# INTRODUCTION TO PYTHON PROGRAMMING

Gael Cascioli



gael.cascioli@uniroma1.it

# OUTLINE

- What is a programming language?
- Why Python?
- How to use Python?
- Basics
  - Basic data types with examples
  - Control flow
  - Functions
  - Object oriented programing (brief introduction)
  - Modules
- Advanced examples
  - Scripting in python
  - Scientific plotting
  - Analyzing geospatial data
- Useful resources

# WHAT IS A PROGRAMMING LANGUAGE?

# WHAT IS A PROGRAMMING LANGUAGE?

”**Programming**” consists in writing a set of instructions to implement algorithms (or procedures) to be executed by a computer.
The language we use to communicate with the computer is the “**programming language**”

**<u>The classical example:</u>**

A fancy bartender/barlady

The programmer

**Mezcal Negroni**

1 oz Campari
1 oz Vermouth
1 oz Mezcal
Orange zest for garnish
Ice

**INSTRUCTIONS**

Combine Campari, Vermouth, and Mezcal in pint glass.
Add ice and stir.
Strain into glass.
Rub orange zest on rim and twist into drink.

The set of instructions

You at home

The computer

The language

# WHAT IS A PROGRAMMING LANGUAGE

- A programming language is an agreement between the human and the computer
- Thousands of languages have been invented

**Two main classes**

| Low Level | High Level |
|---|---|
| Closer to machine code (100110..)<br><br>**Pros:**<br>Fast, Precise<br><br>**Cons:**<br>Difficult for humans to read | Closer to human language<br><br>**Pros:**<br>Easy to understand<br><br>**Cons:**<br>Slower to convert to machine-code (i.e., lower speed) |

# WHAT IS A PROGRAMMING LANGUAGE

**Bonus**: Esoteric languages
https://en.wikipedia.org/wiki/Esoteric_programming_language

**Cow**
Generate the Fibonacci sequence:

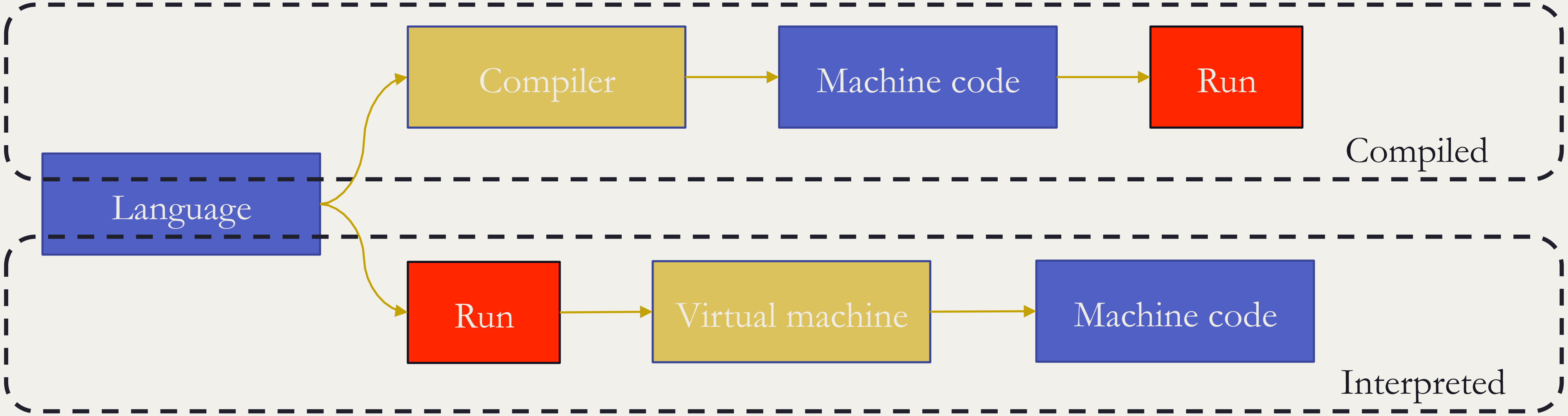
```
MoO moO MoO mOo MOO OOM MMM moO moO MMM mOo mOo moO MMM mOo MMM moO
moO MOO MOo mOo MoO moO moo mOo mOo moo
```

**Malbolge**
Print “Hello World”:

```
(=<`:9876Z4321UT.-Q+*)M'&%$H"!~}|Bzy?=|{z]KwZY44Eq0/{mlk** hKs_dG5[m_BA{?-
Y;;Vb'rR5431M}/.zHGwEDCBA@98\6543W10/.R,+O<
```

# WHAT IS A PROGRAMMING LANGUAGE

Programming languages can also be distinguished between **compiled** and **interpreted**. The main difference consists in **how** and **when** the language is translated in machine code.




**Compiled**
**Pros:** Faster, more control on hardware implementation
**Cons:** Additional step before testing, platform dependent

**Interpreted**
**Pros:** Platform independent
**Cons:** Slower execution

# HOW TO CHOOSE ONE?

The main criterion for the choice is the **intended usage.**

The intended usage defines the requirements in terms of

- **Execution speed**
- **Robustness**
- **Development speed**

| Language | Development speed | Execution speed | Robustness | Developer availability |
|---|---|---|---|---|
| Javascript | ++++ | ++ | + | ++++ |
| Typescript | +++ | ++ | ++ | +++ |
| Python | ++++ | + | + | ++++ |
| Go | +++ | +++ | +++ | ++ |
| Rust | + | ++++ | ++++ | ++ |
| Java | ++ | ++ | +++ | ++++ |
| Haskell | + | ++ | ++++ | + |
| ReasonML | ++ | +++ | ++++ | + |

Source: qvik.com

# WHY PYTHON?

# WHY PYTHON?

**Python implements a high level of abstraction**

- Easy to read and debug
- Automatic memory handling
- It is object-oriented

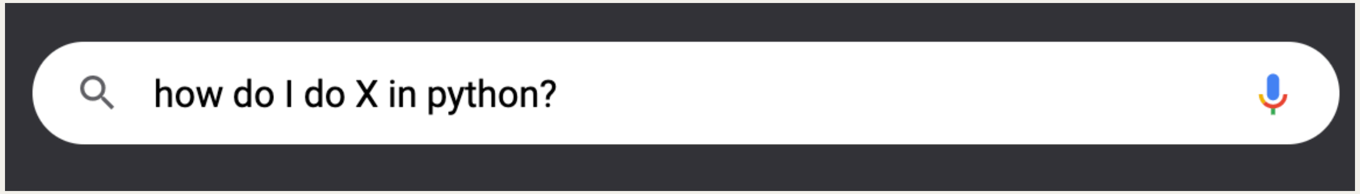
**It is open source**

- The intellectual property behid the code is held by a non-profit organization
- Completely compatible with open source standards
- It’s free!

**It can be easily interfaced with low-level languages**

- Very easy to build Python interfaces to fast-running programs in C/FORTRAN/etc..

**The Python community is LARGE**




Gives you a very good answer 99% of the times!

1 JavaScript
2 Python
2 Java
4 PHP
5 C#
6 C++
7 Ruby
7 CSS
9 TypeScript
9 C
11 Swift
12 Objective-C
13 Scala
13 R
15 Go
15 Shell
17 PowerShell
18 Perl
19 Kotlin
20 Haskell

Popularity on GitHub (2021)
Elaborated by rednonk.com

**Notable users**

# HOW TO USE PYTHON?

**Installation**

The first thing to do is **install** python.
It can be done in several ways depending on the OS (windows, macOS, linux, …)

**The easiest way: Install (ana)Conda!**

"Conda is an open-source package management system and environment management system that runs on Windows, macOS, and Linux. Conda quickly installs, runs, and updates packages and their dependencies. Conda easily creates, saves, loads, and switches between **environments** on your local computer. It was created for Python programs but it can package and distribute software for any language."

# HOW TO USE PYTHON

Useful links for installing python:

The easy way:

**Anaconda:** https://docs.conda.io/projects/conda/en/latest/user-guide/install/

The other way:

**Windows:** https://phoenixnap.com/kb/how-to-install-python-3-windows

**macOS:** https://docs.python-guide.org/starting/install3/osx/

**Linux:** https://docs.python-guide.org/starting/install3/linux/

# HOW TO USE PYTHON

Once installed, it is time to write and run the first python program.
Ok. But how, where??

The standard procedure is the following:

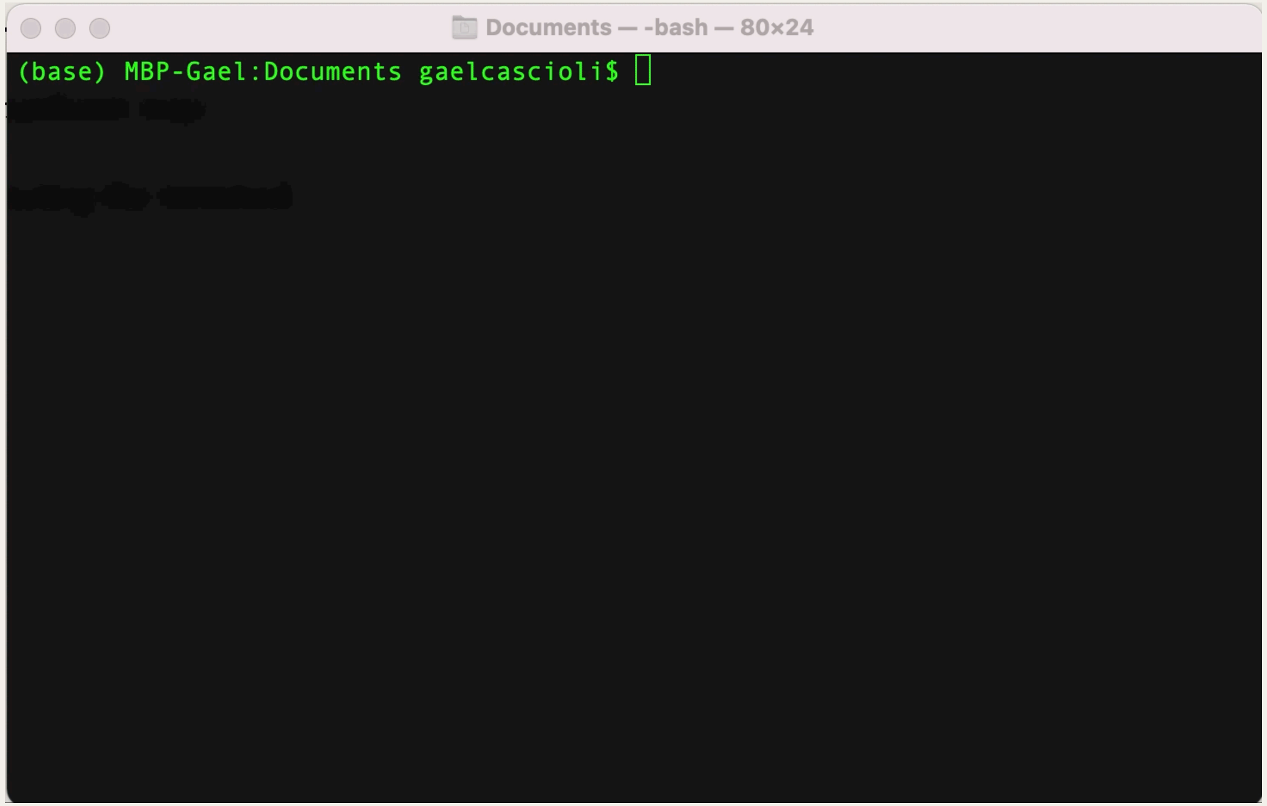
1) Write the program
2) Execute the program

There exist several ways of doing this:

1) **The hardcore way**
2) **The fancy way**
3) **Many other ways**

# HOW TO USE PYTHON

**The hardcore way:** Only using the terminal



# HOW TO USE PYTHON

**The fancy way:** Using an IDE (Integrated Development Environment)

**For example:** SublimeText


But there are many others:

- PyCharm
- Visual Studio Code
- Atmo
- PyDev
- Spyder

# HOW TO USE PYTHON

**The fancy way:** Using an IDE (Integrated Development Environment)

**For example:** SublimeText

But there are many others:

- PyCharm
- Visual Studio Code
- Atmo
- PyDev
- Spyder

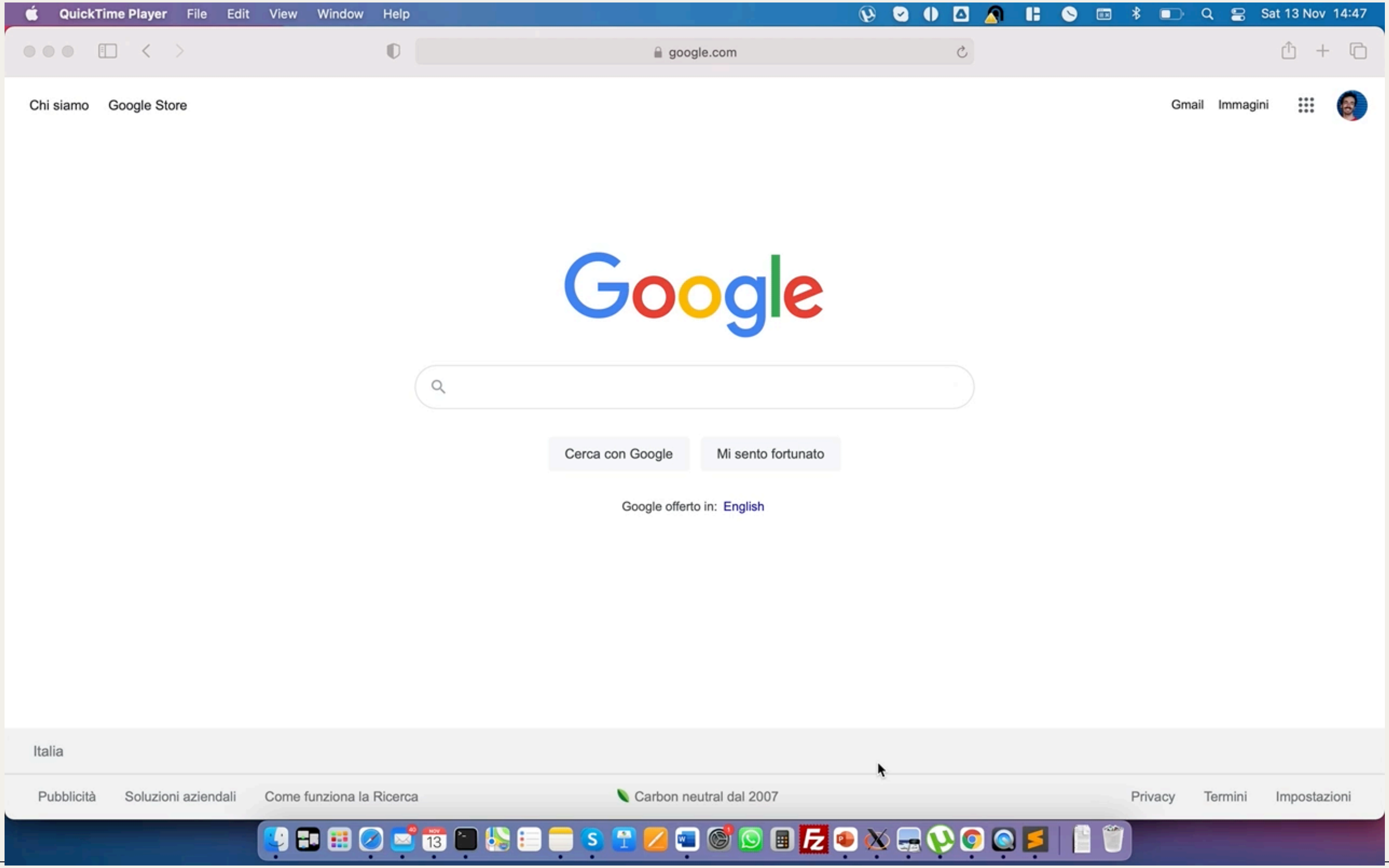
test_embree3.py

```python
# Test Embree3
import trimesh as tm
import embree
import numpy as np
import sys
import utils_rt

def get_centroids(V, F):
    return V[F].mean(axis=1)


def get_cross_products(V, F):
    V0 = V[F][:, 0, :]
    C = np.cross(V[F][:, 1, :] - V0, V[F][:, 2, :] - V0)
    return C


def get_face_areas(V, F):
    C = get_cross_products(V, F)
    C_norms = np.sqrt(np.sum(C**2, axis=1))
    A = C_norms/2
    return A


def get_surface_normals(V, F):
    C = get_cross_products(V, F)
    C_norms = np.sqrt(np.sum(C**2, axis=1))
    N = C/C_norms.reshape(C.shape[0], 1)
    return N

def get_surface_normals_and_face_areas(V, F):
    C = get_cross_products(V, F)
    C_norms = np.sqrt(np.sum(C**2, axis=1))
    N = C/C_norms.reshape(C.shape[0], 1)
    A = C_norms/2
```

# HOW TO USE PYTHON

**Many other ways:**

- **Jupyter Notebook**
- **iPython**
- **Google CoLab**



# BASICS

# BASICS

Basic data types:

Python is a **dynamically and strongly typed language**

The type of variable (int, float, string, etc..) is determined at **execution time**

The data types are imposed.
E.g., an integer cannot be treated as a string without an explicit conversion

Basic variable types in python can be distinguished in **data types** and **data containers**

| **Data Type** | **Name in python** | **Example** |
|---|---|---|
| Integer | int | 3 |
| Floating-point | float | 2.43 |
| Complex | complex | 2+4j |
| String | str | 'Hello world!' |
| Boolean value | bool | True or False |

You can always check what is tye **type** of a variable:

```
>>> a = 2.42
>>> type(a)
<class 'float'>
>>> b = 3+4j
>>> type(b)
<class 'complex'>
>>>
```

# BASICS

The containers, as per their name, are used to contain the previously defined data types.
Each container is designed for a specific use (we will go in more detail later)

| **Data Container** | **Name in python** | **Example** |
|---|---|---|
| List | list | [1,2,3,4] |
| Dictionary | dict | {'Cat': 'Black', 'Dog': 'Brown'} |
| Tuple | tuple | (1,2,3,4) |
| Set | set | {1,2,3,4} |

# SOME EXAMPLES

## Operations on numbers

```
a = 2
b = 3
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a**b)
```

**Output**

```
5
-1
6
0.6666666
8
```

## Operations on Strings

```
a = 'Python'
b = 'Course' print(a+b)
print(a + ' ' + b)
print(2*a)
```

**Output**

'PythonCourse'
'Python Course'
'PythonPython'

## Lists

**Lists are used to store multiple items in a single variable.**

**Example**:
```
mylist = ['Python', 'Programming', 'Language']
mylist2 = [2,3,1,-1,0.4]
```

**Lists are ordered and they can contain duplicates.**

**Example:**
```
mylist = ['A', 'B', 'C', 'C']

print(mylist[0])
print(mylist[1])
print(mylist[2])
print(mylist[3])
```

**Output**
**'A'**
**'B'**
**'C'**
**'C'**

**Indexes in Python always start from 0:**

First element: 0
Second element: 1
Third element: 2
…

## Lists

**Lists are changeable**

**Example:**

```
mylist = ['A', 'B', 'C']
mylist[0] = 'F'
print(mylist)
```

**Output**

```
['F', 'B', 'C']
```

**Lists can contain different data types**

**Example:**

```
mylist = [3, 2+4j, True, 'A', 2.33221]
```

# SOME EXAMPLES

**List operations:**

```
a = [1,2,3]
b = ['A', 'B', 'C']

print( a+b )
print( 2*a )
print( len(a) )
print( a.append(4) )
```

**Output**

```
[1,2,3,'A','B','C']
[1,2,3,1,2,3]
3
[1,2,3,4]
```

## Dictionaries

Dictionaries are used to store data in **key : value** pairs

A dictionary is a container which is ordered (As of Python version 3.7 dictionaries are ordered. In 3.6 and earlier they are unordered)

**Example:**

```
car = {    'Brand': 'FIAT',
           'Model': 'Multipla',
           'year' : 2001,
      }
```

To access an item you can simply call its **key**

**Example:**

```
print( car['Model'] )
```

Output

```
'Multipla'
```

# SOME EXAMPLES

Dictionaries can be extremely useful when manipulating datasets that are naturally ordered as key:value.
Take as an example a list of employees. Every employee has its personal information (e.g., telephone number, address, …)

```
employees = {
                'Mario Rossi': { 'Phone Number' : 625,
                                  'Desk Number' : 21,
                                  'Role': 'Financial Advisor',
                                  },
                'Luisa Verdi': { 'Phone Number' : 226,
                                  'Desk Number' : 11,
                                  'Role': 'HR Manager',
                                  },
                 ...
                }
```

If we need the record of any employee (e.g., Maria Bianchi) we will just need to call:

```
employees['Maria Bianchi']
```

If we need her phone number:

```
employees['Maria Bianchi']['Phone Number']
```

# CONTROL FLOW

# CONTROL FLOW

The **control flow** consists in the set of constructs used to control the execution flow of a code.

This means to control **when and how** to execute parts of the code

The main classification of the control flow constructs is:

- **Conditional constructs**
  The execution of the code depends on some condition

- **Iterative constructs**
  a part of the code gests executed zero or more times

- **Fundamental constructs**
  Functions, methods, …

# IF STATEMENT

The main conditional construct is the **if** clause.

General syntax:

```
if condition1:
    do something
elif condition2:
    do something
elif condition3:
    do something
    ...
elif conditionN:
    do something
else:
    do something
```

N.B.
Indenting the code is **FUNDAMENTAL**

# IF STATEMENT

**Condition specification:**

The condition(s) to be satisfied uses the common logical operators:

- comparison: **<, <=, >, >=, ==, !=**
- Identity: **is, is not**
- Membership: **in, not in**
- Logical: **not, and, or**

**Examples:**

```
x = 20
if x<10:
    print('Small’)
elif 10<=x<=20:
    print('Medium’)
elif 20<x<30:
    print('Large’)
else:
    print('Very Large')
```

```
x = 20
if x == 20:
    print('This is 20’)
else:
    print('This is not 
20')
```

```
x = [1,2,3]
if 1 in x:
    print('Ok!’)
else:
    print('Not Ok!')
```

# IF STATEMENT

**Other examples:**

```
mylist = [1,2,3]
number1 = 1
number2 = 3

if (number1 in x) or (number2 in x):
    print('Ok!’)
else:
    print('Not Ok!')
```

```
control_flag = True

if control_flag:
    do something
else:
    do something else
```

The main **iterative construct** is the **for** construct. It allows to execute a block of code a defined number of times

**Examples**

**Create a list with numbers from 0 to 9**

```
mylist = []

for i in range(10):
    mylist.append(i)
```

**Compute the sum of natural numbers from 0 to 1000**

```
sum_of_numbers = 0

for i in range(1001):
    sum_of_numbers += i
```

We have used the `range()` built-in function. This function is very useful to generate a list of integer numbers.

The generic call to the range function is the following:

`range(start, stop, step = 1)`

**Note:** `range(start, stop)` generates numbers from start to stop-1

**<u>Example</u>: Given a list of words, construct a sentence with the proper spacing between words**

```
mylist = ['These', 'are', 'the', 'words', 'of',
'a', 'sentence’]
mysentence = ‘’

for word in mylist:
    mysentence += word
    mysentence += ' `

print(mysentence)
```

**Output**

```
‘These are the words of a sentence’
```

# ITERATIVE CONSTRUCT

The **while** statement allows to execute a block of instructions indefinitely until a particular condition is **True**

**Example: Sum the natural numbers until their sum is lower than 2000**

```
control_flag = True
sum_of_numbers = 0
number = 0
while control_flag:
    if sum_of_numbers + number < 2000:
        sum_of_numbers += number
        number +=1
    else:
        control_flag = False
```

# FUNCTIONS

# FUNCTIONS

A function is a block of code that only runs when it is called.
They can be extremely useful when some task has to be repeated many times with different inputs.

Example: Conversion between Celsius and Farenheit
$°F = °C \times 1.8 - 32$
$°C = (°F - 32)/1.8$

```
def celsius2farenheit(C):
        F = C*1.8 - 32
        return F

def farenheit2celsius(F):
        C = (F-32)/1.8
        return C
```

# FUNCTIONS

```
def celsius2farenheit(C):
        F = C*1.8 - 32
        return F

def farenheit2celsius(F):
        C = (F-32)/1.8
        return C

def temperature_converter(T, mode = None):
        if mode is None:
                print('Error: A conversion mode must be defined’)
        elif mode == 'c2f’:
                return celsius2farenheit(T)
        elif mode == 'f2c’:
                return farenheit2celsius(T)
        else: print('Error: Conversion mode not recognized')
```

```
t = 80

t_conv = celsius2farenheit(t)
print(t_conv)
>> 112.0

t_conv = temperature_converter(t, mode = 'c2f’)
print(t_conv)
>> 112.0

t_conv = temperature_converter(t, mode = 'cf’)
print(t_conv)
>> Error: A conversion mode must be defined
```

# FUNCTIONS

Functions inputs are divided in
- Positional arguments
- Keyword arguments

```
def a_generic_function(arg1, arg2, ..., kwarg1 = def1, kwarg2 = def2, ... ):
```

Functions can take as input other functions:

```
def generic_converter(T, conversion_function = None):
    return conversion_function(T)
```

```
t = 80
t_conv = generic_converter(t, conversion_function = celsius2farenheit)
print(t_conv)
>> 112.0
```

# OBJECT-ORIENTED PROGRAMMING

# OBJECT ORIENTED PROGRAMMING (BRIEF INTRODUCTION)

Object-oriented programming (OOP) is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

OOP is an approach to model concrete things, define their properties and the relationship between things.

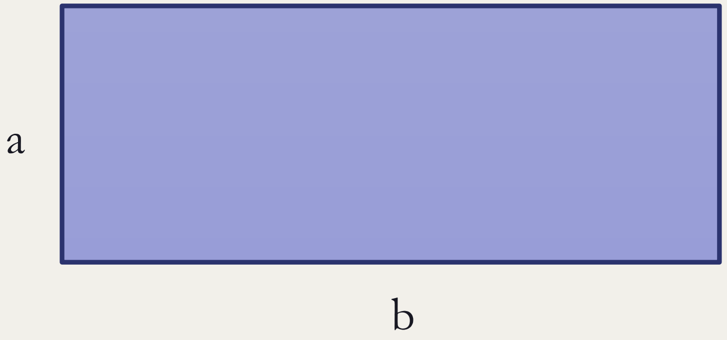
An object has
- Properties
- Methods

**An example**:

We want to define an object representing a rectangle
Later we want to use this object to build two different rectangles and compare them

A rectangle is uniquely defined by the length of its sides **a** and **b.**

The ”rectangle” object should be able to store information about the **properties** of the rectangle (e.g., the **area** and the **perimeter**)



# OBJECT ORIENTED PROGRAMMING (BRIEF INTRODUCTION)

```python
class Rectangle:
        def __init__(self, side1, side2, name):
            self.side1 = side1
            self.side2 = side2
            self.name = name

            # Compute the properties
            self.area = side1*side2
            self.perimeter = 2*side1 + 2*side2




# Define the rectangle R
R = Rectangle(20,3, 'R')

print(type(R))
print(R.area)
print(R.perimeter)

```

Object properties

Initialization of the object

```
<class '__main__.Rectangle'>
60
46
```

# OBJECT ORIENTED PROGRAMMING (BRIEF INTRODUCTION)

We can have our object to do something more interesting. We can add an object **method.** An object method is a function that operates on the object

```python
class Rectangle:
        def __init__(self, side1, side2, name):
            self.side1 = side1
            self.side2 = side2
            self.name = name

            # Compute the properties
            self.area = side1*side2
            self.perimeter = 2*side1 + 2*side2

        def is_square(self):
            if self.side1 == self.side2:
                return True
            else:
                return False


# Define the rectangle R
R = Rectangle(20,3, 'R')
print(R.is_square())
```

```
False
```

The method is_square() operates on the properties of the object itself

The meaning of **self** :

**R.is_square( )**

(it)self

We can also define a method that operates on other objects. For example we would like to compare the area of two rectangles:

```python
class Rectangle:
    def __init__(self, side1, side2, name):
        self.side1 = side1
        self.side2 = side2
        self.name = name

        # Compute the properties
        self.area = side1*side2
        self.perimeter = 2*side1 + 2*side2


    def compare_area(self, other_rectangle):
        if self.area > other_rectangle.area:
            comparison = 'larger'
        else:
            comparison = 'smaller'
        return f'The area of {self.name} is {comparison} than the area of {other_rectangle.name}'

# Define the rectangle R
R = Rectangle(20,3, 'R')
A = Rectangle(18, 5, 'A')
print(R.compare_area(A))
```

```
The area of R is smaller than the area of A
```

Another important aspect when defining objects, is that we can define the relationship rules.

For example, once defined the two rectangles R and A, I would like to know if R>A or R<A.*
So, in my code I'd like to be able to write:

```python
R = Rectangle(20,3, 'R')
A = Rectangle(18, 5, 'A')

print(R<A)
print(R>A)
```

It is possible to define the behavior of standard operations (e.g., +, -, *, >, <, ==, !=, etc) by the so-called **Operator Overloading**

* Here by "greater than" we refer only to the area

# OBJECT ORIENTED PROGRAMMING (BRIEF INTRODUCTION)

```python
class Rectangle:
        def __init__(self, side1, side2, name):
            self.side1 = side1
            self.side2 = side2
            self.name = name

            # Compute the properties
            self.area = side1*side2
            self.perimeter = 2*side1 + 2*side2


        def __gt__(self, other):
            if self.area > other.area:
                return True
            else:
                return False

        def __lt__(self, other):
            return not self > other

# Define the rectangle R

R = Rectangle(20,3, 'R')
A = Rectangle(18, 5, 'A')

print(R<A)
print(R>A)
```

```
True
False
```

# MODULES

# MODULES

Modules (also known as libraries) are collections of (typically) functions.
Modules become extremely useful when writing long programs, to logically organize the code.

Take as example the functions and classes we have defined before. You can write the classes/functions definition in a python file (e.g., **mylibrary.py**)

This way you can easily re-use the code you have already written in another script / from the command line / in another project:

```
import mylibrary
R = mylibrary.Rectangle(20,3, 'R')
```

```
from mylibrary import temperature_converter
t = temperature_converter(40, mode = 'c2f')
```

This way you can logically organize you code in several files. (E.g., only one main file)

# EXTERNAL MODULES

Apart from defining your own module, you can import **external modules** (i.e., modules written by others).

One of the most known is **numpy** (numeric python) which contains many mathematical functions and algorightms.

Suppose you want to create a list of **num** numbers equally spaced between two extremes **a,b**

```python
def my_linspace(a, b, num):
    out = []
    step = (b-a)/(num-1)
    c = a
    for i in range(num):
        c = a + i*step
        out.append(c)

    return out

print(my_linspace(1, 10.0, 5))
```

```
[1.0, 3.25, 5.5, 7.75, 10.0]
```

# EXTERNAL MODULES

Using **numpy** you can do exactly the same thing in **one line**

```python
import numpy
print(numpy.linspace(1, 10.0, 5))
```

```
[ 1.     3.25   5.5    7.75  10.  ]
```

# EXTERNAL MODULES

The main external modules (that usually come directly embedded in the **conda** installation)

| **Module Name** | **Usage** |
|---|---|
| numpy | Numerical operations (arrays, matrices, linear algebra, …) |
| scipy | Numerical methods (optimization, interpolation, integration, differential equations, …) |
| matplotlib | Plotting data |
| os | Interact with operating system |
| sys | Interact with operating system |
| pandas | Manage large datasets |

# EXTERNAL MODULES

External modules need to be installed.

The general way of doing this is:

**(If using anaconda)**

>> conda install modulename

**(If not using anaconda)**

>> pip install modulename

Disclaimer:

Always refer to the module website for installation informations.

# ADVANCED EXAMPLES

# ADVANCED EXAMPLES

**Scripting**

**Scientific plotting**

**Geospatial data visualization**

# USEFUL RESOURCES

Tutorials:
https://www.w3schools.com/python/default.asp
https://www.tutorialspoint.com/python/index.htm

Stack overflow (i.e., where to ask questions and look for already solved problems):
https://stackoverflow.com