = {1: 'apple', 2: 'ball'} # dictionary
```

You can also assign a single value to several variables simultaneously multiple assignments.

Variable a,b and c are assigned to the same memory location, with the value of 1

```
a = b = c = 1
```

Example: dynamically resolved types



```
>>> a = 10
>>> b = 3.4
```

It is possible the assignment:

```
>>> a = b
>>> a
3.4
```

```
>>> a = 3
>>> b = 3.4
```

It is possible the sum:

```
>>> a+b
6.4
```

Example: strongly typed variables



```
>>> a=3          # a is resolved as an integer
>>> str = 'mah'   # str is resolved as a string
```

```
>>> a + str
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

You can not sum a string with an integer.

Python is dynamically and strongly typed



Python is dynamic typed because variable values are checked during execution.

Python is strongly typed because “at runtime” it doesn’t allow implicit conversions.

Type casting



Casting is the operation to convert a variable value from one type to another.

In Python, the casting must be done explicitly with functions such as `int()` or `float()` or `str()`.

Example:

```
>>> x = '100'
```

```
>>> y = '-90'
```

```
>>> print(x + y)
```

```
100-90
```

```
>>> print(int(x) + int(y))
```

```
10
```

That was the `int()` function. There's also another very common one, which is `float()`, which does basically the same thing:

```
>>> print(float(x) + float(y))
```

```
10.0
```

Available type casting



`int(x [,base])` Converts `x` to an integer base specifies the base if `x` is a string.

`long(x [,base])` Converts `x` to a long integer base specifies the base if `x` is a string.

`float(x)` Converts `x` to a floating-point number.

`complex(real [,imag])` Creates a complex number.

`str(x)` Converts object `x` to a string representation.

`repr(x)` Converts object `x` to an expression string.

`eval(str)` Evaluates a string and returns an object.

`tuple(s)` Converts `s` to a tuple.

`list(s)` Converts `s` to a list.

`set(s)` Converts `s` to a set.

`dict(d)` Creates a dictionary. `d` must be a sequence of (key,value) tuples.

`frozenset(s)` Converts `s` to a frozen set.

`chr(x)` Converts an integer to a character.

`unichr(x)` Converts an integer to a Unicode character.

`ord(x)` Converts a single character to its integer value.

`hex(x)` Converts an integer to a hexadecimal string.

`oct(x)` Converts an integer to an octal string.

Be careful in type casting



Type casting is a bit tricky operation.

Example: the string '100.0' can be converted in a float, but not in an integer

```
>>> x = '100.0'
```

```
>>> print(float(x))
```

```
100.0
```

```
>>> print(int(float(x)))
```

```
100
```

```
>>> print(int(x))
```

Traceback (most recent call last):

File "<stdin>", line 1, in ?

ValueError: invalid literal for int(): 100.0

How to comment code



There are two ways to write comments in python code:

- Single line comments

```
# this is a single line comment
```

```
a=13 # this is a single line comment also
```

- More line comments

```
''' You can write a multiple line comment using  
three single quotes'''
```

```
""" This is another way to write a multiple line comment  
using three double quotes"""
```

First python scripts



script_sum.py

```
#!/usr/bin/python3
```

```
one = 1
```

```
two = 2
```

```
three = one + two
```

```
print(three)
```

Launch the script (first way):

Add execution permissions:

```
$ chmod +x script_sum.py
```

Execution:

```
$ ./script_sum.py
```

script_hello.py

```
hello = "hello"
```

```
world = "world"
```

```
helloworld = hello + " " + world
```

```
print(helloworld)
```

Launch the script (second way):

Use the interpreter for execution:

```
$ python3 script_hello.py
```

Note: any var PATH or PYTHONPATH must contain 'python' command location

PYTHONPATH is an environment variable which you can set to add additional directories where python will look for modules and packages.

Indentation and blocks of code



In python blocks of code (set of instructions to be run as a block, like functions) are denoted by line indentation not by curly braces (as in C or Java, for example).

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Example (indentation.py):

```
x = 2.3
```

```
y = 1.2
```

```
# test is a function testing if two numbers are equal or one greater then the other
```

```
def test(x,y):
```

```
    if x==y:
```

```
        print(' the two number are equals')
```

```
    elif x > y:
```

```
        print(' the first number is the greater')
```

```
    else:
```

```
        print(' the last number is the greater')
```

```
print('now testing : ', x, y)
```

```
test(x,y)
```

```
for I in range(2,5,1):
```

```
    for J in range(5,1,-1):
```

```
        print('now testing : ', I,J)
```

```
        test(I,J)
```

Exercise: indentation and blocks of code



Try the two examples in the previous slide writing them on a file and running the file.

Functions: defining a function



A function is a block of code which only runs when it is called and can be run repetitively.

Defining a Function:

- Function blocks begin with the keyword **def** followed by the function name and parentheses ().
- Any input parameter or argument should be placed within these parentheses.
- The first statement of a function can be an optional statement (the documentation string of the function or docstring)
- The code block within every function starts with a colon (:) and is indented.
- The statement `return [value]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

Syntax:

```
def function_name( parameters ):
    "function_docstring"
    instruction_set
    .....
    return [value]
```


Functions: calling a function



Defining a function you specify:
name, parameters and the structure of the block of code.

Given the basic structure of the function, it can be executed by calling it from another function, from a python script or directly from the python prompt.

Example:

```
#!/usr/bin/python3
```

```
# Function definition
```

```
def print_string( str ):  
    "This prints a passed string into this function"  
    print(str)  
    return
```

```
# Function call
```

```
print_string("I'm first call to user defined function!")  
print_string("Again second call to the same function")
```

Function Arguments



You can call a function by using the following types of formal arguments:

- **Required arguments:**
the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.
- **Keyword arguments:**
are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters
- **Default arguments:**
are arguments that assume a default value if a value is not provided in the function call for these arguments
- **Variable-length arguments:**
when a function has to be processed for more arguments than the specified in the function, the variable-length arguments are used. They are not named in the function definition, unlike required, and default arguments (next lesson).

Example: Required Arguments



```
#!/usr/bin/python3
```

```
# Function definition is here
```

```
def my_print( str ):
```

```
    "This function prints the passed string"
```

```
    print(str)
```

```
# Now you can call my_print function
```

```
my_print()
```

When the code is executed, the following is the result:

Traceback (most recent call last):

File "my_print.py", line 11, in <module>

my_print();

TypeError: my_print() takes exactly 1 argument (0 given)

Example: Keyword Arguments



```
#!/usr/bin/python3
```

```
# Function definition is here
```

```
def print_info( name, age ):
```

```
    "This prints the info passed as parameter"
```

```
    print("Name: ", name)
```

```
    print("Age ", age)
```

```
# Now you can call print_info function
```

```
print_info( age=30, name="Silvia" )
```

Example: Default Arguments



```
#!/usr/bin/python3

# Function definition is here
def print_user_info( name, age = 35 ):
    "This prints a passed info into this function"
    print("Name: ", name)
    print("Age ", age)

# Now you can call print_user_info function
print_user_info( age=30, name="Silvia" )
print_user_info( name="Silvia" )
```

When the script is executed, the result is:

```
Name: Silvia
Age 30
Name: Silvia
Age 35
```

Scope of variables



All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The **scope** of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

- Global variables
- Local variables

Global vs. local variables



Variables that are defined inside a function body have a **local scope**, those defined outside have a **global scope**.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Example (try):

```
#!/usr/bin/python3
```

```
total = 0 # This is global variable.
```

```
# Function definition is here
```

```
def sum( arg1, arg2 ):
```

```
    # Add both the parameters and return them.
```

```
    total = arg1 + arg2; # Here total is local variable.
```

```
    print("Inside the function local total : ", total)
```

```
# Now you can call sum function
```

```
sum( 10, 20 );
```

```
print("Outside the function global total : ", total)
```

Output calling the script:

Inside the function local total : 30

Outside the function global total : 0

Global vs. local variables



Variables that are defined inside a function body have a **local scope**, those defined outside have a **global scope**.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

Example (try):

```
#!/usr/bin/python3
```

```
total = 0 # This is global variable.
```

```
# Function definition is here
```

```
def sum( arg1, arg2 ):
```

```
    # Add both the parameters and return them.
```

```
    total = arg1 + arg2; # Here total is local variable.
```

```
    print("Inside the function local total : ", total)
```

```
    return total
```

```
# Now you can call sum function
```

```
total = sum( 10, 20 );
```

```
print("Outside the function global total : ", total)
```

Output calling the script:

Inside the function local total : 30

Outside the function global total : 30

Pass function arguments by reference vs. value



All parameters (arguments) of functions in Python are passed by reference: if you change what a parameter refers to within a function, then the change also reflects back in the calling function.

Example:

```
#!/usr/bin/python3
```

```
# Function definition is here
```

```
def changeme( mylist ):
```

```
    "This changes a passed list into this function"
```

```
    mylist.append([1,2,3,4])
```

```
    print("Values inside the function:\n ", mylist)
```

```
# Now you can call changeme function
```

```
mylist = [10,20,30]
```

```
changeme( mylist )
```

```
print("Values outside the function: \n", mylist)
```

Output:

Values inside the function:

[10, 20, 30, [1, 2, 3, 4]]

Values outside the function:

[10, 20, 30, [1, 2, 3, 4]]

Be careful duplicating variable names



```
#!/usr/bin/python3
# Function definition is here
def changelist( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]      # This assigns new reference in mylist
    print("Values inside the function: ", mylist)

# Function call
mylist = [10,20,30]
changelist( mylist )
print("Values outside the function: ", mylist)
```

The parameter `mylist` is local to the function `changelist`. Changing `mylist` within the function does not affect `mylist` outside the function.

Output:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

Be careful duplicating variable names and return



```
#!/usr/bin/python3
# Function definition is here
def changelist( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]      # This assigns new reference in mylist
    print("Values inside the function: ", mylist)
    return mylist

# Function call
mylist = [10,20,30]
mylist=changelist( mylist )
print("Values after the function: ", mylist)
```

The parameter `mylist` is local to the function `changelist`. Changing `mylist` within the function does not affect `mylist` outside the function.

Output:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [1, 2, 3, 4]

Modules



Modules are file containing Python statements and definitions.

A file containing python code is called a module
If the file is *my_lib.py*, the module name is *my_lib*

Modules are useful to break down large programs into small manageable and organized files.

Modules provide re-usability of code: useful functions can be put in a module and later imported in another module/script and re-used.

How to import module:

import my_module

How to import module by name:

import my_module as example

How to import a single function by a module:

from my_module import my_function | *from my_module import my_function as example*

How to import all names in a module:

*from my_module import **

Example: module imports



File (module) my_libs.py:

```
def add(a, b):
```

```
    #This program adds two numbers and return the result
```

```
    result = a + b
```

```
    return result
```

```
def multiply(a,b):
```

```
    #This program multiply two numbers and return the result
```

```
    result = a * b
```

```
    return result
```

- Import the entire module (my_modules_example_1.py)

```
import my_libs
```

```
print("add 4 and 5.5. Result:", my_libs.add(4,5.5))
```

- Import a single function (my_modules_example_2.py)

```
from my_libs import multiply
```

```
print("Multiply 4 X 5. Result:", multiply(4, 5))
```

- Import the entire module by name(my_modules_example_3.py)

```
import my_libs as example
```

```
print("add 4 and 5.5. Result:", example.add(4,5.5))
```

Example: module os (manage OS dialog operations)



```
import os
```

```
#namespace of module os
```

```
>>> os.curdir
```

```
','
```

```
>>> os.getenv('HOME')
```

```
'/home/bertocco'
```

```
>>> os.listdir('.')
```

```
['my_modules_example_1.py',
```

```
 'read_by_line.py',
```

```
 '.mozilla',
```

```
 '.bash_logout',
```

```
.....
```

```
]
```

```
In [2]: import os
```

```
In [3]: os.defpath
```

```
Out[3]: ':/bin:/usr/bin'
```

Module Search Path



To import a module, Python looks at several places.
Interpreter first looks for a built-in module then the search is in this order:

- The current directory
- PYTHONPATH (an environment variable with a list of directory. It can be modified by either applications or user)
- The installation-dependent default directory

How to reload a Module



The Python interpreter imports a module only once during a session. This makes things more efficient.

If the module is changed during the course of the program, we would have to reload it. To reload the module you have two ways:

- Restart the interpreter (not much clean).
- Use the function `reload()` inside the `importlib` module. Example:

```
In [29]: import my_libs
```

```
In [30]: import importlib
```

```
In [31]: importlib.reload(my_libs)
```

```
Out[31]: <module 'my_libs' from 'my_libs.pyc'>
```


Introspection



Introspection of a language is the ability of the language itself to provide information of its objects at runtime.

Python has a very good support for introspection.

The interpreter can be used interactively to better know and understand the code.

Following the description of useful python built-in functions for introspection

dir()



The `dir()` function can be used to find out names that are defined inside a module.

For example, we have defined the functions `add(a,b)` and `multiply(a,b)` in the module `my_libs`. We can find them:

```
In [32]: dir(my_libs)
```

```
Out[32]:
```

```
['__builtins__',  
  '__doc__',  
  '__file__',  
  '__name__',  
  '__package__',  
  'add',  
  'multiply']
```

names that begin with an underscore are default Python attributes associated with the module (we did not define them ourselves).

All the names defined in the current namespace can be found out using the `dir()` function without any arguments. Example (try):

```
dir()
```

help()



The function `help` is available for each module/object and allows to know the documentation for each function.

Try (in the interpreter) the commands:

```
import math
```

```
dir()
```

```
help(math.acos)
```

Example:

```
In [8]: import math
```

```
In [9]: dir(math)
```

```
Out[9]:
```

```
['__doc__',  
'__name__',  
'__package__',  
'acos',  
'acosh',  
.....
```

```
In [19]: help(math.acos)
```

Help on built-in function `acos` in module `math`:

```
acos(...)
```

```
    acos(x)
```

Return the arc cosine (measured in radians) of `x`.

type()



The type function allows to know the type of the object passed as argument.

Example:

```
In [1]: a=5
```

```
In [2]: type a
```

```
File "<ipython-input-2-e92615c5fd0c>", line 1
```

```
    type a
```

```
      ^
```

```
SyntaxError: invalid syntax
```

```
In [3]: type(a)
```

```
Out[3]: int
```

```
In [5]: l = [1, "alfa", 0.9, (1, 2, 3)]
```

```
In [6]: print([type(i) for i in l])
```

```
[<type 'int'>, <type 'str'>, <type 'float'>, <type 'tuple'>]
```

Pydoc is a python tool for introspection.
It provides information enclosed in a module in a clear and compact manner.

Pydoc uses the doc string `__doc__` and other standard attributes of objects
(`__name__`, `__file__`, ...).

```
$ pydoc os
```

Help on module os:

NAME

os - OS routines for Mac, DOS, NT, or Posix depending on what system we're on.

FILE

```
/usr/lib64/python2.4/os.py
```

DESCRIPTION

This exports:

- all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.
- os.path is one of the modules posixpath, ntpath, or macpath
- os.name is 'posix', 'nt', 'os2', 'mac', 'ce' or 'riscos'
- os.curdir is a string representing the current directory ('.' or ':')

.....

The zen of python



Type in the interpreter: `import this`

Output: the zen of python

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Exercise



Go to page

<https://docs.python.org/2/tutorial/introduction.html>

to the paragraph

“Using Python as a Calculator”

practice with the described operators.

Just to practice with the language,

write a module containing a python function executing the operation for each operator,

write your own script importing the module, reading input from command line,

executing the operations using the functions previously coded,

print the operators and the result for each function.