8-Classi

November 28, 2020

1 Programmazione orientata agli oggetti e Classi

La **programmazione orientata agli oggetti** o object-oriented programming (OOP) in inglese è un paradigma di programmazione basato sulla creazione di software oggetti in grado di comunicare ed interagire tra di loro. La definizione del concetto di classe è l'esempio più comune presente in tali linguaggi, la classe si basa sulla definzione di una interfaccia attraverso cui è possibile interagire con diversi tipi di dati.

Per poter essere definito ad oggetti il linguaggio deve possedere 3 meccanismi: - incapsulamento: separazione tra interfaccia e implementazione - ereditarietà: possibiltà di definire nuovi oggetti a partire da quelli già definiti - polimorfismo: la presenza di un interfaccia comune a diversi tipi di oggetti con comportamenti diversi

Python essendo un linguaggio OOP è esattamente basato su questi meccanismi e sul concetto di classe.

1.1 Classi

Dalla definizione precedente sappiamo che **una classe è un oggetto, in python in particolare ogni cosa è un oggetto!** Una classe però oltre alla possibilità di essere vuota deve contenere degli elementi che possono essere classificati in: - attributi: analoghi alle variabili ed appartenenti solo alla classe - metodi: procedure o funzioni che possono essere applicate sugli attributi

In python per poter inizializzare una classe è necessario usare la parola chiave `class` seguita dal nome della classe, poiché una classe deve essere inizializzata qualora venga invocata è possibile **inizziarla definendo un metodo chiamato costruttore e definito con `__init__`** dove vengono inzializzati gli attributi, la presenza dei caratteri `__` ad inizio e fine del nome contrassegnano il **metodo come magico(magic method in inglese) e permettono di ritornare un nuovo oggetto definito**.Il comando per associare i metodi e gli attributi ad una classe è possibile usare il comando `self`.

[ ]:
```
#definisco una classe usando il comando class e associandole un nome
class Persona: # notare bene che inizio con la lettera maiuscola per convenzione
    #associo alla classe un metodo il costruttore come metodo
    #definedolo esattamente come una funzione con la differenza
    #che per argomenti è necessario sia presente il comando self
    def __init__(self): #ricordatevi di mettere self!
        '''Inizializza la classe persona con attributi
           nome e cognome settati a None'''
        #per creare un attributo è possibile
```

```
        #usare il comando self.nome_attributo
        self.nome = None
        self.cognome = None
```

Abbiamo quindi creato una classe di nome Persona ed creato un costruttore per poter crearla ora bastera invocarla come se fosse una funzione ed associarli ad essa una variabile in modo da poter essere in memoria.

[2]:
```
#umano ora sarà associato a persona
umano = Persona()
print(f"attributo di umano, nome:{umano.nome}, cognome:{umano.cognome}")
#posso modificare ora gli attributi usando il comando umano.attrinuto = valore
umano.nome = "Matteo"
umano.cognome = "Conterno"
print(f"attributo di umano, nome:{umano.nome}, cognome:{umano.cognome}")
```

Come è possibile notare dopo la definizione della classe è possibile modificare gli attributi di essa, la motivazione è dovuta al fatto che l'attributo definito nella classe è **pubblico, ovvero può essere accessibile e modificabile al di fuori della classe**, per rendere questo attributo **privato, ovvero non accessibile o modificabile al di fuori della classe** è possibile porre __ solo prima dell'attributo.

[3]:
```
class Persona_Private:
    def __init__(self): #ricordatevi di mettere self!
        '''Inizializza la classe persona con attributi
           nome e cognome settati a None'''
        #per creare un attributo è possibile
        #usare il comando self.nome_attributo
        self.__nome = None
        self.__cognome = None
```

Attenzione che a differenza di altri linguaggi in python è possibile modificare gli attributi p

[4]:
```
umano = Persona_Private()
#questo darà errore dicendo che non trova l'attributo
umano.nome
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-000bc3ab679f> in <module>
      1 umano = Persona_Private()
      2 #questo darà errore dicendo che non trova l'attributo
----> 3 umano.nome

AttributeError: 'Persona_Private' object has no attribute 'nome'
```

La creazione di altri metodi oltre al costruttore è abbastanza semplice e consiste nella creazione di funzioni all'interno della classe, qualora utilizzaste attributi privati sarebbe consigliabile definire

dei metodi chiamati **getter e setter** ovvero metodi attraverso cui ritornare li attributi privati e risettarli.

[5]:
```
#ridefiniamo la classe Persona in maniera più completa
class Persona:
    def __init__(self, nome = "Matteo", cognome = "Conterno"):
        '''Costruttore della classe persona con argomenti di default
           nome = Matteo, cognome = Conterno'''
        #definisco li attributi in maniera privata in modo
        #che al di fuori della classe sia impossibile
        #modifcarli a meno che si usa un metodo della classe
        self.__nome = nome
        self.__cognome = cognome

    def saluta(self):
        '''Stampa un saluto con nome e cognome, non ritorna nulla'''
        print(f"Salve, io sono {self.__nome} {self.__cognome}")

    def get_name(self):
        '''Ritorna il nome e il cognome '''
        return self.__nome, self.__cognome

    def set_name(self, nome, cognome):
        '''Setta le variabili nome e cognome secondo i nuovi valori,
           sono necessari nuovi valori e non sono presenti di default'''
        self.__nome = nome
        self.__cognome = cognome
```

[6]:
```
umano = Persona()
#invoca il metodo della classe
umano.saluta()
nome, cognome = umano.get_name()
print(f"Valori ottenuti {nome}, {cognome}")
umano.set_name(nome = "Flavio", cognome = "Pirazzi")
umano.saluta()
```

```
Salve, io sono Matteo Conterno
Valori ottenuti Matteo, Conterno
Salve, io sono Flavio Pirazzi
```

Nel caso in cui sia necessario che tutte le classi presentino attributi uguali in ognuna delle loro definizione è possibile associare una variabile di classe che sarà condivisa da ognuna di esse.

[7]:
```
class Persona:
    #variabile di classe, non attributo
    genere = "umano"
    def __init__(self, nome = "Matteo", cognome = "Conterno"):
        '''Costruttore della classe persona con argomenti di default
```

```
            nome = Matteo, cognome = Conterno'''
        self.__nome = nome
        self.__cognome = cognome
```

[8]:
```
umano = Persona()
qualcuno = Persona()
#stampo ala variabile di classe
print(f"variabile di classe di umano è {umano.genere}")
print(f"variabile di classe di qualcuno è {qualcuno.genere}")
```

```
variabile di classe di umano è umano
variabile di classe di qualcuno è umano
```

La variabile di classe può anche essere definita come privata usando i comandi mostrati in precedenza.

1.1.1 Ereditarietà delle classi

L'ereditarietà come abbiamo definito prima è la possibiltà di definire nuove classi a partire da altre già definite, questa è un aspetto molto potente che permette di definire nuove funzionalità ad altre già preesistenti, vediamo un esempio per essere chiari.

[9]:
```
class Calcolatrice:
    def __init__(self):
        '''Definisci un attributo detto risultato a 0'''
        #lo definisco privato così è modificabile solo dalla classe
        self.risultato = 0

    #sarebbe possibile definire un metodo magico che prenda
    #oggetti per restituire un oggetto, ma per semplicità non lo faccio
    def somma(self, x = 0, y = 0):
        '''Calcola la somma di due numeri e somma al risultato'''
        #aggiungiamo la somma al risultato
        self.risultato += x + y
        print(f'La somma di {x} e {y} è {x + y}')
        print(f'risultato delle operazioni fino ad ora {self.risultato}')

    def prodotto(self, x = 1, y = 1):
        '''Calcola il prodotto di due numeri e lo moltiplica al risultato'''
        self.risultato *= (x * y)
        print(f'Il prodotto di {x} e {y} è {x * y}')
        print(f'risultato delle operazioni fino ad ora {self.risultato}')

    def differenza(self, x = 0, y = 0):
        '''Calcola la differenza di due numeri e somma al risultato'''
        self.risultato += (x - y)
        print(f'La differenza di {x} e {y} è {x - y}')
        print(f'risultato delle operazioni fino ad ora {self.risultato}')
```

```
    def divisione(self, x = 1, y = 1):# 1 per evitare problemi di␣
↪ZeroDivisionError
        '''Calcola la divisione tra due numeri e lo divide al risultato'''
        self.risultato /= (x / y)
        print(f'La differenza di {x} e {y} è {x / y}')
        print(f'risultato delle operazioni fino ad ora {self.risultato}')

    def ottieni_risultato(self):
        '''ritorna il risultato delle operazioni'''
        return self.risultato

    def stampa_risultato(self):
        '''Stampa il risultato delle operazioni'''
        print(f'il risultato è {self.risultato}')

    def azzera(self):
        '''azzera il risultato delle operazioni fino ad ora calcolato'''
        self.risultato = 0
        print('risultato azzerato')
```

Proviamo ora a usare la classe calcolatrice, vediamo come funziona.

[10]:
```
calc = Calcolatrice()
calc.somma(x = 1, y = 4)
calc.stampa_risultato()
calc.prodotto( x = 3, y = 5)
calc.differenza(x = 5, y = 10)
calc.azzera()
calc.stampa_risultato()
```

```
La somma di 1 e 4 è 5
risultato delle operazioni fino ad ora 5
il risultato è 5
Il prodotto di 3 e 5 è 15
risultato delle operazioni fino ad ora 75
La differenza di 5 e 10 è -5
risultato delle operazioni fino ad ora 70
risultato azzerato
il risultato è 0
```

Abbiamo quindi creato una classe calcolatrice in grado di effettuare le basiche operazioni, se volessimo aggiungere delle funzionalità alla classe avremmo due possibili strade da seguire:

- modificare la classe originale aggiungendo funzionalità
- creare una nuova classe basandoci su quella già creata aggiungendo funzionalità

Vediamo quindi la seconda strada che corrisponde all'ereditarietà appena mostrata. Per poter ereditare i comportamenti di una classe usando quelli già presenti in una basta mettere tra parentesi

il nome della classe da ereditare subito dopo la dichiarazione della classe, il numero delle classi che sono possibili ereditare non è limitato al singolo in python e qualora sia necessario accedere ad metodi o attributi presenti nelle classi ereditate è possibile usare il comando `super`. Per maggiori informazioni consultate la documentazione di python. Le classi che sono ereditate vengono chiamte **Parent**, mentre le classi che ereditano altre classi sono chiamate **Child**.

`Nel caso di ereditarietà multipla l'accesso ai metodi avviene partendo dalle classi prima dich`

[11]:
```
#metto tra parentesi la classe da cui ereditare le funzionalità
class Scientifica(Calcolatrice):
    def __init__(self):
        '''Inizializzo calcolatrice scientifica settando risultato a zero'''
        #eredito l'attributo risultato della classe originale
        #nel classo di ereditarietà di classe multiple
        #è possibile accedere usando nome_classe.attributo o metodo
        #usiamo il costruttore della calcolatrice mettendo la classe
        #ereditata e l'oggetto da cui ereditare
        super().__init__()
        #alternativamente Calcolatrice.__init__(self)
        # o super(Scientifica, self).__init__()

    def radice_quadrata(self, x = 0):
        '''Calcola la radice quadrata positiva del valore x e la somma al␣
 ↪risultato '''
        self.risultato += x**(0.5)
        print(f'la radice quadrata di {x} è {x**(0.5)}')
        print(f'il risultato è {self.risultato}')

    def potenza(self, x = 0, y = 1):
        '''Calcola x elevato alla y e somma il risultato'''
        self.risultato += x**y
        print(f'{x} elevato alla  {y} è {x**y}')
        print(f'il risultato è {self.risultato}')
```

[12]:
```
scient = Scientifica()
#proviamo a richiamare un metodo della classe ereditata
scient.stampa_risultato()
scient.somma(x = 4, y = 5)
scient.prodotto(x = 6, y = 7)
scient.azzera()
```

```
il risultato è 0
La somma di 4 e 5 è 9
risultato delle operazioni fino ad ora 9
Il prodotto di 6 e 7 è 42
risultato delle operazioni fino ad ora 378
risultato azzerato
```

[13]:
```
#proviamo a usare un metodo della classe nuova
scient.radice_quadrata(x = 4)
scient.potenza(x = 3, y = 4)
scient.azzera()
```

```
la radice quadrata di 4 è 2.0
il risultato è 2.0
3 elevato alla  4 è 81
il risultato è 83.0
risultato azzerato
```

Gli attributi o i metodi privati di una classe non vengono ereditati.

[14]:
```
#proviamo ad usare i metodi di scientifica usando calcolatrice
calc = Calcolatrice()
calc.radice_quadrata()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-42c8935ed1fa> in <module>
      1 #proviamo ad usare i metodi di scientifica usando calcolatrice
      2 calc = Calcolatrice()
----> 3 calc.radice_quadrata()

AttributeError: 'Calcolatrice' object has no attribute 'radice_quadrata'
```

Come possiamo notare l'ereditarietà è in un'unica direzione ovvero le **classi Child ereditano i comportamenti dalle classi Parent, ma le Parent non possiedono le nuove funzionalità dei Child**.

1.1.2 Iteratori

Qualora noi usiamo un loop for quello che accade in python è la creazione di un iteratore ovvero di un oggetto in grado di accedere gli elementi di un altro oggetto in maniera sequenziale, in python è possibile usare il comando `iter` per crearne uno e usare il comando `next` per andare all'elemento successivo.

[15]:
```
saluto = "Ciao"
#creiamo un iteratore
it = iter(saluto)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
C
i
```

```
a
o
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-15-d6a82b121a34> in <module>
      6 print(next(it))
      7 print(next(it))
----> 8 print(next(it))

StopIteration:
```

Come possiamo notare ogni elemento viene iterato al successivo fino all'ultimo dove viene invoca un'eccezione di nome StopIterator poiché non ci sono più elementi da iterare.Qualora volessimo usare un iteratore all'interno della classe è possibile definire un magico `__iter__` e `__next__`.

[16]:
```python
class Persona:
    def __init__(self, nome ='Matteo'):
        '''Inizializza persona con nome = Matteo'''
        self.nome = nome
        self.indice = -1

    def __iter__(self):
        '''Crea un iteratore'''
        return self

    def __next__(self):
        '''Permette di accedere agli elementi successivi'''
        if self.indice == (len(self.nome) - 1):
            raise StopIteration
        else:
            self.indice += 1
            return self.nome[self.indice]
```

[17]:
```python
p = Persona()
for char in p:
    print(char)
```

```
M
a
t
t
e
o
```

Quello che succede nel loop for è che viene invocato il metodo iter e next per iterare all'interno del attributo nome della classe Persona e funziona per qualsiasi lunghezza.

p = Persona(nome = ‘Ermenegildo’) for char in p: print(char)

1.1.3 Generatori

Creare però questi iteraori in molti casi può essere tedioso o complicato ed è per questo motivo che python usa i **generatori** in tali circostanze. In python la creazione di un generatore avviene attraverso il comando `yield`che permette di riotornare l'elemento corrente e tenere memoria del suo indice per andare al successivo, l'unica condizione sul suo utilizzo è che sia all'interno di una funzione.

[18]:
```
def contrario(stringa):
    '''Ritorna gli elementi in ordine inverso'''
    for indice in range(len(stringa)-1, -1, -1):
        yield stringa[indice]

nome = "Matteo"
for char in contrario(nome):
    print(char)
```

```
o
e
t
t
a
M
```

I generatori possono essere creati anche usando delle espressioni tra parentesi simili alle comprensioni di liste con la differenza che devono essere contenute all'interno di parentesi tonde.

[19]:
```
#creiamo un generatore per calcolare la somma di 3n con n da 0 a 9
tot = sum(n for n in range(10))
print(f'La somma di 3n con n da 0 a 9 è {tot}')
```

```
La somma di 3n con n da 0 a 9 è 45
```

COMPLIMENTI AVETE CONCLUSO LA LEZIONE SULLE CLASSI E IL CAPITOLO PYTHON BASE!