Classi e oggetti

Object Oriented Programming

- L' Object Oriented Programming (OOP) è un paradigma di programmazione basato sul concetto di "**Oggetti**"
    - Un oggetto contiene dei dati nella forma di **campi** (anche detti **attributi**)
    - Allo stesso tempo un oggetto è un qualcosa di più complesso di una variabile perchè contiene diversi dati e può offrire dei servizi ***disponendo di funzioni per manipolare i dati che contiene al suo interno*** (chiamati **metodi**)

    - Gli oggetti descrivono entità che hanno un loro **stato interno**
    - Per il momento, pensate agli oggetti come ad una sorta di soluzione alla necessità di dover creare in alcune situazioni un proprio "data type" che non rientri nei classici int, float, bool, String.

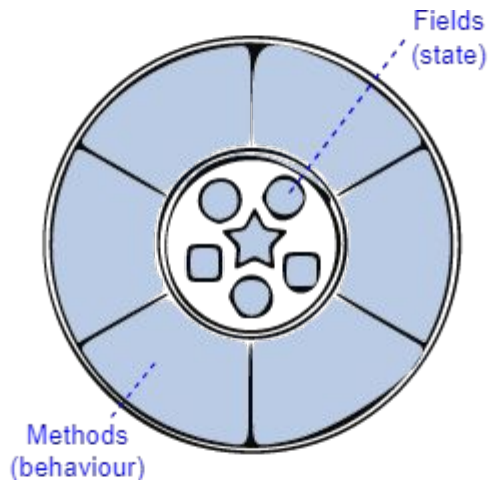
- **Spoiler**: Ogni cosa in Python è un oggetto, ed avete quindi inconsapevolmente già usato gli oggetti per programmare

Cosa è un oggetto?

- Un oggetto rappresenta un modo per modellare a livello logico entità fisiche (e.g., sensori, oggetti reali, un prodotto….un personaggio in un video game)
- Esempio:
-
  - object name: “Mario’s car”
  - class : car #Definiremo presto il concetto di classe
  - attributes:
    - 4 wheels
    - Speed
    - Current gear
  - methods:
    - acceleration()
    - drive()
    - change gear()

Cosa è un oggetto?

- Rappresenta un *oggetto fisico* o un *concetto* del dominio
- Memorizza il suo **stato** interno in *campi privati*
    - *Incapsulamento (black box)*
- Offre un insieme di **servizi**, come *metodi pubblici*
    - Realizza un *tipo di dato astratto (ADT)*

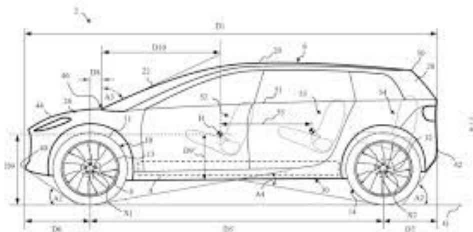


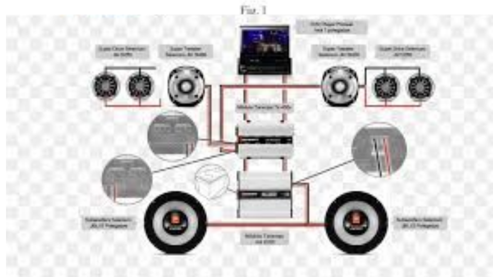
Classi e oggetti

- Ogni *oggetto* ha una **classe** di origine
  - La classe dà la stessa forma iniziale (campi e metodi) a tutti i suoi oggetti
- Ma ogni *oggetto* ha la sua **identità**
  - Stato e locazione in memoria distinti da quelli di altri oggetti
  - Sia istanze di classi diverse che della stessa classe

Classi e oggetti

- **Class**
  - Quale è il concetto da descrivere?
  - Come deve funzionare l'oggetto?
  - Come devono essere costruiti gli oggetti ?





- **Object**
  - E' l'istanza concreta che creiamo
  - E' creato secondo le linee guida e direttive definite nella sua classe

Tutto è un oggetto in Python

- Tutte le “variabili” che avete usato sino ad ora sono in realtà oggetti :)
- Quando definite:
  - area=5 un oggetto chiamato “area” è creato dalla classe “Integer” che definisce cosa è un valore int e quali metodi può offrirvi per cambiare il suo stato interno
  - message = “Hello world” è un oggetto di classe String

- Infatti proprio sulle stringhe avete usato maggiormente la OOP perchè richiedevate dei servizi agli oggetti stringhe tipo
  - .split()
  - .replace()

Classi in Python

- Una classe deve quindi definire:
    - Lo **stato interno** di un oggetto mediante i suoi attributi (campi)
        - Cioè mediante variabili
    - **Metodi** per modificare lo stato interno
        - Cioè delle funzioni, che però operano solo ed unicamente sugli oggetti su cui sono invocate

    - Una speciale funzione chiamata “**Costruttore**” che permette di istanziare un oggetto a partire dalla classe
        - Cioè una funzione che quando invocata chiede virtualmente alla fabbrica di produrre un oggetto di una tal classe
        - La funzione costruttore prende opzionalmente degli input ed ha il compito di definire lo stato iniziale degli oggetti che genera
            - Restituisce (**return**) un oggetto della classe a cui appartiene la funzione costruttore

Esempio

```
class Product:
        def __init__(self, name,price,number):
                self._name = name              # attribute
                self._price = price            # attribute
                self._number = number          # attribute

        def get_name(self):                     #method
                return self._name
        def get_price(self):                    #method
                return self._price
        def set_price(self,new_price):          #method
                self._price=new_price
def main():
        p = Product("pen",1.50,200)             #Instantiation of an object p (Product type)
        print(p.get_name())                     #call of a method on object P
        print(p.get_price())
        p.set_price(2.00)
        print(p.get_price())
main()
```

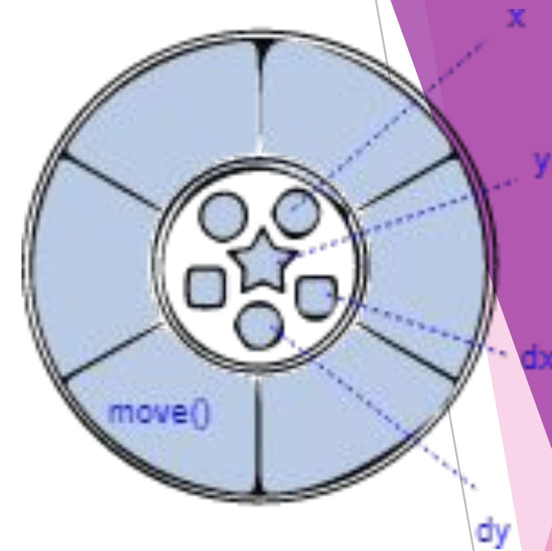
Py

self: Keyword speciale che dice all'interprete Python che metodi e attributi devono essere assegnati ad ogni oggetto costruito a partire da questa classe

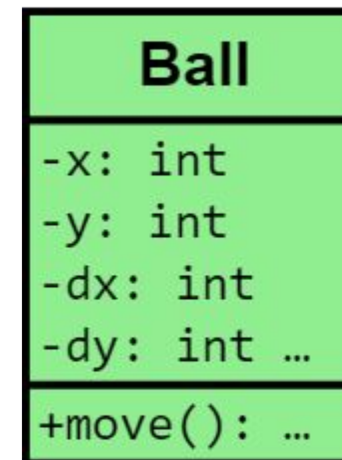
Definizione di una classe

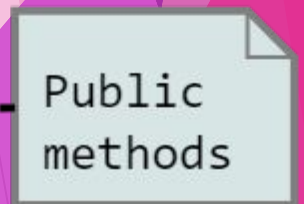
- **Incapsulamento** dei dati: *convenzione* sui nomi
  - Prefisso _ per i nomi dei *campi privati*

```
class Ball:
    # …
    def __init__(self, x0: int, y0: int):
        self._x = x0
        self._y = y0
        self._dx, self._dy = 5, 5
```



- Definiamo tutti i campi necessari alla pallina
  - Rappresentazione completa del suo stato
  - ⇒ `self._x, self._y, self._dx, self._dy`





⭐ Costruzione di un oggetto

- **`__init__`** : metodo *costruttore*
  - Eseguito automaticamente alla creazione di un oggetto
  - *Instantiation is initialization*
- **`self`** : primo parametro di tutti i metodi
  - Rappresenta l'*oggetto* su cui svolgere l'operazione
  - Permette ai metodi di accedere ai campi
  - Non bisogna passare un valore esplicito
- Altri parametri, dopo `self` : li decidiamo noi
  - Non vogliamo creare tutte le palline nella stessa posizione
  - ⇒ Parametri `x0, y0`

```
ball = Ball(40, 80)   # Allocation and initialization
```

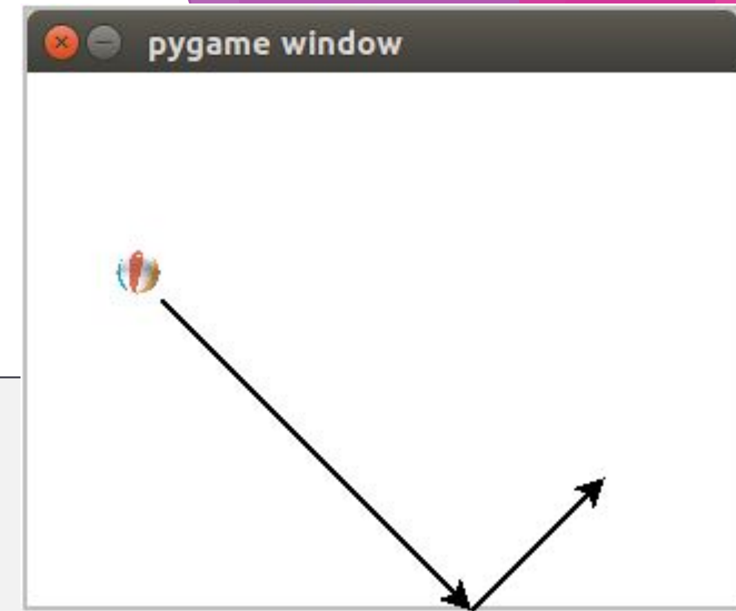
Metodi

- Espongono *servizi* ad altri oggetti

```
ARENA_W, ARENA_H, BALL_W, BALL_H = 480, 360, 20, 20

class Ball:
    # ...
    def move(self):
        if not 0 <= self._x + self._dx <= ARENA_W - BALL_W:
            self._dx = -self._dx
        if not 0 <= self._y + self._dy <= ARENA_H - BALL_H:
            self._dy = -self._dy
        self._x += self._dx
        self._y += self._dy

    def pos(self):  # "getter" method: doesn't modify the state
        return self._x, self._y
```



Uso degli oggetti

```
# Create two objects, instances of the Ball class
b1 = Ball(140, 180)
b2 = Ball(180, 140)

for i in range(25):
    b1.move()
    b2.move()
    print("b1 @", b1.pos(),
          "b2 @", b2.pos())
```

- Nei suoi *campi privati*, ogni oggetto memorizza tutto il suo stato
    - Usiamo i campi al posto delle variabili globali
    - `self._x, self._y, self._dx, self._dy`

https://github.com/tomamic/fondinfo/blob/master/p04_ball.py

Il primo parametro self

- Il primo parametro di ogni metodo si chiama `self` (per convenzione)
- Il valore di `self` viene assegnato *automaticamente*
- Rappresenta l'*oggetto* di cui viene invocato il metodo
- In Python, una chiamata a metodo è interpretata così:

```py
b1 = Ball(140, 180)
b1.move()
```

```py
# ⚠ Python internals, DON'T do this!
b1 = object.__new__(Ball)
Ball.__init__(b1, 140, 180)
Ball.move(b1)
```

Animazione di due palline

```python
b1 = Ball(140, 180)
b2 = Ball(180, 140)

def tick():
    g2d.clear_canvas()
    b1.move()
    b2.move()
    g2d.draw_image("ball.png", b1.pos())
    g2d.draw_image("ball.png", b2.pos())

def main():
    g2d.init_canvas((ARENA_W, ARENA_H))
    g2d.main_loop(tick)
```

Riepilogo dei concetti

- *Campi*: memorizzano i dati caratteristici di una istanza
  - Ogni pallina ha la sua posizione `(self._x, self._y)` e la sua velocità `(self._dx, self._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