

# Python crash course

Machine Learning for Robotics

Máster Universitario en Ingeniería Industrial

Departamento de Automática

## Objectives

1. Overview the Python syntax
2. Being able to program naïve Python scripts

## Bibliography

The Python Tutorial

# Table of Contents

# Introduction

## What is Python? (I)

Python is a general-purpose, high-level, interpreted programming language

- General-purpose: Many applications.
- High-level: Abstract data structures, doing more with less code.
- Interpreted: No need to compile.



It emphasizes code **readability** and programmer's productivity

# Introduction

## What is Python? (II)

### Python

```
#!/usr/bin/python

print("Hello , world !")
```

### Java

```
public class HelloWorld {
    public static void main( String []
        args ) {
        System.out.println( "Hello , world
            !");
    }
}
```

### Hello world! examples

#### C

```
#include <stdio.h>

int main()
{
    printf("Hello , world !\n");
}
```

#### C++

```
#include <iostream>

int main()
{
    std :: cout << "Hello , world !\n"
        ;
}
```

# Introduction

## History

- Python was created by Guido van Rossum in the Netherlands
- Python 2.0: Released on 2000
- Python 3.0: Released on 2008. Backwards-incompatible

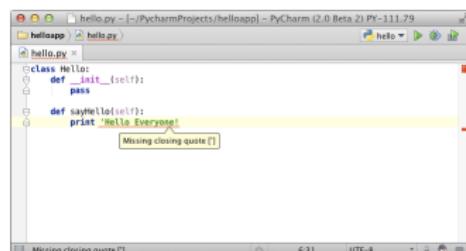
Python 3.X is the present but Python 2.x still popular



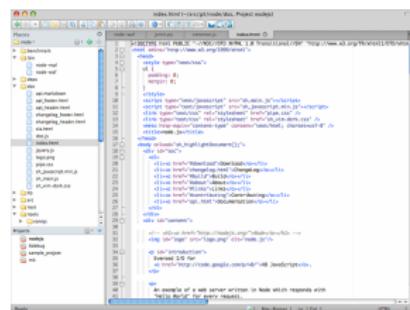
# Introduction

## Installation

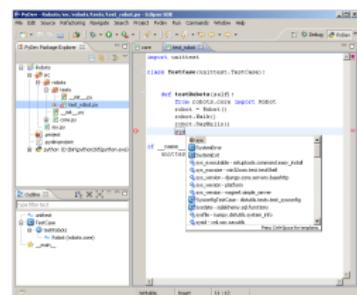
- If you have a good OS such as Linux or Mac, you already have Python!
- Otherwise (Windows), you have to install it
  - Visit <https://www.python.org/downloads/>
- Bad news: There is no “standard” IDE
  - PyCharm, Komodo, PyDev, ...
  - <http://wiki.python.org/moin/PythonEditors>
  - Notebooks are also very popular, particularly in AI



PyCharm



Komodo



PyDev

# The Python interpreter

## Python operation modes

Python is an interpreted language, i.e., it needs an interpreter.

- Interpreted = it is not complied = it needs no compilation.
- Faster development, slower execution.

Two operation modes:

- **Interactive:** The interpreter reads the program from the `stdin` (usually the keyboard).
- **Non-interactive:** The interpreter reads the program from a file (also known as `script`).

# The Python interpreter

## Interactive

Just run `python`

- Different names for different versions to avoid conflicts.
- `python`, `python3.4`, ...

```
localhost:~ user$ python3.4
Python 3.4.2 (v3.4.2:ab2co23a9432, Oct 5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The programmer executes as he writes code down.

# The Python interpreter

## Non-interactive

The program is in a plain text file.

- It can be edited with any text editor.
- Extension “.py”.
- Execution permission (`chmod u+x myscript.py`).
- By default, UTF-8 encoding.

The first line must be `#!/usr/bin/python`

- It is the interpreter location.
- If not present, the interpreted must be invoked.

script.py

```
#!/usr/bin/python  
  
print("Hello, world!")
```

python script.py  
./script.py

# Variables

## Numbers (I)

**Variable:** A name that refers a value.

- No need to declare variables (Python is weakly typed!).
- Python automatically assigns types.
- Basic types: Numbers, strings and booleans.

**Complex data structures:**

- Lists, tuples, dictionaries, associative arrays.

Variables

variable = value

# Variables

## Numbers (II)

```
>> numberA = 4
>> numberB = 2.3
>> numberA + numberB
6.3
>> string = "Spam"
>> boolean = True
>> a = b = c = 0
>> b
0
>>> type(numberA)
<type 'int'>
```

```
a = int(input("Number: "))
b = float(input("Number: "))
c = (a * b) / 2
c += 1
d = c ** 2
print("Result c: ", c)
print("Result d: ", d)
```

New Python elements:

- The `input()` function.
- The `int()` and `float()` functions.
- The `type()` function.

# An informal introduction

## Strings (I)

Of course, variables can contain strings.

```
>>> text = "hello"  
>>> text = 'hello'  
>>> print(text)  
hello
```

Strings contatenation

```
>>> "hello" + " there"  
'hello there'  
>>> "hello" "there"  
'hellothere'
```

Variables with strings

```
>>> a = "hello"  
>>> b = " there"  
>>> a + b  
'hello there'
```

String length

```
>>> len("hello")  
5
```

# An informal introduction

## Strings (II)

### F-strings: From Python 3.6

```
>>> name = 'John'  
>>> age = 22  
>>> print(f"Hi {name}, you are {age}")  
'Hi John, you are 22'
```

# An informal introduction

## Strings (III)

Strings are a sequence of characters: Slice notation.

- Quite common in Python data structures.
- It uses indices (as an array). First index is 0.

### Slice notation

variable[start:stop:step]

```
>>> a = "hello"
>>> a[2]
'1'
>>> a[2:]
'1lo'
>>> a[:2]
'he'
>>> a[2:] + a[:2]
'1lohe'
>>> a[2:4]
'1l'
>>> a[::-2]
'hlo'
>>> a[::-1]
'olleh'
```

# An informal introduction

## Lists (I)

**List:** An ordered collection of mutable data.

- Very powerful data structure, similar to an array.
- Ordered: Data in the list have a location.
- Mutable: Data can be modified.
- Data types can be different.

### List initialization

```
variable = [data1, data2, ..., dataN]
```

# An informal introduction

## Lists (II)

Initialization

```
>>> a = ['spam', 'eggs', 123]
['spam', 'eggs', 123]
```

Slice notation, `len()`  
and + work on lists

```
>>> a[2]
123
>>> a[1:]
['eggs', 123]
>>> a + a
['spam', 'eggs', 123, 'spam', 'eggs', 123]
>>> len(a)
3
```

# Data structures in Python

## Lists (III)

Lists are objects: they have methods

- `list.append(x)`
- `list.insert(i, x)`
- `list.remove(x)`
- `list.pop()`
- `list.index(x)`
- `list.count(x)`
- `list.sort()`
- `list.reverse()`

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

# Control flow

## Conditions (I)

Conditional statements implement decision making

- Decide some code has to be executed or not.
- The result is a boolean.
- Execute code if condition is satisfied.

### if statement

```
if <condition>:  
    # Some code  
else:  
    # Some other code
```

### Example

```
if age > 18:  
    # Some code  
else:  
    # Some other code
```

New Python elements:

- Comments begin with '#'.
- Indentation plays a major role: It defines code bodies.

# Control flow

## Conditions (II)

### Nested condition

```
if age == 18:  
    print("You are 18 years old")  
elif age < 18:  
    print("You are younger than 18 years old")  
elif age > 18:  
    print("You are older than 18 years old")  
else:  
    print("You should not be reading this")
```

# Control flow

## Conditions (III)

SIGN	OPERATOR	SIGN	OPERATOR
<code>==</code>	Equal	<code>and</code>	Logical and
<code>!=</code>	Not equal	<code>or</code>	Logical or
<code>&gt;</code>	Greater	<code>not</code>	Logical not
<code>&lt;</code>	Lower		
<code>&gt;=</code>	Greater or equal		
<code>&lt;=</code>	Lower or equal		

### Nested condition

```
if (age > 18) and (name == "Biggus Dickus"):  
    print("Hi Biggus")
```

# Control flow

## for statements (I)

- Sometimes we have to repeat a task: Loops
  - Other languages iterate over a condition
  - For instance, in C: `for (i=0; i<10; i++)`
- Two loop statements in python: `while` and `for`
- In Python, `for` iterates over a sequence (lists or strings)
  - In each iteration, it assigns a sequence value to a variable

### for statement example

```
for x in [ 'cat' , 'window' , 'dog' ]:  
    print(x)
```

### for statement example

```
for x in "Hello word":  
    print(x)
```

# Control flow

## for statements (II)

Sometimes, we need to iterate over a sequence of numbers

- `range(n)`: It returns a sequence  $0, \dots, n - 1$

### range() example

```
for i in range(5):  
    print(i)
```

### Alternative notation

```
a = [ 'Mary', 'had', 'a']  
  
for i in range(len(a)):  
    print(i, a[i])
```

### Iterate over a sequence of numbers with indices

- `enumerate(list)`

### enumerate() example

```
for i, value in enumerate(['a', 'b', 'c']):  
    print(f"index: {i}, value: {value}")
```

# Control flow

## for statements (III)

List comprehensions are a concise way to create lists

- Widely used and convenient

### List with squares

```
squares = []
for x in range(10):
    squares.append(x**2)
```

### Alternative notation

```
squares = [x**2 for x in range(10)]
```

List comprehensions support conditions

```
list = [x for x in range(10) if x%2 == 0]
```

# Control flow

## While loop

### Fibonacci series

```
#!/usr/bin/python

a, b = 0, 1 # Init variables

while b < 10: # This is a loop
    print("b = ", b)
    print("a = ", a) # Indentation is very important here!
    a, b = b, a+b
```

### New Python elements:

- Multiple assignments

# Functions

## Defining functions (I)

**Function:** A piece of code that can be used several times

- Lazy programmers are good programmers
- Code reuse

Functions can be used with parameters

- Define a function before using it

### Function 1

```
def printHello():
    print("Hello")
printHello()
```

### Function 2

```
def printTwice(string):
    print(string)
    print(string)

printTwice("hello")
```

Hint: If you have to use code more than once, place it in a function

# Functions

## Defining functions (II)

A function can return a value with the keyword `return`

```
def computeSquare(x):
    """ Takes a number and returns its square """
    return x ** 2

x2 = computeSquare(3)
print(f"x2 = {x2}")
```

New Python feature:

- Docstrings

# Functions

## Default argument values and keyword arguments

Python supports default arguments:

- Powerful and simple feature.
- Simpler (and more flexible) function calls.

```
def sum(a, b=10):  
    return a + b  
  
sum(1, 2)  
sum(1)
```

Function arguments can be named:

- It overrides classic positional arguments.
- Positional arguments must be first.

```
def foo(bar, baz):  
    print(bar, baz)  
  
foo(1, 2)  
foo(baz = 2, bar = 1)
```

```
def foo(bar = "hello", baz = "bye"):  
    print(bar, baz)  
  
foo()  
foo("hi")  
foo(baz = "hi")
```

# Data structures in Python

## Tuples

**Tuple:** Ordered sequence of items, very similar to lists.

- However they are not the same.
- Lists are mutable, tuples are immutable.
- Tuples used to contain, **usually**, heterogeneous items.
- Lists used to contain, **usually**, homogeneous items, used to iterate.

### Tuple initialization

```
tuple = (item1, item2, ..., itemN)  
tuple = item1, item2, ..., itemN
```

### Creation

```
tup1 = 1, 2, 3  
tup2 = ("Hi", 1.1, 2)  
tup3 = (0, (1, 3), 2)
```

### Manipulation

```
>>> tup1[0]  
1  
>>> tup1  
(1, 2, 3)  
>>> tup1[1:]  
(2, 3)
```

# Data structures in Python

## Sets

**Set:** A collection of items, unordered with no duplicates.

- Membership testing.
- Eliminating duplicate entries.

### Creation

```
>>> set1 = {"red", "blue"}  
>>> set_mix = set(['a', True, 33])  
>>> set_mix  
{'a', True, 33}  
>>> len(set_mix)  
3  
>>> 33 in set1  
True
```

## Set initialization

```
set = {item1, item2, ..., itemN}
```

**Sequence:** All types that behaves like sequences: Strings, lists and tuples.

# Data structures in Python

## Dictionaries (I)

**Dictionary:** A collection of pairs <key, value>

- Also named as *associative array*, very similar to hash maps.
- Lists are indexed with a number, dictionaries use keys.
- Key: Numbers, strings, tuples and any immutable type.

### Dictionary initialization

```
dict = {key1: value1, key2: value2}
```

### Creation

```
>>> tel = { 'jack' : 4098, 'sape' : 4139 }
```

### Manipulation

```
>>> tel[ 'guido' ] = 4127
>>> del tel[ 'sape' ]
>>> tel
{ 'guido': 4127, 'jack': 4098}
>>> list(tel.keys())
[ 'guido', 'jack' ]
>>> 'guido' in tel
True
```

# Data structures in Python

## Dictionaries (II)

Dictionaries can be iterated by key or by value

- Loop syntax is slightly different
- `items()` method

### Dictionary iteration

```
knights = { 'gallard' : 'the pure', 'robin' : 'the brave' }

for k, v in knights.items():
    print(k, v)

for k in knights.keys():
    print(k)
```

# Data structures in Python

## Summary

DATA STRUCTURE	INITIALIZATION
List	<code>li = [1, 2, 3]</code>
Tuple	<code>tu = (1, 2, 3)</code> <code>tu = 1, 2, 3</code>
Set	<code>se = {1, 2, 3}</code>
Dictionary	<code>dic = {'abc' : 1, 'bca' : 2}</code>

# Modules

## Introduction (I)

You loose everything when exit the interpreter

- Solution: Write it down in a script

When a script becomes big, it is difficult to maintain

- Solution: Split your script in several ones

As you get more scripts, you will need to reuse your functions

- Solution: Create a **module**
- **Module:** A file that contains definitions, functions and classes

If a module is too big, it is too difficult to maintain

- Solution: Create a **package**
- **Package:** A module of modules

# Modules

## Installing packages

Command-line automatic tool: pip

- Very similar to apt-get in Linux

### pip usage

```
$ python -m pip install SomePackage
```

### pip alternative usage

```
$ pip3 install SomePackage
```

Example of the PIL installation:

```
$ pip3 install Pillow
```

(More info)

# Modules

## Namespaces (I)

### Importing packages

```
import <package> [as <name>]
```

A module can import other modules

- Name conflicts may arise: Each module has a symbol table
- It means you should invoke it as `modname.itemname`
- The module name can be changed with the keyword `as`

```
>>> import math  
>>> math.log(10)  
2.302585092994046
```

```
>>> import math as m  
>>> m.log(10)  
2.302585092994046
```

# Modules

## Namespaces (II)

It is possible to import items directly

- `from module import name1, name2`
- `from module import *`
- It uses the global symbol table (no need to use the modname)

```
>>> from math import log  
>>> log(10)  
2.302585092994046
```

```
>>> from math import *  
>>> log(10)  
2.302585092994046
```

# Modules

Example 1: Open a web browser

browser.py

```
import webbrowser  
  
webbrowser.open("https://www.reddit.com")
```

also

browser2.py

```
import webbrowser as w  
  
w.open("https://www.reddit.com")
```

# Modules

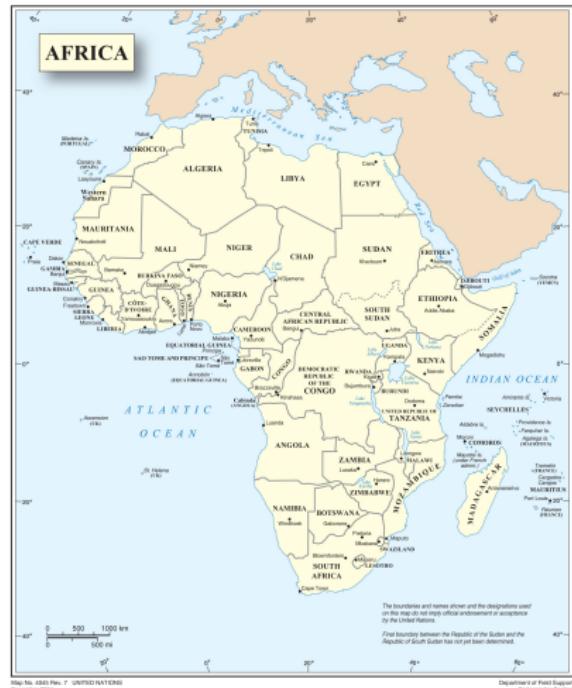
## Example 2: Create a thumbnail

thumbnail.py

```
from PIL import Image

size = (128, 128)

im = Image.open("africa.tif")
im.thumbnail(size)
im.save("africa.jpg")
im.show()
```



(Source)

africa.jpg

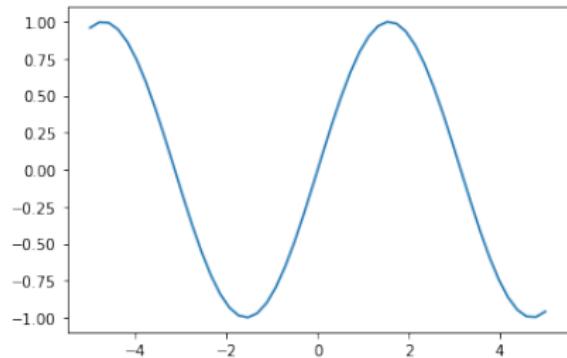
# Modules

## Example 3: Plot

plot.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5)
plt.plot(x, np.sin(x))
```



# Classes in Python (I)

- **Class:** Start with the word **class** followed by class name written in **capital letter** and a colon [Substantives].
- **Attributes:** A lowercase noun.
  - There is no need to declare attributes.
- **Inherited class:** Similar to a class but the class name followed by the class father in brackets.
- **Instance:** Object in lower case followed by the class assignment.

```
car.py
class Vehicle:
    def __init__(self, wheels):
        self.wheels = wheels

class Car(Vehicle):
    def __init__(self, wheels, model):
        Vehicle.__init__(self, wheels)
        self.model = model

ford = Car(4, "mondeo")
myBike = Car(2)
```

# Classes in Python (II)

- **Method:** Start with the word `def`
  - Methods receive automatically a reference to the object (usually named `self`).
- **Constructor:** Method whose name is `__init__()`, the first attribute is `self`.
- All methods and attributes are public.
  - By convention, private members begin with double underscore (`__varName`, `__method_name()`)

# Classes in Python (III)

## Two operations on classes

### Instantiation

Creates a new object

Standard functional notation

```
x = MyClass()
```

### Example

```
time = Time()
```

### Attribute references

Accesses an attribute value

Standard dot syntax

```
obj.name
```

### Example

```
time.hour = 4  
print(time.hour)  
hour = time.hour
```