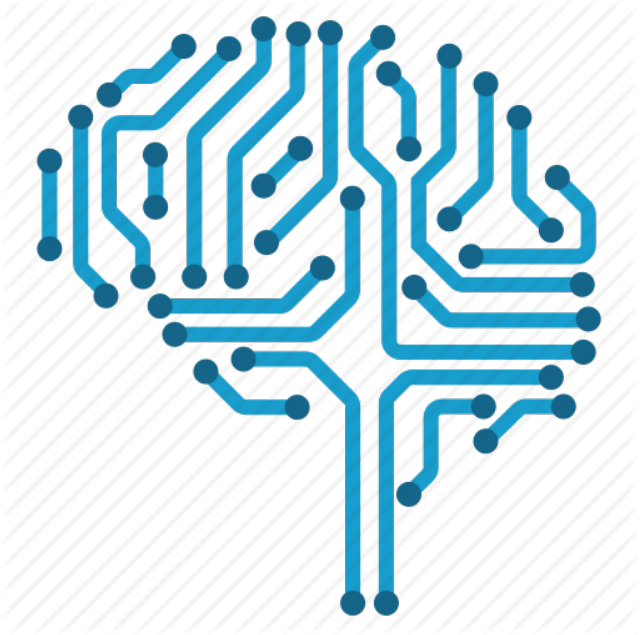
Machine Learning



make software
great again!

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

Il linguaggio Python (1)

Cos'è Python

Linguaggio di programmazione

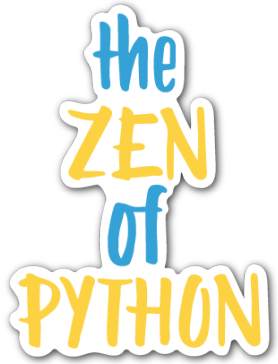
- interpretato
- interattivo
- orientato agli oggetti
- tipizzazione dinamica
- sintassi **molto chiara**
- general purpose
- interfaccia con chiamate e librerie di sistema
- è estendibile in C o C++ (interfaccia anche con Java)
- è **portabile**

Creato nel 1991 da **Guido Van Rossum** deve il suo nome alla trasmissione della BBC ***"Monty Python's Flying Circus"***

Perché Python ?

Lo Zen di Python, di Tim Peters

- Bello è meglio di brutto.
- Esplicito è meglio di implicito.
- Semplice è meglio di complesso.
- Complesso è meglio di complicato.
- Lineare è meglio di nidificato.
- Rado è meglio di denso.
- La leggibilità è importante.
- I casi speciali non sono abbastanza speciali per infrangere le regole.
- Anche se la praticità batte la purezza.
- Gli errori non dovrebbero mai accadere in silenzio.
- Salvo esplicitamente silenziati.
- Davanti all'ambiguità, rifiuta la tentazione di indovinare.
- Se l'implementazione è difficile da spiegare, è una cattiva idea.
- Se l'implementazione è facile da spiegare, potrebbe essere una buona idea.



Dove uso Python ?

Python è un linguaggio di programmazione generico di alto livello e può essere applicato a diverse classi di problemi.

- estesa libreria standard
- manipolazione di stringhe (espressioni regolari, Unicode, calcolo di differenze tra file)
- protocolli Internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, programmazione CGI)
- sviluppo software (unit test, logging, profiling, parsing di codice Python)
- interfacce a sistemi operativi (chiamate di sistema, file system, socket TCP/IP)

librerie python

Sintassi di base

- Python è **case-sensitive**

Naming Convention

- I nomi delle **classi** iniziano con una lettera maiuscola, **tutti gli altri** identificatori iniziano con una lettera minuscola
- Un identificatore che inizi con **un trattino basso** indica che si tratta di un identificatore **privato**
- Un identificatore che inizi con **due trattini bassi** indica che si tratta di un identificatore **fortemente privato**
- Se l'identificatore **inizia e finisce** con due trattini bassi allora è un nome **speciale** definito nel linguaggio.

all'indirizzo *https://www.python.org/dev/peps/pep-0008/* le regole di scrittura

Sintassi di base

Struttura di un programma

- Python (per fortuna) usa l'**indentazione** per il controllo di flusso

PYTHON

```
if voto >= 6:
    print("Sei promosso!")
else:
    print("Ancora qualche sforzo....")
```

Sintassi di base

Variabili e tipi

Python gestisce le **tipizzazione dinamica**

- le variabili sono tutti *oggetti* di un certo *tipo*
- l'identificatore può cambiare tipo durante l'esecuzione
- la tipizzazione è inferita dall'interprete

PYTHON SHELL

```
>>> a = 1
>>> a.__class__.__name__
'int'
>>> a = 2.5
>>> a.__class__.__name__
'float'
>>> a = "ciao"
>>> a.__class__.__name__
'str'
```

Sintassi di base

Variabili e tipi

- anche il tipo ha un tipo...

PYTHON SHELL

```
>>> a = 1
>>> type(a)
<type 'int'>
>>> type(type(a))
<type 'type'>
>>> type(a).__class__.__name__
'type'
```

Sintassi di base

Confronto

- per valore
- vari operatori
  - < > == >= <= <> != # fra valori
  - "is in" "is not in" # con elementi iterabili

PYTHON

```
>>> a = 1
>>> b = 1
>>> a == b
True
>>> b=1.0
>>> a==b # float vs int
True
>>> a=1.2
>>> a==b # float vs float
False
```

Sintassi di base

I/O da stdin/stdout : la funzione **print**

- la sintassi completa è

SYNTAX

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- dove *objects è una lista di elementi da stampare (da 0 a N elementi)

PYTHON

```
>>> print()

>>> print(a)
1.2
>>> print(a, b)
1.2, 1.0
>>> print("a+b=", a+b)
a+b= 2.2
>>> print("a+b=", a+b, sep='')
a+b=2.2
```

Sintassi di base

I/O da stdin/stdout : la funzione **input**

PYTHON

```
>>> age=input("Inserisci la tua età: ")
Inserisci la tua età: 18
>>> print(age)
18
>>> print(type(age))
<class 'str'>
```

- usare il *cast* al volo (attenzione all'input però...)

PYTHON

```
>>> numero = int(input("dammi un numero:")) # 10
dammi un numero:10
>>> numero+1
11
>>> numero = int(input("dammi un numero:")) # ciao
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ciao'
```

Sintassi di base

condizione: IF, IF ELSE, IF ELIF ELSE

```python
numero=int(input("numero"))
if numero > 10:
    print("numero è maggiore di 10")
elif numero < 10:
    print("numero è minore di 10")
else:
    print("numero è uguale a 10.")
```

espressione condizionale

```python
maggiorenne = True if age >= 18 else False
```

Sintassi di base

cicli

- cicli a numerosità definita
- cicli condizionali
- cicli su oggetti iterabili

PYTHON

```
# definito - classico ciclo for
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Sintassi di base

cicli

PYTHON

```
# anche dal maggiore al minore
# equivale a
# for (int i=10; i<20; i=i+3)
>>> for i in range(10, 20, 3):
...     print(i)
...
10
13
16
19

# ciclo while
while numero < 10:
    numero=int(input("inserisci un numero "))
    print("Hai inserito ", numero)
```

Stringhe

inizializzazione

- sono **oggetti** e non semplici array di caratteri
- definite come sequenza di caratteri fra apici o doppi apici (o tripli se multilinea)

```
>>> s1 = 'una stringa'
>>> s2 = "un'altra stringa"
>>> s3 = """una stringa un po' più lunga
che mi servono due righe"""
>>> s1
'una stringa'
>>> s2
"un'altra stringa"
>>> s3
"una stringa un po' più lunga\nche mi servono due righe"
```

Stringhe

formattazione

- metodo **format** di `<class 'str'>`

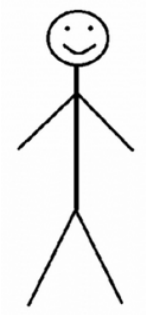
PYTHON

```
>>> s = "Oggi in lab. {} una bella lezione di {}"
>>> labo = 'T'
>>> materia = 'python'
>>> print(s.format(labo, materia))
Oggi in lab. T una bella lezione di python
>>> labo = 'G'
>>> materia = 'assembly'
>>> print(s.format(labo, materia))
Oggi in lab. G una bella lezione di assembly
```

Stringhe

formattazione

- metodo **format** di `<class 'str'>`
- anche su stringhe multilinea (template)



PYTHON

```
>>> t = """
... {0} è un bravo studente.
... {0} usa {1} tutti i giorni.
... {0} è intelligente.
...
... Fai come {0}!""".format("Bill", "python")
>>> print(t)

Bill è un bravo studente.
Bill usa python tutti i giorni.
Bill è intelligente.

Fai come Bill!
```

Stringhe

interpolazione : operatore %

PYTHON

```
>>> s = """Ciao, %s! Vi piace %s ? """ % ("ragazze", "python")
>>> print(s)
Ciao, ragazze! Vi piace python ?
```

- Ma anche coi numeri...

PYTHON

```
>>> costo = 10.3
>>> s = "Il totale è %s"
>>> s % costo
'Il totale è 10.3'
>>> "Il totale è %d" % costo
'Il totale è 10'
>>> "Il totale è %f" % costo
'Il totale è 10.300000'
>>> "Il totale è %.2f" % costo
'Il totale è 10.30'
```

Oggetti iterables

- liste (list)
- tuple (tuple)
- insiemi (set)
- dizionari (dict)
- contenitori di oggetti di tipo **qualsiasi**
- esiste una funzione *iter()* per accedere agli elementi di un contenitore

Liste

- contenitori modificabili
- accessibili per indice

PYTHON

```
>>> L = [] # lista vuota
>>> len(L)
0
>>> L = [1,2,3,"quattro", 5.0]
>>> len(L)
5
>>> L[0]
1
>>> L[-1]
5.0
>>> L[2:4]
[3, 'quattro']
```

Liste

- gli **array** sono *liste*
- le **matrici** sono *liste di liste*

PYTHON

```
>>> a = [2,7,4,1,9]
>>> a[1]
7
>>> m = [
... [1, 3, 2],
... [5, 6, 0],
... [-1, 0, 3]
... ]
>>> m
[[1, 3, 2], [5, 6, 0], [-1, 0, 3]]
>>> m[1][2] # terzo elemento della seconda riga
0
>>> m[-1] # ultima riga
[-1, 0, 3]
```

Liste

- le liste sono **modificabili**

PYTHON

```
>>> L = [1,3,4]
>>> L.append(9)
>>> L
[1, 3, 4, 9]
>>> L.pop()
9
>>> L
[1, 3, 4]
>>> M = ['a', 'b', 'c']
>>> L+M # somma di liste
[1, 3, 4, 'a', 'b', 'c']
>>> L-M # e la differenza???
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

Liste

- altri metodi utili

PYTHON

```
>>> L = [1, 3, 6, 5, 1, 2, 4, 6, 1]
>>> L.index(6)  # posizione in cui si trova il primo '6'
2
>>> L.count(1)  # quante volte è presente '1'
3
>>> L.remove(2) # rimuove il primo '2' che trova
>>> L
[1, 3, 6, 5, 1, 4, 6, 1]
>>> L.sort()    # ordinamento crescente numerico-lessicografico
>>> L
[1, 1, 1, 3, 4, 5, 6, 6]
>>> L.reverse() # inverte la lista
[6, 6, 5, 4, 3, 1, 1, 1]
>>> L.index(-1) # se non è presente l'elemento restituisce un errore
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: -1 is not in list
```

Tuple

- non modificabili
- no sort
- no reverse
- no assignment

PYTHON

```
>>> t = ("uno", "due", 3, 4.0, ['a', 'b', 'c'], "sei")
>>> t[0]
'uno'
>>> t[5][1]
'e'
>>> t[-2]
['a', 'b', 'c']
>>> t[-2][0]
'a'
>>> t[0] = "cinque"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Set

- modificabili ma **devono** avere oggetti distinti

PYTHON

```
>>> s = {1,2,3}
>>> s
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(1)
>>> s
{1, 2, 3, 4}    # compare una sola volta
```

Set: unione, intersezione, differenza

PYTHON

```
>>> s = {1,2,3}
>>> t = {2,3,4}
>>> s | t   # unione
{2, 3}
>>> s & t   # intersezione
{1, 2, 3, 4}
>>> s - t   # differenza
{1}
```

- Uhm... e quando mai lo userò?

Set e List

- per operazioni fra liste possiamo passare dai set

PYTHON

```
>>> L = [1,2,3,4,5,6]
>>> N = [4,5,6,7,8,9]
>>> lista_unione = list(set(L) | set (N))
>>> lista_unione
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista_intersezione = list(set(L) & set (N))
>>> lista_intersezione
[4, 5, 6]
>>> lista_differenze = list(set(L) - set (N))
>>> lista_differenze
[1, 2, 3]
```

Dizionari

- una delle strutture più **comode** di python
- formato JSON nativo
- array associativo **chiave => valore**
  - chiave deve essere serializzabile - se complessa immutabile
    - numeri
    - stringhe
    - tuple NO LISTE

Sintassi di base

Dizionari

PYTHON

```
>>> d = {}
>>> d.__class__
<class 'dict'>
>>> d = {"giulio": "33312345678", "jenny": "34987654321"}
>>> d["giulio"]
'33312345678'
>>> d["marco"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'marco'
>>> d[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

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

Python e Machine Learning

5 minuti di pausa...

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', 'classe': '5A'}}
>>> 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

```
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

```
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

```
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

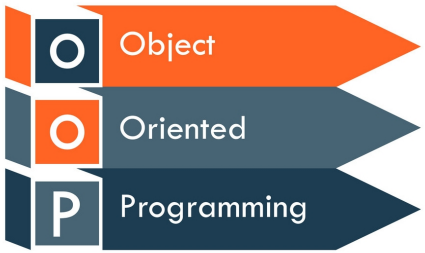
```python
def somma_sicuramente_interi(a: int, b: int) -> int:
    assert(a.__class__.__name__ == 'int')
    assert(b.__class__.__name__ == 'int')
    return a+b

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:

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

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

PYTHON
```
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())
```

PYTHON

```
Cell:  3456789012
```

OUTPUT

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

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

PYTHON

OOP: Metodi Statici

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

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"]
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

- i metodi che come primo parametro **NON HANNO** self sono considerati statici
- 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