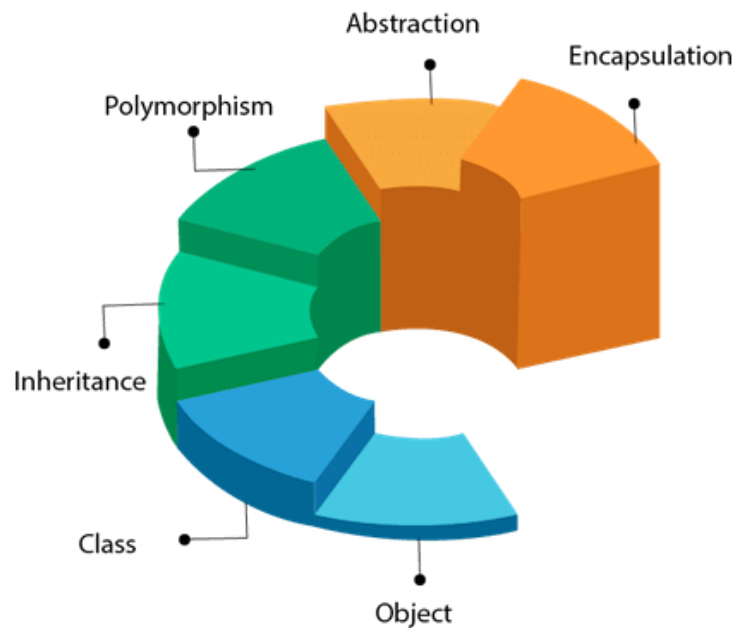


# Object Oriented Programming



# *Object Oriented Programming*

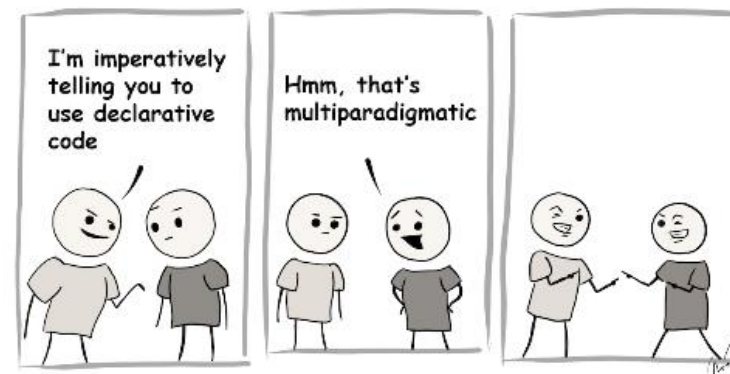
- paradigmi di programmazione
  - OOP
- oggetto
- classe
  - attributi
  - costruttore
  - metodi

SUMMARY



## *paradigma di programmazione*

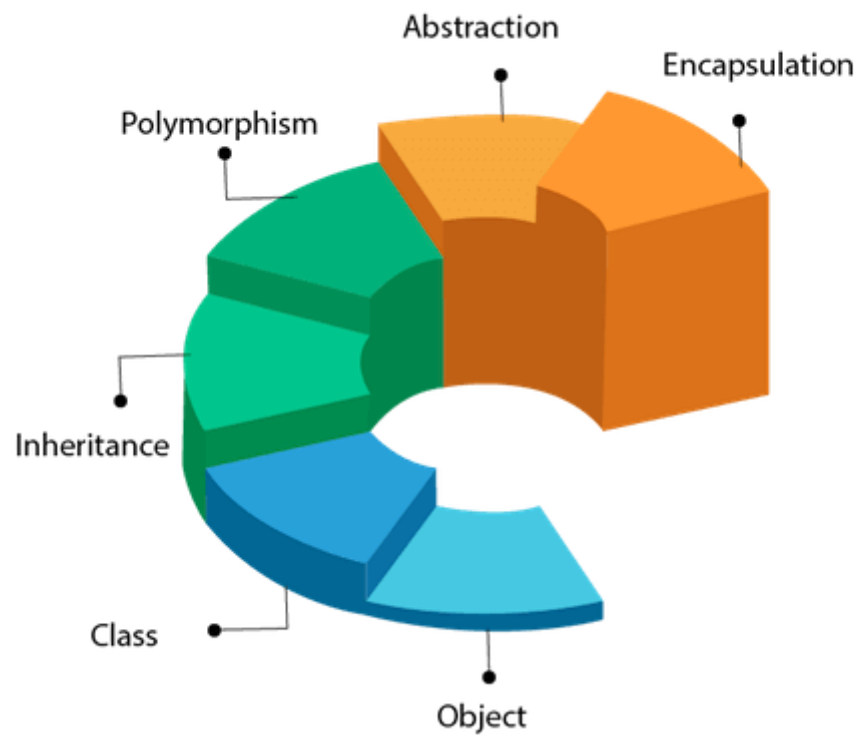
- il ***paradigma di programmazione*** definisce il modo in cui il programmatore concepisce il programma
- i vari paradigmi si ***differenziano***
- per le ***astrazioni*** usate per rappresentare gli elementi di un programma (funzioni, oggetti, variabili ...)
- per i ***procedimenti*** usati per l'elaborazione dei dati (assegnamento, iterazione, gestione del flusso dei dati ...)



## *paradigmi*

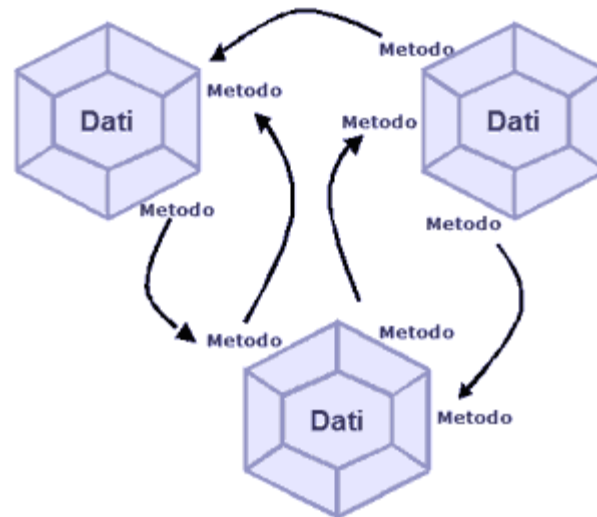
- paradigma ***imperativo***
  - programmazione procedurale ('60)
  - programmazione strutturata ('60-'70)
- programmazione orientata agli ***eventi***
  - *interfacce grafiche*
- programmazione ***logica***
  - *intelligenza artificiale*
- programmazione ***funzionale***
  - *applicazioni matematiche e scientifiche*
- programmazione ***orientata agli oggetti***

## *programmazione orientata agli oggetti*



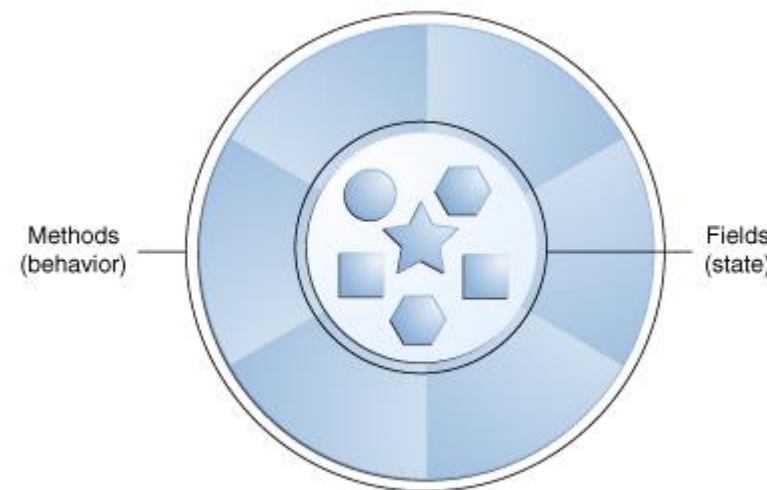
## *Object Oriented Programming*

- la *programmazione orientata agli oggetti* (**OOP**, **O**bject **O**riented **P**rogramming) permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi



## *oggetto*

- analisi della realtà e definizione del ***dominio applicativo***
  - evidenziare informazioni essenziali eliminando quelle non significative per il problema
- un ***oggetto*** rappresenta un oggetto fisico o un concetto del dominio
  - memorizza il suo ***stato*** interno in campi privati (attributi dell'oggetto)
    - concetto di ***incapsulamento*** (black box)
  - offre un insieme di ***servizi***, come ***metodi*** pubblici (comportamenti dell'oggetto)
- realizza un ***tipo di dato astratto***
  - (ADT - *Abstract Data Type*)



## *astrazione*

- *astrazione*
  - *evidenziare* alcune *proprietà*, ritenute *significative*, relative ad un determinato fenomeno osservato
  - *escludendone* altre considerate *non rilevanti* per la sua comprensione
- astrazione → creazione di *modelli*
  - nella OOP i modelli sono rappresentati dalle *classi*
- *Object Oriented Analysis*
  - modellare la realtà di interesse in un sistema software attraverso l'astrazione



## *esempio: automobile di Mario*



- nome dell'oggetto: «AutoDiMario»
- classe: ***Automobile***
- ***attributi***
  - velocità\_massima → 220
  - velocità\_attuale → 120
  - tipo\_freni → tamburo
- ***metodi***
  - accelera(n) → aumenta velocità nKm/h
  - frena()

## *classi e oggetti*

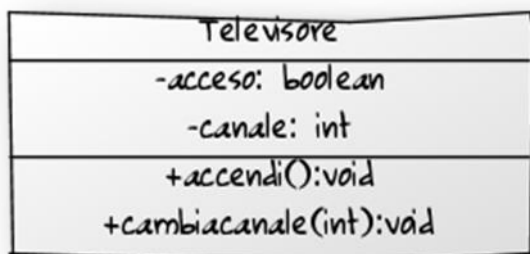
- ogni oggetto ha una **classe** di origine (*è istanziato da una classe*)
- la classe definisce la stessa **forma iniziale** (campi e metodi) a tutti i suoi oggetti
- ma ogni oggetto
  - ha la sua **identità**
  - ha uno stato e una locazione in memoria distinti da quelli di altri oggetti
    - sia istanze di classi diverse che della stessa classe



## *classi e tipi di dato*

- una classe è a tutti gli effetti un ***tipo di dato*** (come gli interi e le stringhe e ogni altro tipo già definito)
- un tipo di dato è definito dall'insieme di ***valori*** e dall'insieme delle ***operazioni*** che si possono effettuare su questi valori
- nella programmazione orientata agli oggetti, è quindi possibile sia utilizzare tipi di dato esistenti, sia definirne di nuovi tramite le classi
- i nuovi tipi di dato si definiscono ***ADT*** (*Abstract Data Type*)

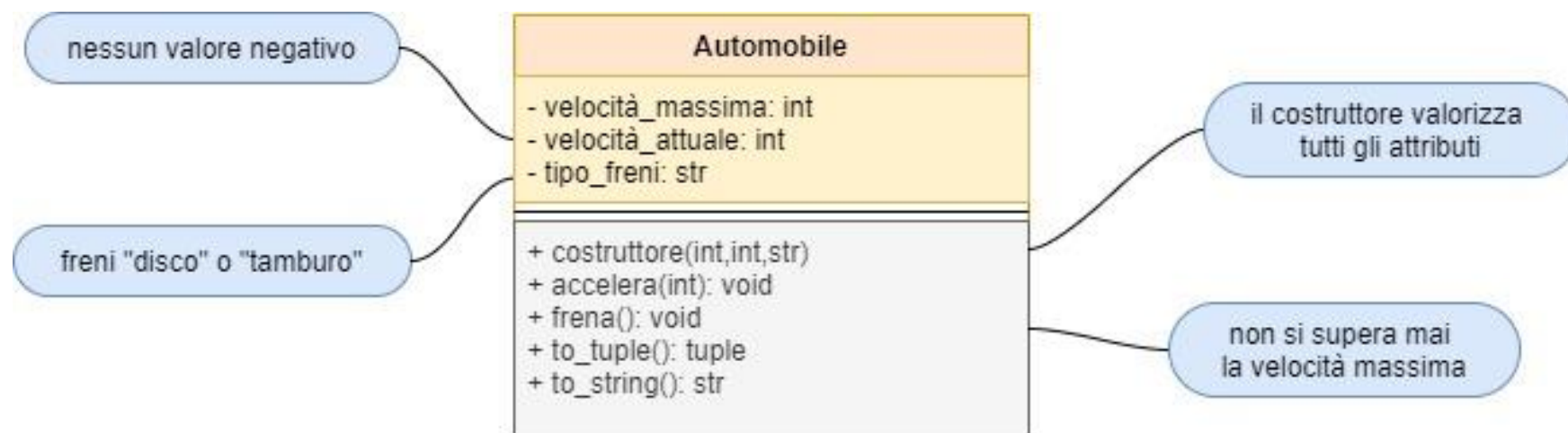
## *diagramma delle classi*



- la prima sezione contiene il ***nome*** della classe
- la seconda sezione definisce i suoi ***attributi***
- la terza i ***metodi***, le operazioni che si possono compiere sull'oggetto

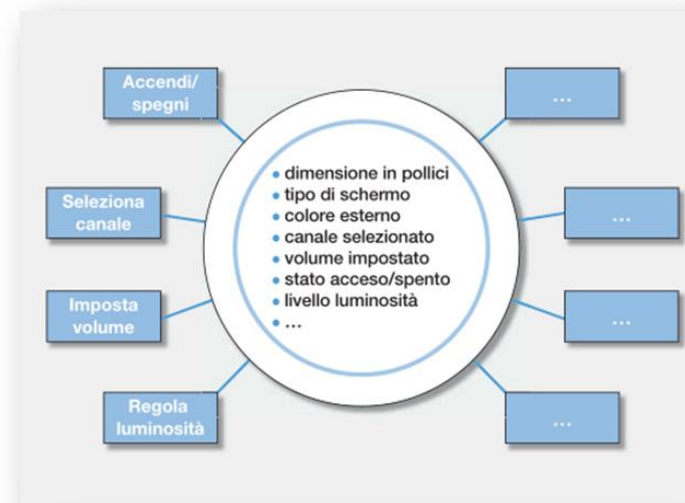
<http://www.draw.io>

## *classe Automobile*



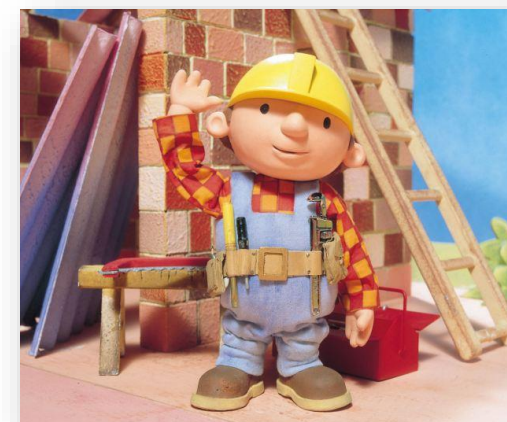
## *incapsulamento (information hiding)*

- ***nascondere*** il funzionamento interno (la struttura interna)
- fornire un'***interfaccia*** esterna che permetta l'utilizzo senza conoscere la struttura interna



## *Python – self - \_\_init\_\_*

- costruzione di oggetti (*istanziamento*)
- **\_\_init\_\_**: metodo *costruttore*
- eseguito *automaticamente* alla creazione di un oggetto
  - *instantiation is initialization*
- **self**: primo parametro di tutti i metodi
  - non bisogna passare un valore esplicito
  - rappresenta l'oggetto di cui si chiama il metodo
  - permette ai metodi di accedere ai campi



## *Python definizione della classe*

```
class Automobile:

    def __init__(self, vm: int, va: int, freno: str):
        '''
        vm velocità massima
        va velocità attuale
        freno tipo freno
        '''
        self._vm = max(vm, 0)
        self._va = max(va, 0)
        if freno == 'tamburo':
            self._freno = 'tamburo'
        else:
            self._freno = 'disco'
```



## *Python metodi*

```
def accelera(self, km: int):  
    ''' aumenta la velocità di km chilometri orari '''  
    if km <= 0:  
        return  
    self._va += km  
    if self._va > self._vm:  
        self._va = self._vm
```

```
def frena(self):  
    ''' dipende dal tipo dei freni '''  
    if self._freno == 'disco':  
        self._va -= 20  
    else:  
        self._va -= 10  
    self._va = max(self._va, 0)
```

```
def to_tuple(self) -> tuple:  
    return self._vm, self._va, self._freno  
  
def to_string(self) -> str:  
    s = 'velocità massima: ' + str(self._vm)  
    s += ' velocità attuale: ' + str(self._va)  
    s += ' freni a ' + self._freno  
    return s
```

## *oggetti e classi*



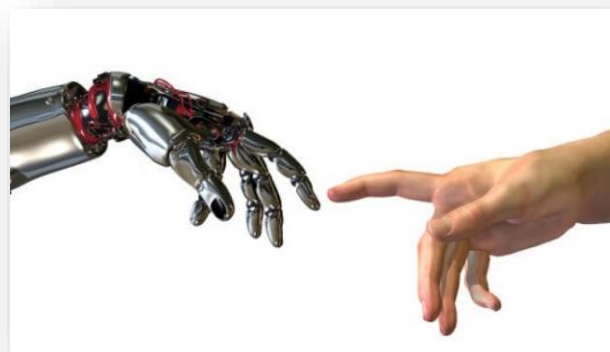
- da una ***classe*** possono essere istanziati (creati) ***più oggetti***
- tutti con gli ***stessi attributi***
  - ma ognuno ha una sua ***valorizzazione*** degli attributi
- tutti hanno lo stesso ***comportamento***
  - rispondono ai messaggi attivando i ***metodi*** della classe

## *gli oggetti*

- gli oggetti sono le **entità** di un programma che **interagiscono** tra loro per raggiungere un obiettivo
- vengono **creati** (istanziati) in fase di esecuzione
- ognuno di essi fa parte di una categoria (una **classe**)
- ogni classe può creare **più oggetti**, ognuno dei quali, pur essendo dello stesso tipo, è **distinto** dagli altri
- un oggetto è una **istanza** di una classe

## *creazione di un oggetto*

- per creare un oggetto si effettua una ***istanziatura*** di una classe
- in questa fase viene riservato uno spazio di memoria per conservare i valori degli attributi dell'oggetto che si sta creando (per mantenere memorizzato lo stato dell'oggetto)



## *interazione tra gli oggetti*

- per comunicare gli oggetti utilizzano i metodi, scambiandosi messaggi l'uno con l'altro
- quando si invoca un metodo l'oggetto reagisce eseguendo il metodo opportuno
- l'invocazione dei metodi può richiedere parametri di qualsiasi tipo, compresi quindi oggetti
- un oggetto può passarne un altro attraverso un metodo, o addirittura potrà passare se stesso
- un messaggio ha la seguente sintassi:  
    **<nomeOggetto>.<nomeMetodo>(<parametri>)**

## *oggetti*

```
from automobile import Automobile

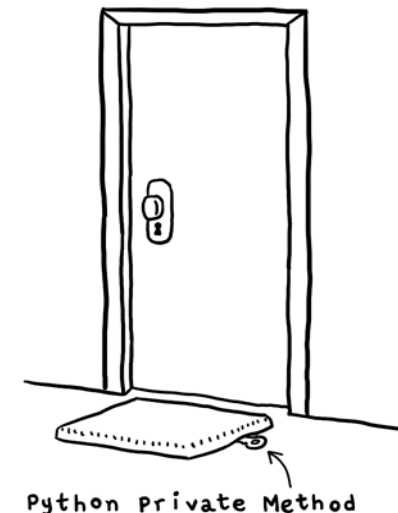
mario = Automobile(220,120,'tamburo')
print(mario.to_tuple())
print(mario.to_string())
mario.accelera(5)
print(mario.to_string())
mario.accelera(500)
print(mario.to_string())
for i in range(25):
    mario.frena()
    print(mario.to_string())
```

## *definizioni*

- ***campi***: memorizzano i ***dati caratteristici*** di una istanza
  - ogni pallina ha la sua posizione (x, y) e la sua direzione (dx, dy)
- ***parametri***: ***passano*** altri ***valori*** ad un metodo
  - se alcuni dati necessari non sono nei campi
- ***variabili locali***: memorizzano ***risultati parziali***
  - generati durante l'elaborazione del metodo
  - nomi ***cancellati*** dopo l'uscita dal metodo
- ***variabili globali***: definite ***fuori*** da tutte le funzioni
  - usare sono se strettamente necessario
  - meglio avere qualche parametro in più, per le funzioni

## *incapsulamento*

- l'**incapsulamento** (*information hiding*) è un concetto fondamentale dell'ingegneria del software
- questo principio prevede che si possa **accedere** alle informazioni di un oggetto **unicamente attraverso i suoi metodi**
- in **Python** l'incapsulamento non è vincolante; per consuetudine attributi privati vengono rappresentati mediante un identificatore che inizia con **\_** (underscore)
- in **Java** l'incapsulamento si avvale dei **modificatori di visibilità** per nascondere gli attributi di un oggetto
  - mettere in atto questa tecnica significa **non avere** mai **attributi** di un oggetto di tipo **public**, salvo eccezioni particolari per costanti o attributi di classe da gestire in base al caso specifico





## *accesso agli attributi*

- *information hiding*
  - gli attributi sono definiti *privati* (convenzione nome inizia con \_)
- metodi accessors (*getter*)
  - permettono l'accesso 'in lettura' agli attributi privati
- metodi mutators (*setter*)
  - permettono l'accesso 'in scrittura' agli attributi privati

```
• def __init__(self, a):  
    ## private varibale in Python  
    self.__a = a  
  
    ## getter method to get the properties  
    def get_a(self):  
        return self.__a  
  
    ## setter method to change the value 'a'  
    def set_a(self, a):  
        self.__a = a
```

## *in Python tutto è un oggetto*

```
n = 15                                # viene istanziato l'oggetto n della classe int
print('tipo di n ', type(n))          # type funzione
print('rappresentazione binaria di n ', bin(n)) # bin funzione
print('cifre binarie di n ', n.bit_length())    # bit_length metodo
```

```
tipo di n <class 'int'>
rappresentazione binaria di n 0b1111
cifre binarie di n 4
```

```
f = 3.25                                # f oggetto della classe float
print('tipo di f ', type(f))            # type funzione
print(f.as_integer_ratio(), 'frazione generatrice di', f) # as_integer_ratio metodo
```

```
tipo di f <class 'float'>
(13, 4) frazione generatrice di 3.25
```

## *Classi in Java*

```
public class Automobile {  
  
    private int vm;           // velocità massima  
    private int va;           // velocità attuale  
    private String freno;     // tipo freno  
  
    public Automobile(int vm, int va, String freno) {  
        this.vm = Math.max(vm, 0);  
        this.va = Math.max(va, 0);  
        if (freno.equals("tamburo"))  
            this.freno = "tamburo";  
        else  
            this.freno = "disco";  
    }  
}
```

```
public void accelera(int km){ // aumenta la velocità di km chilometri orari
    if (km <= 0)
        return;
    this.va += km;
    if (this.va > this.vm)
        this.va = this.vm;
}

public void frena(){ // dipende dal tipo dei freni
    if (this.freno.equals("disco"))
        this.va -= 20;
    else
        this.va -= 10;
    this.va = Math.max(this.va, 0);
}

public String toString(){
    String s = "velocità massima: " + this.vm;
    s += " velocità attuale: " + this.va;
    s += " freni a " + this.freno;
    return s;
}

}
```

```
public class AutomobileMain {  
  
    public static void main(String[] args) {  
        Automobile mario = new Automobile(220,120,"tamburo");  
        System.out.println(mario.toString());  
        mario.accelera(5);  
        System.out.println(mario);  
        mario.accelera(500);  
        System.out.println(mario);  
        for (int i = 0; i < 25; i++) {  
            mario.frena();  
            System.out.println(mario);  
        }  
    }  
}
```

## *dunder methods (metodi speciali)*

- metodi predefiniti utilizzati implicitamente da Python
- il loro nome inizia e finisce con due trattini bassi (underscore)
- ci si riferisce a questi metodi come ***dunder methods*** dove ***dunder*** sta per ***double underscore***
- esempi:
  - il metodo dunder init (**`__init__`**) viene automaticamente invocato al momento della creazione di un oggetto
  - il metodo **`__str__`** fornisce una rappresentazione sotto forma di stringa degli attributi dell'oggetto
    - viene chiamato automaticamente dalla funzione **`print`**

## *dunder methods (1)*

- **object.\_\_del\_\_(self)**
  - called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor.
- **object.\_\_lt\_\_(self, other)**
- **object.\_\_le\_\_(self, other)**
- **object.\_\_eq\_\_(self, other)**
- **object.\_\_ne\_\_(self, other)**
- **object.\_\_gt\_\_(self, other)**
- **object.\_\_ge\_\_(self, other)**
  - the correspondence between operator symbols and method names is as follows:  $x < y$  calls `x.__lt__(y)`,  $x \leq y$  calls `x.__le__(y)`,  $x == y$  calls `x.__eq__(y)`,  $x != y$  calls `x.__ne__(y)`,  $x > y$  calls `x.__gt__(y)`, and  $x \geq y$  calls `x.__ge__(y)`.

## *dunder methods (2)*

- `object.__add__(self, other)`
- `object.__sub__(self, other)`
- `object.__mul__(self, other)`
- `object.__and__(self, other)`
- `object.__xor__(self, other)`
- `object.__or__(self, other)`
  - these methods are called to implement the binary arithmetic operations (+, -, \*, &, ^, |)



## *variabili di istanza e variabili di classe*

- ogni oggetto (istanza di una classe) dispone di ***attributi*** che sono propri di se stesso
  - gli attributi di un oggetto sono chiamati ***variabili di istanza***
  - per accedere alle variabili di istanza si utilizza il riferimento ***self***
- le ***variabili di classe*** sono attributi ***condivisi*** da tutte le istanze della classe
  - per accedere alle variabili di classe si utilizza il nome della classe

## *variabili di classe esempio*

```
class Gatto:
    numero_gatti = 0

    def __init__(self,nome: str):
        ''' costruttore '''
        self.nome = nome
        Gatto.numero_gatti += 1
        print('ora esiste il gatto',self.nome)

    def __del__(self):
        ''' distruttore '''
        Gatto.numero_gatti -= 1
        print('ora non esiste più il gatto',self.nome)
```

```
g1 = Gatto('micio')
print('gatti presenti:',Gatto.numero_gatti)
g2 = Gatto('fufi')
print('gatti presenti:',Gatto.numero_gatti)
del g1
print('gatti presenti:',Gatto.numero_gatti)
del g2
print('gatti presenti:',Gatto.numero_gatti)
```

```
ora esiste il gatto micio
gatti presenti: 1
ora esiste il gatto fufi
gatti presenti: 2
ora non esiste più il gatto micio
gatti presenti: 1
ora non esiste più il gatto fufi
gatti presenti: 0
```

## *aggregazione - composizione*

- un attributo di un oggetto è un oggetto di un'altra classe
- esempio:
  - Punto: punto sul piano cartesiano
  - Segmento: gli estremi sono due punti

## *classe Punto*

```
class Punto:

    def __init__(self, x: float, y: float):
        self._x = x
        self._y = y

    def coordinate(self) -> (float, float):
        return self._x, self._y

    def __str__(self) -> str:
        return "(" + str(self._x) + "," + str(self._y) + ")"

    def sposta(self, dx: float, dy: float):
        self._x += dx
        self._y += dy
```

## *classe Punto (segue)*

```
def distanza_origine(self) -> float:
    '''
    restituisce la distanza del punto dall'origine degli assi
    '''
    return (self._x**2 + self._y **2) ** 0.5

def distanza_punto(self, p: 'Punto') -> float:
    '''
    restituisce la distanza dal punto p
    '''
    dx = self._x - p._x
    dy = self._y - p._y
    return (dx ** 2 + dy ** 2) ** 0.5
```

## *istanziamento di punti (oggetti della classe Punto)*

```
p1 = Punto(1,1)
print('punto p1',p1)
print("distanza di p1 dall'origine degli assi",p1.distanza_origine())
p1.sposta(1,2)
print('dopo lo spostamento p1',p1)
print("distanza di p1 dall'origine degli assi",p1.distanza_origine())
p2 = Punto(5,7)
print('punto p2',p2)
print('distanza fra p1 e p2',p1.distanza_punto(p2))
```

```
punto p1 (1,1)
distanza di p1 dall'origine degli assi 1.4142135623730951
dopo lo spostamento p1 (2,3)
distanza di p1 dall'origine degli assi 3.605551275463989
punto p2 (5,7)
distanza fra p1 e p2 5.0
```

## *classe Segmento*

```
from Punto import Punto

class Segmento:

    def __init__(self, a: Punto, b: Punto):
        self._a = a
        self._b = b

    def lunghezza(self) -> float:
        return self._a.distanza_punto(self._b)

    def __str__(self) -> str:
        return "estremo a = " + str(self._a) + " estremo b = " + str(self._b)
```

## *oggetti delle classi Punto e Segmento*

```
p1 = Punto(1,1)
print('punto p1',p1)
print("distanza di p1 dall'origine degli assi",p1.distanza_origine())
p1.sposta(1,2)
print('dopo lo spostamento p1',p1)
print("distanza di p1 dall'origine degli assi",p1.distanza_origine())
p2 = Punto(5,7)
print('punto p2',p2)
print('distanza fra p1 e p2',p1.distanza_punto(p2))
s = Segmento(p1,p2)
print(s)
print('lunghezza=',s.lunghezza())
p1._x = 0
print('lunghezza=',s.lunghezza())
```