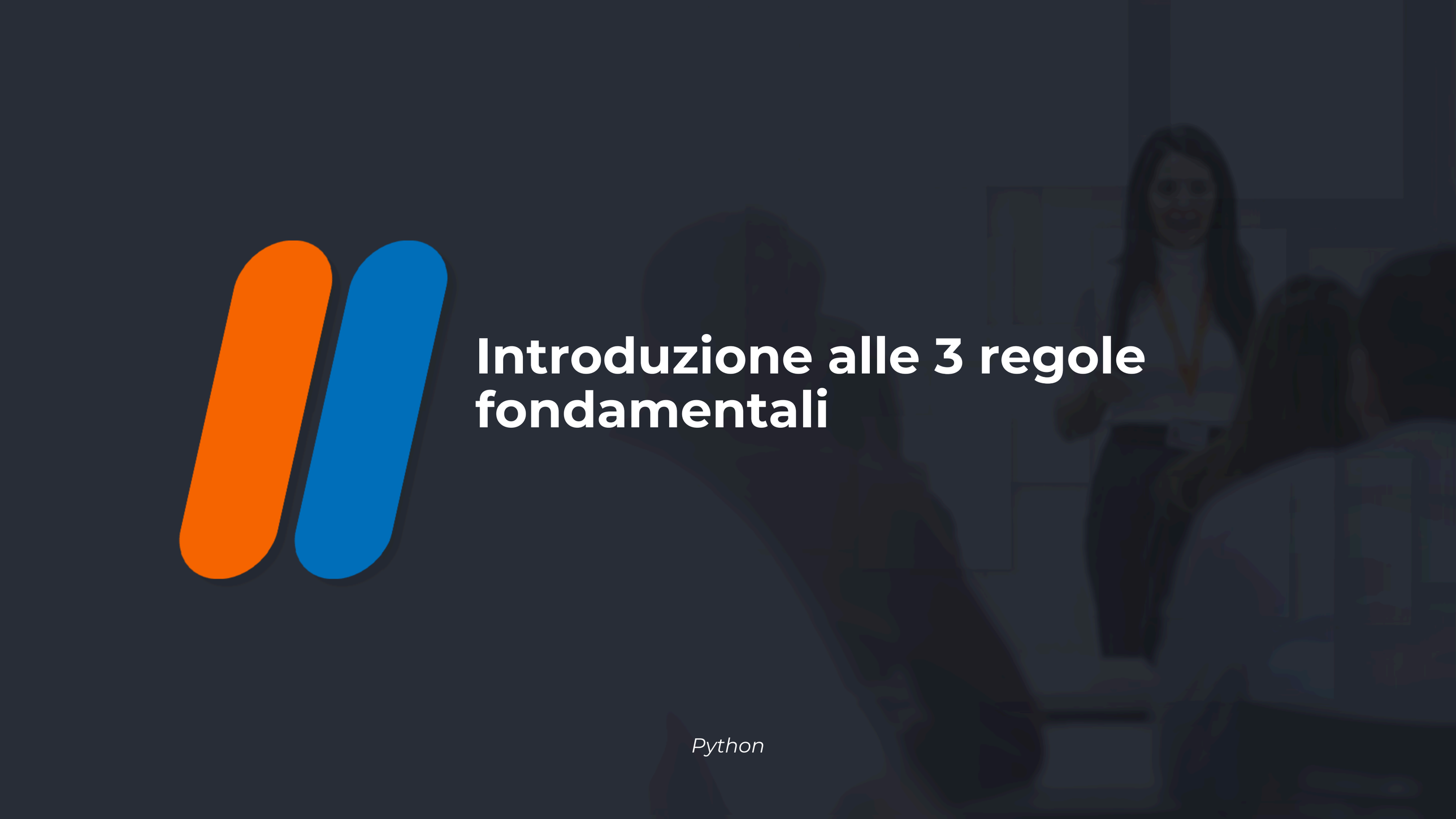


Introduzione alle 3 regole fondamentali

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:

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.

L'ereditarietà in Python è un meccanismo fondamentale che permette a una classe (detta classe derivata) di ereditare attributi e metodi da un’altra classe (detta classe base).

Questo approccio facilita il riuso del codice e la creazione di gerarchie logiche nelle applicazioni, migliorando la manutenibilità e l'estendibilità del software.

La sintassi tipica per definire una classe derivata è class Derived(Base):, dove Derived acquisisce automaticamente i comportamenti e le proprietà definiti in Base.

Inoltre, Python supporta il concetto di overriding, che consente alla classe derivata di ridefinire metodi ereditati per modificarne il comportamento in maniera specifica, e l'ereditarietà multipla, permettendo a una classe di ereditare da più classi base, sebbene questo richieda una particolare attenzione alla gestione dell'ordine di risoluzione dei metodi (MRO).

Un elemento chiave nell'implementazione dell'ereditarietà è l'uso della funzione super(), che permette di accedere in modo dinamico ai metodi della classe base senza dover specificare esplicitamente il nome della classe.

Questa tecnica è particolarmente utile per mantenere un flusso logico coerente nelle chiamate ai metodi, soprattutto in contesti di ereditarietà multipla, dove il corretto ordine di invocazione può influire significativamente sul comportamento complessivo del programma.

In sintesi, l'ereditarietà in Python e l'uso appropriato di super() sono strumenti potenti per costruire strutture di codice modulari e riutilizzabili, elementi essenziali per lo sviluppo di applicazioni complesse e ben organizzate.

Inoltre, l'ereditarietà in Python si integra perfettamente con il concetto di polimorfismo, che permette a funzioni e metodi di trattare in maniera uniforme istanze di classi differenti ma correlate.

Ad esempio, una funzione può operare su oggetti di classi diverse che condividono un'interfaccia comune, senza dover conoscere i dettagli specifici di ciascuna implementazione.

Questo livello di astrazione non solo semplifica la scrittura di codice generico, ma favorisce anche l'estendibilità, poiché nuovi tipi di oggetti possono essere introdotti nel sistema senza richiedere modifiche sostanziali al codice esistente.

Un ulteriore aspetto teorico da considerare è l'utilizzo dei mixin, che rappresentano classi progettate specificamente per fornire funzionalità riutilizzabili e che vengono ereditate da altre classi senza essere considerate parte della gerarchia principale.

I mixin consentono di "mescolare" comportamenti aggiuntivi nelle classi target, promuovendo la composizione e riducendo la duplicazione del codice.

Questa tecnica, combinata con l'ereditarietà multipla, permette di creare sistemi altamente modulari e flessibili, sebbene richieda una progettazione attenta per evitare conflitti e ambiguità nella risoluzione dei metodi.

Di seguito una lista dei metodi e delle funzioni intrinseci all'ereditarietà in Python, con una breve spiegazione per ciascuno:

- **super(): Funzione che permette di richiamare metodi della classe base in modo dinamico, facilitando l’overriding e la gestione dell’ereditarietà multipla.**
- **isinstance(obj, Class): Funzione built-in che verifica se un oggetto è un'istanza di una determinata classe o di una sua sottoclasse.**
- **issubclass(Class1, Class2): Funzione built-in che controlla se la prima classe è una sottoclasse della seconda.**
- **mro: Attributo che restituisce il "Method Resolution Order", ovvero l’ordine in cui vengono risolti i metodi nelle gerarchie di ereditarietà.**

Di seguito una lista dei metodi e delle funzioni intrinseci all'ereditarietà in Python, con una breve spiegazione per ciascuno:

- **bases: Attributo che restituisce una tupla contenente le classi base immediate da cui una classe eredita.**
- **subclasses(): Metodo che, invocato su una classe, restituisce una lista delle sue sottoclassi immediate.**
- **init_subclass(): Metodo speciale chiamato automaticamente ogni volta che una classe viene sottoclasseggiata, utile per personalizzare il processo di creazione di sottoclassi.**
- **new(): Metodo statico utilizzato per creare una nuova istanza della classe, spesso ridefinito per controllare il processo di istanziazione in contesti ereditari.**

```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()
```

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.

L'incapsulamento è un principio fondamentale della programmazione orientata agli oggetti che consiste nel nascondere i dettagli interni di un oggetto, esponendo solo ciò che è strettamente necessario tramite un'interfaccia pubblica.

In Python, sebbene non esista una vera e propria "privatizzazione" degli attributi come in altri linguaggi, la convenzione prevede l'uso del prefisso di un underscore (ad esempio, _attributo) per indicare che un attributo o un metodo è destinato ad essere usato solo all'interno della classe o dei suoi derivati.

Questa pratica contribuisce a mantenere l'integrità dello stato interno dell'oggetto, evitando accessi o modifiche accidentali dall'esterno.

Un aspetto importante dell'incapsulamento riguarda anche il controllo sull'accesso e la manipolazione dei dati interni attraverso metodi specifici, comunemente noti come getter e setter.

Questi metodi consentono di implementare logiche di validazione o di trasformazione dei dati prima che vengano letti o modificati, garantendo così che lo stato dell'oggetto rimanga coerente e valido.

Inoltre, Python supporta il meccanismo di name mangling (utilizzando il doppio underscore, ad esempio __attributo), che, pur non rendendo l'attributo completamente inaccessibile, ne complica l'accesso diretto e favorisce l'uso delle interfacce pubbliche progettate per interagire con l'oggetto.

Ecco una lista dei metodi e delle funzioni che, in Python, sono comunemente impiegati per gestire l'incapsulamento, permettendo di controllare l'accesso agli attributi interni degli oggetti:

- **__getattribute__(self, name): Metodo speciale invocato per ogni accesso a un attributo. Può essere ridefinito per controllare o registrare gli accessi agli attributi interni.**

- **__getattr__(self, name): Metodo chiamato quando un attributo richiesto non viene trovato tramite il normale meccanismo di ricerca. È utile per implementare comportamenti di fallback o per gestire attributi "virtuali".**

- **__setattr__(self, name, value): Metodo che intercetta ogni assegnazione ad un attributo. Ridefinirlo consente di applicare logiche di validazione o trasformazione dei dati prima della memorizzazione.**

- __delattr__(self, name): Metodo chiamato quando si tenta di eliminare un attributo. Può essere usato per impedire la rimozione accidentale di attributi critici.

- property(fget=None, fset=None, fdel=None, doc=None): Funzione built-in che permette di definire proprietà con getter, setter e deleter in maniera elegante, facilitando l'incapsulamento e il controllo degli accessi agli attributi.

- getattr(object, name[, default]): Funzione built-in per accedere in maniera dinamica ad un attributo, con la possibilità di specificare un valore di default nel caso in cui l'attributo non esista.

- **setattr(object, name, value): Funzione built-in per assegnare un valore ad un attributo, utile per operazioni dinamiche.**

- **delattr(object, name): Funzione built-in che consente di eliminare dinamicamente un attributo da un oggetto.**

- **hasattr(object, name): Funzione built-in che verifica la presenza di un attributo in un oggetto, favorendo la scrittura di codice difensivo.**

```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())
```

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.

In Python il polimorfismo si realizza grazie alla natura dinamica del linguaggio, che permette alle funzioni e ai metodi di operare su oggetti di tipi diversi purché abbiano un comportamento compatibile.

Il meccanismo si fonda sul concetto di "duck typing": se un oggetto implementa i metodi richiesti, il codice può interagire con esso senza fare controlli espliciti sul tipo.

Questo approccio favorisce la scrittura di funzioni generiche e flessibili, capaci di gestire vari tipi di dati basandosi sulle capacità reali dell’oggetto piuttosto che sulla sua identità formale.

Il comportamento polimorfico si manifesta internamente attraverso l’ereditarietà e il metodo di risoluzione dei metodi (MRO).

Le classi base definiscono interfacce comuni che le classi derivate possono ridefinire, permettendo così ad oggetti di classi differenti di rispondere in modo specifico a una chiamata di metodo comune.

Durante l’esecuzione, Python utilizza il binding dinamico per determinare quale implementazione di un metodo invocare, rendendo la scelta dell’azione dipendente dall’oggetto istanziato anziché dalla dichiarazione statica.

Questo paradigma si riscontra in numerose aree del linguaggio e delle sue librerie: dai metodi speciali come str, add o len, che consentono la personalizzazione del comportamento per operazioni standard, alle API di librerie esterne che sfruttano il polimorfismo per offrire interfacce uniformi su strutture dati diverse.

In sostanza, il polimorfismo in Python si integra in tutto l'ecosistema, facilitando l'estensione e la manutenzione del codice e promuovendo uno sviluppo modulare e riutilizzabile.

```
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.

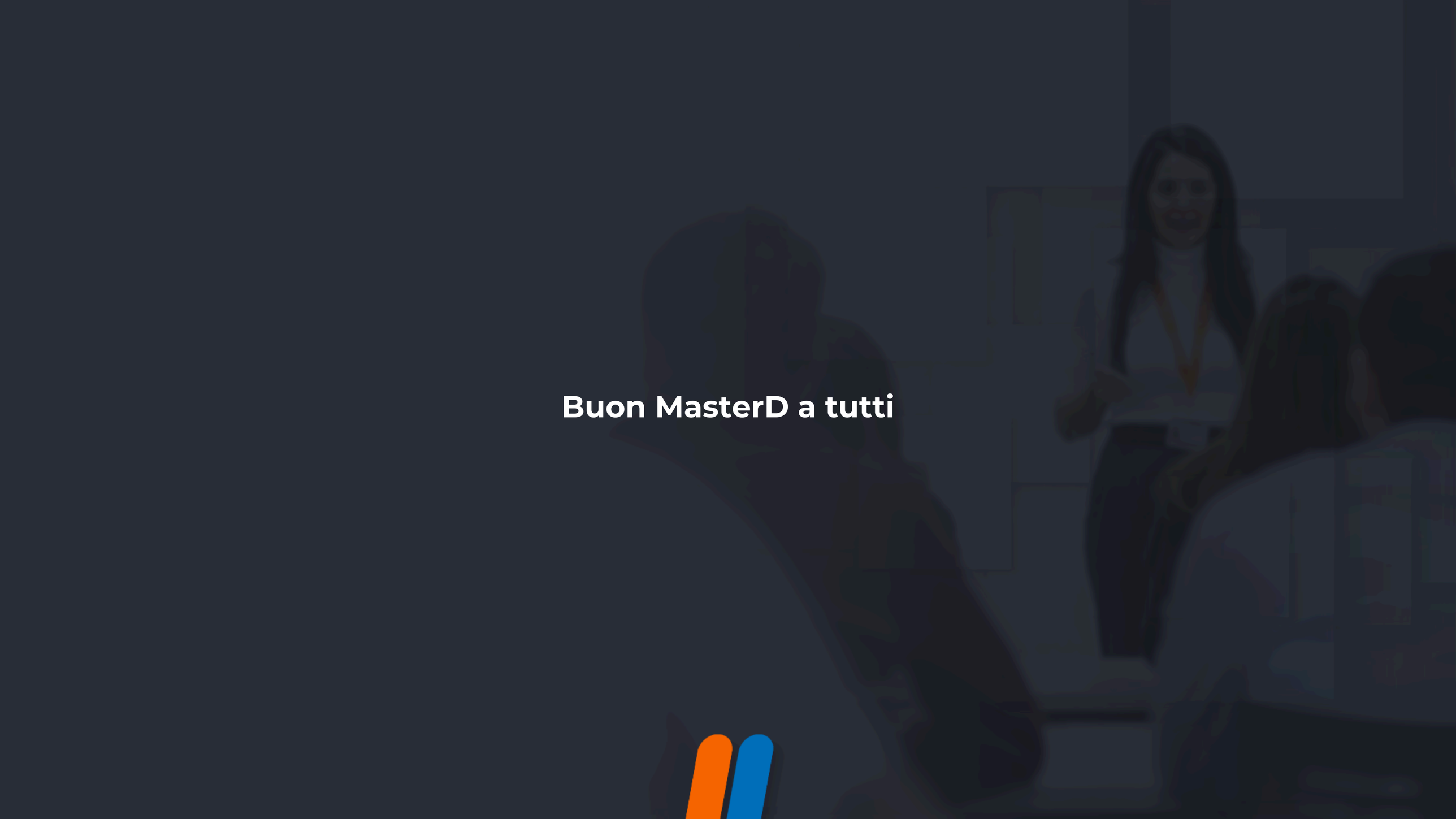
Questi idiomi aiutano a sfruttare appieno le capacità di Python e a rendere il codice più efficiente e facile da seguire.

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