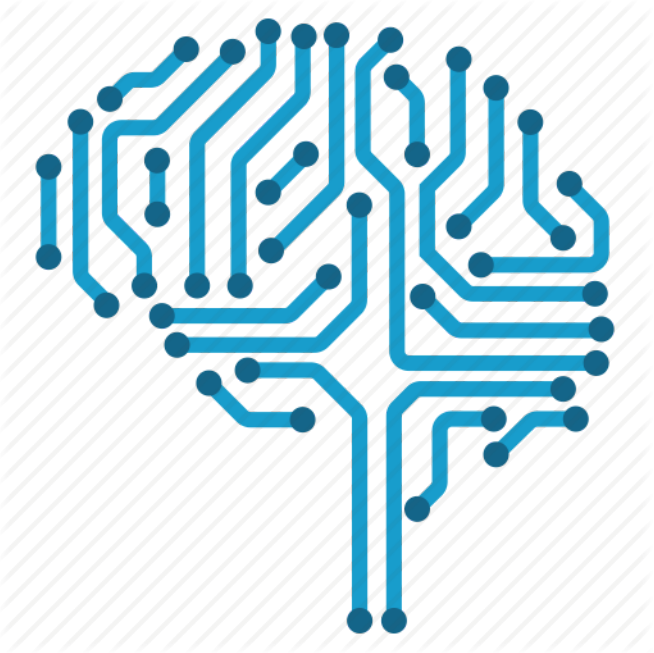
Machine Learning



make software
great again!

Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia

Il linguaggio Python (2)

Sintassi di base : Dizionari

- struttura complessa **non necessariamente** simmetrica **né** omogenea

PYTHON

```
>>> d = {
... (1, "giulio"): "374632768372",
... ("marco") : {
... "telefono": "34637647321",
... "indirizzo": "via Roma, 2",
... "classe": "5A" },
... (3, "luca") : [10, 20, "trenta"]}
>>> import pprint   # pretty print
>>> pprint.pprint(d)
{'marco': {'classe': '5A',
           'indirizzo': 'via Roma, 2',
           'telefono': '34637647321'},
 (1, 'giulio'): '374632768372',
 (3, 'luca'): [10, 20, 'trenta']}
```

Iterazione su liste, tuple, dizionari

- "qualunque oggetto in grado di essere trattato come una sequenza è definito un oggetto iterable (iterabile)"
- su ogni oggetto iterabile si può applicare un iteratore

PYTHON

```
>>> L = ['a', 'b', 'c', 'd']
>>> for elem in L:
...     print(elem)
...
a
b
c
d
```

Iterazione su liste, tuple, dizionari

- potrei fare anche così ma è utile?? Lo faccio solo quando serve....

PYTHON

```
>>> L = ['a', 'b', 'c', 'd']
>>> len(L)
4
>>> range(len(L))
range(0, 4)
>>> for i in range(len(L)):
...     print(L[i])
...
a
b
c
d
>>> for i in range(100):    # ciclo a numerosità predefinita
        faiqualcosa()
```

Iterazione su liste, tuple, dizionari

- sulle tuple è simile, sui dizionari itera solo sulla lista delle chiavi

PYTHON

```
>>> t = ("uno", "due", "tre")
>>> for elem in t:
...     print(elem)
...
uno
due
tre
>>> d
{(3, 'luca'): [10, 20, 'trenta'], (1, 'giulio'): '374632768372', 'marco': {'telefono': '34637647321', 'indirizzo': 'via Roma, 2'
>>> for elem in d:
...     print("Chiave: ", elem)
...
Chiave:  (3, 'luca')
Chiave:  (1, 'giulio')
Chiave:  marco
```

Iterazione su liste, tuple, dizionari

- per vedere anche i valori

PYTHON

```
>>> for elem in d:
...     print("Chiave: ", elem, "Valore", d[elem])
...
Chiave:  (3, 'luca') Valore:  [10, 20, 'trenta']
Chiave:  (1, 'giulio') Valore:  374632768372
Chiave:  marco Valore:  {'telefono': '34637647321', 'indirizzo': 'via Roma, 2', 'classe': '5A'}
```

Qualche perla....

- da liste a dizionario

PYTHON

```
>>> chiavi = ["a","b","c"]
>>> valori = [1, 2, 3]
>>> zip(chiavi, valori) # associa elementi della prima lista e della seconda
<zip object at 0x7fea328a0ac8>
>>> list(zip(chiavi, valori)) # crea un lista di coppie
[('a', 1), ('b', 2), ('c', 3)]
>>> dict(zip(chiavi, valori)) # crea un dizionario chiave-valore
{'c': 3, 'a': 1, 'b': 2}
```

Funzioni : definizione

- possono restituire da 0 a N valori
- i parametri di tipo semplice sono passati per valore (int, string, float)
- dizionari, liste, oggetti per riferimento
- anche le tuple, ma sono immutabili

SINTASSI

```
def <nomefunzione>(*args, **kwargs):
    < corpo
      della
      funzione
    >
    [return tuplavalori]
```

- dove ***args** sono i parametri posizionali (non-keyworded variable)
- e ***kwargs** sono i parametri per chiave (keyworded variable)

Funzioni : esempi

PYTHON

```python
def somma(a, b):
    somma = a+b
    return somma    # un solo risultato

def somma_e_differenza(a, b):
    somma = a+b
    differenza = a-b
    return somma, differenza    # una tupla di risultati

a = 2
b = 3
print(somma(a, b))
print(somma_e_differenza(a, b))
```

OUTPUT

```
5
(5, -1)
```

Funzioni : esempi

PYTHON

```python
def potenza(base, esponente=2):
    """
        per default la funzione calcola il quadrato di base
    """
    result = base
    for i in range(esponente-1):
        result = result*base
    return result

print(potenza(3))
print(potenza(2, 4))
```

OUTPUT

```
9
16
```

Funzioni : esempi

- le funzioni supportano la tipizzazione forte

PYTHON

```python
def somma_interi(a: int, b: int) -> int:
    return a+b

print("somma interi :", somma(2,3))
print("somma float  :", somma(2.5,3.5))
```

OUTPUT

```
somma interi : 5
somma float  : 6.0
```

Python philosophy: We're all consenting adults here

- e quindi??

Gestione delle eccezioni

- costrutto **try..except**
- simile a gestione di Java ma non deve essere dichiarato

PYTHON

```python
def dividi(dividendo, divisore):
    result = dividendo/divisore
    return result

print("10:2 = ", dividi(10,2))
print("15:2 = ", dividi(15,2))
print("15:0 = ", dividi(15,0))
```

OUTPUT

```
10:2 =  5.0
15:2 =  7.5
Traceback (most recent call last):
  File "01_funzioni.py", line 97, in <module>
    print("15:0 = ", dividi(15,0))
  File "01_funzioni.py", line 92, in dividi
    result = dividendo/divisore
ZeroDivisionError: division by zero
```

Gestione delle eccezioni

- costrutto **try..except**

PYTHON

```
def dividi(dividendo, divisore):
    try:
        result = dividendo/divisore
    except:
        result = "As pòl mìa fér!"
    return result

print("10:2 = ", dividi(10,2))
print("15:2 = ", dividi(15,2))
print("15:0 = ", dividi(15,0))
```

OUTPUT

```
10:2 =  5.0
15:2 =  7.5
15:0 =  As pòl mìa fér!
```

Gestione delle eccezioni

- il costrutto **try..except** può essere sequenziato e terminare con finally
- Il codice in **else** verrà eseguito solo se non sono state generate eccezioni.
- Il codice in **finally** verrà eseguito sempre

PYTHON

```
def test():
    f = None
    try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())
    except OSError as err:
        print("OS error: {0}".format(err))
    except ValueError:
        print("Errore: il file non contiene un numero intero!")
    except:
        print("Errore inaspettato!")
    else:
        # viene eseguito sempre
        print("Nessun errore")
        print("Il file contiente il numero "+str(i))
        f.close()
    finally:
        if f:
            print("chiusura file...")
            f.close()
        print("fine procedura")
```

Gestione delle eccezioni : Assertion Error

- un altro modo per controllare l'input è la gestione con **assert**

PYTHON

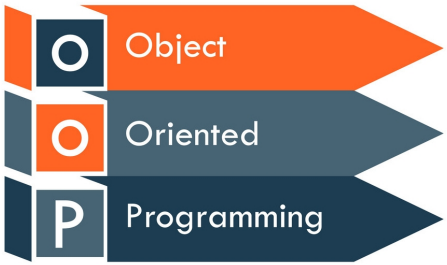
```python
def somma_sicuramente_interi(a: int, b: int) -> int:
    assert(a.__class__.__name__ == 'int')
    assert(b.__class__.__name__ == 'int')
    return a+b

print("somma interi :", somma_sicuramente_interi(2,3))
print("somma float  :", somma_sicuramente_interi(2.5,3.5))
```

OUTPUT

```
somma interi : 5
Traceback (most recent call last):
  File "01_funzioni.py", line 142, in <module>
    print("somma float  :", somma_sicuramente_interi(2.5,3.5))
  File "01_funzioni.py", line 137, in somma_sicuramente_interi
    assert(a.__class__.__name__ == 'int')
AssertionError
```

OOP

- Python è un linguaggio fortemente orientato agli **oggetti**
- Ogni elemento è un oggetto
- lo stesso **type** di un oggetto è un oggetto di tipo type... [uhm....]



- La sintassi minima per definire una **classe** in python è:

```
class <classname>:
    pass
```

- costruttore (metodo **init**)
- non supporta costruttori multipli perché esistono i **kwargs
- il primo parametro dei metodi di istanza è sempre **self**

```
class <classname>:
    def __init__(self, **kwargs):
        <code>
```

OOP: Esempio

PYTHON

```python
class Studente:
    def __init__(self, nome, cognome, cellulare=""):
        self.nome = nome
        self.cognome = cognome
        self.cellulare = cellulare

s1 = Studente("Antonella", "Catellani")
print(s1)
s2 = Studente("Alessandro", "Muzzini", "3456789012")
print(s2)
s3 = Studente()
print(s3)
```

OUTPUT

```
S1 =  <__main__.Studente object at 0x7fad44eadcc0>
S2 =  <__main__.Studente object at 0x7fad44eadcf8>
Traceback (most recent call last):
  File "02_classi_2.py", line 19, in <module>
    s3 = Studente()
TypeError: __init__() missing 2 required positional arguments: 'nome' and 'cognome'
```

OOP: Esempio

- il primo parametro dei metodi di istanza è sempre **self**

PYTHON

```
def getCellulare(self):
    return self.cellulare

...

print("Cell: ", s2.getCellulare())
```

OUTPUT

```
Cell:  3456789012
```

- equivale a invocare il metodo con la sintassi **<Classe>.<metodo>(oggetto)**
- esempio:

PYTHON

```
print("Cell: ", Studente.getCellulare(s2))
```

OOP: Metodi Statici

- i metodi che come primo parametro **NON HANNO** self sono considerati statici
- sono preceduti dal decoratore **@staticmethod**

PYTHON

```python
class Pizza:

    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validate_topping(topping):
        if topping == "pineapple":
            raise ValueError("No pineapple")
        else:
            return True

    def getToppings(self):
        return self.toppings

ingredients = ["cheese","onions","tomato"]
# la funzione predefinita all restituisce True se tutti gli elementi sono veri
if all(Pizza.validate_topping(i) for i in ingredients):
    pizza = Pizza(ingredients)
print(pizza.getToppings())
```

OUTPUT

```
['cheese', 'onions', 'tomato']
```

OOP: Metodi e attributi di classe

- esiste anche il decoratore **@classmethod** che prende come parametro la classe

PYTHON

```
class Pet:
    pets = 0

    def __init__(self, petname):
        self.petname = petname
        Pet.pets += 1

    def __str__(self):
        return "My name is " + self.petname

    @classmethod
    def quanti(cls):
        return cls.pets

b = Pet("Baffo")
print(b)
m = Pet("Molly")
print("sono in ", b.quanti())
```

OUTPUT

```
My name is Baffo
sono in  2
```

Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia