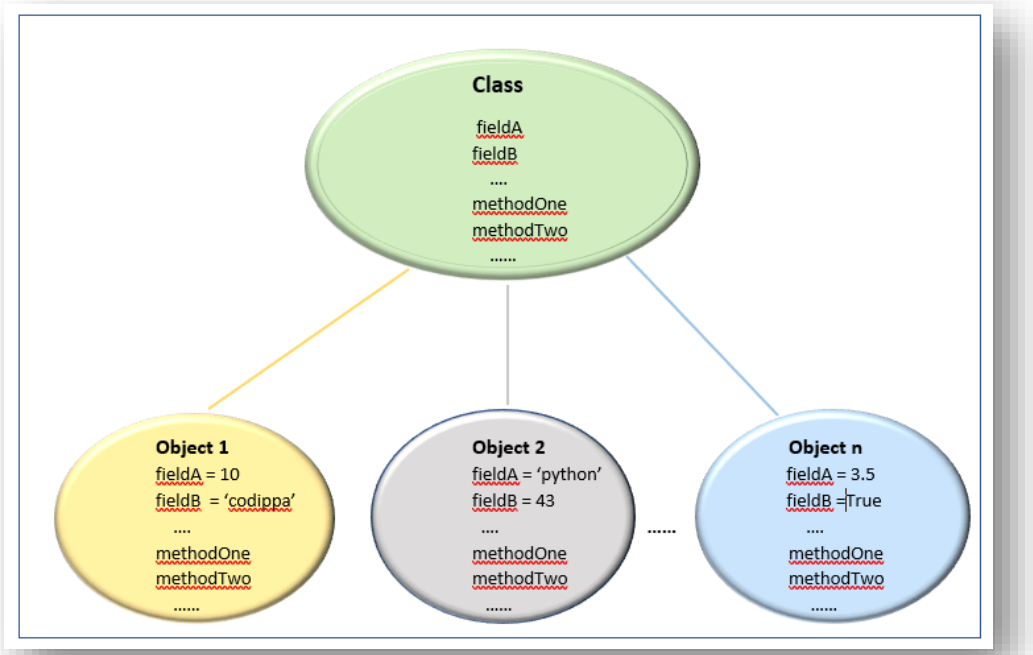
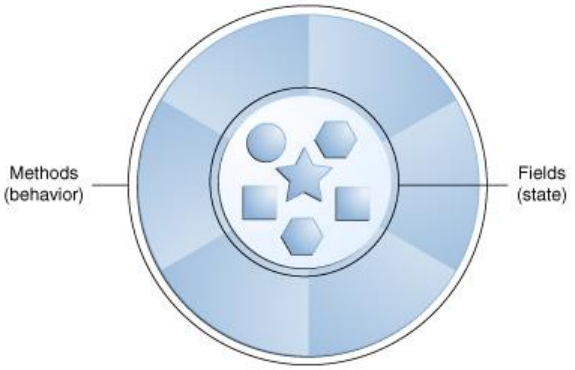


Oggetti

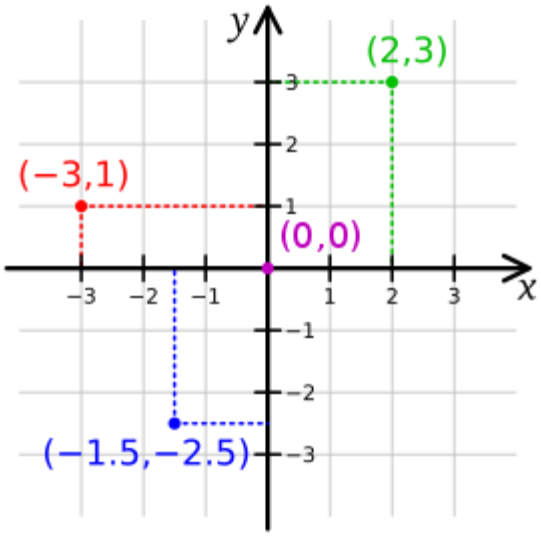
python

- 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*)



- 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

- 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
  - 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

```
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

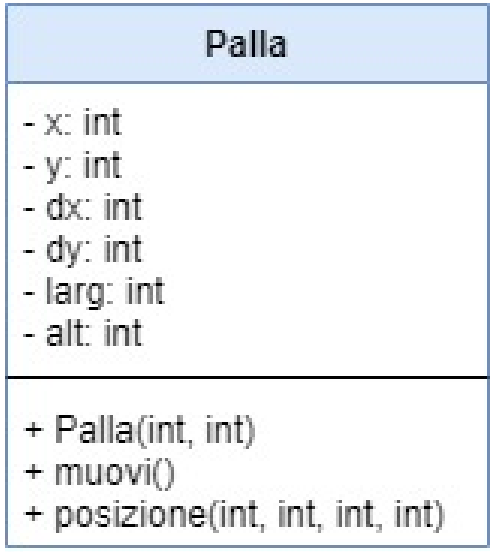
A. Ferrari

```
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

A. Ferrari

```
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()
```

```
from cl_ball import Ball  # Ball is defined in cl_ball.py

# Create two objects, instances of the Ball class
b1 = Ball(40, 80)      # allocation and initialization
b2 = Ball(80, 40)

for i in range(25):
    print('Ball 1 @', b1.position())
    print('Ball 2 @', b2.position())
    b1.move()
    b2.move()
```

```
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()
```

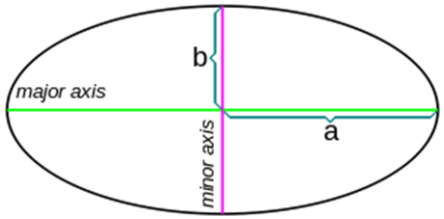
- ***campi***: memorizzano i ***dati caratteristici*** di una istanza
  - ogni pallina ha la sua posizione (x, y) e la sua direzione (dx, dy)
- ***parametri***: ***passano*** altri ***valori*** ad un metodo
  - se alcuni dati necessari non sono nei campi
- ***variabili locali***: memorizzano ***risultati 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

oggetti in python 3

esercizi

4.1 classe ellisse

- definire una ***classe*** che modella un'ellisse
- ***campi privati*** (parametri del costruttore)
  - semiassi: a, b
- ***metodi pubblici*** per ottenere:
  - area: π·a·b
  - distanza focale: $2 \cdot \sqrt{|a2 - b2|}$
- nel corpo principale del ***programma***
  - creare un oggetto con dati forniti dall'utente
  - visualizzare area e distanza focale dell'ellisse



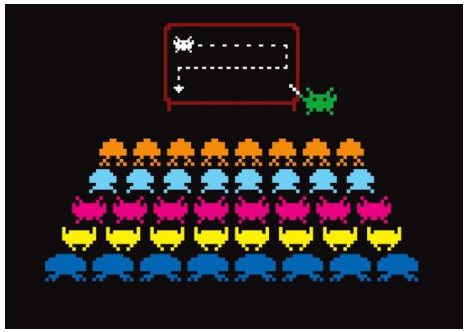
4.2 animazione di una pallina

- partire dalla classe ***Ball***
- eseguire l'animazione:
  - per ogni frame, chiamare il metodo ***move*** della pallina
  - rappresentare un rettangolo o un cerchio nella ***posizione aggiornata*** della pallina
- ***modificare*** però il metodo move
  - la pallina si sposta sempre di pochi pixel in orizzontale
  - la pallina non si sposta verticalmente
  - se esce dal bordo destro, ricompare al bordo sinistro e viceversa

4.3 classe degli alieni

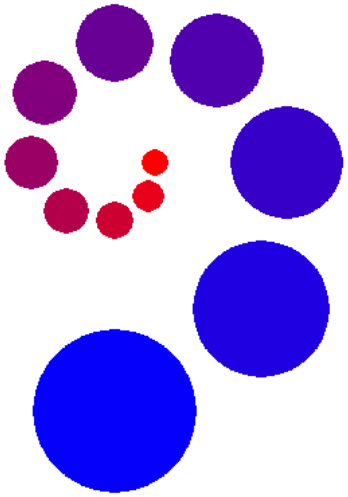
- creare una classe ***Alien***
  che contenga i ***dati*** ed il ***comportamento*** dell'alieno
  - campi privati: x, y, dx
  - metodo ***move*** per avanzare
  - metodo ***position*** per ottenere la posizione attuale
- istanziare un ***oggetto*** Alien e farlo muovere sullo schermo
  - chiamare il metodo move ad ogni ciclo
  - visualizzare un rettangolo nella posizione corrispondente

definire nella classe delle opportune costanti



4.4 spirale

- mostrare l'animazione di un cerchio lungo una ***spirale***
- ruotare attorno ad un ***centro fisso*** (xc, yc)
- ***aumentare*** la ***distanza*** r dal ***centro*** ad o
- cancellare lo sfondo ad ogni passo
- disegnare un cerchio sempre ***più grande***
- dopo ***n*** passi, ricominciare da capo



http://www.ce.unipr.it/brython/?p2_fun_spiral.py

4.5 classe spirale

- mostrare l'animazione di un cerchio lungo una spirale
- realizzare una classe per gestire dati e comportamento del cerchio
- implementare il movimento in un metodo ***move()***
- campi: ***xc, yc, i***
- ***i*** conta i passi; se eccede il limite, torna a 0