Object Oriented Programming

Prof. Ing. Loris Penserini, PhD

https://orcid.org/0009-0008-6157-0396

POLIMORFISMO e COSTRUTTORI MULTIPLI

Caratteristiche

Il polimorfismo è una delle caratteristiche più potenti della programmazione orientata agli oggetti:

- permette di usare un'unica interfaccia per oggetti di classi diverse, che tuttavia possono comportarsi in modo differente.
- In pratica, più classi possono condividere lo stesso metodo (cioè lo stesso nome), ma implementarlo secondo la propria logica interna.
- Questa flessibilità è utile per istanziare oggetti con stati diversi (cioè con attributi o valori iniziali differenti), ad esempio grazie a costruttori con parametri diversi, pur mantenendo la stessa struttura generale del codice.

Utilizzi

Assieme alle proprietà di Ereditarietà e all'Incapsulamento facilita il programmatore a seguire la filosofia dell'ingegneria del software dell'usabilità e della modularità delle applicazioni.

In sintesi, nella OOP, il Polimorfismo è la proprietà di una classe di poter generare diverse specializzazioni, semplicemente fornendo alle classi figlie la possibilità di svolgere:

Overriding: **riscrivere uno stesso metodo della classe padre.**

Overloading: **invocare uno stesso metodo con tipologia e parametri diversi.**

Il polimorfismo rende possibile trattare oggetti diversi come se appartenessero alla stessa famiglia di classi, pur permettendo loro di reagire in modo personalizzato. Questa capacità di avere oggetti con lo stesso nome di metodo ma con comportamenti diversi e stati propri è ciò che rende la OOP così potente, estendibile e facile da mantenere nel tempo.

Overriding

L'overriding consiste nel ridefinire, in una classe derivata, un metodo già presente nella classe base. In altre parole, **il method overriding si verifica quando la classe derivata ridefinisce un metodo della classe base con lo stesso nome e stessa firma, ma con un comportamento diverso.**

In questo modo, l'oggetto figlio mantiene la stessa interfaccia ma modifica il comportamento, adattandolo al proprio contesto.

Esempio: un metodo **parla()** può stampare "Ciao" per un oggetto **Persona**, ma "Miagolo" per un oggetto **Gatto**.

L'overriding rappresenta la forma più comune di polimorfismo in OOP.

Esempio «persona.py»: Overriding

Le variabili sono «private»

I metodi «get...» si usano per accedere alle variabili interne «private», per cui sono solo in lettura.

OOP > oop_pw4 > persona.py > Persona

```python
class Persona:
    def __init__(self, nome, cognome, eta, interessi):
        self.__nome = nome
        self.__cognome = cognome
        self.__eta = eta
        self.__interessi = interessi
        self.saluto = f"Buongiorno, sono {self.__nome} {self.__cognome}"

    def get_pag_benvenuto(self):
        return self.saluto

    def getNome(self):
        return self.__nome

    def getCognome(self):
        return self.__cognome

    def getEta(self):
        return self.__eta

    def getInteressi(self):
        return self.__interessi
```

Esempio «persona.py»: Overriding

OOP > oop_pw4 > persona.py > ...

```python
class Persona:
    def __init__(self, nome, cognome, eta, interessi):
        self.__nome = nome
        self.__cognome = cognome
        self.__eta = eta
        self.__interessi = interessi
        self.saluto = f"Buongiorno, sono {nome} {cognome}"

    def get_pag_benvenuto(self):
        return self.saluto

```

Esempio «studente.py»: Overriding

OOP > oop_pw4 > studente.py > ...

```python
from persona import Persona

class Studente(Persona):
    def __init__(self, nome, cognome, eta, interessi, ind_studio):
        super().__init__(nome, cognome, eta, interessi)
        self.ind_studio = ind_studio

    # OVERRIDING: ridefinizione del metodo del genitore
    def get_pag_benvenuto(self):
        # richiama comunque il metodo del genitore (opzionale)
        saluto_base = super().get_pag_benvenuto()
        return f"{saluto_base}, e frequento {self.ind_studio}"

```

Esempio «main.py»: Overriding

OOP > oop_pw4 > main.py > ...

```python
from persona import Persona
from studente import Studente

Persona1 = Persona("Luca", "Rossi", 40, "lettura")
Studente1 = Studente("Giulia", "Bianchi", 22, "musica", "Ingegneria Informatica")

print(Persona1.get_pag_benvenuto())
print(Studente1.get_pag_benvenuto())   # chiamerà la versione ridefinita

```

output

Obj: Persona1 → Buongiorno, sono Luca Rossi

Obj: Studente1 → Buongiorno, sono Giulia Bianchi, e frequento Ingegneria Informatica

Overloading

L'overloading si riferisce alla possibilità di definire più versioni di uno stesso metodo con parametri diversi, ovvero «**firme diverse**», così da adattarne il comportamento a seconda dei dati in ingresso.

- Alcuni linguaggi (come Java o C++) lo supportano in modo diretto;
- in Python non esiste formalmente, ma è possibile simularlo tramite valori di default, ***args** o **@singledispatch**, permettendo comunque una forma di polimorfismo basato sui parametri.

Esempio «persona.py»: Overloading

OOP > oop_pw4 > persona.py > Persona

```python
class Persona:
    def __init__(self, nome, cognome, eta, interessi):
        self.__nome = nome
        self.__cognome = cognome
        self.__eta = eta
        self.__interessi = interessi
        self.saluto = f"Buongiorno, sono {self.__nome} {self.__cognome}"

    def get_pag_benvenuto(self):
        return self.saluto

    def getNome(self):
        return self.__nome

    def getCognome(self):
        return self.__cognome

    def getEta(self):
        return self.__eta

    def getInteressi(self):
        return self.__interessi
```

Esempio «studente.py»: Overloading

OOP > oop_pw5 > studente.py > Studente > __init__

```python
from persona import Persona

class Studente(Persona):
    # Simulazione di OVERLOADING del costruttore
    def __init__(self, nome="", cognome="", eta=None, interessi=None, ind_studio=None):
        # costruttore “flessibile”: accetta diversi tipi di input
        if nome and cognome and eta and interessi:
            super().__init__(nome, cognome, eta, interessi)
        else:
            super().__init__("Sconosciuto", "", 0, "nessuno")
        self.ind_studio = ind_studio or "corso non specificato"

    def get_pag_benvenuto(self):
        return f"Buongiorno, sono {self.getNome()} {self.getCognome()}, e frequento {self.ind_studio}"

```

Soluzione con «wrapper»

Perché nell'else, riga n.10, ci sono solo 4 parametri: richiama il costruttore della classe madre (Persona), non quello di Studente.

Il quinto parametro serve solo nella classe Studente, e infatti si gestisce subito dopo, alla riga n.11, che inizializza l'attributo specifico della sottoclasse, indipendente dalla logica del genitore.

Osservando il main.py:

- In **Studente1**, tutti e 5 i parametri sono forniti, quindi entra nel **ramo if** e chiama **super().__init__()** con i 4 parametri effettivi del genitore.
- In **Studente2**, mancano nome/cognome/ecc., quindi entra nel **ramo else** e chiama **super().__init__("Sconosciuto", " ", 0, "nessuno").**
- Infine, imposta **ind_studio** di default.

Esempio «main.py»: Overloading

```
OOP > oop_pw5 > main.py > ...
1   from persona import Persona
2   from studente import Studente
3
4   Persona1 = Persona("Luca", "Rossi", 40, "lettura")
5   Studente1 = Studente("Giulia", "Bianchi", 22, "musica", "Ingegneria Informatica")
6   Studente2 = Studente(ind_studio="Matematica")
7
8   print(Persona1.get_pag_benvenuto())
9   print(Studente1.get_pag_benvenuto())    # chiamerà la versione ridefinita (IF)
10  print(Studente2.get_pag_benvenuto())    # chiamerà la versione ridefinita (ELSE)
11
```

Obj: Persona1

Obj: Studente1

Obj: Studente2

output

```
Buongiorno, sono Luca Rossi
Buongiorno, sono Giulia Bianchi, e frequento Ingegneria Informatica
Buongiorno, sono Sconosciuto , e frequento Matematica
```

@singledispatcher

Il decoratore @singledispatch è uno strumento molto elegante di Python per simulare l'overloading dei metodi o delle funzioni (cioè lo stesso nome, ma comportamenti diversi a seconda del tipo di parametro).

@singledispatch è un decoratore del modulo functools che permette di definire una funzione "generica", cioè una funzione che cambia comportamento in base al tipo dell'argomento che riceve.

È come una **forma di overloading basato sui tipi**.

Esempio «persona.py»: @dispatcher

OOP > oop_pw6 > persona.py > Persona > _

```python
from functools import singledispatchmethod

class Persona:
    # wrapper "pubblico": accetta anche nessun argomento
    def saluta(self, arg=None):
        return self._saluta(arg)   # passa SEMPRE un argomento al dispatcher

    # dispatcher "privato"
    @singledispatchmethod
    def _saluta(self, arg):
        print("Ciao, piacere di conoscerti!")  # fallback

    @_saluta.register(type(None))
    def _(self, _):
        print("Ciao, piacere di conoscerti!")

    @_saluta.register(str)
    def _(self, nome):
        print(f"Ciao {nome}, benvenuto!")

    @_saluta.register(int)
    def _(self, eta):
        print(f"Hai {eta} anni? Piacere di conoscerti!")
```

Esempio di «main.py» con @dispatcher

OOP > oop_pw6 > main.py > ...

```python
from persona import Persona

p = Persona()
p.saluta()          # funziona: wrapper passa None al dispatcher
p.saluta("Luca")    # str
p.saluta(30)        # int
```

p.saluta()

p.saluta("Luca")

p.saluta(30)

output

```
Ciao, piacere di conoscerti!
Ciao Luca, benvenuto!
Hai 30 anni? Piacere di conoscerti!
```

GRAZIE!