Introduction to Python

What is Python

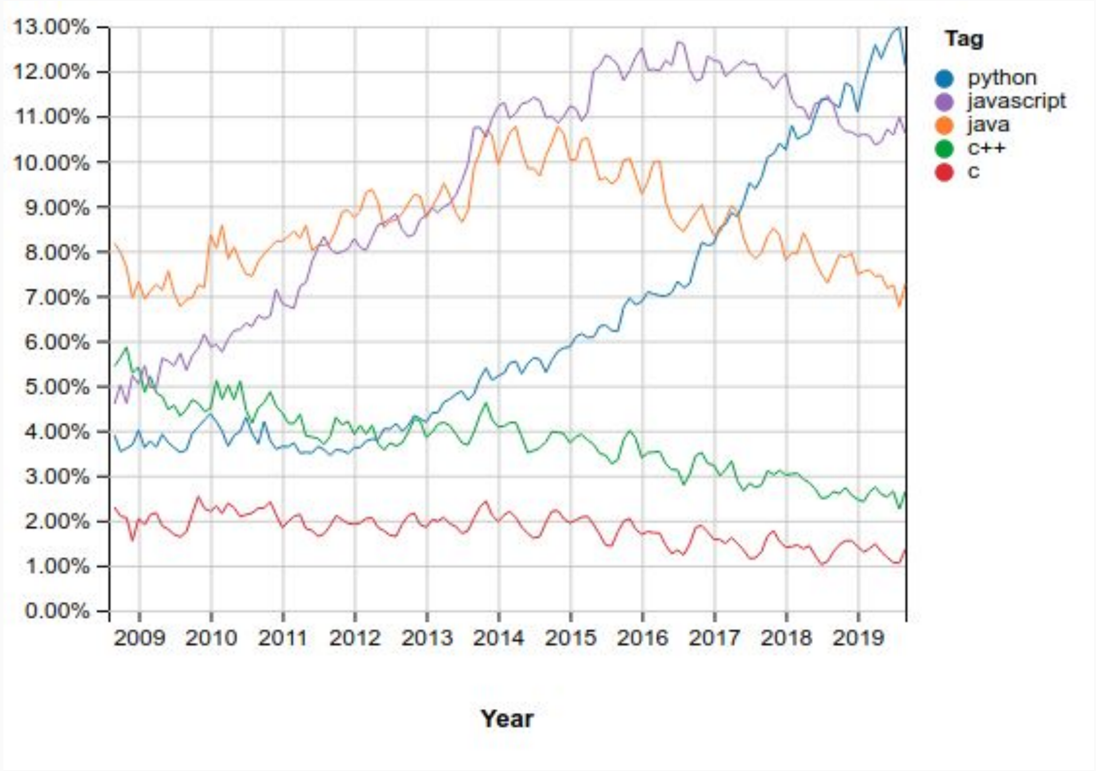
Python is a programming language originally developed by Guido van Rossum and now supported by a large community.

It is object oriented and dynamically typed, particularly suited for scripting but it can be used also for developing applications (web, GUI, etc)

Nowadays, it is largely adopted by researchers since it is easy to learn and use, allowing to fast prototype applications and tests

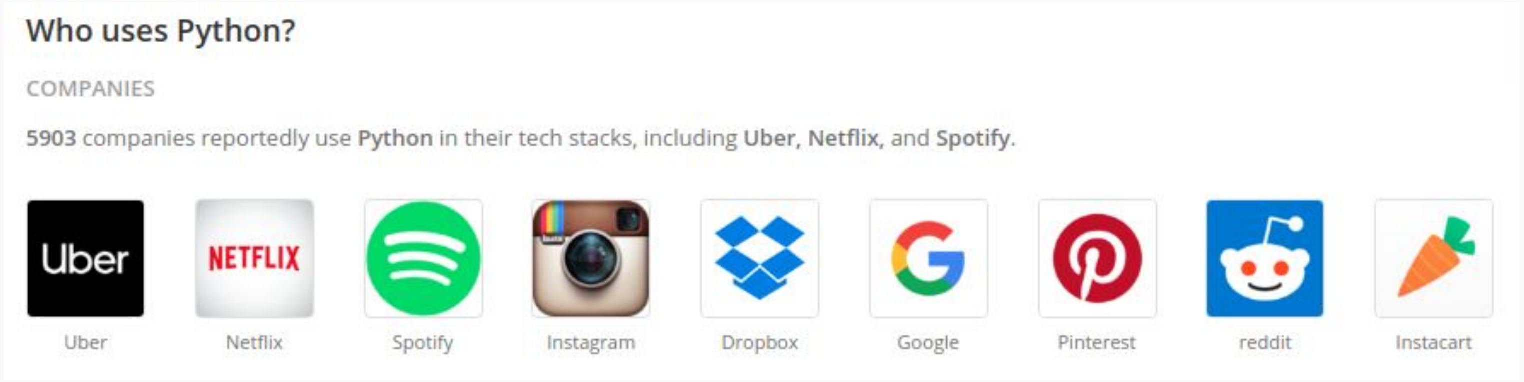
What is Python

Programming languages trend (last updated 14/10/2019) on [StackOverflow]



What is Python

Some companies that uses Python in their stack (according to stackshare)



Who uses Python?

COMPANIES

5903 companies reportedly use **Python** in their tech stacks, including **Uber, Netflix,** and **Spotify**.

What is Python

Unfortunately, there exists two main versions of Python: Python 2.x and Python 3.x

Despite their are quite similar, a program written using Python 2.x may not work for Python 3.x and viceversa

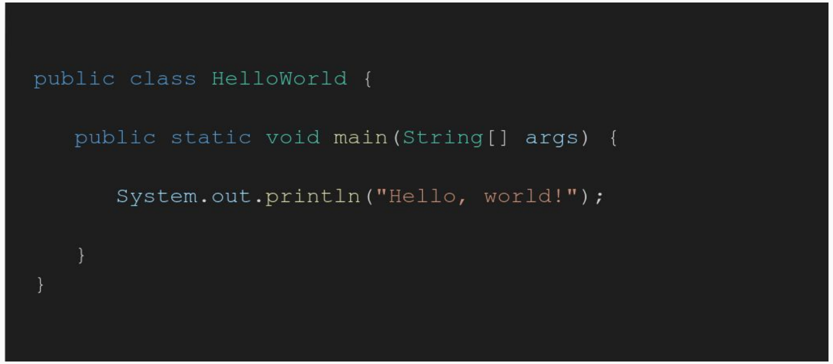
Most of libraries and frameworks (e.g., TensorFlow, PyTorch, NumPy and many more) are available for both

In this course we will use Python 3.x since Python 2.x won’t be supported anymore starting from 2020, but many programmers are still using it

Hello, World

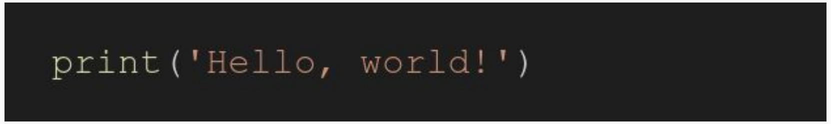
Let’s write our first application, considering the case of the traditional “*Hello, world!*”.

Java source code



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

Python source code



```python
print('Hello, world!')
```

How to run the code

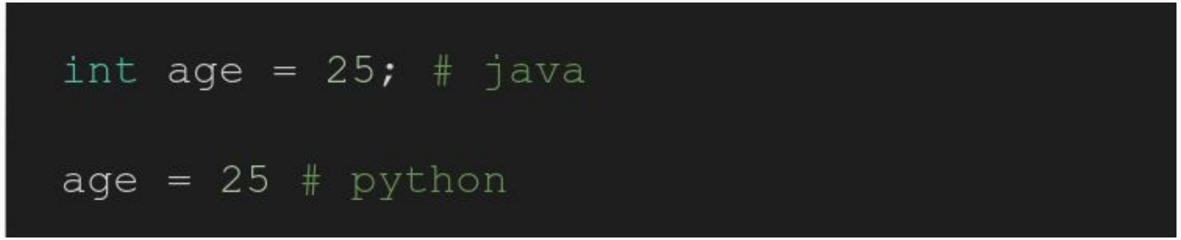
```
javac HelloWorld.java && java HelloWorld
```

How to run the code

```
python hello_world.py
```

Data types

As we said before, Python is a dynamically typed language. This means that we are not forced to explicit the type of each variable, since the compiler is smart enough to understand the type by himself.

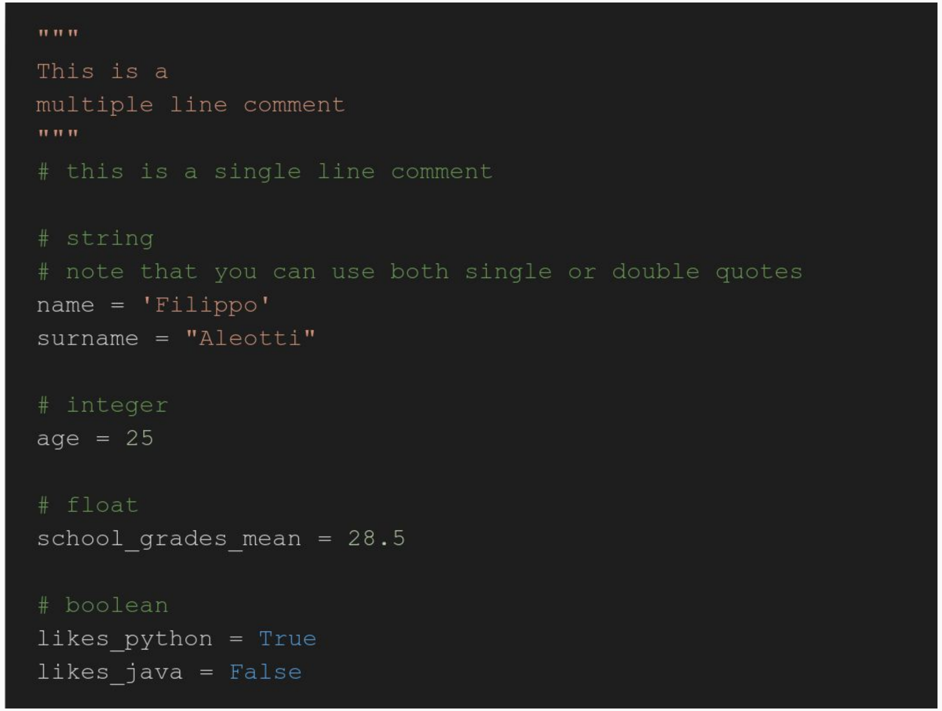


```
int age = 25; # java

age = 25 # python
```

Notice also that the ; character is no more required at the end of the line

Data types

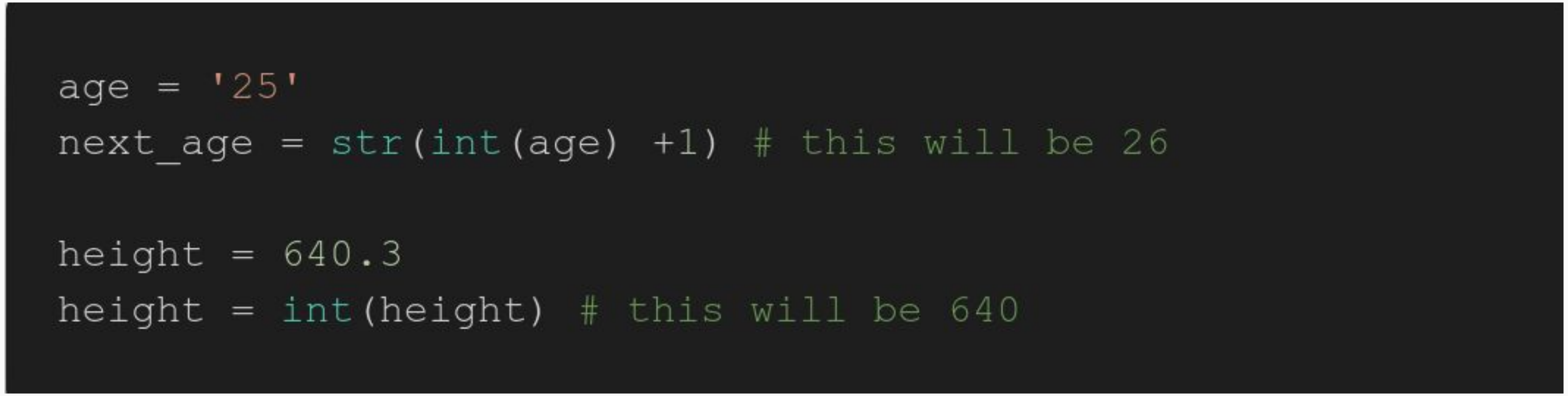
Primitive types of the language are integer, float, strings and boolean



```
"""
This is a
multiple line comment
"""
# this is a single line comment

# string
# note that you can use both single or double quotes
name = 'Filippo'
surname = "Aleotti"

# integer
age = 25

# float
school_grades_mean = 28.5

# boolean
likes_python = True
likes_java = False
```

Data types and primitives: casting

Sometimes we need to change from a type to another.
To do so, we can cast the type (of course paying attention, otherwise an exception will be raised)

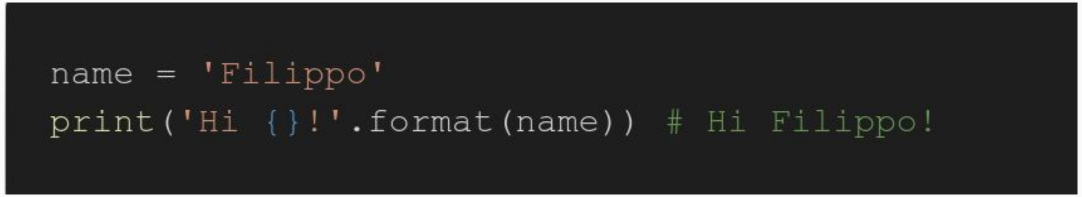


```python
age = '25'
next_age = str(int(age) +1) # this will be 26

height = 640.3
height = int(height) # this will be 640
```

Data types: string formatting

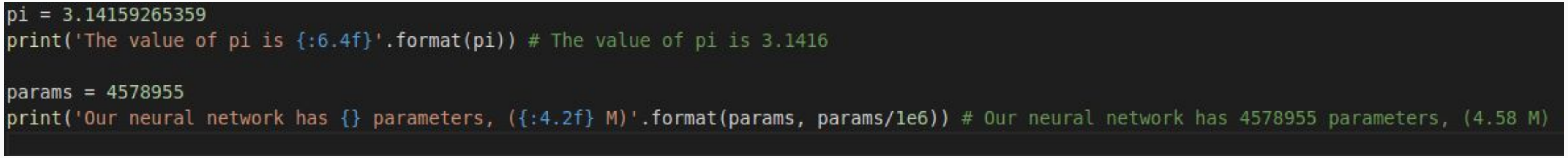
Lot of times we need to format a string to improve the readability

We can format our string using the format method of strings



```python
name = 'Filippo'
print('Hi {}!'.format(name)) # Hi Filippo!
```

but we can even apply more complex operations, such as reducing the numbers of decimals

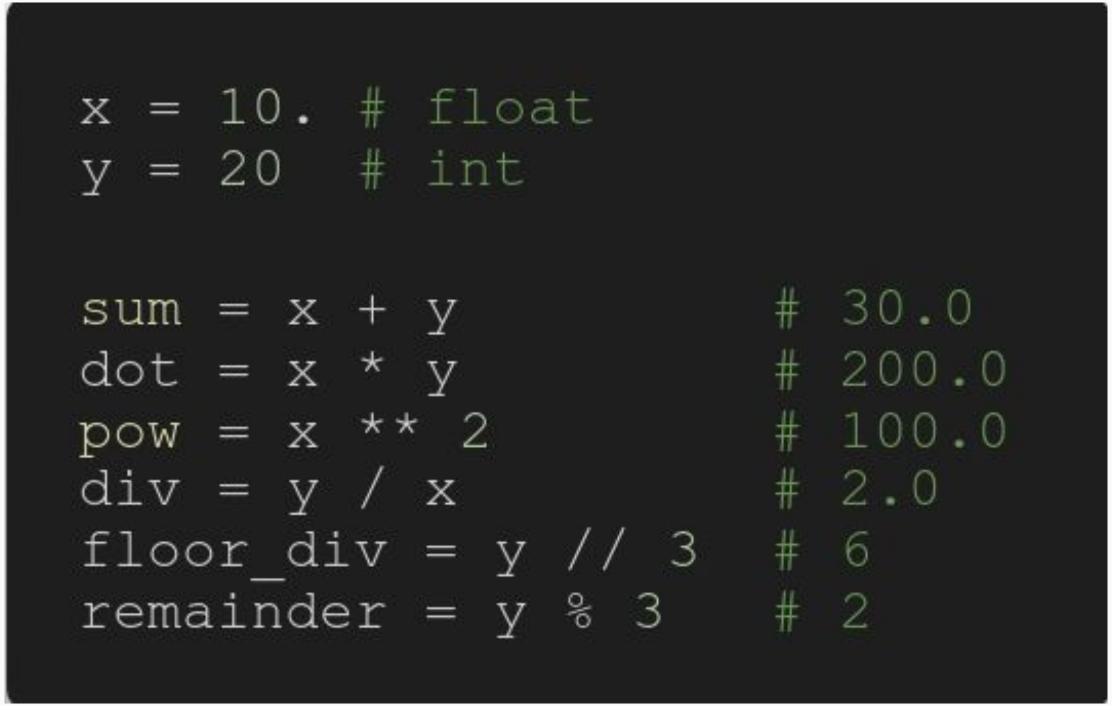


```python
pi = 3.14159265359
print('The value of pi is {:6.4f}'.format(pi)) # The value of pi is 3.1416

params = 4578955
print('Our neural network has {} parameters, ({:4.2f} M)'.format(params, params/1e6)) # Our neural network has 4578955 parameters, (4.58 M)
```

More informations are available here

Data types: math with integers and floats

Mathematical operations are performed as usual



```
x = 10. # float
y = 20  # int

sum = x + y         # 30.0
dot = x * y         # 200.0
pow = x ** 2        # 100.0
div = y / x         # 2.0
floor_div = y // 3  # 6
remainder = y % 3   # 2
```

Data types: strings

Strings are sequences of characters

```
name = 'Filippo'
first_char = name[0] # F
last_char = name[-1] # o
# as for list, len returns the number of elements
length = len(name)  # 7
# as for list, the + allows to concat strings
full_name = name + ' Aleotti' # Filippo Aleotti
name = name.replace('F','f')  # replacing all F with f
```

They <u>do not</u> support item assignment or mixing characters with int or float

```
name[0] = 'F'    # TypeError: 'str' object does not support item assignment
name = name + 2 # TypeError: cannot concatenate 'str' and 'int' objects
```

Data types: tuples

Python offers more complex data structures to handle data. Some of the most important are tuples, lists and dictionaries

Tuples are <u>immutable</u> list of values comma-separated

```
languages = 'python', 'java', 'c++', 'c#', 'ruby'   # this is a tuple
languages = ('python', 'java', 'c++', 'c#', 'ruby') # but this is the convention for tuple
```

They can be accessed by index

```
first = languages[0]  # get the first element of the tuple
last  = languages[-1] # get the last element of the tuple
third_and_fourth = languages[2:4] # it returns the tuple ('c++','c#')
```

But the values cannot be changed (otherwise TypeError will be raised)

```
languages[1] = 'javascript' # TypeError: 'tuple' object does not support item assignment
```

Data types: lists

If you plan to change your data, a better option consist in using lists instead of tuples.

```
languages = ['python', 'java', 'c++', 'c#', 'ruby']  # this is a list
void = list() # empty list
void = []      # another empty list
mixed_list = ['1', 1, True, None] # we can create list with element of different types
```

As for tuples, we can get elements by index

```
first = languages[0]   # get the first element
last  = languages[-1] # get the last element
third_and_fourth = languages[2:4] # it returns the list ['c++','c#']
```

But, differently from tuples, lists are editable

```
languages[1] = 'javascript' # now the list will be ['python', 'javascript', 'c++' , 'c#', 'ruby']
languages.append('c') # now the last element of the list is 'c'
languages.insert(3,'go') # insert at index 3 the string 'go'
```

Data types: dictionaries

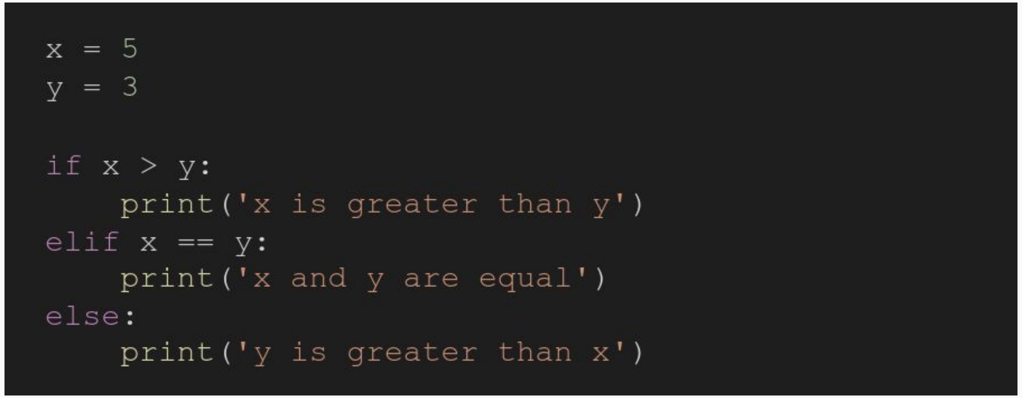
Dictionaries allow to store key-value couples

```
person = {}      # this is a dict
person = dict() # another dict

# adding elements to the dict
person['name'] = 'Filippo'
person['surname'] = ' Aleotti'

# in-line dict creation
person = {
     'name': 'Filippo',
     'surname': 'Aleotti',
     'age': 25
}

name = person['name'] # 'Filippo'
```

Statements

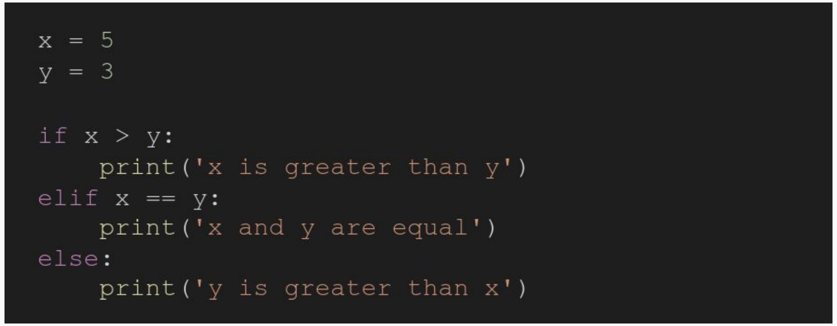
As for other programming languages, Python exposes **if** statement for conditions and **for** and **while** to iterate.

Using the if, we can execute a set of instructions a given a certain condition



```
x = 5
y = 3

if x > y:
    print('x is greater than y')
elif x == y:
    print('x and y are equal')
else:
    print('y is greater than x')
```

Statements: if



```python
x = 5
y = 3

if x > y:
    print('x is greater than y')
elif x == y:
    print('x and y are equal')
else:
    print('y is greater than x')
```

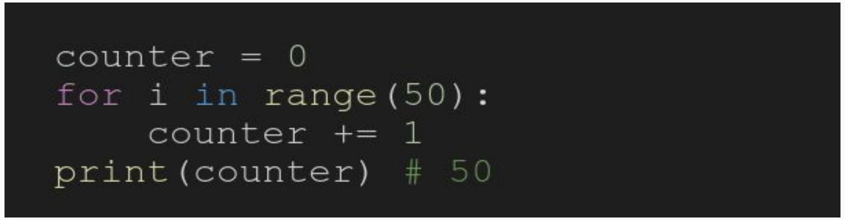
NOTE: you can notice that the instruction to execute are indented with respect to the condition. In Python indentation is mandatory and if not respected the compiler will throw an exception.

You are free to indent with tabs or spaces, but you can not use both in the same script

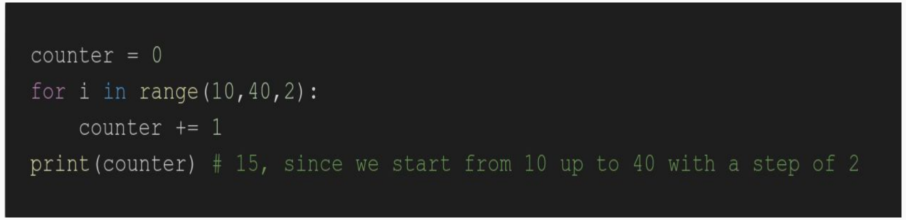
Indenting a block is the equivalent of wrapping a set of instructions with { } in Java

Statements: for

For loops are useful to do some operations N times

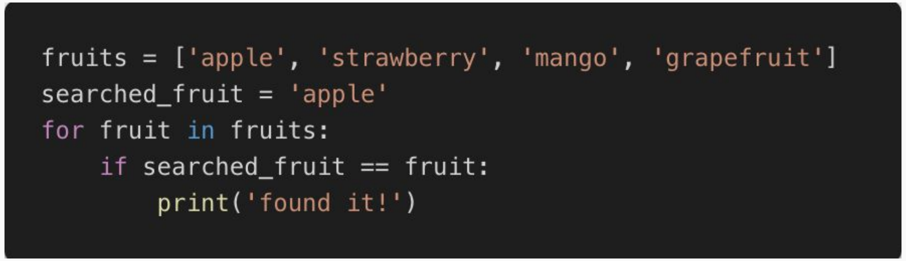


```python
counter = 0
for i in range(50):
    counter += 1
print(counter) # 50
```

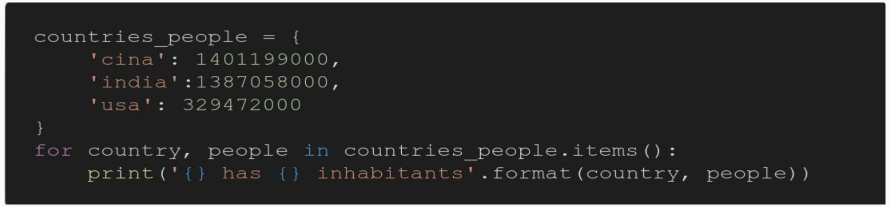


```python
counter = 0
for i in range(10,40,2):
    counter += 1
print(counter) # 15, since we start from 10 up to 40 with a step of 2
```

We can iterate also over collections



```python
fruits = ['apple', 'strawberry', 'mango', 'grapefruit']
searched_fruit = 'apple'
for fruit in fruits:
    if searched_fruit == fruit:
        print('found it!')
```



```python
countries_people = {
    'cina': 1401199000,
    'india':1387058000,
    'usa': 329472000
}
for country, people in countries_people.items():
    print('{} has {} inhabitants'.format(country, people))
```

Statements: while

While loop is used to continue iterating until a given condition is no more verified

```
found = False
searched_fruit = 'apple'
index = 0
fruits = ['mango', 'peach', 'pineapple', 'apple', 'strawberry']

while not found and index < len(fruits):
    if fruits[index] == searched_fruit:
        found = True
    else:
        index += 1

result = '{} is at index {}'.format(searched_fruit, index) if found else '{} not found'.format(searched_fruit)
print(result)
```

This script will print “apple is at index 3”

Functions

In order to improve the readability but also the reusability of your code, it is a good practice to encapsulate a set of instruction inside a **function**. This function may be called both from the same and even also from other scripts.

Functions are defined using the **def** keyword

```python
def dot(x,y):
    return x*y
```

In this case, **x** and **y** are arguments of the **dot** function

Functions: default arguments

In Python, we can assign default values to arguments of the function

```
def pow(x, y=2):
    return x**y

x1 = pow(5)   # x1=25 , y assumes the default value 2
x2 = pow(5,3) # x2=125, y is 3
```

We can also change a specific set of values

```
def my_transformation(x, alpha=1, beta=5):
    return x**alpha - beta

x1 = my_transformation(5, beta=2) #x1=3, we set beta=3 and alpha=1
```

Scopes

The scope of a variable defines its visibility: it depends by where the variable has been defined and influence how we can use that variable

In Python we are not forced to declare always a variable (even it may be a good practice, to improve the readability of the code)

```
x = 5
if x > 3:
    b = True

result = b
```

In this case b=True since the condition x>3 is verified, but with x=1 the program would raise *NameError: name 'b' is not defined*

Scopes

All the variables <u>declared </u>inside a function are local, so they are not visible outside that function

```python
def print_name():
    name = 'Filippo'
    print(name)

print_name()    # Filippo
my_name = name # NameError: name 'name' is not defined
```

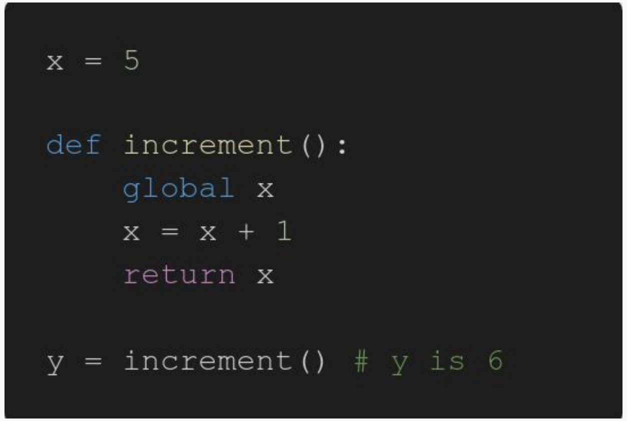
Variables defined in global scope are visible inside local ones

```python
surname = 'Aleotti'

def print_full_name():
    name = 'Filippo'
    print(name+ ' '+ surname)

print_full_name() # Filippo Aleotti
```

Scopes

Global variables are visible but not editable inside a function

Using the **global** we are able to modify global variables



```
x = 5

def increment():
    global x
    x = x + 1
    return x

y = increment() # y is 6
```

If we had forgotten the **global**, the program would have raised *UnboundLocalError: local variable 'x' referenced before assignment*

Objects

In OOP, objects wrap data and methods to handle with a specific element of the domain.

For instance, in our exam-assistant application we can model a student as an object to keep trace of all the **properties** of a particular student (e.g., his ID) and specify **methods** valid for all the students

In particular, **Student** will be a **class** in our application, while the specific student Filippo Aleotti (with name="Filippo Aleotti" and ID="0123456") is an **object** (i.e., an instance of class **Student**)

Objects

Let’s create the **class Student** and our first **object** of that class

```python
class Student(object):

    def __init__(self, name, id):
        self.name = name
        self.id = id

    def introduce_yourself(self):
        print('Hello, my name is {} and my ID is {}'.format(self.name, self.id))

student = Student('Filippo Aleotti', '0123456')
student.introduce_yourself()
```

Objects

In particular:

- In Python, by default all objects inherit by object

```
class Student(object):
```

- We defined the **constructor** of our object using the reserved method __init__.

  The keyword **self** is like the **this** operator in Java, allowing to refer to the object. In the constructor we set the two properties (name and id) of object

```
def __init__(self, name, id ):
    self.name = name
    self.id = id
```

Objects

- We defined the method *introduce_yourself*

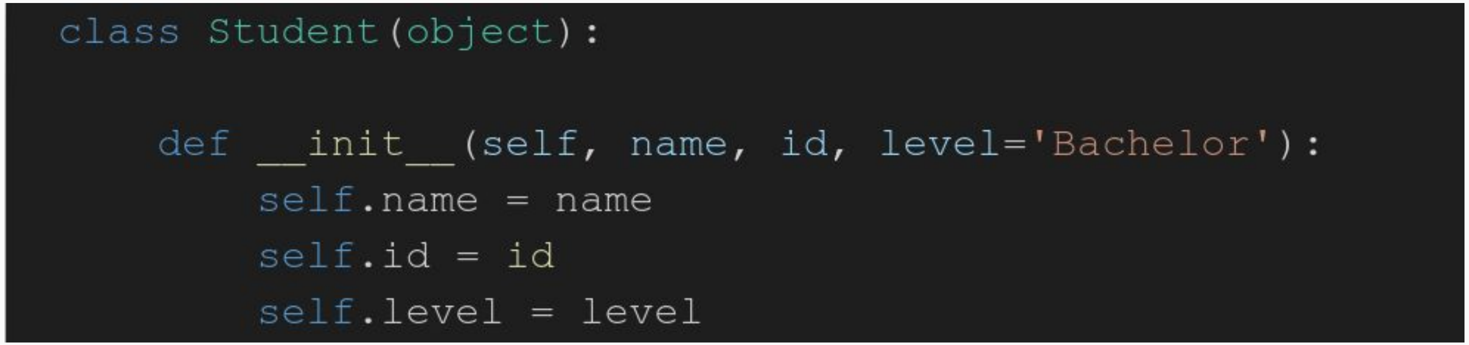
```
def introduce_yourself(self):
    print('Hello, my name is {} and my ID is {}'.format(self.name, self.id))
```

- We then created an instance of **class Student** and invoked his method *introduce_yourself*

```
student = Student('Filippo Aleotti', '0123456')
student.introduce_yourself()
```

Objects: defaults in constructor

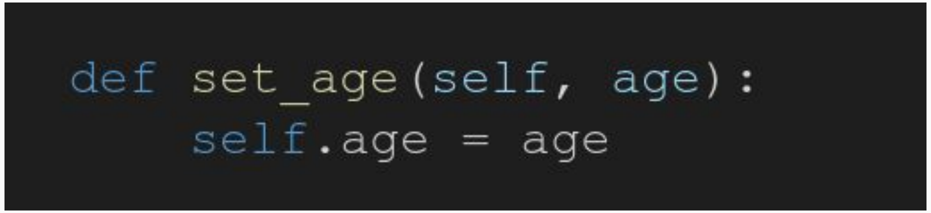
In Python you cannot define multiple constructors for your class, but you are free to use default values



```python
class Student(object):

    def __init__(self, name, id, level='Bachelor'):
        self.name = name
        self.id = id
        self.level = level
```

Objects: properties definition

You are free to define wherever you want properties of your object

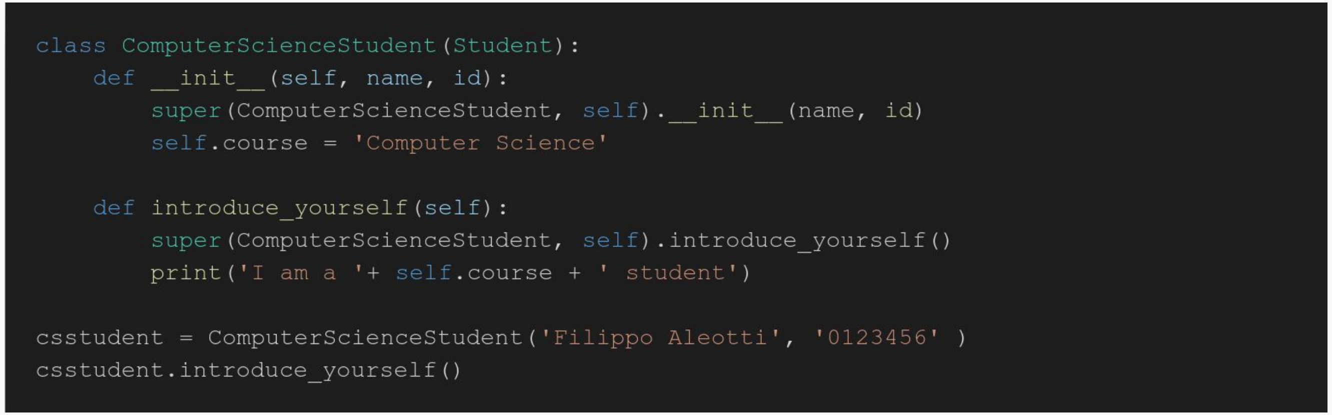


```python
def set_age(self, age):
    self.age = age
```

Objects: inheritance

Inheritance is useful to recycle parent's methods and properties, avoiding to rewrite already defined pieces of code

You can access to parent's method using the **super** keyword



```python
class ComputerScienceStudent(Student):
    def __init__(self, name, id):
        super(ComputerScienceStudent, self).__init__(name, id)
        self.course = 'Computer Science'

    def introduce_yourself(self):
        super(ComputerScienceStudent, self).introduce_yourself()
        print('I am a '+ self.course + ' student')

csstudent = ComputerScienceStudent('Filippo Aleotti', '0123456' )
csstudent.introduce_yourself()
```

Multiple inheritance is allowed (but you can avoid it exploiting the same consideration used for languages in which it is forbidden)

Imports

Sometimes we have to import functions or objects created by us in a different file (e.g., utils) or even third-party libraries (e.g., NumPy, TensorFlow)

To do so, Python offers the **import** keyword

```
import numpy
import tensorflow as tf
from utils import read_image, write_image
```

NOTE: Third-party libraries firstly have to be installed on your system/environment. Package manager (Pip, Anaconda etc) can help you to tackle this task

Some examples

Installation

Before we start to code, we need to install Python 3.x if not already installed on our machine

Linux machines already have Python, while Mac and Windows users have to download and install it respectively from here and here

It is not mandatory, but for this course we suggest to use **Ubuntu** when it is possible: some installations (e.g., TensorFlow) may be simpler than using other solutions.

Creation of a virtual environment

Even if your system has Python already installed, it is a good practice to create a virtual environment, since it allows to install dependencies and packages without create conflicts with other environments (like in a sandbox).

```
sudo apt update
sudo apt install python3 python3-pip
sudo pip3 install virtualenv
virtualenv -p python3 venv
source venv/bin/activate
```

venv is the name of the new environment that will be created, while the last command activate the env

Pip

In the previous slide, we ran the command "*pip3 install virtualenv*"

Pip (for Python 3.x is Pip3) is a package manager: it allows to install modules and packages available in a store

For instance, if we need OpenCV, which is a wide use open source computer vision and machine learning library, we need to run

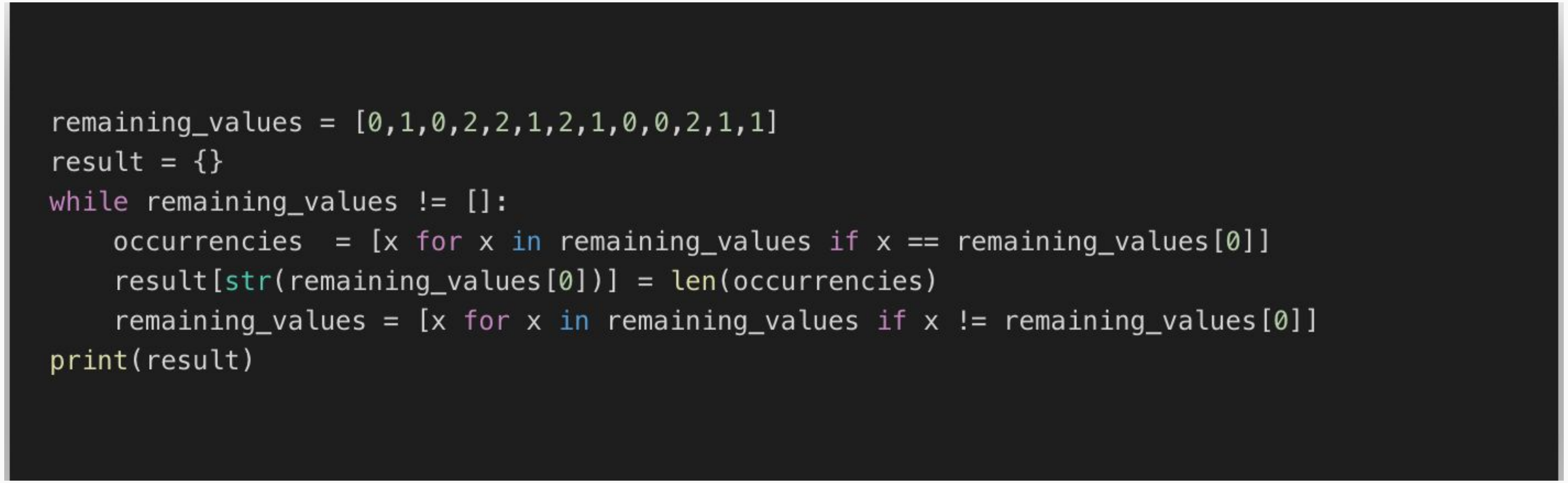
```
pip3 install opencv-python
```

In the PyPI store you can search if the package you need is available

Exercise 1

Given the list [0,1,0,2,2,1,2,1,0,0,2,1,1], store the frequency of each value in a dict, where the key is the value.

Exercise 1



```python
remaining_values = [0,1,0,2,2,1,2,1,0,0,2,1,1]
result = {}
while remaining_values != []:
    occurrencies  = [x for x in remaining_values if x == remaining_values[0]]
    result[str(remaining_values[0])] = len(occurrencies)
    remaining_values = [x for x in remaining_values if x != remaining_values[0]]
print(result)
```

In this solution, we used the list comprehension to iterate over the list with remaining values
It returns a new list, made up by those values that satisfies a given condition

Exercise 2

We stored into the file *temperatures.txt* many of temperatures (expressed in census degrees) for a set of places. Each line contains the values, separated by space, for a specific place.

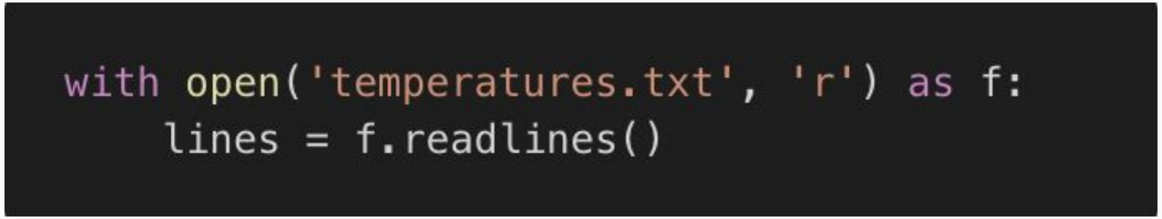
Find the maximum and minimum temperature for each place

Find the global maximum and the minimum temperature

Exercise 2: tips

- First of all, we need to read the values from the *temperatures.txt* file

In Python, we can manage with files using the **with** statement



```python
with open('temperatures.txt', 'r') as f:
    lines = f.readlines()
```

The **with** block masks the opening and the closing of the file
In the example we opened the file *temperatures.txt* in *read* modality (**‘r’**), getting back the file pointer as **f.** Then, we use **f** to store each line of the file into a list

Other modalities are write (**‘w’**), read-write(**‘rw’**), read binary (**‘rb’**) and write-binary (**‘wb’**)

Exercise 2: tips

- ***sys.float_info.max*** and ***sys.float_info.min*** return respectively the maximum and the minimum float values
- We need to remove the end-line character. The function strip() of string do the task
- We read strings from the file, so we need to cast them into float values if we have to perform

Exercise 2: solution

```python
import sys
with open('temperatures.txt', 'r') as f:
    lines = f.readlines()

mins = []
maxs = []
global_min = sys.float_info.max
global_max = sys.float_info.min

for line in lines:
    local_min = sys.float_info.max
    local_max = sys.float_info.min
    local_temperatures = line.strip().split(' ')

    for temp in local_temperatures:
        temp = float(temp)
        if temp >= local_max:
            local_max = temp
        if temp <= local_min:
            local_min = temp
    mins.append(local_min)
    maxs.append(local_max)

    if local_min <= global_min:
        global_min = local_min
    if local_max >= global_max:
        global_max = local_max
```

Exercise 2: another solution

```python
with open('temperatures.txt', 'r') as f:
    lines = f.readlines()

mins = []
maxs = []
for line in lines:
    local_temperatures = line.strip().split(' ')
    float_temperatures = map(float, local_temperatures)
    local_min = min(float_temperatures)
    local_max = max(float_temperatures)
    mins.append(local_min)
    maxs.append(local_max)
global_max = max(maxs)
global_min = min(mins)
```

In this solution we exploit the built-in functions **min** and **max** to find the minimum and maximum of a collection.
Moreover, we used the **map** function to apply an operation(in our case, the casting) to each element of a collection