Introduction to Python

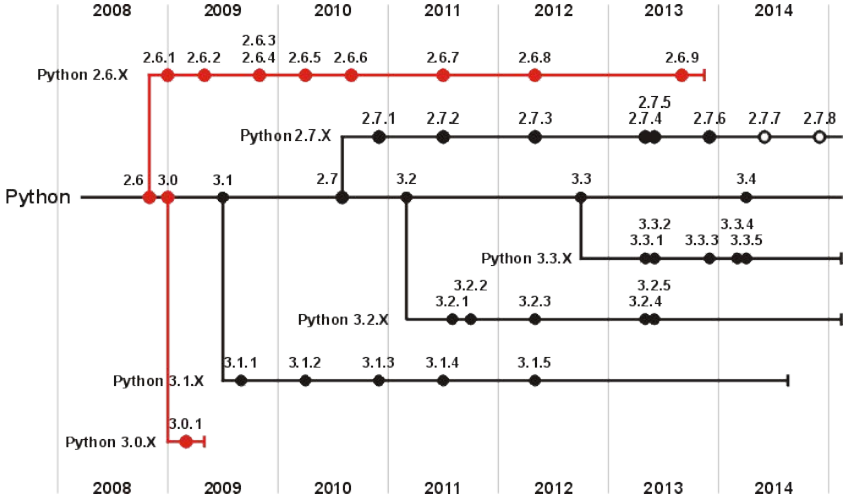
Anna Ivagnes, Federico Pichi, Gianluigi Rozza

September 30, 2025

Slides inspired by previous courses given by Marco Tezzele and Nicola Demo

What is Python?

- Python is an ***interpreted***, ***high-level***, ***general-purpose*** programming language;
- Focused on code readability;
- Two main versions: **2.x** (since 2020 2.7 will not be officially supported) and **3.x**.

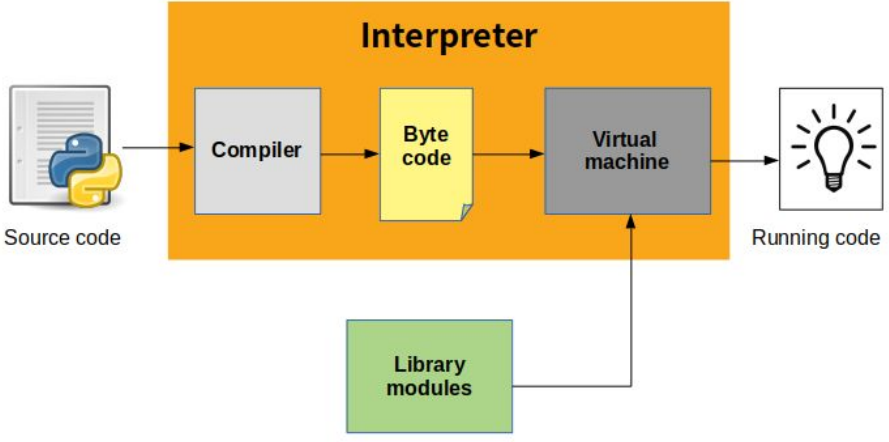


The Zen of Python

- Beautiful is better than ugly
- Explicit is better than implicit
- Simple is better than complex
- Complex is better than complicated
- Readability counts
- ...

All the 20 aphorisms about Python guidelines are available at (or try to “import this”):
https://www.python.org/dev/peps/pep-0020/

The Python Interpreter



In interpreted languages, the **order** of statements matter!

Installing Python interpreter

In the most common Linux distro, as well as in OSX systems, Python is usually already installed.

If you need to manually install the intepreter visit https://www.python.org/.

It is highly recommended to install the Python >3.8 version (Python >3 versions also include the package manager **pip**)

Running Python

Python shell from a terminal run the command '`python`' to open the interactive shell

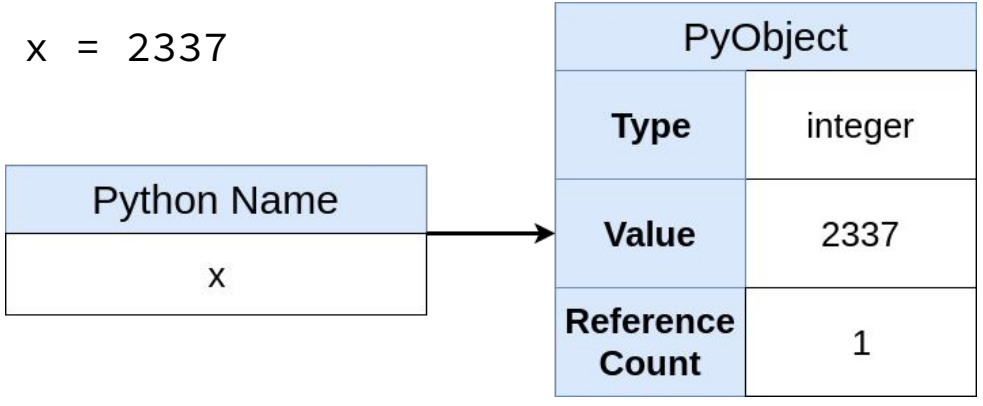
Python script run a Python script (extension *.py) by typing '`python myscript.py`'

```
$ python
Python 3.8.8 (default, Apr 13 2021, 19:58:26)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 3+2
5
```

Python variables

In Python all the variables are references to an object allocated in the memory.

Take care when you want to **copy** variables!



Operations with Numbers

Addition	+	3+4
Subtraction	-	1-9
Multiplication	*	5*3
Floating point division	/	7/2
Integer division	//	7//2
Modulus (reminder)	%	7%3
Exponentiation	**	2**5

Python Types

- Python is called ***dynamically typed***: the type of a variable is declared at runtime:

  `x = 10` ⟶ `x` is automatically of type `int`

- All the variables are objects;
- You can use the built-in method `type()` to control a generic variable.

Primitive Types

```
int      [x = 10]
float    [x = 3.14]          64bit
complex  [x = 2+6j]          128bit
bool     [x = True]
str      [x = 'Hello']       # single or double quotes
```

All the primitive types have already implemented the basic operation, e.g. sum, product, etc...

See the notebook!

Strings

- Strings are enclosed by single or double quotes (1 or 3)
  - `word = 'bug' or "bug" or '''bug''' or """bug"""`
- You can combine string variables by using the + operator
  - `'inventory:' + '11' + 'computers'`
  - `base = 'inventory: '`
  - `base += '11 computers' # base = base + '11 computers'`
  - `print(base)`
- Escape with \ for newlines and tabs
  - `newline = \n`
  - `tabs = \t`
  - `print('ab\tcd\nefgh')`

Strings

- Duplicate with *
  - `start = 'NA ' * 6`
  - `end = 'BATMAAAN'`
  - `print(start + end)`
- Extract a character with []
  - `person = 'marco polo'`
  - `person[0] >>> m`
  - `person[-4] >>> p`
- Strings are **immutable**, so you cannot insert a character directly
  - `person[0] = 'M'`
  - `>>> TypeError`
- Use the replace() function
  - `person.replace('m', 'M')`
  - `>>> Marco polo`

See the notebook!

Strings Slicing

- You can extract substring by using a **slice**. A general slice is the following
  - `[start:end:step]`
- The slice will include characters from offset *start* to one before *end*, skipping characters by *step*.
  - `[:]` extracts the entire sequence
  - `[start:]` from start to the end
  - `[:end]` from the beginning to the end offset minus 1
  - `[start:end]` from start to end minus 1

See the notebook!

Strings Slicing

- The step can also be negative
  - `[-1::-1]`
  - `[::-1]`
  - `Try using the variable person`
- Length of a string
  - `len(person)`
  - `len('')`
- Split a string using a separator
  - `person.split(' ')`
  - `person.split()`

Strings Case

- Remove a character from a string
  - `person.strip('o')`
- Capitalize the first word
  - `person.capitalize()`
- Convert to lowercase/uppercase
  - `person.lower() or .upper()`
- Remember the strings are **immutable!**

Type Conversions

- `int()`
  - `int(True)`
  - `int(37.5)`
  - `int('12')`
- `float()`
  - `float(False)`
  - `float(-82)`
- `str()`
  - `str(24+5)`
  - `str(True)`

List

- implemented as dynamic array
- contains heterogeneous elements
- operator `[]` is used to initialize lists (or `list()`) or to select an element

```
>>> empty_list = []
>>> mylist = [1, 3, ‘pippo’, []]
>>> mylist[2]
pippo
```

List Operations

Lists have already implemented most of the needed operations:

- length of list: `len([1, 2, 4]) = 3`
- sum: `[1, 2, 4] + [3, 5] = [1, 2, 4, 3, 5] # concatenation`
- add an element at the end of in a custom position (`append()` or `insert()`)
- remove elements by index or value (`pop()` or `remove()`)
- check presence of an item: `3 in [3, 5]`
- for docs: https://docs.python.org/3/tutorial/datastructures.html

See the notebook!

Tuples

See the notebook!

- tuples are sequences of arbitrary items
- tuples, unlike lists, are **immutable**
- operator `()` is used to initialize lists: `empty_tuple = ()`
- tuples use less space than lists
- there is no `append(), insert(), pop(), remove()` but there is the `[]`

```
>>> cars_tuple = (‘ferrari’, ‘porsche’, ‘maserati’)
>>> a, b, c = cars_tuple
>>> # a now corresponds to ‘ferrari’, b to ‘porsche’ etc.
```

Dictionary

- implemented as hashtable
- contains (unordered) pairs of (*keys*, *values*)
- operator `{}` is used to initialize dict (or `dict()`)

```
>>> city = {'name': 'Trieste', 'pop': 204338}
>>> city['pop'] = 205000
>>> city['postal_code'] = 34100
```

Dictionary Operations

- the method `get()` can be use to access to an element (or `[]`)
- the method `keys()` return the list of all the keys of the dictionary
- the method `values()` returns the list of values
- the method `items()` returns a list of tuples containing the (key, value) pairs
- Keep in mind that the elements in a dictionary are unordered!

See the notebook!

Indentation

- Python adopt just indentation to delimit blocks of code;
- From the PEP8 standard: "*Use 4 spaces per indentation level*";
- NO TABS!

```
class Foo(object):

    def generic_method(self, v1):
        pass
```

The IF Statement

- The general syntax is (pay attention to the **indentation**)

```
>>> if something_true:
>>>     blabla
>>> elif something_else:
>>>     blabla
>>> else:
>>>     print(‘not passed’)
```

Logical Operators

- equal: `x == y`
- not equal: `x != y`
- greater than: `x > y`
- less than: `x < y`
- greater or equal: `x >= y`
- less or equal: `x <= y`
- obj equality: `x is y`
- membership: `x in y`

To combine conditional statements, you can use `and`, `or`, `not`.

```
>>> if x <= 0 or x >= 100:
>>>     pass
```

What is False?

- a false value does not need to be explicitly `False`! The following are all considered false:
  - `False`
  - `None`
  - `0`
  - `0.0`
  - `‘’`
  - `[]`
  - `()`
  - `{}`

The WHILE Statement

- you can do a simple loop with `while` (pay attention to the **indentation**)

```
>>> count = 1
>>> while count < 6:
>>>     print(count)
>>>     count += 1
```

- you can stop a `while` using the keyword `break`

The WHILE Statement

- `while` is particularly suited for all the uncounted loops (number of iterations is not known a priori)
- take into account that the condition is checked before enter in the loop
- it is useful to check user input

```
>>> x = int(input('Digit number less than 100: '))
>>> while x >= 100:
>>>     x = int(input('Too high number, please retry: '))
```

See the notebook!

The FOR Statement

- to cycle over elements of iterators, without knowing how many elements there are, the Pythonic way is the following

```
>>> cities = ['Pordenone', 'Udine', 'Palmanova']
>>> for city in cities: # for i in range(len(cities)):
>>>     print(city)     #     print(cities[i])
```

- you can also exploit number sequences with `range(start, stop, step)`
- try to print the numbers from 10 to 20 every 2

List Comprehensions

- a *comprehension* is a Pythonic way to create data structures from one or more iterators.
- let us see how can we create a list of consecutive numbers. We have seen

```
>>> number_list = []
>>> number_list.append(2)
>>> number_list.append(3) # and so on
```

List Comprehensions

- We can also use a `for` statement

```
>>> number_list = []
>>> for number in range(2, 6):
>>>     number_list.append(number)
```

- or we can be better pythonistas with

```
>>> number_list = [number for number in range(2, 6)]
```

List Comprehensions

- You can use with all **iterable** objects:

```
>>> d = {k: v for k, v in my_list}
```

- It is not only a matter of compactness in the code: being Python an interpreted language, *comprehension* changes a bit how the code is run, so the performances

List Comprehensions

- it is also possible to use conditions and the general list comprehension can be

```
>>> [expression for item in iterable if condition]
>>> odds = [num for num in range(2, 10) if num % 2 == 1]
[3, 5, 7, 9]
```

See the notebook!

Reading and Writing Text Files

- In order to read or write a text file first you need to `open` it

```
>>> fileobj = open(filename, mode)
```

- *fileobj* is the file object returned by `open()`
- *filename* is the string name of the file
- *mode* is a string indicating the file's type and what you want to do with it

Reading and Writing Text Files

- The first letter of *mode* indicates the operation
  - `r` for reading
  - `w` for writing. If the file exists it is overwritten, otherwise it is created
  - `x` for writing a file but only if the file does not already exist
  - `a` for appending at the end of an existing file
- The second letter of *mode* indicates file's type
  - `t` or nothing for text
  - `b` for binary

Reading and Writing Text Files

- to write a file use the `write()` method:

```
>>> fout = open('awesome_file', 'wt')
>>> my_long_string = 'blablabla'
>>> fout.write(my_long_string)
>>> fout.close()
```

Reading and Writing Text Files

- you can write a file in a more safe way and automatically closing it using `with`

```
>>> my_long_string = ‘blablabla’
>>> with open(‘awesome_file’, ‘wt’) as fout:
>>>     fout.write(my_long_string)
```

Reading and Writing Text Files

- To read a text file we can either read it line by line or all at once

```
>>> poem = ‘’
>>> with open(‘awesome_file’, ‘rt’) as fin:
>>>     for line in fin:
>>>         poem += line

>>> with open(‘awesome_file’, ‘rt’) as fin:
>>>     lines = fin.readlines()
```

Functions

A function is a block of code which only runs when it is called: you can pass data and the function will return the result of the computation.

Use `def` to create new functions.

```
>>> def myfunction():
>>>     print('Hello World!')
>>> myfunction()
```

Functions

- You can pass information to the function: they are called **arguments**.
- You can also specify default arguments: in the case you not pass argument, the function will use the default value.

```
>>> def hello(name, surname='Demo'):
>>>     print('Hello {} {}'.format(name, surname))
>>> hello('Marco')          # output: Hello Marco Demo
>>> hello(surname='Tezzele', name='Marco')
    # output: Hello Marco Tezzele
>>> hello()                 # Error!
```

Functions

- Functions can also return a value using the keyword `return` (the interpreter will return to *caller*)

```
>>> def sum(a, b):
>>>     return a+b
>>> primo, secondo = 3, 5
>>> print(sum(primo, secondo)) # output: 8
```

Functions and Scope

```
>>> a = [5, 2]
>>> def foo(a):
>>>     a[0] += 1
>>>     return a[0], 3
>>> print(a)       # output: [5, 2]
>>> first, second = foo(a)
>>> print(a)       # output: [6, 2]
```

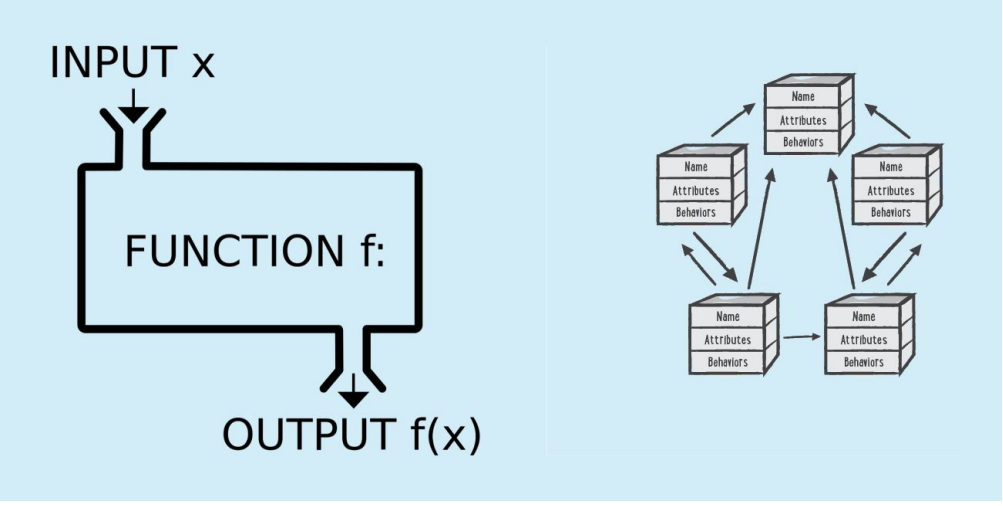
Import External Packages

In Python you can use external packages (only if installed in your machine!) by simply using `import`.

Take care that you can import the entire package (module), classes or just functions!

```
>>> import mymodule                  -> mymodule.MyClass.my_method()
>>> from mymodule import MyClass     -> MyClass.my_method()
```

Object Oriented



- closer to real world
- extremely useful for medium/big projects
- increase the code modularity and reusability

Object Oriented Python

A **class** is an extensible template for creating object, providing initial values for state (attributes) and implementations of behavior (methods).

An **object** refers to a particular instance of a class.

```
class Dog(object):

____def __init__(self, name='Pippo'):
________self.name = name

my_dog = Dog('Pluto')
print(my_dog.name)
```

Object Oriented Python

Attributes are variables that refer to an object (or to a class).

class attributes: *species*

object attributes: *name*

self refers to the object itself

```
class Dog(object):

     species = ‘Canis lupus familiaris’

     def __init__(self, name=’Pippo’):
          self.name = name

my_dog = Dog(‘Pluto’)
your_dog = Dog()
print(my_dog.name)
print(Dog.species)
```

Object Oriented Python

Methods are basically the function available for the object:

- `__init__(self, ...)` is a special method called constructor (the method to create a new object)
- `run()` is a generic method (the first argument `self` is mandatory)

```
class Dog(object):

     species = ‘Canis lupus familiaris’

     def __init__(self, name=’Pippo’):
          self.name = name

     def run(self):
     pass

my_dog = Dog(‘Pluto’)
my_dog.run()
```

Object Oriented Python

The meaning of the `self` argument is simple: is the variable that indicates the instantiated object. Usually methods are called using:

```
<object>.<method>(arguments)
```

But technically the interpreter executes:

```
<class>.<method>(<object>, arguments)
```

That is the reason of the mandatory first argument in object methods.

Object Oriented Python

```
class Dog():

____def __init__(self):
________pass

____def run(self):
________print('im running')

dog = Dog()
dog.run()
Dog.run(dog)      # perfectly fine
```

Object Oriented Python

Inheritance helps to create specialized classes that share methods: `Dog` inherits all methods and attributes from `Animal` (but not viceversa).

the children is Dog
the parent is Animal

For methods/attributes with same names, Python goes for the children implementation.

```
class Animal(object):

____def __init__(self, name='Pippo'):
________self.name = name

class Dog(Animal):

____species = 'Canis lupus familiaris'

____def bau(self):
________print('Bau!')
```

Object Oriented Python

```
class Triangle(object):

____def __init__(self, l1, l2, l3):
________self.l = [l1, l2, l3]

____def perimeter(self):
________return sum(self.l)
```

```
class EquilateralTriangle(Triangle):

     # Complete the implementation
     # considering that equilateral
     # triangles have equal length sides
     # YOU NEED NO MORE THAN 2 LINES
     def __str__(self):
          return str(l1) + str(l2)

my_tria = EquilateralTriangle(5.2)
print(my_tria.perimeter()) # 15.6
```

Naming Convention

- **`lower_case_with_underscores`** for methods, variables, attributes;
- **`CamelCase`** for classes;
- **`_single_leading_underscore`** for “conventionally” private methods;
- **`__double_leading_underscore`** for private methods (mangling!);
- **`__double_leading_and_trailing_underscore__`** for special methods.

```
class Foo(object):

    def generic_method(self, v1):
        pass

    def __str__(self):
        return ‘Generic <Foo> object’
```

Documentation

1. **docstring** (a string literal that occurs as the first statement in a module, function, class, or method definition.)
2. **comment**

```
def bar():
    """
    A multi-line
    docstring.
    """
    pass
```

```
# Sum the input
a = b + c

''' Check if input
is odd by looking at
final bit '''
is_odd = a & 1
```

Programming Best Practices

Why Code Quality?

High quality code means:

- ➜ less time to read and understand the code;
- ➜ less time to extend the code and/or fix a bug;
- ➜ more stable code thanks to the tests.

... but of course this implies more time in the earliest development stage.

Code Style

- follow the code standard;
- well-designed structure (no global variables, object oriented paradigm, design patterns, ...);
- avoid **code duplication** (it means twice to test, to fix, to edit);
- implement **small procedures** with few variables instead big procedure, in order to increase the modularity and the readability;
- use **meaningful names** for all the variables;
- avoid too many nested blocks (max 3);
- assign the right scope to the variables;
- break the lines after 80 characters!
- ...

Code Testing

Tests are very important to make software reliable!
HINT: *test your code since the early development stage*

unit tests: check the functionality of a small block of code, typically a procedure (e.g. a generic function that computes the norm of a given vector);

functional test: check the program works as expected (e.g. my ROM software return the right POD basis).

Testing in practice

```
def my_sorting_alg(tmp_list):

    # Here the implementation
    # of the function to sort
    # list in-place
```

```
def testing_my_alg():

    l1 = [4, 6, 1, 7]
    l2 = [4, 6, 1, 7]

    my_sorting_alg(l1)
    l2.sort()

    if l1 == l2:
        print('Test OK')
```