Introduzione alle 3 regole fondamentali

Python

Questi concetti sono essenziali per strutturare programmi efficaci e riutilizzabili, specialmente in linguaggi che supportano l'OOP come Python, Java e C#.

Ecco una breve spiegazione di ciascuno:

Incapsulamento: L'incapsulamento è il principio di nascondere i dettagli interni dell'implementazione di una classe e di esporre solo quelle funzionalità che sono necessarie per il resto del sistema.

In pratica, ciò significa definire le variabili di una classe come private o protette e fornire metodi pubblici (getter e setter) per accedere e modificare tali variabili.

Questo aiuta a proteggere l'integrità dei dati e a minimizzare l'interdipendenza tra componenti software.

```python
class Account:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # Il saldo è nascosto dall'esterno

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Aggiunti {amount} al saldo")
        else:
            print("Il deposito deve essere positivo")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Prelievo di {amount} effettuato")
        else:
            print("Fondi insufficienti o importo non valido")

    def get_balance(self):
        return self.__balance

# Esempio di utilizzo
account = Account("Mario")
account.deposit(500)
print(account.get_balance())
account.withdraw(200)
print(account.get_balance())
```

Ereditarietà: L'ereditarietà permette di creare nuove classi basate su classi esistenti.

Una classe figlia eredita metodi e variabili dalla classe genitore, ma può anche aggiungere nuove funzionalità o modificare quelle esistenti.

L'ereditarietà facilita il riutilizzo del codice e può rendere il design del software più logico e organizzato.

```python
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display_info(self):
        print(f"Veicolo: {self.brand} {self.model}")

class Car(Vehicle):  # Car eredita da Vehicle
    def __init__(self, brand, model, horsepower):
        super().__init__(brand, model)
        self.horsepower = horsepower

    def display_info(self):
        super().display_info()
        print(f"Potenza: {self.horsepower} CV")

# Esempio di utilizzo
car = Car("Tesla", "Model S", 670)
car.display_info()
```

Polimorfismo: Il polimorfismo è la capacità di una funzione di operare su oggetti di diverse classi.

In Python, il polimorfismo si manifesta principalmente attraverso l'overriding dei metodi (quando una classe figlia reimplementa un metodo della classe genitore) e il duck typing (quando un oggetto è valutato in base ai metodi e attributi che supporta, piuttosto che sulla sua effettiva ereditarietà).

Questo permette di scrivere codice più flessibile e facilmente estendibile.

```
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Bau Bau!")

class Cat(Animal):
    def make_sound(self):
        print("Miao Miao!")

def animal_sound(animal):
    animal.make_sound()

# Esempio di utilizzo
dog = Dog()
cat = Cat()

animal_sound(dog)  # Output: Bau Bau!
animal_sound(cat)  # Output: Miao Miao!

```

Astrazione nella Programmazione Orientata agli Oggetti (OOP)

Teoria dell'Astrazione L'astrazione è un concetto fondamentale nella programmazione orientata agli oggetti (OOP) che mira a ridurre la complessità dei sistemi, nascondendo i dettagli meno rilevanti e mettendo in evidenza le caratteristiche essenziali. Questo principio aiuta i programmatori a concentrarsi su interazioni ad alto livello, omettendo dettagli di basso livello.

In pratica, l'astrazione è realizzata definendo delle classi che rappresentano astrazioni di entità o concetti del mondo reale.

Una classe astratta può includere metodi astratti, che sono metodi dichiarati ma non implementati nella classe astratta stessa.

Le classi derivate sono poi obbligate a implementare questi metodi, fornendo comportamenti specifici che dipendono dalla particolare sottoclasse utilizzata.

```python
from abc import ABC, abstractmethod

class Shape(ABC):  # ABC indica che Shape è una classe astratta
    def __init__(self, color):
        self.color = color

    @abstractmethod
    def area(self):
        pass  # Nessuna implementazione qui, deve essere implementato nelle sottoclassi

    @abstractmethod
    def perimeter(self):
        pass  # Nessuna implementazione qui, deve essere implementato nelle sottoclassi

    def display_color(self):
        print(f"The color of the shape is {self.color}")

class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14 * self.radius

class Rectangle(Shape):
    def __init__(self, color, width, height):
        super().__init__(color)
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Esempio di utilizzo
circle = Circle("red", 5)
rectangle = Rectangle("blue", 4, 6)

print(f"Circle area: {circle.area()}")
print(f"Rectangle area: {rectangle.area()}")
circle.display_color()
rectangle.display_color()
```

In Python, come in molti altri linguaggi di programmazione, ci sono diverse regole e principi che sono fondamentali per scrivere codice efficace e mantenibile.

Ecco tre regole fondamentali che ogni programmatore Python dovrebbe conoscere:

Leggibilità del codice:

Python è noto per la sua leggibilità e semplicità.

La Guida di Stile per Python, conosciuta come PEP 8, fornisce linee guida su come formattare il codice Python per renderlo più leggibile.

Questo include convenzioni su spaziature, nomi di variabili, lunghezza delle linee, commenti e molto altro. Mantenere il codice leggibile è cruciale, perché rende il software più facile da comprendere e mantenere.

Uso di "Pythonic" idioms:

Scrivere codice "Pythonic" significa utilizzare le strutture e gli idiomi del linguaggio Python nel modo più efficace possibile.

Questo implica sfruttare le caratteristiche del linguaggio come le comprensioni di lista, le espressioni generatori, e l'uso del duck typing.

Ad esempio, preferire for item in iterable: invece di usare indici quando non sono strettamente necessari.

Questi idiomi aiutano a sfruttare appieno le capacità di Python e a rendere il codice più efficiente e facile da seguire.

Gestione delle eccezioni:

In Python, è importante gestire correttamente le eccezioni per evitare crash del programma e per gestire situazioni impreviste durante l'esecuzione.

L'uso di blocchi try e except permette di intercettare errori e di reagire a essi, anziché lasciare che l'intero programma fallisca.

Inoltre, è buona pratica utilizzare le eccezioni specifiche piuttosto che catturare genericamente tutte le eccezioni, per evitare di mascherare altri errori non correlati.


Buon MasterD a tutti