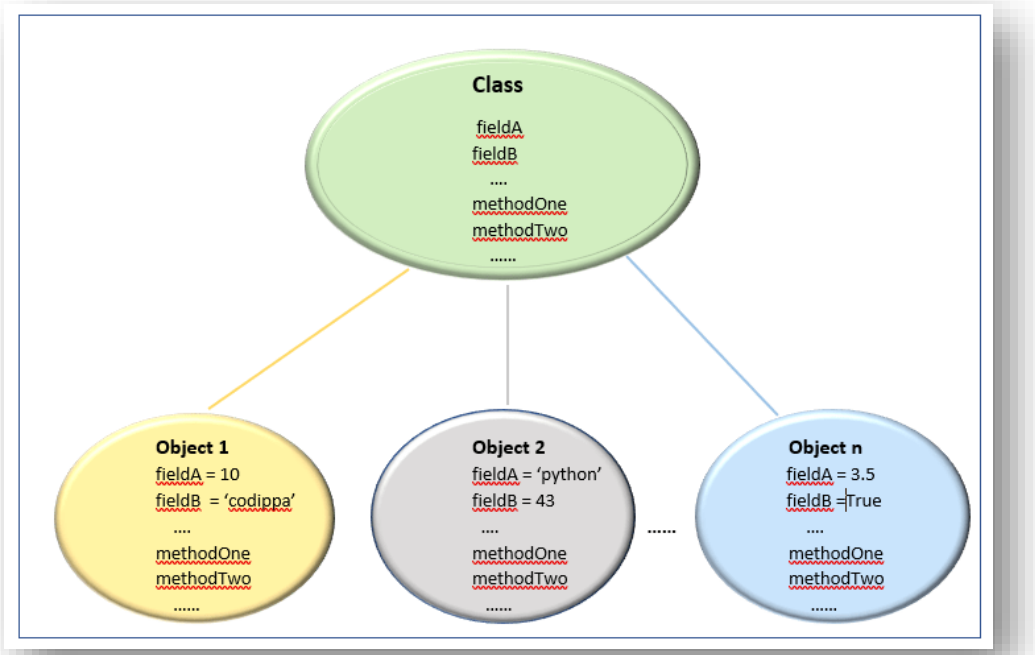
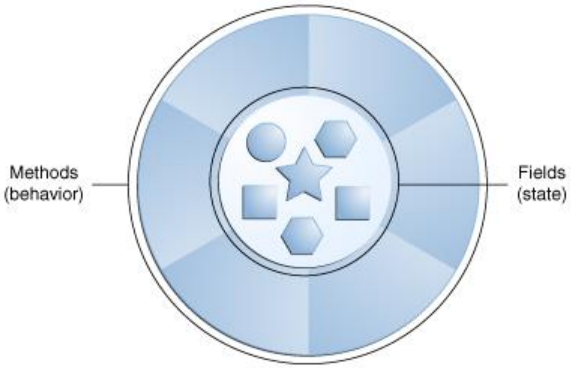


Programmazione Orientata agli Oggetti

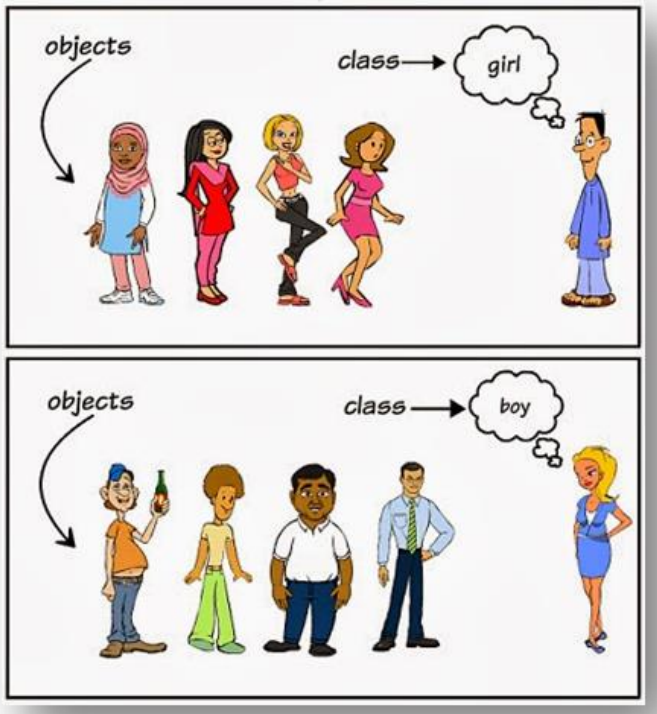
python

- un ***programma*** è un insieme di ***oggetti*** che ***interagiscono*** tra loro scambiandosi ***messaggi***
  - ogni oggetto mantiene il proprio stato (i propri dati)
- un ***oggetto*** appartiene a una ***classe*** che ne definisce le caratteristiche
  - la classe che serve a modellare un insieme di oggetti con le stesse caratteristiche in grado di compiere le stesse azioni
- una ***classe*** definisce un nuovo ***tipo di dato*** *astratto* (ADT)

- 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)

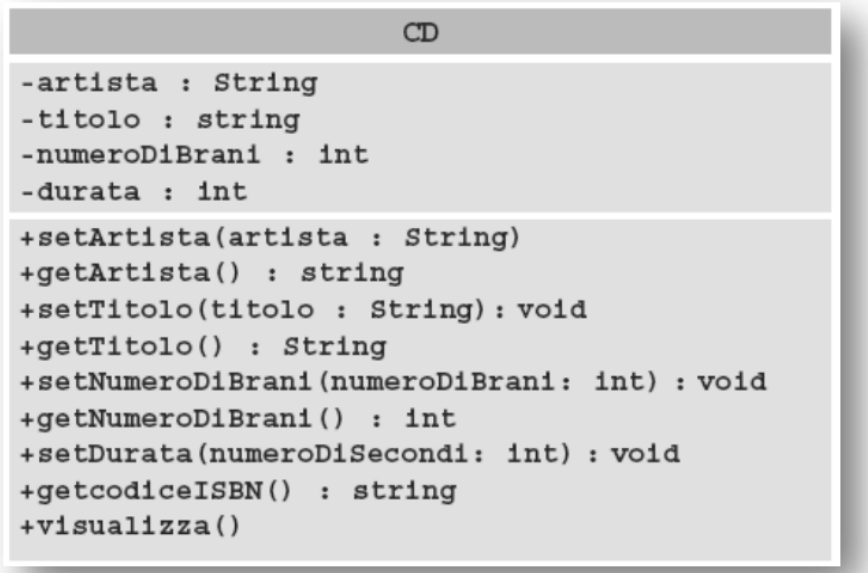


- ogni oggetto ha una ***classe*** di origine (*è istanziato da una classe*)
- la classe definisce le ***caratteristiche comuni*** (campi e metodi) a tutti i suoi oggetti
- ma ogni oggetto
  - ha la sua ***identità***
  - ha uno stato e una locazione in memoria distinti da quelli di altri oggetti



esempio di classe

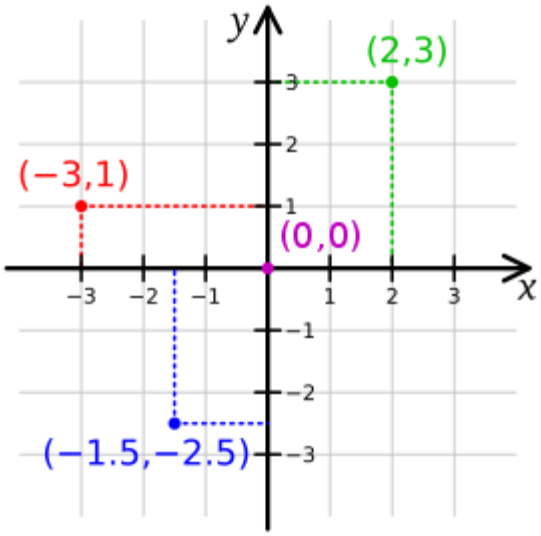
- per catalogare i cd musicali abbiamo bisogno di implementare un programma nel cui dominio applicativo è presente la classe CD
- i metodi della classe CD servono per impostare e recuperare i valori degli attributi



esempio di oggetti

- i diagrammi che rappresentano gli oggetti *(**Object Diagram** in **UML**)* mettono in luce i ***valori*** che assumono gli attributi

- punto sul piano cartesiano 2d
  - coordinate x y
  - distanza dall'origine degli assi
  - distanza da un altro punto



Punto
- x: float - y:float
+ Punto(float, float) + coordinate(): float, float + distanza_origine(): float + distanza_punto(Punto): float

- costruzione di oggetti (***istanziazione***)
- **`__init__`**: metodo ***inizializzatore***
- eseguito ***automaticamente*** alla creazione di un oggetto
- **`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

```
p1 = Punto(40, 80)  # Allocatione e inizializzazione
```

Punto – implementazione (1)

A. Ferrari

```
class Punto:
    '''
    rappresenta un punto sul piano cartesiano
    in uno spazio bidimensionale
    '''

    def __init__(self, x: float, y: float):
        '''
        inizializzazione attributi (coordinate)
        '''
        self._x = x
        self._y = y

    def coordinate(self) -> (float, float):
        '''
        coordinate del punto
        '''
        return self._x, self._y
```

Punto – implementazione (2)

A. Ferrari

```
    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
```

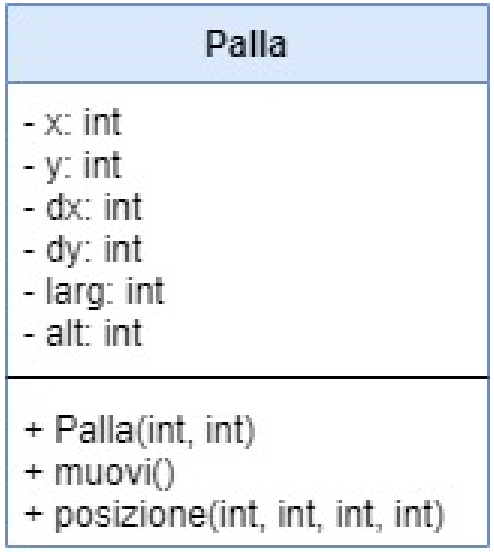
Punto – oggetti

```
def main():
    p1 = Punto(3,4)
    x , y = p1.coordinate()
    # x = p1.coordinate()[0]
    # y = p1.coordinate()[1]
    print(p1,end=' ')
    #print('punto (',x,',',y,')',end = ' ')
    print("dista dall'origine",p1.distanza_origine())
    p2 = Punto(5,5)
    print(p2,end=' ')
    print("dista dal punto",p2.coordinate(),p1.distanza_punto(p2))
```

Palla
(si muove in canvas 2d)

A. Ferrari

- attributi
  - x, y coordinate
  - dx, dy spostamento orizz / vert
  - larg, alt dimensione
- costruttore
  - Palla
- metodo
  - muovi (spostamento dx e dy)
    - logica di spostamento
  - posizione restituisce posizione e dimensioni



Palla – implementazione (1)

A. Ferrari

```
class Palla:
    '''
    rappresenta un oggetto che si muove
    in uno spazio bidimensionale
    '''

    def __init__(self, x: int, y: int, dx: int = 5, dy: int  = 5):
        '''
        inizializzazione attributi
        '''
        self._x = x
        self._y = y
        self._dx = dx
        self._dy = dy
        self._w = 20
        self._h = 20
```

Palla – implementazione (2)

A. Ferrari

```
    def muovi(self):
        '''
        sposta la pallina secondo le direzioni dx e dy
        '''
        if not (0 <= self._x + self._dx <= ARENA_L - self._w):
            self._dx = -self._dx
        if not (0 <= self._y + self._dy <= ARENA_H - self._h):
            self._dy = -self._dy
        self._x += self._dx
        self._y += self._dy

    def posizione(self) -> (int, int, int, int):
        '''
        restituisce coordinate e dimensioni della pallina
        '''
        return self._x, self._y, self._w, self._h
```

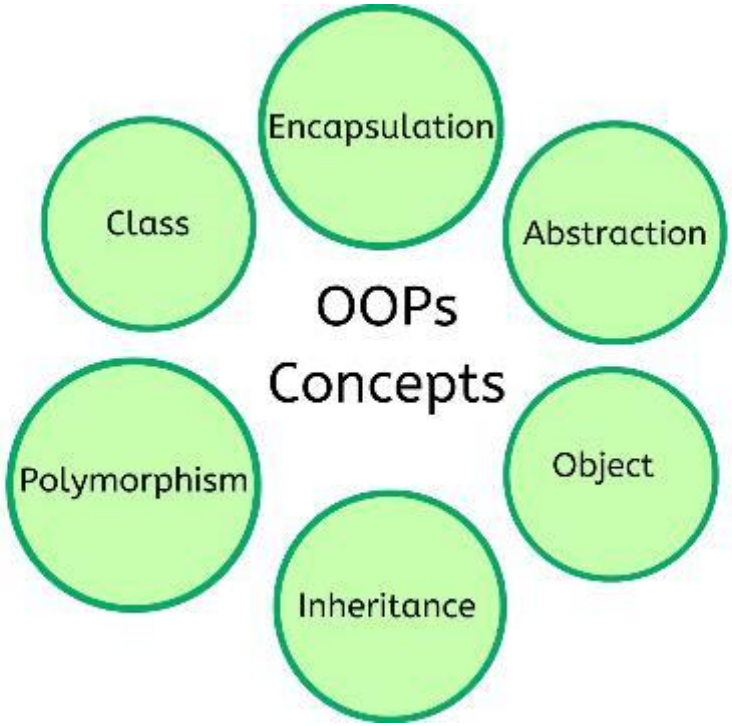
Palla - test

```
ARENA_L = 320              # larghezza arena
ARENA_H = 240              # altezza arena

def main():
    p1 = Palla(10,20)
    p2 = Palla(34,40,12,23)
    for i in range(1,80):
        print('p1 si trova in posizione',p1.posizione())
        print('p2 si trova in posizione',p2.posizione())
        p1.muovi()
        p2.muovi()
```

```
import g2d
from sl04_01_Palla import Palla, ARENA_L, ARENA_H

def avanza():
    g2d.clear_canvas()                  # "pulisce il background
    p1.muovi()
    p2.muovi()
    g2d.set_color((0, 0, 255))          # colore prima palla
    g2d.fill_rect(p1.posizione())       # disegno prima palla
    g2d.set_color((0, 255, 0))          # colore seconda palla
    g2d.fill_rect(p2.posizione())       # disegno seconda palla

p1 = Palla(40, 80)                          # movimento dx 5 dy 5
p2 = Palla(80, 40, 3, 2)                    # movimento dx 3 dy 2

def main():
    g2d.init_canvas((ARENA_L, ARENA_H))
    g2d.main_loop(avanza, 1000 // 30)  # Millisecondi (frame rate)

if __name__ == "__main__":  # solo se è il modulo principale
    main()
```

- ***campi***: memorizzano i ***dati caratteristici*** di una istanza
  - ogni pallina ha la sua posizione (x, y), la sua direzione (dx, dy) e la sua dimensione (w,h)
- ***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

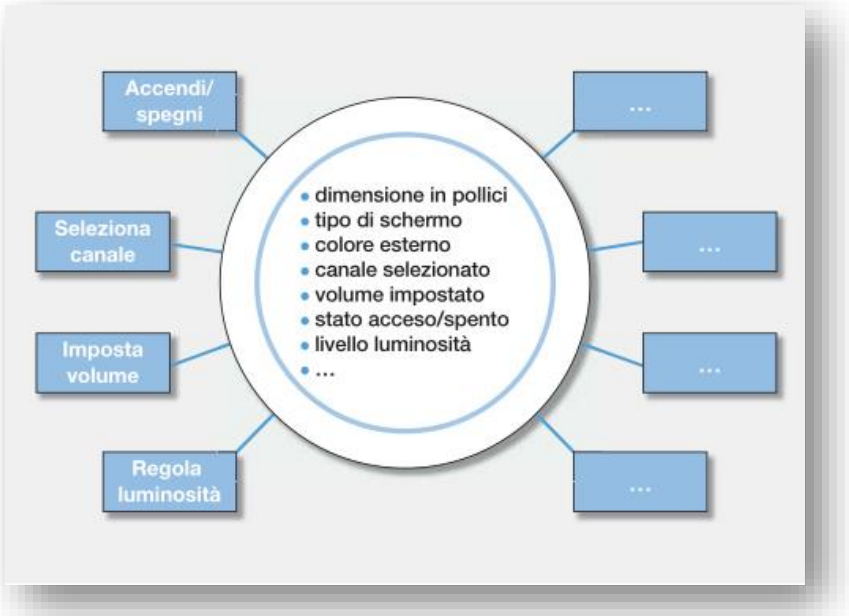
- ***astrazione***
- ***incapsulamento***
- ***composizione***
- ***ereditarietà***
- ***polimorfismo***



- (*definizione*) l'***astrazione*** è un procedimento mentale che permette di ***evidenziare*** alcune ***proprietà***, ritenute significative, relative ad un determinato fenomeno osservato escludendone altre considerate non rilevanti per la sua comprensione
- l'astrazione comporta la creazione di modelli
  - nel caso dell'OOP i modelli sono rappresentati dalle classi

incapsulamento *(information hiding)*

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



- un oggetto di una classe è composto da oggetti di altre classi

composizione esempio (segmento)

A. Ferrari

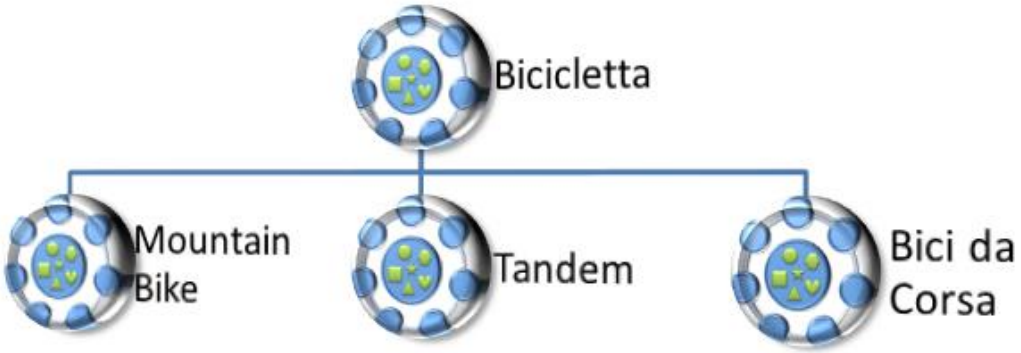
```
class Segmento:
    '''
    rappresenta un segmento sul piano cartesiano
    in uno spazio bidimensionale
    '''
    def __init__(self, p1: Punto, p2:Punto):
        '''
        inizializzazione attributi (coordinate)
        '''
        self._p1 = p1
        self._p2 = p2

    def lunghezza(self) -> float:
        '''
        lunghezza del segmento
        '''
        return self._p1.distanza_punto(self._p2)

    def __str__(self) ->str:
        '''
        rappresentazione del segmento
        '''
        return "[" + str(self._p1) + " - " + str(self._p2) + "]"
```

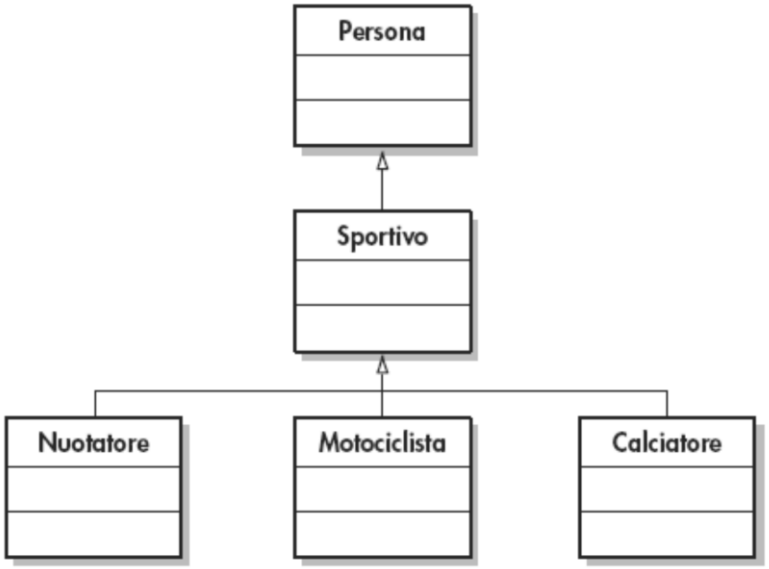
composizione esempio (triangolo)

A. Ferrari

```
class Triangolo:
    '''
    rappresenta un triangolo sul piano cartesiano
    '''
    def __init__(self, a: Punto, b:Punto, c:Punto):
        ''' inizializzazione attributi (lati) '''
        self._lati = [Segmento(a,b),Segmento(b,c),Segmento(c,a)]
    def perimetro(self) -> float:
        ''' perimetro '''
        p = 0
        for l in self._lati:
            p += l.lunghezza()
        return p
    def area(self) ->float:
        ''' area (formula di Erone) '''
        p = self.perimetro() / 2
        a , b, c = self._lati
        return sqrt(p*(p-a.lunghezza())*(p-b.lunghezza())*(p-c.lunghezza()))
```

- l'ereditarietà permette di **definire nuove classi** partendo da classi sviluppate in precedenza
- la nuova classe viene definita esprimendo solamente le **differenze** che essa possiede rispetto alla classe di partenza
- l'ereditarietà permette di specificare "il punto di partenza", cioè la **classe base**, e le **differenze** rispetto a questa

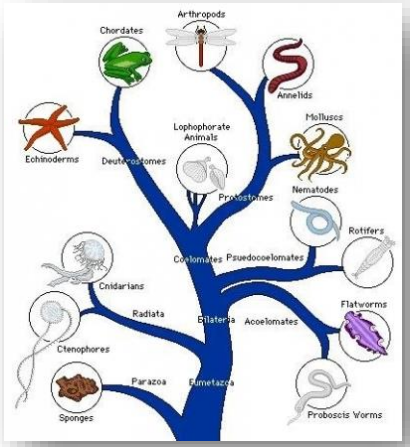




- l'ereditarietà può estendersi a più livelli generando una **gerarchia di classi**
- una classe derivata può, a sua volta, essere base di nuove sottoclassi
- nell'esempio: Sportivo è sottoclasse di Persona ed è superclasse di Nuotatore, Motociclista e Calciatore
- nella parte alta della gerarchia troviamo le **classi generiche**, scendendo aumenta il **livello di specializzazione**

ereditarietà
relazione is-a

- relazione ***is-a***
  - ***classificazione***, es. in biologia
    - vertebrati ***sottoclasse*** di animali
    - mammiferi ***sottoclasse*** di vertebrati
    - felini ***sottoclasse*** di mammiferi
    - gatti ***sottoclasse*** di felini
- relazione ***is-a tra classi***: ogni sottoclasse...
  - ***eredita*** le caratteristiche della classe base
  - ma introduce delle ***specializzazioni***



- il ***polimorfismo*** è la capacità espressa dai metodi ridefiniti di assumere ***forme*** (implementazioni) ***diverse*** all'interno di una gerarchia di classi o all'interno di una stessa classe
- il polimorfismo indica la possibilità dei ***metodi*** di possedere ***diverse implementazioni***

- definiremo una ***classe base*** come ***interfaccia astratta***
  - es. Animale:
- tutti gli animali fanno un verso (***interfaccia***)
- ogni animale fa un verso diverso (***polimorfismo***)

```
class Animale:
   def parla(self):
     raise NotImplementedError("metodo astratto")
```

```
class Cane(Animale):
    def __init__(self, nome):
        self._nome = nome
    def parla(self):
        print("sono", self._nome, ", un cane ... bau")
class Gatto(Animale):
    def __init__(self, nome):
        self._nome = nome
    def parla(self):
        print("sono", self._nome, ", un gatto ... miao")
```