Classi e oggetti (OOP)

Cos'è la programmazione a oggetti?

Fino ad ora abbiamo usato variabili separate per rappresentare le cose. Ma nel mondo reale, le "cose" hanno **proprietà** e **comportamenti** che stanno insieme. Un'auto ha colore, marca, velocità (proprietà) e può accelerare, frenare, sterzare (comportamenti). Sarebbe un casino gestire tutto con variabili separate!

La **Programmazione Orientata agli Oggetti** (OOP) ci permette di creare i nostri **tipi di dato personalizzati** che raggruppano dati e funzioni in un unico pacchetto. Tipo costruire i tuoi LEGO personalizzati invece di usare solo i mattoncini base!

La prima classe

Una **classe** è un "modello" (o stampo) per creare oggetti. Un **oggetto** è un'istanza concreta di quella classe. Tipo: la classe `Studente` è lo stampo, e tu sei un oggetto creato da quello stampo!

```
class Studente:
    def __init__(self, nome, eta, classe):
        self.nome = nome
        self.eta = eta
        self.classe = classe

# Creare oggetti (istanze) — usiamo lo stampo!
s1 = Studente("Mario", 16, "3A")
s2 = Studente("Luigi", 17, "4B")

print(f"{s1.nome} ha {s1.eta} anni, classe {s1.classe}")
print(f"{s2.nome} ha {s2.eta} anni, classe {s2.classe}")
```

Spiegazione (non saltarla!)

- `class Studente:` — definisce una nuova classe chiamata `Studente`

- `__init__` — è il **costruttore**: viene chiamato automaticamente quando crei un nuovo oggetto. I doppi underscore si chiamano "dunder" (double underscore)
- `self` — è un riferimento all'oggetto stesso (tipo "io" in italiano). È obbligatorio come primo parametro di ogni metodo
- `self.nome`, `self.eta` ecc. — sono gli **attributi** dell'oggetto, le sue caratteristiche

---

Metodi

I **metodi** sono funzioni che appartengono a una classe e descrivono i **comportamenti** dell'oggetto. Tipo le azioni che può fare:

```
class Studente:
    def __init__(self, nome, voti):
        self.nome = nome
        self.voti = voti

    def media(self):
        return sum(self.voti) / len(self.voti)

    def promosso(self):
        return self.media() >= 6

    def aggiungi_voto(self, voto):
        self.voti.append(voto)

    def presentati(self):
        stato = "promosso" if self.promosso() else "bocciato"
        print(f"Sono {self.nome}, media {self.media():.2f} ({stato})")

# Uso
s = Studente("Mario", [7, 8, 6, 9])
s.presentati()

s.aggiungi_voto(5)
s.presentati()
```

Attributi vs metodi

Concetto	Cos'è	Esempio
Attributo	Un dato dell'oggetto (cosa HA)	`self.nome`, `self.eta`
Metodo	Un'azione dell'oggetto (cosa FA)	`self.presentati()`

Pensa a un oggetto come a un **personaggio di un videogioco**: ha delle statistiche (attributi: vita, forza, velocità) e delle azioni (metodi: attacca, difendi, corri).

Il metodo `__str__`

Il metodo speciale `__str__` definisce come l'oggetto viene mostrato quando usi `print()`. Senza di lui, Python stampa un messaggio tipo `<Punto object at 0x7f...>` che non dice niente a nessuno:

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

    def distanza_origine(self):
        return (self.x**2 + self.y**2) ** 0.5

p = Punto(3, 4)
print(p)  # Chiama automaticamente __str__ → (3, 4)
print(f"Distanza dall'origine: {p.distanza_origine():.2f}")
```

Esempio completo: Conto bancario

Un classico degli esercizi OOP! Simuliamo un conto in banca:

```
class ContoBancario:
    def __init__(self, titolare, saldo=0):
        self.titolare = titolare
        self.saldo = saldo

    def deposita(self, importo):
        if importo > 0:
            self.saldo += importo
            print(f"Depositati {importo}€. Saldo: {self.saldo}€")
        else:
            print("Importo non valido!")

    def preleva(self, importo):
        if importo > self.saldo:
            print("Fondi insufficienti!")
        elif importo <= 0:
            print("Importo non valido!")
        else:
            self.saldo -= importo
            print(f"Prelevati {importo}€. Saldo: {self.saldo}€")

    def __str__(self):
        return f"Conto di {self.titolare}: {self.saldo}€"

# Uso
conto = ContoBancario("Mario", 1000)
print(conto)

conto.deposita(500)
conto.preleva(200)
conto.preleva(2000)  # Fondi insufficienti!
print(conto)
```

Ereditarietà

L'**ereditarietà** è tipo il DNA delle classi: puoi creare una nuova classe che **eredita** attributi e metodi da una classe "genitore". Tipo: `Cane` eredita da `Animale`, perché un cane È

un animale!

```
class Animale:
    def __init__(self, nome, zampe):
        self.nome = nome
        self.zampe = zampe

    def parla(self):
        print("...")

    def __str__(self):
        return f"{self.nome} ({self.zampe} zampe)"

class Cane(Animale):  # Cane eredita da Animale
    def __init__(self, nome, razza):
        super().__init__(nome, 4)  # Chiama il costruttore del genitore
        self.razza = razza

    def parla(self):  # Sovrascrive il metodo del genitore
        print("Bau bau!")

class Gatto(Animale):
    def __init__(self, nome):
        super().__init__(nome, 4)

    def parla(self):
        print("Miao!")

class Pesce(Animale):
    def __init__(self, nome):
        super().__init__(nome, 0)

    def parla(self):
        print("Blub blub!")

# Uso — il polimorfismo in azione!
animali = [
    Cane("Fido", "Labrador"),
    Gatto("Micio"),
    Pesce("Nemo")
]
```

```
for a in animali:
    print(a, end=" → ")
    a.parla()
```

Come funziona?

- `class Cane(Animale)` — il `Cane` **eredita** tutto da `Animale`
- `super().__init__(...)` — chiama il costruttore della classe genitore (tipo dire "prima fai le cose che fanno tutti gli animali, poi le cose specifiche del cane")
- Il metodo `parla()` viene **sovrascritto** in ogni sottoclasse — questo si chiama **polimorfismo** (stessa funzione, comportamento diverso a seconda dell'oggetto)

Esempio completo: Forme geometriche

Un esempio più completo che mette insieme tutto: ereditarietà, metodi, `__str__`:

```
class Forma:
    def __init__(self, nome):
        self.nome = nome

    def area(self):
        return 0

    def perimetro(self):
        return 0

    def __str__(self):
        return f"{self.nome}: area={self.area():.2f}, perimetro={self.pe

class Rettangolo(Forma):
    def __init__(self, base, altezza):
        super().__init__("Rettangolo")
        self.base = base
        self.altezza = altezza

    def area(self):
        return self.base * self.altezza

    def perimetro(self):
        return 2 * (self.base + self.altezza)

class Cerchio(Forma):
    def __init__(self, raggio):
        super().__init__("Cerchio")
        self.raggio = raggio

    def area(self):
        return 3.14159 * self.raggio ** 2

    def perimetro(self):
        return 2 * 3.14159 * self.raggio

class Triangolo(Forma):
    def __init__(self, base, altezza, lato1, lato2, lato3):
        super().__init__("Triangolo")
```

```
        self.base = base
        self.altezza = altezza
        self.lato1 = lato1
        self.lato2 = lato2
        self.lato3 = lato3

    def area(self):
        return self.base * self.altezza / 2

    def perimetro(self):
        return self.lato1 + self.lato2 + self.lato3

# Uso — ciclo su forme diverse, stesso codice!
forme = [
    Rettangolo(5, 3),
    Cerchio(4),
    Triangolo(6, 4, 5, 5, 6)
]

for f in forme:
    print(f)
```

Esercizi

Esercizio 1: Classe Libro

Crea una classe `Libro` con attributi titolo, autore, pagine e un metodo che dice se è un libro "lungo" (oltre 300 pagine). Aggiungi anche `__str__` per stamparlo in modo carino!

```
class Libro:
    # Completa la classe
    pass

# Test
libro1 = Libro("Il Signore degli Anelli", "Tolkien", 1200)
libro2 = Libro("Il Piccolo Principe", "Saint-Exupéry", 96)

# Stampa le info e se è lungo o corto
```

Esercizio 2: Classe Dado

Crea una classe `Dado` con un metodo `lancia()` che restituisce un numero casuale da 1 a 6, e un metodo `statistiche(n)` che lancia N volte e mostra quante volte esce ogni faccia. Tipo fare un esperimento di probabilità!

```
import random

class Dado:
    # Completa la classe
    pass

# Test: lancia il dado 100 volte e mostra quante volte esce ogni numero
```