SVILUPPO APPLICAZIONI E ANALISI DATI CON PYTHON

OOP in Python

Programmazione ad Oggetti

Docente: Marino Speciale

Indice della Lezione

Classi e Oggetti

Classe: Progetto/Template per creare oggetti

Oggetto: Istanza concreta di una classe

```
# Definire una classe
class Automobile:
    def __init__(self, marca, colore):
        self.marca = marca
        self.colore = colore

# Creare oggetti (istanze)
auto1 = Automobile("Ferrari", "rosso")
auto2 = Automobile("Fiat", "blu")
```

auto1 e auto2 sono due oggetti diversi della stessa classe

Attributi e Metodi

```
class Persona:
    def __init__(self, nome, età):
        # Attributi (variabili dell'oggetto)
        self.nome = nome
        self.età = età

    # Metodo (funzione dell'oggetto)
    def presentarsi(self):
        print(f"Ciao, mi chiamo {self.nome}")

    # Metodo che modifica attributi
    def invecchiare(self):
        self.età += 1

# Uso
p = Persona("Mario", 25)
p.presentarsi()      # Ciao, mi chiamo Mario
p.invecchiare()      # Ora ha 26 anni
```

Incapsulamento: Attributi Privati

```
class ContoCorrente:
    def __init__(self, saldo):
        self.__saldo = saldo  # Privato: doppio underscore

    def deposita(self, importo):
        """Metodo pubblico per depositare"""
        self.__saldo += importo

    def preleva(self, importo):
        if importo <= self.__saldo:
            self.__saldo -= importo
        else:
            print("Saldo insufficiente")

    def get_saldo(self):
        """Metodo getter per leggere il saldo"""
        return self.__saldo

conto = ContoCorrente(1000)
conto.deposita(500)
print(conto.get_saldo())      # 1500
# conto.__saldo = -999  # ERRORE: non accessibile direttamente
```

Ereditarietà: Classe Padre e Figlio

```
# Classe Padre (Base)
class Animale:
    def __init__(self, nome):
        self.nome = nome

    def suona(self):
        return "Suono generico"

# Classe Figlio (Derivata) - Eredita da Animale
class Cane(Animale):
    def suona(self):  # Override del metodo
        return "Bau bau"

class Gatto(Animale):
    def suona(self):  # Override del metodo
        return "Miao"

# Uso (Polimorfismo)
animali = [Cane("Rex"), Gatto("Micia"), Animale("Creatura")]
for animale in animali:
    print(f"{animale.nome} fa: {animale.suona()}")
```

Super() e Inizializzazione

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

class Auto(Veicolo):
    def __init__(self, marca, colore, num_porte):
        # Chiama il costruttore della classe padre
        super().__init__(marca, colore)
        self.num_porte = num_porte

    def descrizione(self):
        return f"{self.marca} {self.colore} con {self.num_porte} porte"

auto = Auto("Toyota", "nero", 4)
print(auto.descrizione())  # Toyota nero con 4 porte
```

Polimorfismo: Duck Typing

```
# Diversi oggetti con lo stesso metodo
class Papera:
    def verso(self):
        return "Quack!"

class Gatto:
    def verso(self):
        return "Miao"

# Funzione polimorfa: funziona con qualsiasi oggetto
# che abbia il metodo verso()
def fai_verso(animale):
    print(animale.verso())

fai_verso(Papera())  # Quack!
fai_verso(Gatto())   # Miao

# Duck typing: "Se cammina come papera e starnazza..."
```

Metodi Speciali (__str__, __eq__, etc.)

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

    # __str__: rappresentazione stringa (print)
    def __str__(self):
        return f"Punto({self.x}, {self.y})"

    # __repr__: rappresentazione per debug
    def __repr__(self):
        return f"Punto(x={self.x}, y={self.y})"

    # __eq__: uguaglianza
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # __add__: operatore +
    def __add__(self, other):
        return Punto(self.x + other.x, self.y + other.y)

p1 = Punto(1, 2)
p2 = Punto(1, 2)
print(p1)           # Punto(1, 2)
print(p1 == p2)     # True
print(p1 + p2)      # Punto(2, 4)
```

Classi Astratte (ABC)

```
from abc import ABC, abstractmethod

# Classe astratta: non può essere istanziata
class Forma(ABC):
    @abstractmethod
    def area(self):
        """Metodo astratto: deve essere implementato da figli"""
        pass

    @abstractmethod
    def perimetro(self):
        pass

# Implementazione concreta
class Quadrato(Forma):
    def __init__(self, lato):
        self.lato = lato

    def area(self):
        return self.lato ** 2

    def perimetro(self):
        return 4 * self.lato

# forma = Forma()  # ERRORE: non può istanziare classe astratta
quad = Quadrato(5)
print(quad.area())      # 25
```

@property e @setter

```
class Persona:
    def __init__(self, nome, età):
        self.nome = nome
        self._età = età  # Privata per convenzione

    # @property: accesso come attributo
    @property
    def età(self):
        return self._età

    # @setter: modifica con controllo
    @età.setter
    def età(self, valore):
        if valore < 0:
            raise ValueError("Età non può essere negativa")
        self._età = valore

p = Persona("Marco", 25)
print(p.età)    # 25 (sembra attributo)
p.età = 30      # Valida il valore con setter
# p.età = -5    # ERRORE: ValueError
```

@staticmethod vs @classmethod

```
class Utilità:
    versione = "1.0"

    # @staticmethod: non accede a self né cls
    @staticmethod
    def somma(a, b):
        return a + b

    # @classmethod: accede a cls (la classe stessa)
    @classmethod
    def versione_info(cls):
        return f"Versione: {cls.versione}"

# Uso
print(Utilità.somma(5, 3))           # 8 (senza istanza)
print(Utilità.versione_info())       # Versione: 1.0

# Non servono istanze!
# u = Utilità()  NON necessario
```

Composizione vs Ereditarietà

```
# EREDITARIETÀ (IS-A)
class Animale:
    pass
class Cane(Animale):  # Cane IS-A Animale
    pass

# COMPOSIZIONE (HAS-A) - Preferita quando possibile
class Motore:
    def accendi(self):
        print("Motore acceso")

class Auto:
    def __init__(self):
        self.motore = Motore()  # Auto HAS-A Motore

    def avvia(self):
        self.motore.accendi()

auto = Auto()
auto.avvia()  # Motore acceso

# Composizione = flessibilità + semplicità
```

Progetto: Sistema Bancario (Part 1)

```
from abc import ABC, abstractmethod

class Conto(ABC):
    def __init__(self, titolare, saldo_iniziale):
        self._titolare = titolare
        self._saldo = saldo_iniziale

    @abstractmethod
    def calcola_interessi(self):
        pass

    def deposita(self, importo):
        self._saldo += importo

    def preleva(self, importo):
        if importo > self._saldo:
            raise ValueError("Saldo insufficiente")
        self._saldo -= importo

    @property
    def saldo(self):
        return self._saldo
```

Progetto: Sistema Bancario (Part 2)

```
# Sottoclassi concrete
class ContoCorrente(Conto):
    def calcola_interessi(self):
        return 0  # Niente interessi

class ContoRisparmio(Conto):
    def __init__(self, titolare, saldo, tasso):
        super().__init__(titolare, saldo)
        self.tasso = tasso

    def calcola_interessi(self):
        # Deposita gli interessi
        interesse = self._saldo * (self.tasso / 100)
        self._saldo += interesse
        return interesse

# Uso
cc = ContoCorrente("Mario", 1000)
cr = ContoRisparmio("Luigi", 1000, 2.5)

cc.deposita(500)
print(f"CC saldo: {cc.saldo}")  # 1500

interesse = cr.calcola_interessi()
print(f"Interesse: {interesse}")  # 25.0
```

Errori Comuni in OOP

- ❌ Dimenticare nei metodi
- ❌ Condividere liste/dizionari tra istanze (mutable defaults)
- ❌ Non usare super() in ereditarietà complessa
- ❌ Overengineering: classi troppo complesse e annidate
- ❌ Esporre attributi privati direttamente

Best Practices OOP

- ✅ Preferisci composizione all'ereditarietà
- ✅ Usa Abstract Base Classes per interfacce
- ✅ Scrivi docstring per classi e metodi
- ✅ Incapsula: attributi privati, metodi pubblici
- ✅ Scrivi test unitari per le classi

Strumenti Consigliati

PyCharm Professional

Refactoring automatico, analisi codice, debugging visuale

→ Produzione

VS Code + Extensions

Pylance per type checking, Pylint per analisi

→ Leggero

Testing Unitario con unittest

```
import unittest

class TestConto(unittest.TestCase):
    def setUp(self):
        """Preparazione prima di ogni test"""
        self.conto = ContoCorrente("Test", 1000)

    def test_deposita(self):
        """Test del metodo deposita"""
        self.conto.deposita(500)
        self.assertEqual(self.conto.saldo, 1500)

    def test_preleva_insufficiente(self):
        """Test quando il saldo è insufficiente"""
        with self.assertRaises(ValueError):
            self.conto.preleva(2000)

if __name__ == '__main__':
    unittest.main()
```

Design Pattern: Singleton

```
class DatabaseSingleton:
    """Solo una istanza per tutta l'applicazione"""
    _istanza = None

    def __new__(cls):
        if cls._istanza is None:
            cls._istanza = super().__new__(cls)
        return cls._istanza

db1 = DatabaseSingleton()
db2 = DatabaseSingleton()
print(db1 is db2)  # True - stessa istanza!
```

Design Pattern: Factory

```
class ContoFactory:
    """Factory per creare conti diversi"""
    @staticmethod
    def crea_conto(tipo, titolare, saldo, **kwargs):
        if tipo == "corrente":
            return ContoCorrente(titolare, saldo)
        elif tipo == "risparmio":
            return ContoRisparmio(titolare, saldo, kwargs.get('tasso', 2.0))
        else:
            raise ValueError(f"Tipo {tipo} non supportato")

# Uso (il client non conosce i dettagli)
c1 = ContoFactory.crea_conto("corrente", "Mario", 1000)
c2 = ContoFactory.crea_conto("risparmio", "Luigi", 5000, tasso=3.5)
```

Riepilogo Lezione

- ✓ Classi, oggetti, attributi, metodi
- ✓ Incapsulamento e attributi privati
- ✓ Ereditarietà e polimorfismo
- ✓ Classi astratte e design patterns

Domande?

Approfondimenti:

→ Testing e TDD (Test-Driven Development)

→ Web Framework: Django e Flask

→ Documentazione: Sphinx e docstring