Modulo 7 Programmazione ad oggetti in Python

46. **Python Object Oriented Programming** 🚗

Obiettivi di apprendimento:

- Comprendere i concetti di **Classe** (blueprint) e **Oggetto** (istanza).
- Definire una classe usando la parola chiave `class`.
- Comprendere e usare il metodo speciale `__init__` (costruttore).
- Distinguere tra **attributi** (dati) e **metodi** (funzioni di classe).

Contenuto teorico:

La **Programmazione Orientata agli Oggetti (OOP)** è un paradigma che organizza il software attorno ai **dati** (attributi) e alle funzioni che operano su tali dati (metodi), raggruppati in **Classi**.

- **Classe:** Un modello o *blueprint* per creare oggetti.
- **Oggetto (Istanza):** Una specifica entità creata dalla classe.

```
class Automobile:
    # ATTRIBUTI DI CLASSE (condivisi da tutte le istanze)
    ruote = 4

    # METODO COSTRUTTORE: Eseguito quando viene creato un nuovo oggetto
    def __init__(self, marca, modello, colore="bianco"):
        # ATTRIBUTI DI ISTANZA (proprietà specifiche di ogni oggetto)
        self.marca = marca      # Yamaha
        self.modello = modello  # R1
        self.colore = colore    # rosso
        self.accesa = False

    # METODO: Funzione che opera sull'oggetto
    def accendi(self):
        if not self.accesa:
            self.accesa = True
            print(f"La {self.marca} {self.modello} si è accesa.")
        else:
            print(f"La {self.modello} è già accesa.")

    def descrivi(self):
        stato = "accesa" if self.accesa else "spenta"
        return f"Auto: {self.marca} {self.modello}, Colore: {self.colore},
Stato: {stato}, Ruote: {self.ruote}"

# Creazione di Oggetti (Istanze)
auto1 = Automobile("Fiat", "Panda", "verde")
auto2 = Automobile("Audi", "A4") # Usa il colore di default

# Accesso ad Attributi e Metodi
```

```
print(auto1.descrivi()) # Stampa: Auto: Fiat Panda, Colore: verde...
auto1.accendi()
print(auto1.descrivi())

print(auto2.descrivi()) # Stampa: Auto: Audi A4, Colore: bianco...
```

---

47. **Class Variables** 🎓

Obiettivi di apprendimento:

- Distinguere chiaramente tra **variabili di classe** e **variabili di istanza**.
- Comprendere quando e come accedere o modificare le variabili di classe.

Contenuto teorico:

1. **Variabili di Classe:** Dichiarate direttamente all'interno del corpo della classe (prima di `__init__`). Sono **condivise** tra tutte le istanze della classe.
2. **Variabili di Istanza:** Dichiarate all'interno di un metodo (tipicamente `__init__`) usando `self.`. Sono **uniche** per ogni oggetto creato.

```
class Universita:
    # VARIABILE DI CLASSE: Condivisa da tutti gli studenti
    nome_ateneo = "Università degli Studi di Torino"
    tassa_base_annuale = 800.00

    def __init__(self, nome, matricola, tasse_pagate=0.0):
        # VARIABILI DI ISTANZA: Uniche per ogni studente
        self.nome = nome
        self.matricola = matricola
        self.tasse_pagate = tasse_pagate

    def mostra_info(self):
        return f"Studente: {self.nome} | Ateneo: {Universita.nome_ateneo}
| Tasse dovute: €{self.tassa_base_annuale - self.tasse_pagate:,.2f}"

# Creazione di istanze
studente_a = Universita("Marco Bianchi", "S001")
studente_b = Universita("Giulia Rossi", "S002")

print("=== VARIABILI DI CLASSE ===")
print(studente_a.mostra_info())
print(studente_b.mostra_info())

# Modifica della variabile di classe tramite la CLASSE stessa
Universita.tassa_base_annuale = 900.00

print("\n--- Dopo l'aumento delle tasse ---")
# La modifica si riflette su tutte le istanze
print(studente_a.mostra_info())
```

```
# Tentativo di modifica tramite istanza (crea solo una nuova variabile
d'istanza)
studente_b.tassa_base_annuale = 500.00 # Crea una variabile D'ISTANZA (non
modifica la variabile di classe)
print(f"Tassa di Giulia (istanza): {studente_b.tassa_base_annuale}")
print(f"Tassa dell'Università (classe): {Universita.tassa_base_annuale}")
```

---

48. **Inheritance** 👨‍👦

Obiettivi di apprendimento:

- Comprendere il principio di **Ereditarietà** (is-a relationship).
- Definire una **classe figlia** che eredita attributi e metodi da una **classe padre**.
- Usare la funzione `super()` per accedere ai metodi della classe padre.

Contenuto teorico:

L'**Ereditarietà** permette di definire una nuova classe (sottoclasse o *child class*) che eredita tutti gli attributi e i metodi da una classe esistente (superclasse o *parent class*).

- **Vantaggio:** Riusabilità del codice.

```
class Veicolo: # CLASSE PADRE (Superclasse)
    def __init__(self, marca, modello, anno):
        self.marca = marca
        self.modello = modello
        self.anno = anno

    def accendi_motore(self):
        return "Motore avviato."

    def info(self):
        return f"Veicolo: {self.marca} {self.modello} ({self.anno})"

class Auto(Veicolo): # CLASSE FIGLIA (Eredita da Veicolo)
    def __init__(self, marca, modello, anno, numero_porte):
        # Chiama il costruttore della classe padre (Veicolo)
        super().__init__(marca, modello, anno)
        self.numero_porte = numero_porte # Nuovo attributo

    # Overriding: Ridefinizione di un metodo ereditato
    def info(self):
        # Usa il metodo del padre e aggiunge un dettaglio
        info_padre = super().info()
        return f"{info_padre}, Porte: {self.numero_porte}"

    def parcheggia(self): # Nuovo metodo
        return f"L'auto {self.modello} sta parcheggiando."
```

```
# Creazione di istanze
mia_auto = Auto("Fiat", "Tipo", 2020, 5)

print("=== EREDITARIETÀ ===")
# Usa il metodo ereditato (accendi_motore)
print(mia_auto.accendi_motore())

# Usa il metodo ridefinito (info)
print(mia_auto.info())
```

---

49. **Multiple Inheritance** 🐟

Obiettivi di apprendimento:

- Comprendere il concetto di **Ereditarietà Multipla** (una classe figlia eredita da più padri).
- Conoscere il problema del "diamante" e il meccanismo **MRO** (Method Resolution Order) di Python.

Contenuto teorico:

L'**Ereditarietà Multipla** si verifica quando una classe figlia eredita direttamente da due o più classi padre.

```
class Volante: # PADRE 1
    def vola(self):
        return "Posso volare."

class Nuotante: # PADRE 2
    def nuota(self):
        return "Posso nuotare."

class AnimaleFantastico(Volante, Nuotante): # FIGLIO (Eredita da Volante E
Nuotante)
    def descrivi(self):
        # Chiama metodi ereditati da diversi padri
        return f"Sono un animale ibrido: {self.vola()} e {self.nuota()}"

# Creazione di istanza
ibrido = AnimaleFantastico()

print("=== EREDITARIETÀ MULTIPLA ===")
print(ibrido.vola())
print(ibrido.nuota())
print(ibrido.descrivi())

# Method Resolution Order (MRO)
# MRO definisce l'ordine in cui Python cerca i metodi e gli attributi
# nelle classi padre in caso di ereditarietà multipla.
# La MRO garantisce che i metodi vengano cercati in modo coerente (Depth-
First Left-to-Right).
print("\nOrdine di Risoluzione dei Metodi (MRO):")
print(AnimaleFantastico.mro())
# Output MRO: [AnimaleFantastico, Volante, Nuotante, object]
```

---

50. **super()** 🔴

Obiettivi di apprendimento:

- Comprendere il ruolo della funzione `super()` nell'ereditarietà.
- Utilizzare `super()` per accedere e chiamare metodi della classe padre (superclasse).
- Applicare `super()` nel costruttore `__init__` per inizializzare gli attributi ereditati.

Contenuto teorico:

La funzione `super()` è fondamentale nell'ereditarietà. Permette di accedere direttamente ai metodi e agli attributi della **classe padre** o della superclasse in generale, rispettando l'Ordine di Risoluzione dei Metodi (**MRO**).

L'uso più comune è all'interno del costruttore `__init__` della classe figlia per assicurarsi che tutti gli attributi ereditati dal genitore vengano correttamente inizializzati.

```
class Veicolo:
    def __init__(self, marca, modello):
        self.marca = marca
        self.modello = modello

    def muovi(self):
        return f"{self.marca} {self.modello} si sta muovendo."

class Auto(Veicolo):
    def __init__(self, marca, modello, porte):
        # 1. Chiama il costruttore della superclasse (Veicolo) per
inizializzare
        #    'marca' e 'modello'. Questo evita la duplicazione del codice.
        super().__init__(marca, modello)

        # 2. Inizializza gli attributi specifici della classe figlia
        self.porte = porte

    def muovi(self):
        # Chiama il metodo muovi() della superclasse (Veicolo) e lo
estende
        risultato_padre = super().muovi()
        return f"{risultato_padre} con {self.porte} porte."

mia_auto = Auto("Fiat", "Panda", 5)
print(mia_auto.muovi()) # Output: Fiat Panda si sta muovendo. con 5 porte.
```

---

51. **Polymorphism** 🎭

Obiettivi di apprendimento:

- Comprendere il concetto di **Polimorfismo** (molte forme).
- Applicare il polimorfismo tramite l'uso di interfacce comuni su classi diverse.

Contenuto teorico:

Il **Polimorfismo** (dal greco "molte forme") è la capacità di utilizzare la stessa interfaccia (lo stesso nome di metodo) per oggetti di classi diverse. In Python, questo è supportato in modo implicito.

Se classi diverse (spesso legate dall'ereditarietà) definiscono lo **stesso metodo**, possiamo scrivere codice che non ha bisogno di sapere esattamente *quale* tipo di oggetto sta gestendo.

```
class Cane:
    def parla(self):
        return "Bau!"

class Gatto:
    def parla(self):
        return "Miao!"

class Anatra:
    def parla(self):
        return "Quack!"

def fai_parlare(animale):
    """Questa funzione non si preoccupa del tipo di animale,
    basta che l'oggetto abbia il metodo 'parla()'."""
    print(animale.parla())

# Polymorphism in azione: lo stesso metodo 'fai_parlare' funziona
# con oggetti di classi completamente diverse
fai_parlare(Cane())
fai_parlare(Gatto())
fai_parlare(Anatra())
```

---

52. **Duck Typing** 🦆

Obiettivi di apprendimento:

- Comprendere il principio del **Duck Typing** (tipizzazione dell'anatra).
- Capire come il Duck Typing si relaziona con il Polimorfismo in Python.

Contenuto teorico:

Il **Duck Typing** è il modo in cui Python implementa il polimorfismo. Il principio si riassume nella frase: **"Se cammina come un'anatra, nuota come un'anatra e starnazza come un'anatra, allora deve essere un'anatra."**

In Python, non importa qual è la classe dell'oggetto, importa **solo quali metodi e attributi ha**. Finché un oggetto ha il metodo che stiamo cercando (ad esempio `parla()` come nell'esempio precedente), Python lo tratterà come se fosse il tipo di oggetto desiderato.

Questo rende Python molto flessibile e non richiede l'implementazione formale di interfacce come in altri linguaggi.

```
class Persona:
    def parla(self):
        return "Ciao!"

# La funzione 'fai_parlare' della sezione 51 (Polimorfismo) funziona
# perfettamente anche con la classe Persona, anche se non eredita da
'Animale',
# perché possiede il metodo 'parla()'.

def prova_duck_typing():
    # Cane, Anatra e Persona sono tutti "tipi anatra" in questo contesto.
    oggetti = [Cane(), Anatra(), Persona()]

    print("\n--- Duck Typing in azione ---")
    for oggetto in oggetti:
        fai_parlare(oggetto)

# prova_duck_typing()
```

---

53. **Static Methods** ⚡

Obiettivi di apprendimento:

- Definire un metodo statico usando il decoratore `@staticmethod`.
- Comprendere che un metodo statico appartiene alla classe, non all'istanza.
- Sapere che i metodi statici non hanno accesso a `self` (attributi di istanza) o `cls` (attributi di classe).

Contenuto teorico:

I **Metodi Statici** sono funzioni che si trovano all'interno di una classe, ma che **non operano sull'istanza** della classe (non prendono `self` come primo argomento) né sulla classe stessa (non prendono `cls`).

Sono essenzialmente funzioni normali che per ragioni organizzative (namespace, coerenza) vengono raggruppate all'interno di una classe.

```
class Utility:
    # Metodo statico: non ha bisogno di accedere a nessuna istanza o
attributo di classe
    @staticmethod
    def formatta_data(data_greggia):
        """Converte la data greggia nel formato italiano."""
        # Supponiamo che data_greggia sia "YYYY-MM-DD"
        parti = data_greggia.split('-')
        if len(parti) == 3:
            return f"{parti[2]}/{parti[1]}/{parti[0]}"
```

```
        return data_greggia

    # Questo metodo richiede un'istanza (self)
    def calcola_qualcosa(self, valore):
        return valore * 2

print("=== METODI STATICI ===")

# Il metodo statico può essere chiamato direttamente dalla classe, senza
istanza
data_formattata = Utility.formatta_data("2025-12-04")
print(f"Data formattata: {data_formattata}")

# Un metodo normale richiede prima la creazione di un'istanza
u = Utility()
print(f"Calcolo istanza: {u.calcola_qualcosa(5)}")
```

---

54. **Class Methods** 🏫

Obiettivi di apprendimento:

- Definire un metodo di classe usando il decoratore `@classmethod`.
- Comprendere che un metodo di classe prende la classe stessa (`cls`) come primo argomento.
- Utilizzare i metodi di classe come costruttori alternativi.

Contenuto teorico:

I **Metodi di Classe** sono legati alla classe e non all'istanza dell'oggetto. Prendono la classe stessa (convenzionalmente chiamata `cls`) come primo argomento implicito.

Sono spesso utilizzati per:

1. **Costruttori Alternativi:** Creare istanze della classe da dati in formati diversi (es. da una stringa, da un dizionario).
2. **Operare su Variabili di Classe:** Modificare o accedere alle variabili che sono condivise da tutte le istanze.

```
class Pizza:
    # Variabile di classe che tiene traccia del numero di pizze create
    pizze_vendute = 0

    def __init__(self, dimensione, topping):
        self.dimensione = dimensione
        self.topping = topping
        Pizza.pizze_vendute += 1

    # Metodo di Classe: usa cls per fare riferimento alla classe (Pizza)
    @classmethod
    def crea_margherita(cls):
        """Costruttore alternativo: crea una pizza Margherita standard."""
```

```
        # Ritorna una nuova istanza della classe corrente (cls)
        return cls("Media", ["Mozzarella", "Pomodoro"])

    @classmethod
    def registra_vendita(cls, numero):
        """Metodo per operare su una variabile di classe."""
        cls.pizze_vendute += numero
        print(f"Registrate {numero} vendite extra. Totale:
{cls.pizze_vendute}")

# Uso del costruttore standard
pizza1 = Pizza("Grande", ["Salame", "Peperoni"])

# Uso del costruttore alternativo (Metodo di Classe)
pizza_margherita = Pizza.crea_margherita()

print("=== METODI DI CLASSE ===")
print(f"Pizza 1 (Standard): {pizza1.dimensione} con {pizza1.topping}")
print(f"Pizza Margherita: {pizza_margherita.dimensione} con
{pizza_margherita.topping}")
print(f"Totale pizze iniziali: {Pizza.pizze_vendute}")

Pizza.registra_vendita(10)
```

---

55. **Magic Methods** 🌟

Obiettivi di apprendimento:

- Riconoscere i **Magic Methods** (o *Dunder Methods*).
- Comprendere il ruolo di `__str__` e `__repr__` per la rappresentazione degli oggetti.
- Usare `__len__` e `__add__` per estendere il comportamento di operatori built-in.

Contenuto teorico:

I **Magic Methods** (metodi magici), noti anche come *Dunder Methods* (da **d**ouble **under**score), sono metodi speciali che iniziano e finiscono con doppi underscore (es. `__init__`, `__str__`). Non vengono chiamati direttamente dall'utente, ma vengono invocati automaticamente da Python in risposta a specifiche operazioni o eventi.

Metodo	Invocato da...	Scopo
`__init__`	`Oggetto()`	Inizializzazione di un nuovo oggetto.
`__str__`	`print(Oggetto)` o `str(Oggetto)`	Rappresentazione "leggibile per l'utente".
`__repr__`	Shell interattiva o `repr(Oggetto)`	Rappresentazione "leggibile dal programmatore".
`__len__`	`len(Oggetto)`	Definisce cosa restituisce l'operazione di lunghezza.
`__add__`	`Oggetto1 + Oggetto2`	Definisce il comportamento dell'operatore `+`.

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

    # 1. __str__: per la stampa
    def __str__(self):
        return f"Punto(x={self.x}, y={self.y})"

    # 2. __repr__: per il debugging (spesso identico a __str__ per oggetti
semplici)
    def __repr__(self):
        return f"Punto({self.x}, {self.y})"

    # 3. __add__: per l'operatore +
    def __add__(self, altro_punto):
        nuovo_x = self.x + altro_punto.x
        nuovo_y = self.y + altro_punto.y
        return Punto(nuovo_x, nuovo_y)

p1 = Punto(10, 5)
p2 = Punto(2, 3)

# Invocazione implicita di __str__
print(f"\nStampa: {p1}")

# Invocazione implicita di __add__
p3 = p1 + p2 # python chiama p1.__add__(p2)
print(f"Somma (P1 + P2): {p3}") # Output: Punto(x=12, y=8)
```

---

56. **@property** ⚙️

Obiettivi di apprendimento:

- Comprendere il ruolo del decoratore `@property`.
- Utilizzare `@property` per trattare un metodo come un attributo.
- Implementare *getter*, *setter* e *deleter* per controllare l'accesso e la modifica degli attributi.

Contenuto teorico:

Il decoratore **`@property`** è un modo "alla Python" per implementare i principi dell'**Incapsulamento**. Permette di definire metodi per ottenere (`getter`), impostare (`setter`) e cancellare (`deleter`) il valore di un attributo, ma trattando il metodo come se fosse un attributo normale (non richiede parentesi tonde `()`).

Questo è utile per aggiungere logica di validazione o calcolo senza cambiare il modo in cui l'utente interagisce con l'oggetto.

```python
class Utente:
    def __init__(self, nome, prezzo_orario):
        self._nome = nome # Uso dell'underscore per indicare che
l'attributo è 'privato' (convenzione)
        self._prezzo_orario = prezzo_orario

    # GETTER: Definisce l'attributo virtuale 'prezzo_orario'.
    # Chiamato quando si accede a utente.prezzo_orario
    @property
    def prezzo_orario(self):
        return self._prezzo_orario

    # SETTER: Definisce la logica per impostare il valore di
'prezzo_orario'.
    # Chiamato quando si assegna un valore: utente.prezzo_orario = 15
    @prezzo_orario.setter
    def prezzo_orario(self, nuovo_valore):
        if nuovo_valore < 0:
            raise ValueError("Il prezzo non può essere negativo.")
        self._prezzo_orario = nuovo_valore

    # PROPERTY CALCOLATA (sola lettura)
    @property
    def costo_giornaliero(self):
        # Calcola e restituisce il valore; non memorizzato come attributo.
        return self._prezzo_orario * 8

utente = Utente("Mario Rossi", 10.00)

print(f"\nCosto giornaliero iniziale: €{utente.costo_giornaliero:,.2f}") #
Chiama il getter del costo

# Chiamata al SETTER implicito
utente.prezzo_orario = 12.50

print(f"Nuovo prezzo orario (getter): €{utente.prezzo_orario:,.2f}")
print(f"Nuovo costo giornaliero: €{utente.costo_giornaliero:,.2f}")

# Tentativo di errore (verrebbe intercettato dal setter)
# utente.prezzo_orario = -5 # Solleva ValueError
```

---

57. **Decorators** 🎁

Obiettivi di apprendimento:

- Comprendere che un **Decoratore** è una funzione che prende un'altra funzione e ne estende il comportamento.
- Implementare un decoratore semplice (es. per misurare il tempo di esecuzione).

Contenuto teorico:

Un **Decoratore** è una funzione che *avvolge* un'altra funzione, permettendo di aggiungere funzionalità al codice esistente **senza modificarlo** direttamente. In Python, i decoratori sono invocati usando la sintassi `@nome_decoratore` subito prima della definizione della funzione.

I decoratori sono una forma avanzata di funzioni di ordine superiore (funzioni che prendono altre funzioni come argomenti o le restituiscono).

```
import time

# 1. Definizione del Decoratore
def misura_tempo(func):
    """Questo decoratore misura quanto tempo impiega una funzione a essere
eseguita."""
    def wrapper(*args, **kwargs):
        tempo_inizio = time.time()

        # Chiama la funzione originale
        risultato = func(*args, **kwargs)

        tempo_fine = time.time()
        print(f"Tempo di esecuzione di '{func.__name__}': {tempo_fine -
tempo_inizio:.4f}s")
        return risultato
    return wrapper

# 2. Applicazione del Decoratore
@misura_tempo # Equivalente a: calcola_lunga = misura_tempo(calcola_lunga)
def calcola_lunga():
    """Simula una lunga operazione (es. I/O o calcoli complessi)."""
    somma = 0
    for i in range(1000000):
        somma += i
    return somma

print("=== DECORATORS ===")
calcola_lunga()
# Quando chiami calcola_lunga(), in realtà stai chiamando wrapper(),
# che misura il tempo e poi esegue la funzione originale.
```

---

58. **Exception Handling** 🚦

Obiettivi di apprendimento:

- Comprendere la necessità della gestione delle eccezioni.
- Utilizzare i blocchi **`try`**, **`except`** e **`finally`** per intercettare e gestire gli errori.
- Sollevare eccezioni manualmente con la parola chiave `raise`.

Contenuto teorico:

L'**Exception Handling** (Gestione delle Eccezioni) permette di scrivere codice che può gestire errori e situazioni inaspettate senza che il programma si arresti in modo anomalo.

- **`try`:** Contiene il codice che *potrebbe* sollevare un'eccezione.
- **`except`:** Contiene il codice da eseguire se un'eccezione del tipo specificato si verifica nel blocco `try`.
- **`finally`:** Contiene il codice che *deve sempre* essere eseguito, indipendentemente dal fatto che si sia verificata un'eccezione o meno (spesso usato per operazioni di pulizia, come chiudere file).

```
def gestisci_divisione(numeratore, denominatore):
    try:
        # Codice che potrebbe generare una ZeroDivisionError o TypeError
        risultato = numeratore / denominatore

    except ZeroDivisionError:
        # Gestisce specificamente la divisione per zero
        print("❌ Errore: Non puoi dividere per zero.")
        risultato = None

    except TypeError:
        # Gestisce specificamente tipi non compatibili (es. 10 / "due")
        print("❌ Errore: Assicurati di usare solo numeri.")
        risultato = None

    except Exception as e:
        # Gestisce qualsiasi altra eccezione non catturata
        print(f"❌ Errore generico: {e}")
        risultato = None

    else:
        # Codice eseguito SOLO se il blocco 'try' ha avuto successo
        print("✅ Operazione completata con successo.")

    finally:
        # Codice eseguito sempre
        print("--- Tentativo di calcolo terminato. ---")
        return risultato

print("=== EXCEPTION HANDLING ===")
gestisci_divisione(10, 2)
gestisci_divisione(10, 0)
gestisci_divisione(10, "due")
```
