ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI INGEGNERIA CIVILE, CHIMICA, AMBIENTALE E DEI MATERIALI

PROFESSIONAL MASTER'S PROGRAMME 2ND LEVEL
SUSTAINABLE AND INTEGRATED MOBILITY IN URBAN REGIONS

Introduction to Python

Filippo Aleotti:
DISI– University of Bologna

MODULO:
Autonomous vehicles , Electric cars and recharging systems

Imola, 29/11/2019

With the unconditional support:

With the contribution of:

Hello!

Speaker: Filippo Aleotti

I pursued my Bachelor and Master degree in Computer Science at University of Bologna.

Currently, I am a PhD student of the course Structural and Environmental Health Monitoring and Management at University of Bologna.

I am working on Monocular depth estimation.

What we will see today

In this lecture, you will be introduced to core concepts of programming languages, then we will look at Python. Finally, we will do some practice.

Introduction to programming languages

Introduction to programming languages

Electronic devices are everywhere: smartphones, tablets, personal computers and sensors are just some examples of tools we use in our day life.

They are able to solve complex tasks quickly, automatically, with less errors.

For instance, consider you have to check a long document: it would be time-consuming and probably you will make some errors (due to distractions, tiredness etc). Let a pc do it for us!

Introduction to programming languages

Computers can solve a problem N times, automatically in a low amount of times

Introduction to programming languages

To do so, we need to program the device: in fact, electronic devices do not "talk" human language, but machine language (a language in which only 0 and 1 are allowed)!

```
0010101110
1011010100
1101011010
1111110101
0010110101
```

Unfortunately, machine language is extremely hard to read and write for humans… so, how can we program an electronic device easily? We can do it by using a **programming language**!

Introduction to programming languages

As the name suggests, a programming language is a formal set of rules and keywords we use for programming.

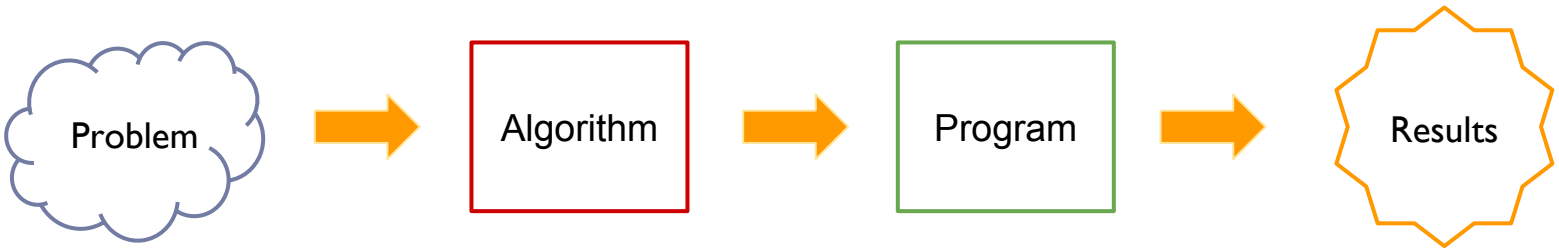
By using a programming language, we can define in a **program** the set of instructions the computer has to apply to solve the task. Then, the program is turned into machine code (generally using a **compiler**), so the computer can run it to solve the problem.

Introduction to programming languages

Given a problem, we first need an **algorithm**, then we turn this algorithm into a **program**.

Algorithm: set of logical steps we have to perform to solve the problem.

Program: implementation of the algorithm using a programming language.



Introduction to programming languages

program

```
#include <stdio.h>
main(){

printf("HelloWorld");
}
```

Compiler

Binary code

```
0010101110
1011010100
1101011010
1111110101
0010110101
```

Results

Introduction to programming languages

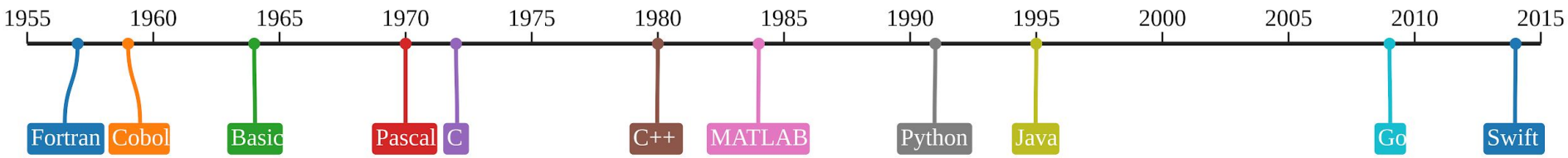
Some languages are not compiled (turned into binary form), but they are parsed at runtime by an **interpreter** of the language.

The interpreter is a computer program that read a program written in a particular language and executes it line by line.

Sometimes, the interpreter creates intermediate outputs (called bytecode). The bytecode is then translated into specific machine code using a Virtual Machine. This allows to use the same bytecode on different machines.

Introduction to programming languages

Starting from ‘50, a large number of programming languages have been developed.



The picture shows the timeline of some famous programming languages, but the real number of programming languages is larger than this (~700 notable for Wikipedia).

Introduction to programming languages

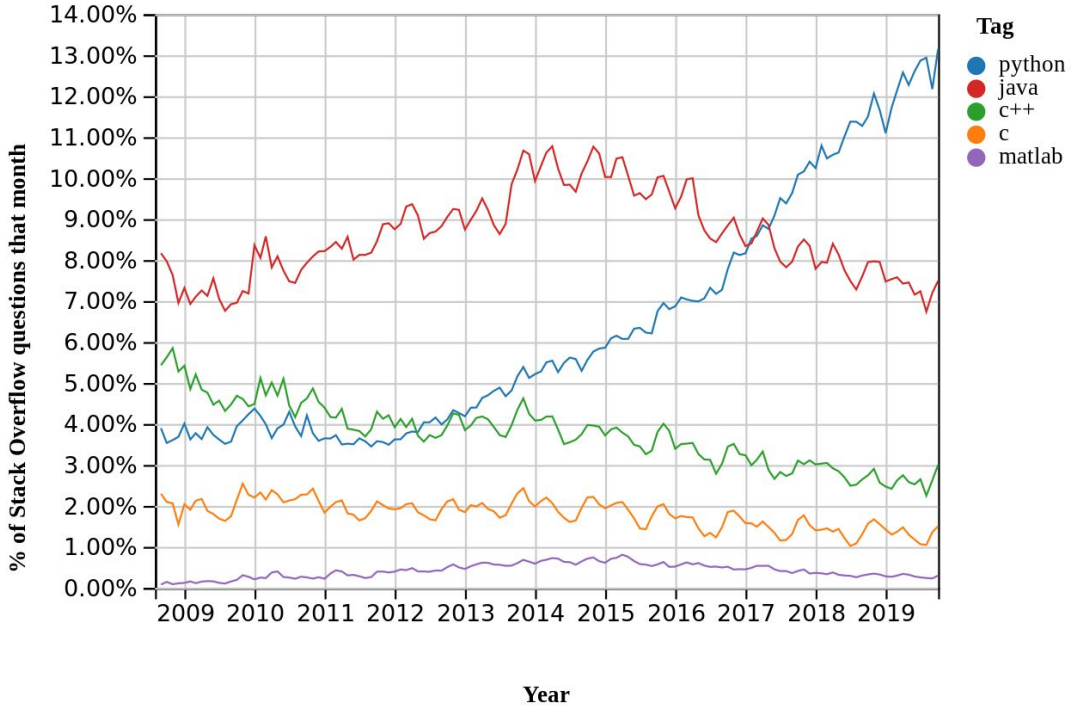
Why so many programming languages?

Many of these languages have the same capability power (so we can solve a problem using different languages), but they may differ greatly in terms of libraries, supports and communities, security, maintainability (legacy language vs active one).

For instance, it is easier to develop a web server in Java or Python than in C, given a large number of available libraries.

Introduction to programming languages

Some programming languages are not used anymore, some other instead are extremely adopted. Let's have a look at StackOverflow to discover programming languages trends.



Introduction to programming languages

As you can see, Python is exhibiting an exponential increasing curve.

This is mainly due to:

- Large number of additional libraries
- Large support
- Easy to learn
- Simple to read and understand
- It allows fast prototyping and scripting style

In the following, we will take a closer look at it.

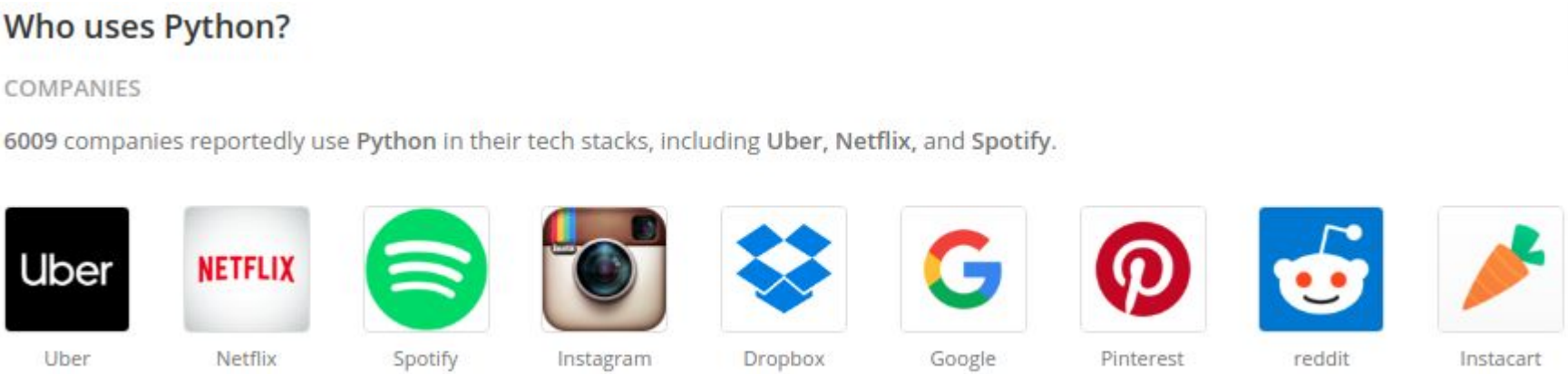
Python

Python

Python is an interpreted* programming language created by Guido Van Rossum and released in 1991. Today, Python is the second most used language on GitHub.

*Actually, a Python program is first converted into bytecode, then interpreted

Python



Picture from stackshare

According to stackshare, these are just few of the big IT companies that uses Python to realise their products.

Python

Python is mostly used for:

Web developing

Machine Learning / Deep Learning

Data Visualization and analysis

logos are from ml-logos project, original web sites and Wikipedia

Python

There exists two main families of Python: Python 2.x and Python 3.x. They are quite similar, they differ in some details (e.g., print).

From January 2020 Python 2.x won't be supported anymore.

In this lecture, we will use Python 3.x.

Python

We can use Python in various way:

- in many OS (macOS, Ubuntu, Fedora, etc) Python is installed by default
- for Windows users, Python installer can be downloaded for free from the official site
- we can use online editors, such as Jupyter

Hello, world!

Let's write our first Python program: we desire to write in our terminal the "Hello, world!" sentence.

To do so, we have to create a **hello.py** file, open it with a editor, write the following instruction and save.

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

Now, the file hello.py contains the instruction to print out the desired sentence, we have just to invoke the interpreter.

Hello, world!

Python interpreter can be invoked inside a terminal.

Once the terminal (PowerShell on Microsoft) is opened, we can run the interpreter as it follows:

python3 hello.py

At the end, we will obtain the desired outcome

```
faleotti@pascal:~/Documents/MASTER/python/scripts$ python hello.py
Hello, world!
```

Variables and Data types

Variables

Inside your programs you expect to read data, apply operations and transformations on them and finally return a results.

Programming languages offer you **variables** to store data, and operations on variables to change them

In general, creating a variable means allocating a portion of the RAM memory of your pc. When you do not need anymore such space, you (or eventually someone for you, e.g. the garbage collector) deallocate the variable, making the space available again.

Data type

In Python, variables have a data type depending on how we are going to use them in the program.

For instance, we expect to handle differently your name and the number of rooms in your house, since the former is related to text while the latter is a numerical entity.

Python offers many data types, including:

- Numeric, so Integer, Float and Complex
- Boolean
- Sequence, so String, List, Tuple and Dictionary

Empty variable

We can use the value None to create a variable that currently has no value. It will be (possibly) modified later in the code

```
age    = None
people = None
```

Maybe now we do not know the correct value for these variables, but we have already prepared them, so they can be used and changed later

Comment

Sometimes we want to add comments to our program. Comments may help us and other people when reading the code, and commented lines won't be executed

You can comment a line using #, and more lines opening and closing three times the single or the double quotes

```
# this line is commented
'''
  multiple
  line
  comment
'''
```

NOTE: do not abuse of comments!

Integer

We use integer variables to represent elements of $\mathbb{Z}$ set

For instance, the age of a person or the number of people in this room can be stored in integer variables

```
age    = 26 # integer
people = 20 # integer
```

Float

We use float variables to approximate elements of $\mathcal{R}$ set

The height in meter of a person or a payment in $ you made with your credit card are examples of float values

```
height  = 1.87  # float
payment = 200.2 # float
```

Math

With integer and float variables we can perform mathematical operations, such as addition, subtraction, multiplication and division

```
x = 10.  # float
y = 20   # int
add = x + y          # 30.0 float
dot = x * y          # 200.0 float
power = x ** 2       # 100.0 float
div = y / x          # 2.0 float
int_div = y // 3   # 6, int
modulo = y % 3    # 2 (remainder of modulo division), int
```

Math

We can check equality with == and inequality with != .

Moreover, we can force the order of operations using ()

```
x = 10. # float
y = 20  # int
z = 2
equality = x == y   # False, boolean
inequality = x != y # True, boolean
result1 = x+y*z     #  50, the result of y*z is added to x
result2 = (x+y)*z   # 60, the result of x+y is multiplied by z
```

Boolean

We use boolean variables to store True or False values. For instance, if a payment has be done we can set a flag payment equals to True, while if a flight passenger hasn't request to check-in yet, we can create a checked variable set to False

```
checked = False
payed   = True
```

We can perform logical operation on booleans

```
in_airport = True
in_bus_station = False
checked = False
ready_to_flight = in_airport and checked # False
ready_to_move = in_airport or in_bus_station # True
```

String

We use string variables to store textual information.

Your name and surname are examples of strings. In Python, we can create a string variable using single or double quote

```
name    = 'Filippo'
surname = "Aleotti"
city    ='Bologna'
empty   = ''
```

Notice that we can create also empty string, just opening and closing the string

String

We can think about strings as sequences of characters, and we are able to access these characters using **index notation**.

Moreover, we can exploit methods of string to perform quite common operations (e.g., to replace some characters)

```
name = 'Filippo'
first_char = name[0] # F
last_char = name[-1] # o
length = len(name) # 7
name = name.replace('F','f')  # replacing all F with f
```

Note: the index starts from 0, so first char is placed at index 0, the second at index 1 and so on.

String

Given two or even more strings, we can merge them into a single, longer string using the + operator. This operation is called **concatenation** of strings

```
name      = 'Filippo'
surname   = "Aleotti"
full_name= name + ' ' + surname
```

In this case, *full_name* is a new string obtained concatenating name, surname and a single space string

String

Strings are immutable objects: it means that, once created, they cannot be modified. So, using **index notation** we are able to read elements but not to change them.

In addiction, we cannot mix characters and integers or float using +

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

String

Given two strings, we can check if they are equal or not using ==, and != to check the inequality

```
name      = 'Filippo'
name2     = "Filippo"
surname   = "Aleotti"
surname2  = "Aleottii"
same_name = name == name2 # True, boolean
same_surname = surname == surname2 # False, boolean
different_surname = surname != surname2 # True, boolean
```

String

The **len** function returns the length of a string (i.e., its number of characters).

```
name     = 'Filippo'
l = len(name) # 7
```

Print

Given a variable, we are able to print its value using the print function

```
name = 'Filippo'
print(name) # Filippo
age  = 26
print(age) # 26
print('your name is '+ name) # your name is Filippo
```

We have already seen the print function in the *Hello, world!* example.

Cast

Sometimes we have to change the type of a variable.

Suppose you have read data from an input form in a web page, but such value represents a payment. We have to **cast** the value from text to float.

```
txt_payment   = '260.12'
float_payment = float(txt_payment) # 260.12
int_payment   = int(float_payment) # 260
str_payment   = str(int_payment)   # '260'
```

Note: you might lose accuracy with casting!

Cast

Not all casting transformations are allowed: for instance, string to float casting is granted if and only if the string contains only values that represent numbers

```
txt_payment   = '260.12DOLLARS'
float_payment = float(txt_payment) # ValueError: invalid literal for float(): 260.12DOLLARS
```

In case of failure, Python will throw you an error message

Format

To improve the readability of our strings while printing, we may want to format them.

```
name    = 'Filippo'
print('Hello {}!'.format(name)) # Hello Filippo!
pi = 3.14159265359
print('The value of pi is {:6.4f}'.format(pi)) # The value of pi is 3.1416
print('Hello {}, the value of pi is {:6.4f}'.format(name, pi)) # Hello Filippo, the value of pi is 3.1416
```

We can even format numbers in strings: a placeholder with f will contain a number, and we are able to set both the number of digits (6) and decimals (4)

More info here

Collections

So far, we have used variables able to store a single value, with Strings as only exception, since they contain a sequence of characters.

However, in many practical situations we desire to store together many values: for instance, collect all the credit card payments each person in this room did last month.

It is preferable to group values, in order to look in a single place when we have to read/modify them

Python offers various data structures for this purpose, such as Tuples, Lists and Dictionaries

Tuple

A tuple is an immutable 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
```

Using index notation we access to elements of the tuple

```
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#')
```

However, we can't modify the elements

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

Slice notation

In the previous example we saw also how to get more than a single element from tuple using index notation

```
languages = 'python', 'java', 'c++', 'c#', 'ruby'    # this is a tuple
third_and_fourth = languages[2:4] # it returns the tuple ('c++','c#')
```

The notation [start : stop] is called slice notation in Python, since it allows to extract a slice of data from our data structure

The outcome slice contains all elements starting from start index until stop-1 index (i.e., element at stop index is not included)

Slice notation

Slice notation in valid for all sequences (i.e., also for strings).

We can also use negative index:

```
languages = 'python', 'java', 'c++', 'c#', 'ruby'    # this is a tuple
subset = languages[1:-1] # it returns the tuple ('java', 'c++', 'c#')
```

In this case, we are considering start as 1, while stop as length of the sequence - 1 (i.e., 5-1=4). In other words, we are taking all the elements at indexes 1,2,3.

Slice notation

Actually, the full notation is [start : stop: step], with 1 as default value for step

We can use the step to change the distance between two consecutive values

```
languages = 'python', 'java', 'c++', 'c#', 'ruby'
subset = languages[1:4:2] # it returns the tuple ('java', 'c#')
```

Moreover, using step=-1 we can obtain the reversed collection:

```
languages = 'python', 'java', 'c++', 'c#', 'ruby'
subset = languages[::-1] # it returns the tuple ('ruby', 'c#', 'c++', 'java', 'python')
```

List

If you plan to change your data frequently, a better data structure is list.

```
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 get elements using index notation

```
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#']
```

List

Differently from tuples, we are able to modify elements of lists using **index notation** or exploiting methods of lists (e.g., **append** and **insert**)

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

Dictionary

Dictionaries allow to store key-value couples. For instance, we can use this data structure to save all the information about a person, e.g. its name and age

```
person = {}      # this is a dict
person = dict() # another dict
person['name'] = 'Filippo'
person['surname'] = ' Aleotti'
```

We can also create a dict using inline notation

```
person = {
    'name': 'Filippo',
    'surname': 'Aleotti',
    'age': 25
}
```

Dictionary

We can retrieve an element of the dictionary through its key

```
person = {
    'name': 'Filippo',
    'surname': 'Aleotti',
    'age': 25
}
name = person['name']
```

If the key doesn't exist, Error will be displayed

```
job = person['job'] # KeyError: 'job'
```

Dictionary

Given a dict, you can add new value just adding the correspondent key.

For instance, we can insert a new key *job* to the dict person, setting it to student value.

```
person['job'] = 'student'
print('job: {}'.format(person['job']))
```

Doing so, if we look at the key job we will find student value.

Using assignment, if a key already exists its value will be replaced with the new one.

Dictionary

We can obtain a list with all the key-value couples of a dict by running the **items** method. Instead, the method **keys** returns all the keys of a dict.

```
print(person.items()) # [('age', 25), ('surname', 'Aleotti'), ('name', 'Filippo')]
print(person.keys())  # ['age', 'surname', 'name']
```

Statements

Statements

Repeating a set of instruction N times or check if a given condition is verified or not, and act accordingly is quite common scenario in algorithms.

For instance, suppose we have to solve the following problem:

If the age of a person is lower than 18 years old then display the message "You are not qualified to drive a car", otherwise display three times the message "have a good day"

To solve the problem, we have to perform checks and to repeat instructions!

Conditions

To check if a given condition is True or False, and act accordingly we can use the **if** statement

```
x = 5
y = 3
if x > y:
    print('x is greater than y')
else:
    print('y is greater or equal than x')
```

Given the condition x > y, if it is verified then we will print *x is greater than y*, otherwise *y is greater or equal than x*

Conditions

We can use the **elif** reserved key to add more than a single 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')
```

First, the condition x > y is checked. If not verified, then the second condition x == y will tested and if also this condition fails then we will print *y is greater than x*

Indent

Notice that in the previous example the print functions were not aligned with the conditions. In Python, indenting is mandatory, since it allows **group** a set of instructions.

You are free to indent using tab or spaces, but the choice must be consistent in the full script.

```
x = 5
y = 3
if x > y:
    print('x is greater than y')
    x = x-2
else:
    print('y is greater or equal than x')
    x = x+2
```

For loop

For loop allows to repeat a given set of instructions N times

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

The range function creates a list of indexes [0,50[(so, 50 indexes), then for each index we increase the counter

Doing so, we have increased for 50 times the counter variable

For loop

Changing the parameters of the range function we can obtain different results. For instance, we can increase the starting index, and even the step between consecutive indexes (default step is 1)

```
counter = 0
for i in range(10,40,2):
    counter += 1
print(counter) # 15
```

In this case, counter is 15 because we start from 10 up to 40 (with 40 excluded) and the step between each index and the following one is 2 (so 10, 12, 14 …, 38)

For loop

Moreover, we can use the for loop to iterate over each element of a list

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

For loop

We can iterate also on elements of a dictionary

```
person = {
    'name': 'Filippo',
    'surname': 'Aleotti',
    'age': 25,
    'job': 'student'
}
for k,v in person.items():
    print('key:{} value:{}'.format(k,v))
```

The previous code will print each key-value couple of the dict

While loop

Python while loops are used to iterate as long as a given condition is True

```
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)
```

In this case, the condition is found == False (i.e., **not** found) **and** index < length of the list (i.e., the index is valid)

Functions

Functions

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

Functions are defined using the **def** keyword.

```
def dot(x,y):
    result = x*y
    return result
```

In this case, we have created the dot function. x and y are arguments of the function

Functions

The reserved keyword **return** allows to exit the function, giving back the value stored in the variable.

For instance, in the dot function we have stored into the result variable the dot value, and when the function will be called we will return such value.

We can return even more than one value (comma separated), but in general it is a good practice return back a dict or a list

Functions

We can run the function as many time as we desire, and the function may contains as many instructions as you want.

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

dot1 = dot(5,3)
dot2 = dot(2,4)
print(dot1) # 15
print(dot2) # 8
```

In this example, we have called the dot function two times, with different parameters

Functions

We can also set default values for parameters: when the function is called, if a parameter is not set the default value will be used

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

x1 = my_transformation(5)           # 0, we are setting alpha=1, beta=5
x2 = my_transformation(5, alpha=3) # 120, we are setting alpha=3, beta=5
x3 = my_transformation(5,  beta=2) # 3, we are setting alpha=1, beta=2
x4 = my_transformation(5, alpha=2, beta=2) # 23, we are setting alpha=2, beta=2
```

Of course, if a parameter doesn't have a default, it must be set when the function is called, otherwise error will be raised

```
x5 = my_transformation() # TypeError: my_transformation() takes at least 1 argument (0 given)
```

Examples

Python for image processing

Python is becoming the standard programming language for AI, and it offers many libraries to help you create incredible applications.

Among these libraries, a very popular and used one for computer vision is OpenCV

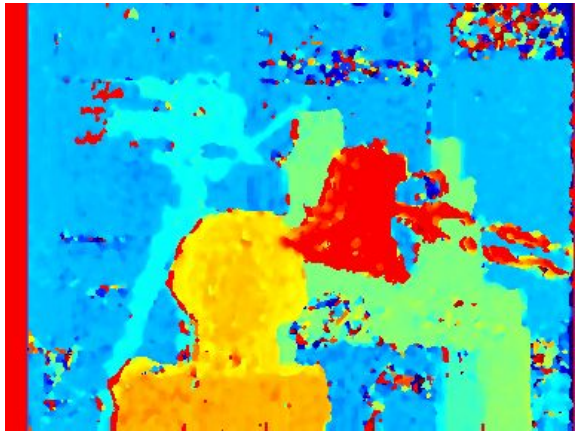
OpenCV offers lots of methods and functions to realize applications based on Stereo Matching, Object Detection, Optical Flow estimation, Tracking and so on.

Stereo Matching

Stereo Matching is straightforward using OpenCV

```
left = cv2.cvtColor(cv2.imread('image/left.png'),cv2.COLOR_BGR2GRAY)
right = cv2.cvtColor(cv2.imread('image/right.png'), cv2.COLOR_BGR2GRAY)
stereo_matcher = cv2.StereoSGBM_create(numDisparities = 16, blockSize = 5)
disparity = stereo_matcher.compute(left, right)
disparity = (disparity).astype(np.uint8)
disparity = cv2.applyColorMap(disparity, cv2.COLORMAP_JET)
cv2.imwrite('./disparity.png',disparity)
```

from Middlebury Dataset



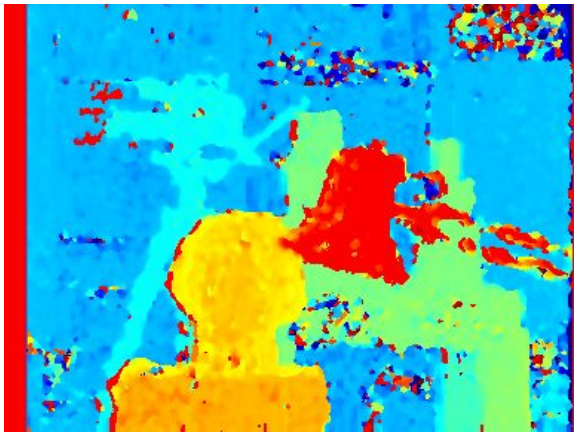
FAR

CLOSE

Stereo Matching

With few lines of code, we can also to reprojects points from 2D to 3D space

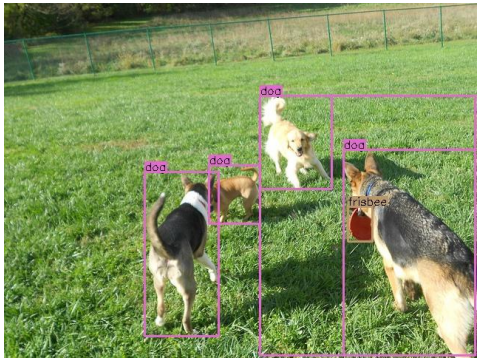
```
img = cv2.imread('image/left.png')
left = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
right = cv2.cvtColor(cv2.imread('image/right.png'), cv2.COLOR_BGR2GRAY)
stereo_matcher = cv2.StereoSGBM_create(numDisparities = 16, blockSize = 5)
disparity = stereo_matcher.compute(left, right)
focal_length=1.
ppm =  np.float32([[1,0,0,0],[0,-1,0,0], [0,0,focal_length,0],[0,0,0,1]])
points_3D = cv2.reprojectImageTo3D(disparity, ppm)
colors = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
generate_pointcloud(colors,points_3D,'3D.ply')
```



from Middlebury Dataset

Python for Deep Learning

Python is highly supported by Deep Learning frameworks such as TensorFlow and PyTorch



Python for Deep Learning

We few lines of code, we can realize for instance object detection applications, leveraging on AI

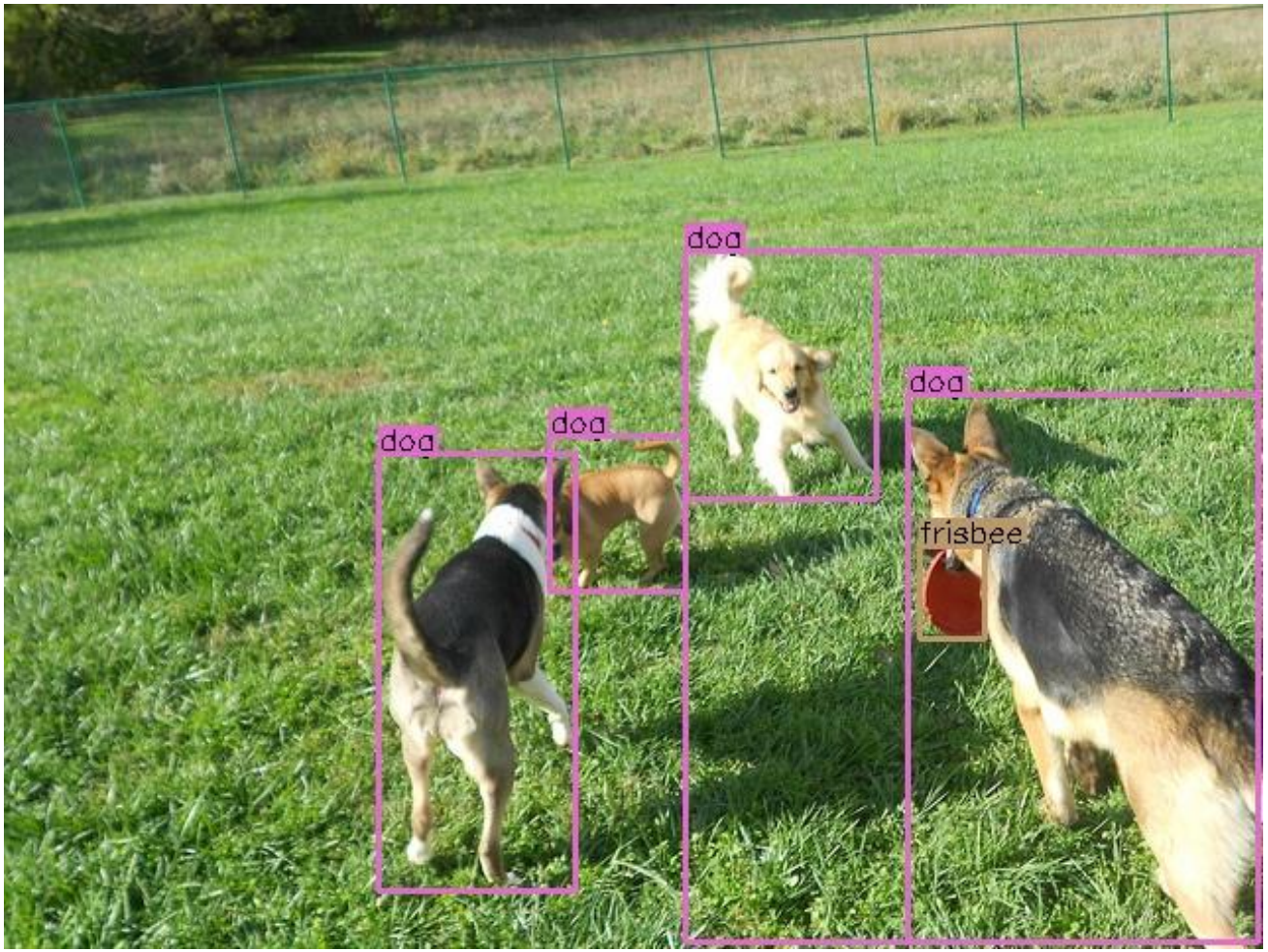
```
import cv2
from core.detectors import CornerNet_Squeeze
from core.vis_utils import draw_bboxes

detector = CornerNet_Squeeze()
image    = cv2.imread("demo.jpg")

bboxes = detector(image)
image  = draw_bboxes(image, bboxes)
cv2.imwrite("demo_out.jpg", image)
```

Code available here

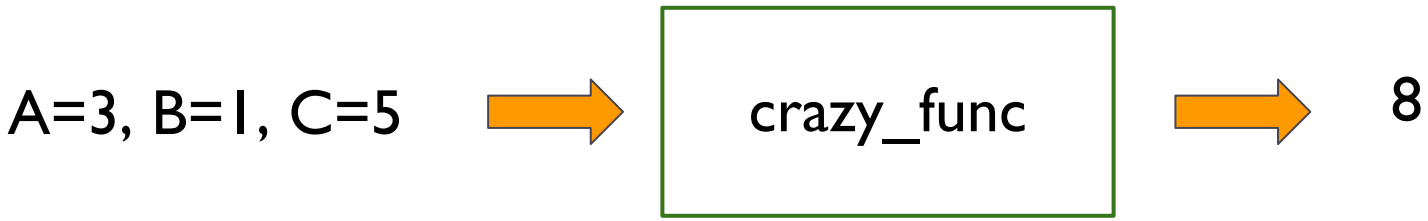
Python for Deep Learning

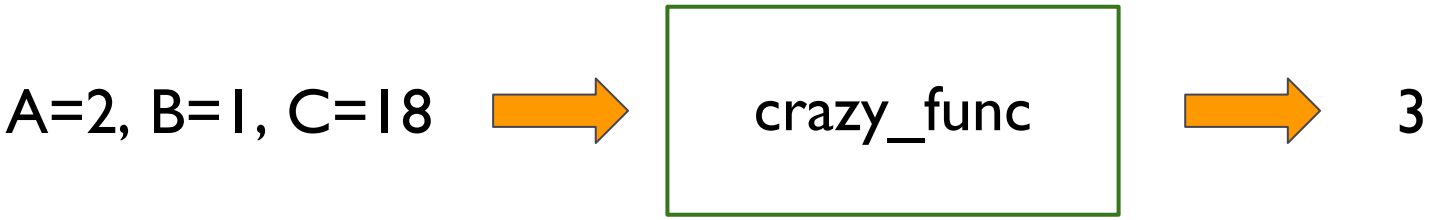


Exercises

Exercise 1

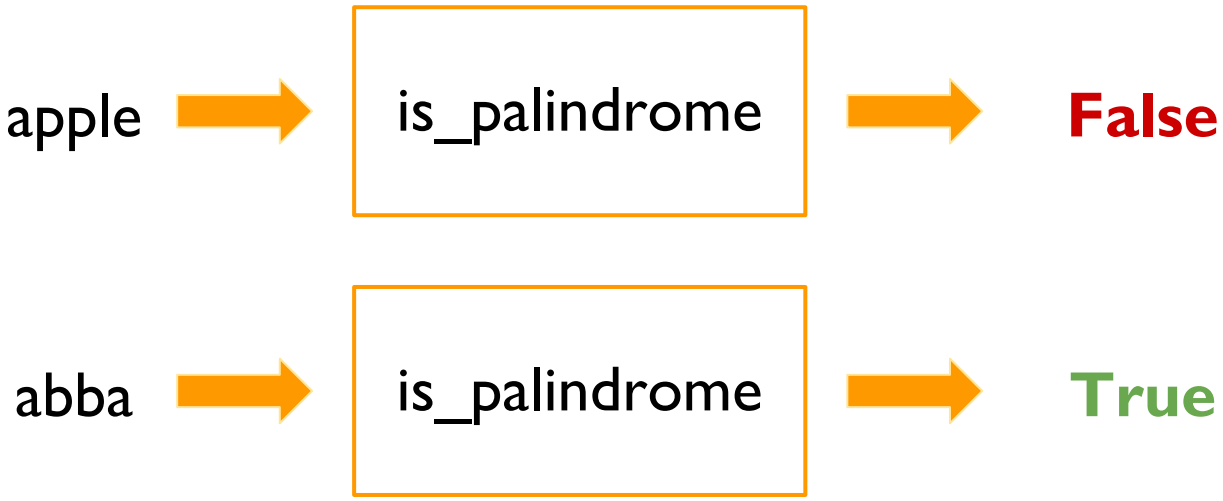
Create a Python function that, given three numbers as input A,B and C, return the sum of A and B if C is multiple of 9, otherwise the subtraction between C and 3 times B, powered by A.





Exercise 2

Create a Python function that, given a word as input, return True if that world is palindrome, False otherwise



Exercise 2

Tips 1: a word is palindrome if the the first character is equal to the last, the second character is equal to the second-last and so on.

Tips 2: reversing a palindrome word (so, last character becomes the first, second-last the second and so on) we obtain the same word

Exercise 3

Given a list of integer, store the frequency of each value in a dict, where the key is the value.

[0,1,0,2,2,1,2,1,0,0,2,1,1]

frequency_extractor

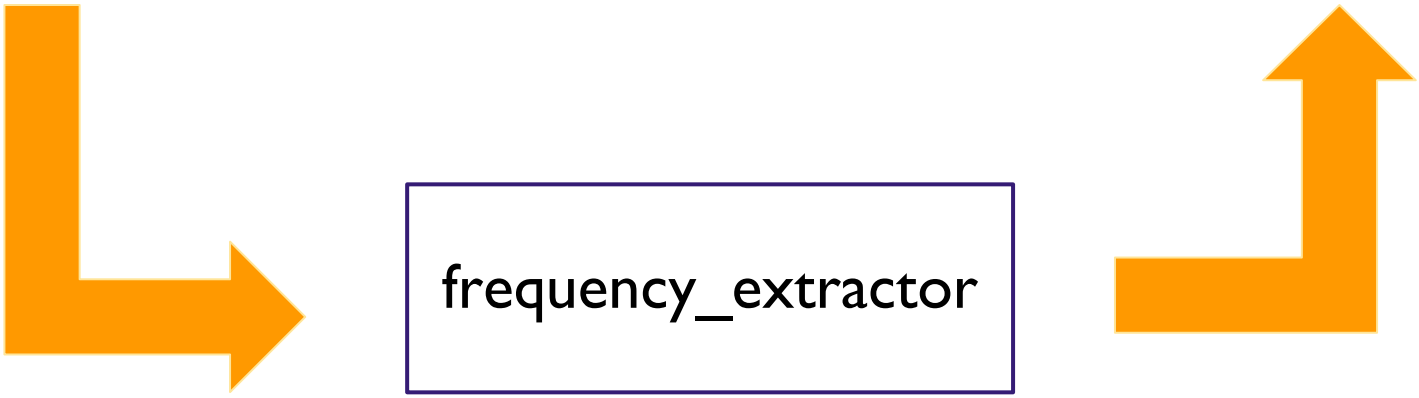
dict{ '0': 4, '1': 5, '2': 4 }

Exercise 4

Modify the function realised in the previous exercise to return (inside the dict, using the key min) also the value with minimum frequency. If more values have the same minimum frequency, then return the lowest one

[0,1,0,2,2,1,2,1,0,0,2,1,1]

dict{ '0': 4, '1': 5, '2': 4, 'min': 0 }



Thank you for the attention

Filippo Aleotti

DISI – University of Bologna

filippo.aleotti2@unibo.it

References

- Python course by W3schools
- Programming Python, Mark Lutz, Edited by O'Reilly
- Learning Python, Mark Lutz, Edited by O'Reilly
- Python Exercises by W3Resource
- OpenCV
- Python Documentation