lez_06

January 21, 2026

1 Cenni di OOP

La programmazione orientata agli oggetti (Object Oriented Programming - OOP) è un paradigma di programmazione che ci consente di modellare strutture dati come se fossero *oggetti*, che interagiscono tra loro attraverso lo scambio di *messaggi*. Questo è basato sul concetto fondamentale di **classe** e presenta, tra gli altri, i seguenti vantaggi:

1. **Modellazione**: la OOP consente di modellare in modo intuitivo oggetti reali e astratti nel software;
2. **Gestibilità**: l'organizzazione del codice così prodotto ne consente una manutenzione e una modifica relativamente semplici;
3. **Modularità**: l'organizzazione del software in classi favorisce il riutilizzo delle sue parti.

Questo paradigma è nato agli inizi degli anni '70 con linguaggi come *Simula* e *Small-talk*, per poi diventare negli anni '90 il paradigma predominante per lo sviluppo di software. Linguaggi che fanno utilizzo intensivo del paradigma ad oggetti sono, tra gli altri: *C++*, *Java*, *Python*, *Ruby*.

Affinché un linguaggio sia definito orientato agli oggetti deve rispettare 3 caratteristiche:

1. **Incapsulamento**: la definizione della classe e l'interfaccia attraverso la quale si può interagire con essa sono **separate** e l'utente finale ha accesso esclusivamente alla seconda;
2. **Ereditarietà**: deve essere possibile definire una classe a partire da un'altra classe;
3. **Polimorfismo**: classi diverse possono essere utilizzate con un'interfaccia comune.

Tutti questi aspetti saranno introdotti e chiariti nel corso della lezione con esempi estremamente semplici, quindi: **niente paura**.

In termini meno precisi ma più concreti, il modello ad oggetti cerca di fornire una rappresentazione che rappresenti le relazioni tra i diversi elementi che compongono il problema reale. Prendiamo ad esempio un'azienda e supponiamo di voler progettare un software che gestisca i dipendenti. Se adottiamo un approccio orientato agli oggetti, possiamo vedere i dipendenti come una **Classe**, e ogni dipendente sarà un **Oggetto** o istanza di quella classe.

Se allarghiamo lo sguardo, ci rendiamo conto che questo ragionamento è estendibile a molti altri aspetti di un sistema aziendale: ad esempio troveremo tipi diversi di dipendenti e ciascun tipo può essere descritto con una classe diversa. Avremo quindi la classe impiegati, la classe direttori, ecc.

Anche quando non si parla di persone possiamo ragionare in termini di oggetti: un software gestionale tiene solitamente traccia degli ordini, pertanto potremmo pensare di definire una classe *Ordini*. Ciascun oggetto di questa classe sarebbe un ordine, con la sua data, il suo valore, il commerciale di riferimento, ecc.

Inoltre un ordine può cambiare il suo stato nel tempo: può passare da richiesto ad approvato ad evaso e così via. In questo caso quello che faremo sarà definire un **metodo** che modifichi lo stato di un ordine.

Per passare agli aspetti pratici, vediamo ora come si definisce una classe, quali sono le sue caratteristiche e come istanziare degli oggetti appartenenti a quella classe in Python.

[1]:
```
print(type(1))
print(type('a'))
print(type([1, 2, 3]))
```

```
<class 'int'>
<class 'str'>
<class 'list'>
```

Come vediamo dall'output di *type()*, abbiamo già incontrato qualche classe lungo la nostra strada. Tutti i cosiddetti *tipi* di variabile in python sono definiti attraverso l'utilizzo di classi. Nella lezione successiva creeremo anche noi i nostri personalissimi *tipi*, utilizzando la OOP per definire due tipi di strutture dati, la pila e la coda.

Per il momento restiamo sulle cose semplici ed iniziamo con la cosa più familiare che abbiamo a disposizione: definiamo una classe che definisce una *Persona*:

[7]:
```
class Person:
    def __init__(self, nome, età):
        self.name = nome
        self.age = età

persona1 = Person('Lorenzo', 27)
print(type(persona1))
```

```
<class '__main__.Person'>
```

Smontiamo questo codice riga per riga per capire cosa sta succedendo. Iniziamo col definire una classe: una classe in python si definisce con la keyword **class**, seguita dal nome che vogliamo assegnare a quella classe. Fino a qui, nulla di astruso.

Subito dopo aver assegnato il nome alla classe, troviamo quella che sembra la definizione di una funzione. In realtà, quando siamo all'interno di una classe, non definiamo funzioni, ma **metodi**. Questi metodi esisteranno SOLO per gli oggetti di quella classe, e saranno applicabili solo a questi.

Il metodo che stiamo definendo è il metodo *__init__*(), ed è un metodo speciale. Ogni volta che vediamo un metodo che inizia e finisce con due underscore ("___"), sappiamo che si tratta di un metodo riservato da python che svolge una funzione ben precisa.

Nel nostro caso, init specifica quali parametri dobbiamo fornire quando decidiamo di creare un oggetto appartenente a quella classe. Nel caso della nostra persona, ci siamo limitati a richiedere nome ed età.

self è il primo parametro di TUTTI i metodi che definiamo, e fa riferimento all'oggetto stesso.

Il metodo init prende il parametro nome e lo assegna ad un **attributo** della classe chiamato *name*. Stessa cosa per quanto riguarda l'età. Questi attributi prendono il nome di **instance variables** o

variabili appartenenti all'oggetto, ed ogni oggetto della classe Person avrà il suo valore per *name* e il suo valore per *age*.

Questi attributi sono accessibili con la notazione *oggetto.nome_attributo*:

[9]:
```
print('Questa persona si chiama', persona1.name)
print('Ed ha', persona1.age, 'anni')
```

```
Questa persona si chiama Lorenzo
Ed ha 27 anni
```

Notare l'assenza di parentesi dopo *name* ed *age*: questo perché stiamo parlando di attributi, di variabili, non di metodi o funzioni!

Ora definiremo qualche nuova persona, e poi definiremo dei metodi per lavorare con questi oggetti:

[10]:
```
persona2 = Person('Chiara', 21)
persona3 = Person('Giorgia', 26)
persona4 = Person('Bruno', 53)
```

Vediamo ora un secondo metodo speciale che è possibile definire su una classe: il metodo *__str__()*

Questo metodo ci consente di definire una stringa *informativa* che verrà stampata quando cerchiamo di passare un oggetto di quella classe alla funzione *print()*.

Infatti se cerchiamo di stampare i nostri oggetti, otteniamo cose poco informative:

[11]:
```
print(persona1)
print(persona4)
```

```
<__main__.Person object at 0x7efff032d1f0>
<__main__.Person object at 0x7efff032d730>
```

Definendo il metodo *__str__()* invece:

[16]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return self.name + ' ha ' + str(self.age) + ' anni'

p1 = Person('Lorenzo', 27)
p2 = Person('Giorgia', 26)
p3 = Person('Bruno', 52)

print(p1)
print(p3)
```

```
Lorenzo ha 27 anni
Bruno ha 52 anni
```

Abbiamo ridefinito la classe aggiungendo il metodo *__str()__* e abbiamo creato 3 nuove istanze (o oggetti) di quella classe. Vediamo come stavolta, quando andiamo a stampare le istanze, otteniamo un messaggio decisamente più informativo.

> **NOTA**: ovviamente non è buona pratica riaprire la classe di volta in volta nel codice per fare delle modifiche. Ciascuna classe è solitamente definita in un file a parte, che solitamente si trova in una cartella appositamente creata chiamata *classes* e vengono importate quando necessario. Ricordo che la OOP viene utilizzata nella creazione di grandi progetti software, e raramente torna utile o è di come utilizzo quando il nostro lavoro è fare scripting per analisi dati ecc.. Questa lezione vuole solo essere un'introduzione ai concetti principali, in modo da riconoscere le classi quando le si incontra e capire come funzionino e cosa dobbiamo aspettarci dal loro utilizzo

Allontaniamoci ora dai metodi *speciali* e definiamo un metodo tutto nostro. Vogliamo un metodo che aumenti l'età delle nostre persone di un anno, e lo chiameremo *happy_birthday()*:

[18]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return self.name + ' ha ' + str(self.age) + ' anni'

    def happy_birthday(self):
        print('È il compleanno di ' + self.name + '!')
        self.age += 1
        print('Ora ha ' + str(self.age) + ' anni')

p1 = Person('Lorenzo', 27)
p2 = Person('Giorgia', 26)
p3 = Person('Bruno', 52)
print(p1)
p1.happy_birthday()
```

```
Lorenzo ha 27 anni
È il compleanno di Lorenzo!
Ora ha 28 anni
```

Qui vediamo il primo esempio di **incapsulamento**: abbiamo definito un metodo *happy_birthday()* che agisce modificando uno degli attributi del nostro oggetto della classe Person, l'età. Utilizzando questo metodo abbiamo fornito all'utente un'interfaccia che gli consente di modificare quell'attributo, ma solamente nel modo in cui abbiamo stabilito. L'utente non può infatti (per ora, almeno) aggiungere 3 all'età, o sottrarre 159, o dividerla per 2. Può soltanto eseguire le operazioni che noi abbiamo previsto nella definizione della classe. Quindi in un certo senso, gli attributi sono *protetti* dalla classe, e questo è ciò che viene chiamato *data encapsulation*.

> **NOTA**: quando definiamo un metodo su una classe, anche se questo non prende argomenti in input, è sempre necessario specificare l'argomento *self*.

Passiamo ora al concetto di ereditarietà definendo una nuova classe, che chiameremo **Impiegato**.

Se siamo tutti d'accordo sul fatto che gli impiegati sono anche persone, allora forse possiamo utilizzare molte delle cose che sono già definite nella classe persona per creare una classe impiegato.

Questo è possibile, se definiamo la classe impiegato come classe *figlia* della classe persona. Phyton ci consente di fare questo con una sintassi abbastanza banale:

```
class Figlia(classe_genitore):     ........     ........     .......
```

Ma cosa significa che Impiegato sarà la classe figlia di Person? Significa che erediterà tutti i metodi e gli attributi della classe Person. Ovviamente possiamo aggiugnere alla classe figlia altri metodi, che saranno specifici per essa (come ad esempio un metodo per calcolare la paga di un impiegato) e possiamo inoltre **sovrascrivere** alcuni metodi che in impiegato dovranno comportarsi diversamente. Vediamo un esempio:

[9]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return self.name + ' ha ' + str(self.age) + ' anni'

    def happy_birthday(self):
        print('È il compleanno di ' + self.name + '!')
        self.age += 1
        print('Ora ha ' + str(self.age) + ' anni')


class Impiegato(Person):
    def __init__(self, name, age, id_no):
        self.name = name
        self.age = age
        self.id_no = id_no

    def __str__(self):
        return 'Impiegato No. ' + self.id_no + ' - ' + self.name

imp1 = Impiegato('Lorenzo', 27, '12345')
imp2 = Impiegato('Giorgia', 19, '23134')

print(imp1)
imp1.happy_birthday()
```

```
Impiegato No. 12345 - Lorenzo
È il compleanno di Lorenzo!
Ora ha 28 anni
```

Ricapitoliamo velocemente quello che abbiamo fatto con il codice qui sopra: - Abbiamo definito una classe *Person*, con i suoi metodi *init*, *str*, ed *happy_birthday*; - Abbiamo definito una nuova classe *Impiegato*, figlia della classe *Person*; - La classe *Impiegato* ha ereditato tutti i metodi della

classe *Person*; - Abbiamo sovrascritto i metodi *init* (per aggiungere un *id*) e il metodo *str* (perché vogliamo informazioni diverse quando stampiamo l'oggetto) - Non abbiamo ridefinito il metodo *happy_birthday*, tuttavia questo è ancora presente perché è stato ereditato dalla classe *Person*. Ovviamente questo è solo un esempio, e chiaramente non è molto utile ricorrere all'ereditarietà delle classi solo per conservare un attributo. Tuttavia in situazioni più complesse questa *tecnica* è largamente utilizzata e consente una buona organizzazione del codice.

Vediamo ora un semplice esempio che introduce l'ultimo concetto fondamentale in OOP, il **polimorfismo**:

[12]:
```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def info(self):
        print('{} is a {} cat'.format(self.name, self.color))

    def make_sound(self):
        print('Meow')


class Dog:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def info(self):
        print('{} is a {} dog'.format(self.name, self.color))

    def make_sound(self):
        print('Woof')

cat1 = Cat('Leo', 'Black')
dog1 = Dog('Kira', 'Brown')

cat1.info()
cat1.make_sound()
dog1.info()
dog1.make_sound()
```

```
Leo is a Black cat
Meow
Kira is a Brown dog
Woof
```

Abbiamo definito due classi, *Cat* e *Dog*. Ciascuna di queste classi ha un suo metodo *init*, un suo metodo *info*, ed un suo metodo *make_sound*.

Il **polimorfismo** sta nel fatto che le due classi hanno metodi con lo stesso nome (i.e. la stessa

interfaccia) ma questi metodi si comportano in modo diverso a seconda della classe in cui sono definiti.

Per finire questa breve introduzione alla programmazione orientata agli oggetti, vediamo velocemente come le classi possono essere utilizzate per la costruzione di **strutture dati**. Nel prossimo esempio vedremo come costruire una struttura dati **pila**, ossia una struttura FILO (First in - Last out), definendo una classe che si comporti come ci si aspetterebbe da una pila:

[15]:
```
class MyStack():
    def __init__(self):
        self.stack = []

    def __str__(self):
        return str(self.stack)

    def get_len(self):
        return len(self.stack)

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if self.get_len() > 0:
            self.stack.pop()
        else:
            print('WARNING: Stack is empty')

    def flush(self):
        self.stack.clear()

stack1 = MyStack()
stack1.push(1)
stack1.push(2)
print(stack1.get_len())
print(stack1)
stack1.pop()
stack1.pop()
stack1.pop()
print(stack1)
for i in range(5):
    stack1.push(i)
print(stack1, stack1.get_len())
stack1.flush()
print(stack1)
```

```
2
[1, 2]
WARNING: Stack is empty
[]
```

```
[0, 1, 2, 3, 4] 5
[]
```

Ricapitolando:

- Abbiamo definito una classe pila (MyStack);
- Il suo unico attributo sarà una lista, e la inizializziamo vuota con *init*;
- Se stampiamo un oggetto pila, vedremo la lista (metodo *str*);
- Abbiamo definito un metodo per sapere quanti elementi contiene la nostra pila (*get_len*);
- Abbiamo definito un metodo per aggiungere elementi (*push*);
- Abbiamo definito un metodo per rimuovere elementi, che utilizza il metodo *get_len* per sapere se la pila è vuota (*pop*)
- Abbiamo definito un metodo che svuota la pila (*flush*)

1.1 Esercizio:

Definire una classe *MyQueue* che si comporti come la struttura dati coda. La coda è una struttura dati FIFO, ossia First in - First Out. Il primo elemento aggiunto sarà anche il primo ad essere rimosso. In particolare la classe dovrà rispettare queste caratteristiche:

1. Mostrare la coda quando vengono stampate le istanze (gli oggetti) della classe;
2. Un metodo *enqueue*, che aggiunge oggetti in cima alla coda;
3. Un metodo *dequeue*, che rimuove gli oggetti in cima alla coda;
4. Un metodo *flush* che ripulisce la coda.

   NOTA: Data l'estrema similarità con la classe MyStack, per quanto possibile evitare di cercare suggerimenti nel box di codice qui sopra. Sbatteteci un po' la testa!

..

..

..

..

..

..

..

..

..

..

..

..

1.2 Soluzione:

[18]:
```
class MyQueue:
    def __init__(self):
        self.queue = []

    def __str__(self):
        return str(self.queue)

    def get_len(self):
        return len(self.queue)

    def enqueue(self, elemento):
        self.queue.append(elemento)

    def dequeue(self):
        if self.get_len() > 0:
            self.queue.pop(0)
        else:
            print('WARNING: Queue is empty')

    def flush(self):
        self.queue.clear()

queue1 = MyQueue()
queue1.enqueue(1)
queue1.enqueue(2)
print(queue1.get_len())
print(queue1)
queue1.dequeue()
queue1.dequeue()
queue1.dequeue()
print(queue1)
for i in range(5):
    queue1.enqueue(i)
print(queue1, queue1.get_len())
queue1.flush()
print(queue1)
```

```
2
[1, 2]
WARNING: Queue is empty
[]
[0, 1, 2, 3, 4] 5
[]
```