Programmazione Orientata agli Oggetti in Python

Benvenuti a questo corso sulla Programmazione Orientata agli Oggetti (OOP) in Python. Attraverso questo percorso didattico, esploreremo i fondamenti teorici e pratici della OOP, una metodologia di programmazione che consente di strutturare il codice in modo efficiente e riutilizzabile.

Questo corso è pensato per studenti con conoscenze base di Python che desiderano approfondire un paradigma di programmazione fondamentale nello sviluppo software moderno. Dall'incapsulamento all'ereditarietà, analizzeremo tutti i concetti chiave con esempi pratici e spiegazioni dettagliate.

Paradigmi di Programmazione a Confronto

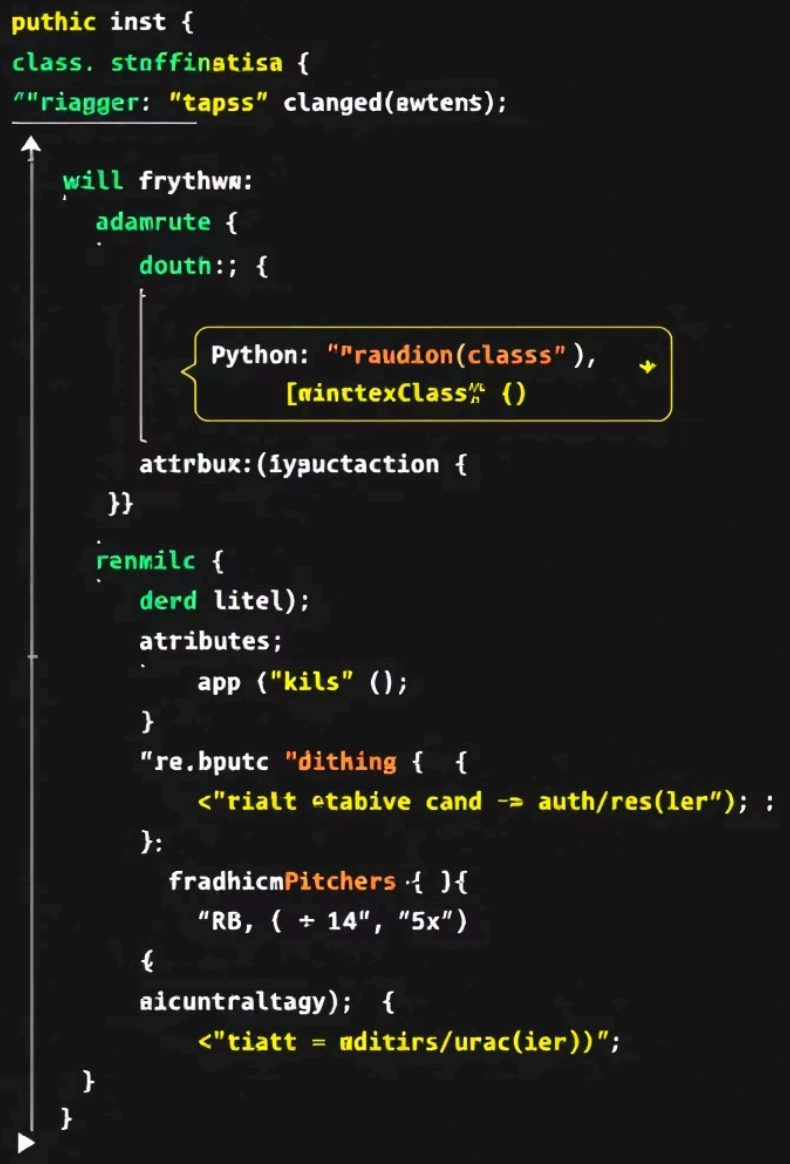
Programmazione Procedurale

- Focalizzata sulle procedure (funzioni)
- Dati e funzioni sono entità separate
- Struttura lineare del codice
- Difficile da mantenere su larga scala

Programmazione Orientata agli Oggetti

- Focalizzata sugli oggetti che contengono dati e metodi
- Incapsulamento di dati e comportamenti
- Struttura modulare e riutilizzabile
- Maggiore manutenibilità e scalabilità

La differenza fondamentale tra i due paradigmi risiede nell'organizzazione del codice: mentre la programmazione procedurale separa dati e funzioni, la OOP li unisce in entità chiamate oggetti. Questa integrazione porta a un codice più organico, modulare e facilmente estendibile.



Definizione di una Classe in Python

Struttura Base

Una classe è definita con la parola chiave "class" seguita dal nome (in CamelCase per convenzione) e due punti. Il corpo della classe contiene attributi e metodi.

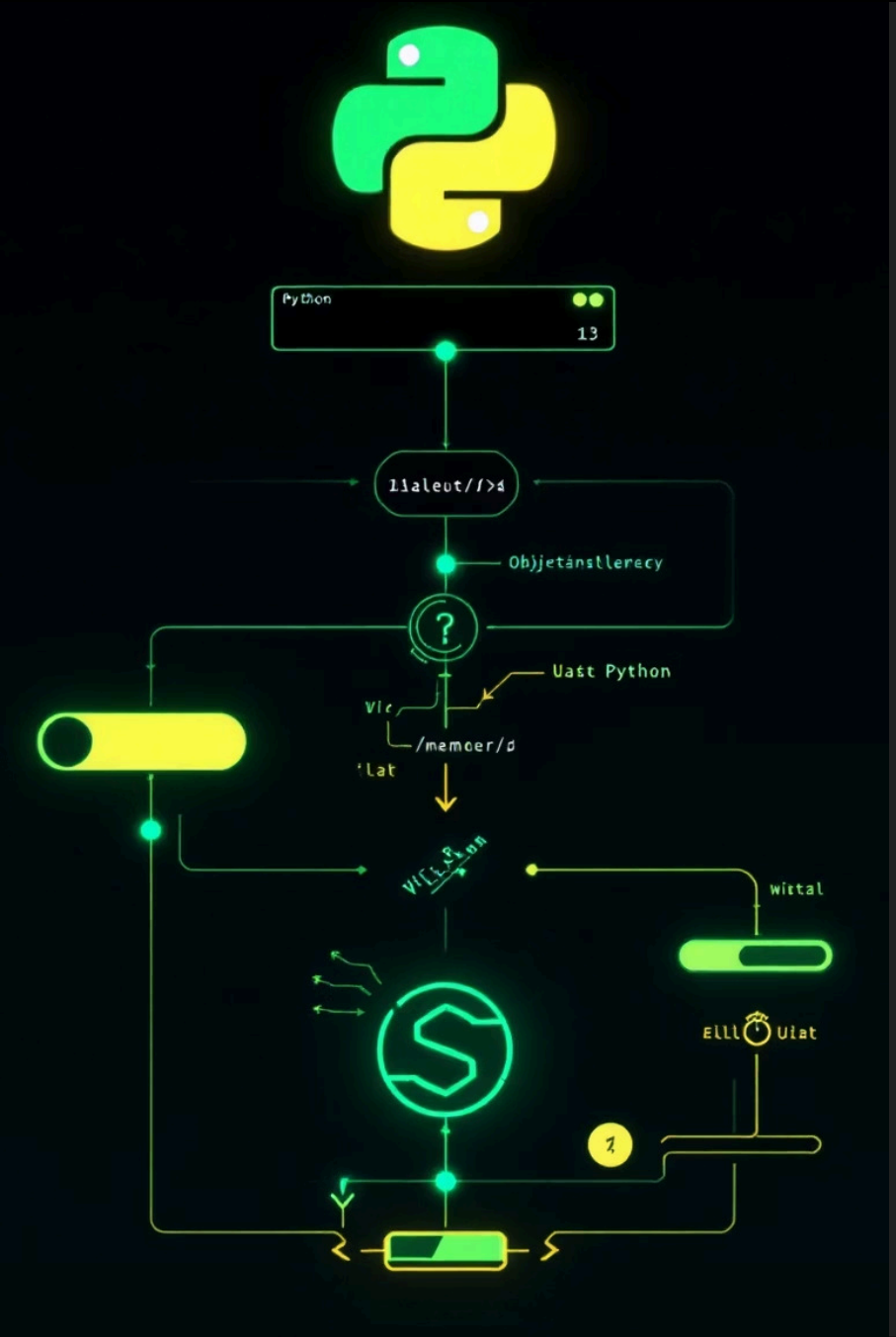
Attributi

Rappresentano i dati o le proprietà che caratterizzano la classe. Possono essere definiti a livello di classe o di istanza.

Metodi

Sono funzioni definite all'interno della classe che determinano il comportamento degli oggetti. Ricevono sempre "self" come primo parametro.

Le classi fungono da "blueprint" per la creazione di oggetti. Definiscono la struttura e il comportamento che tutte le istanze della classe condivideranno. Una buona progettazione delle classi è fondamentale per sfruttare appieno i vantaggi della programmazione orientata agli oggetti.



Istanziare Oggetti dalla Classe

Definizione della Classe

Creiamo il modello con attributi e metodi

Chiamata al Costruttore

Utilizziamo il nome della classe seguito da parentesi

Creazione dell'Oggetto

Python alloca memoria e restituisce un riferimento all'oggetto

Utilizzo dell'Oggetto

Accediamo agli attributi e invochiamo i metodi

L'istanziazione è il processo che trasforma una classe (modello astratto) in un oggetto concreto nella memoria del computer. Ogni oggetto istanziato ha il proprio spazio di memoria per gli attributi, ma condivide i metodi definiti nella classe.

Attributi di Istanza e Metodi

Attributi di Istanza

Dati specifici per ogni oggetto

Metodi

Comportamenti definiti nella classe

Notazione con punto

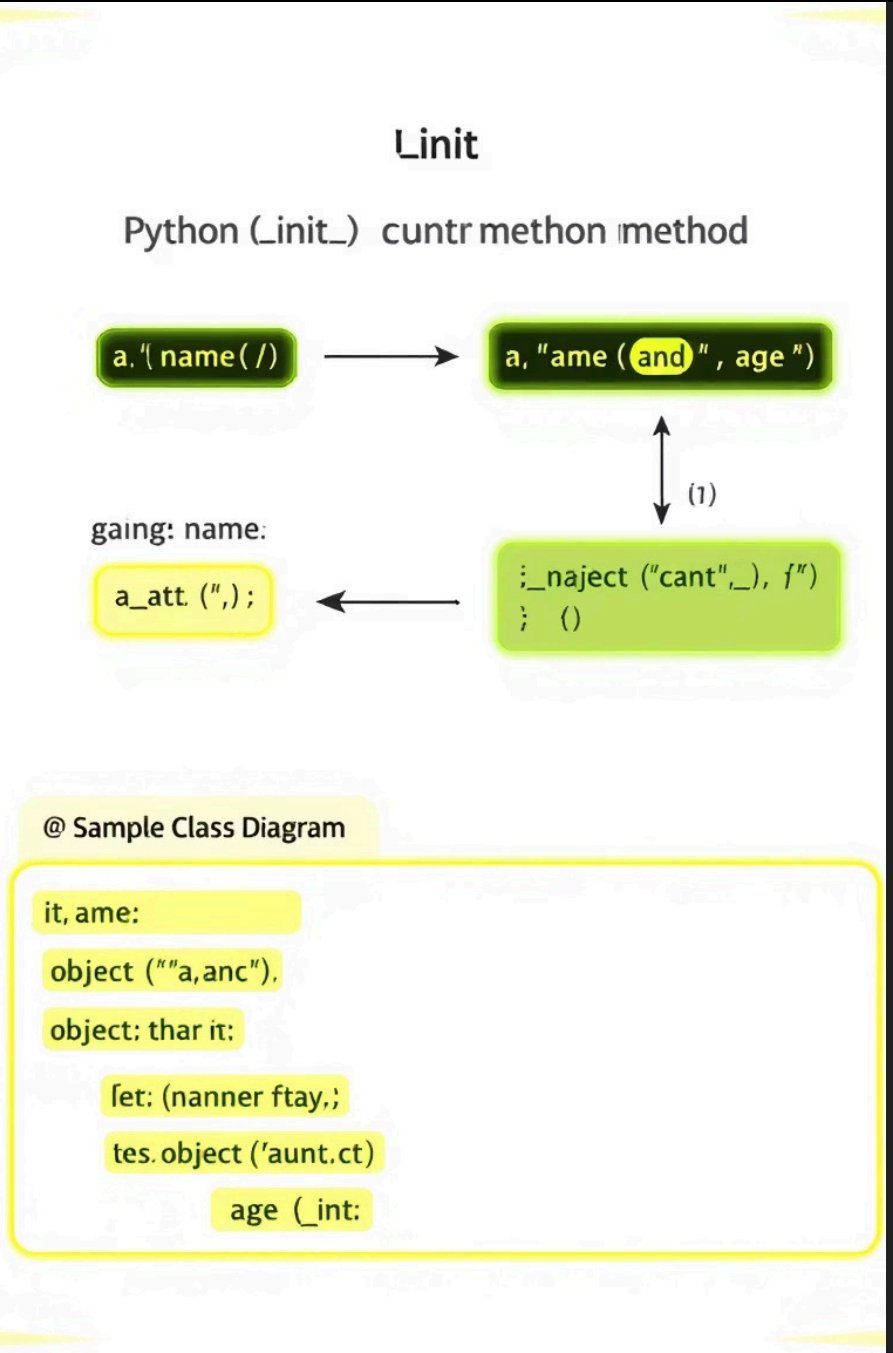
Accesso a attributi e metodi

Parametro self

Riferimento all'istanza corrente

Gli attributi di istanza memorizzano dati specifici per ogni oggetto creato dalla classe. Sono accessibili tramite il parametro "self" all'interno dei metodi della classe, che rappresenta l'istanza specifica su cui il metodo viene chiamato.

I metodi definiscono i comportamenti che gli oggetti possono eseguire e hanno sempre "self" come primo parametro, permettendo loro di accedere e modificare gli attributi dell'istanza specifica.



Il Metodo Costruttore __init__()

Invocazione

Chiamato automaticamente alla creazione dell'oggetto

Parametrizzazione

Riceve i dati necessari per inizializzare l'oggetto

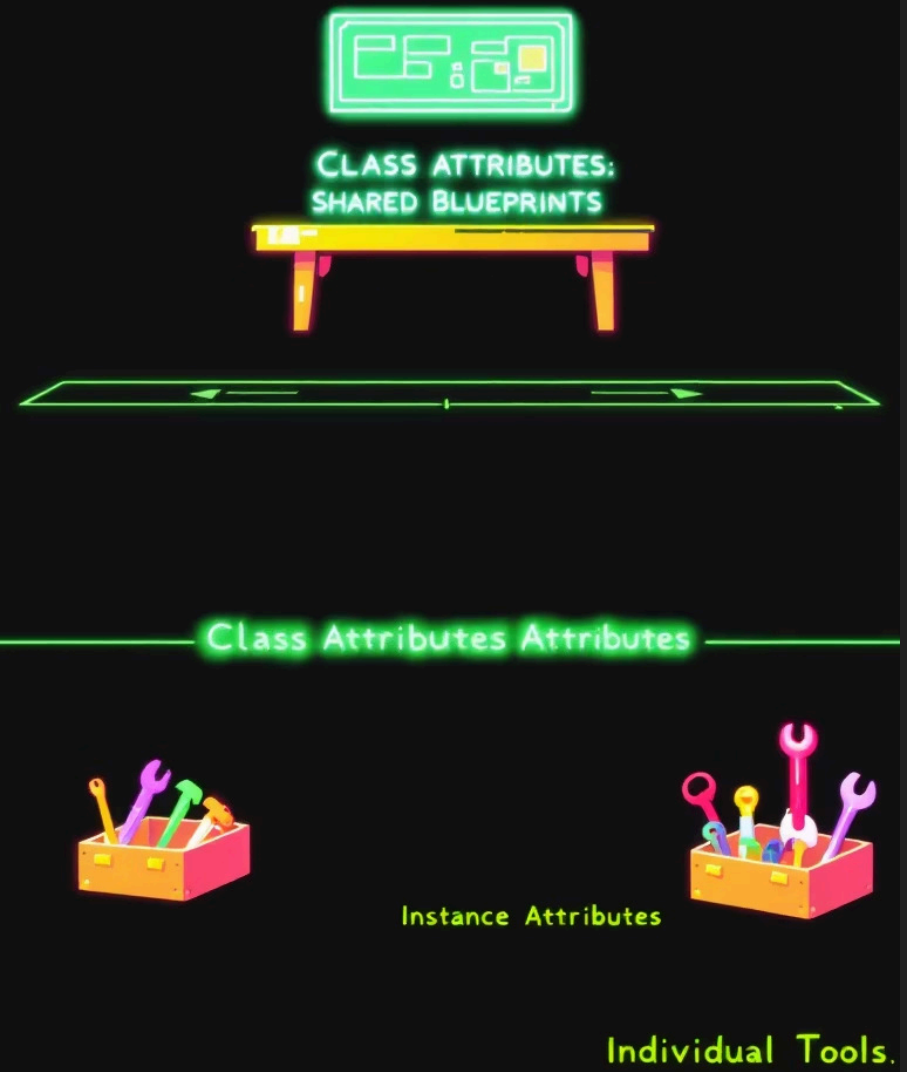
Inizializzazione

Definisce e assegna valori agli attributi di istanza

Completamento

Non ritorna valori (implicitamente ritorna None)

Il metodo __init__() è un metodo speciale chiamato costruttore che viene eseguito automaticamente quando si crea una nuova istanza della classe. Il suo scopo principale è inizializzare gli attributi dell'oggetto con valori specifici, garantendo che ogni nuovo oggetto abbia uno stato iniziale valido.



Attributi di Classe vs Attributi di Istanza

Caratteristica	Attributi di Classe	Attributi di Istanza
Definizione	Definiti nel corpo della classe	Definiti nei metodi (in genere in __init__)
Memoria	Condivisi tra tutte le istanze	Unici per ogni istanza
Accesso	NomeClasse.attributo o self.attributo	self.attributo
Uso tipico	Costanti, contatori, configurazioni	Stato specifico dell'oggetto

Gli attributi di classe sono variabili condivise tra tutte le istanze di una classe, utili per memorizzare informazioni comuni o valori costanti. Vengono definiti direttamente nel corpo della classe, fuori da qualsiasi metodo.

Gli attributi di istanza, invece, sono specifici per ogni oggetto creato e vengono tipicamente definiti nel metodo __init__(). Rappresentano lo stato individuale di ciascun oggetto.

Metodi Speciali in Python

__str__(self)

Definisce la rappresentazione leggibile dell'oggetto quando viene usato str() o print(). Dovrebbe restituire una stringa informativa e comprensibile per l'utente finale.

__repr__(self)

Fornisce una rappresentazione "ufficiale" dell'oggetto, idealmente una stringa che potrebbe essere usata per ricreare l'oggetto. Usata dagli sviluppatori per il debugging.

__add__(self, other)

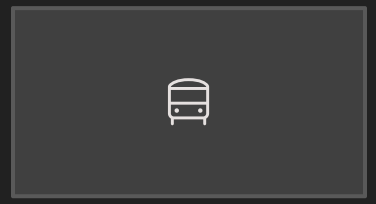
Permette di definire il comportamento dell'operatore + tra istanze della classe. Consente di implementare operazioni personalizzate.

__eq__(self, other)

Implementa il controllo di uguaglianza (==) tra oggetti. Definisce quando due istanze della classe sono considerate uguali.

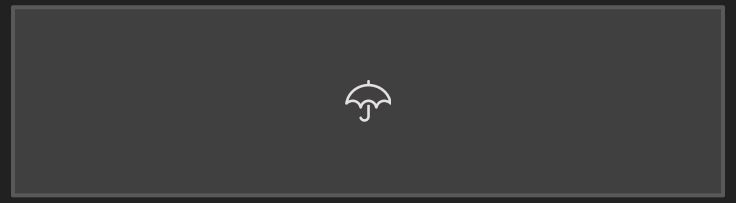
I metodi speciali, anche chiamati "metodi magici" o "dunder methods" (da double underscore), permettono di integrare le classi personalizzate con le funzionalità built-in di Python, rendendo gli oggetti più intuitivi da usare.

Incapsulamento e Attributi Privati



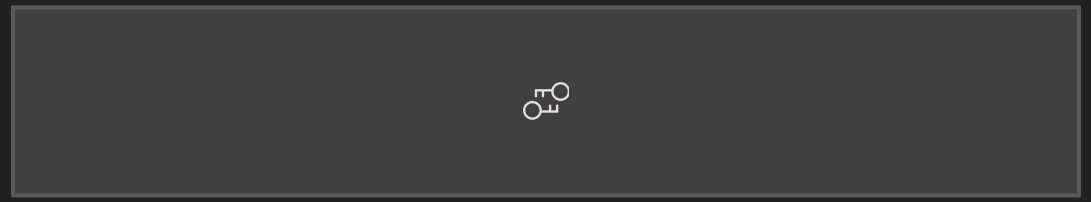
Attributi pubblici

Accessibili liberamente da ovunque



Attributi protetti (_nome)

Convenzione per indicare uso interno



Attributi privati (__nome)

Name mangling per limitare l'accesso

L'incapsulamento è un principio della OOP che nasconde i dettagli implementativi di una classe, esponendo solo le interfacce necessarie. In Python, a differenza di altri linguaggi, l'incapsulamento è basato principalmente su convenzioni anziché su restrizioni rigide.

Gli attributi preceduti da un singolo underscore (_attributo) sono considerati "protetti" per convenzione, indicando che dovrebbero essere usati solo all'interno della classe e delle sue sottoclassi. Gli attributi con doppio underscore (__attributo) subiscono il "name mangling", rendendo più difficile l'accesso diretto dall'esterno.

Ereditarietà: Concetti Fondamentali

Riuso del codice

Evita duplicazioni implementando funzionalità comuni

Gerarchia di classi

Organizzazione in relazioni parent-child

Estensione delle funzionalità

Aggiunge comportamenti specifici alle sottoclassi

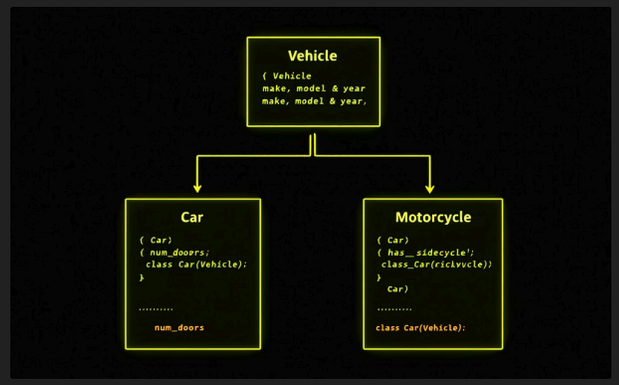
Polimorfismo

Utilizzo uniforme di oggetti di classi diverse

L'ereditarietà è un meccanismo fondamentale della OOP che permette di definire una nuova classe basata su una classe esistente. La nuova classe (sottoclasse) eredita attributi e metodi dalla classe esistente (superclasse) e può estenderli o modificarli secondo necessità.

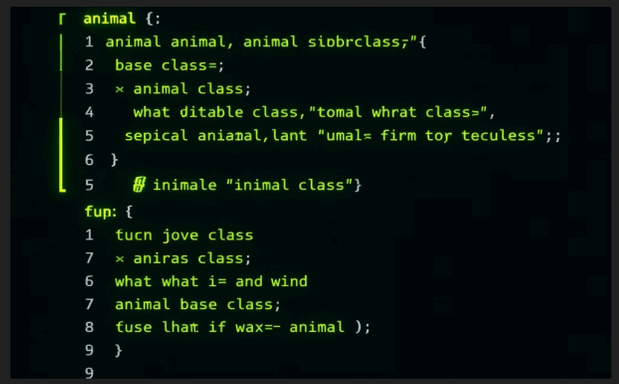
Questo approccio promuove il riutilizzo del codice e consente di creare gerarchie di classi che riflettono relazioni "è un tipo di" del mondo reale, come "un gatto è un tipo di animale".

Classe Genitore e Sottoclasse



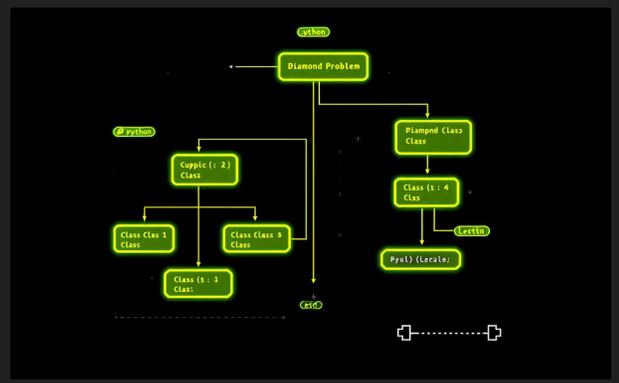
Sintassi dell'Ereditarietà

In Python, l'ereditarietà si implementa specificando la classe genitore tra parentesi nella definizione della sottoclasse. Questa semplice notazione stabilisce una relazione gerarchica tra le classi, consentendo alla sottoclasse di ereditare tutti gli attributi e i metodi della classe genitore.



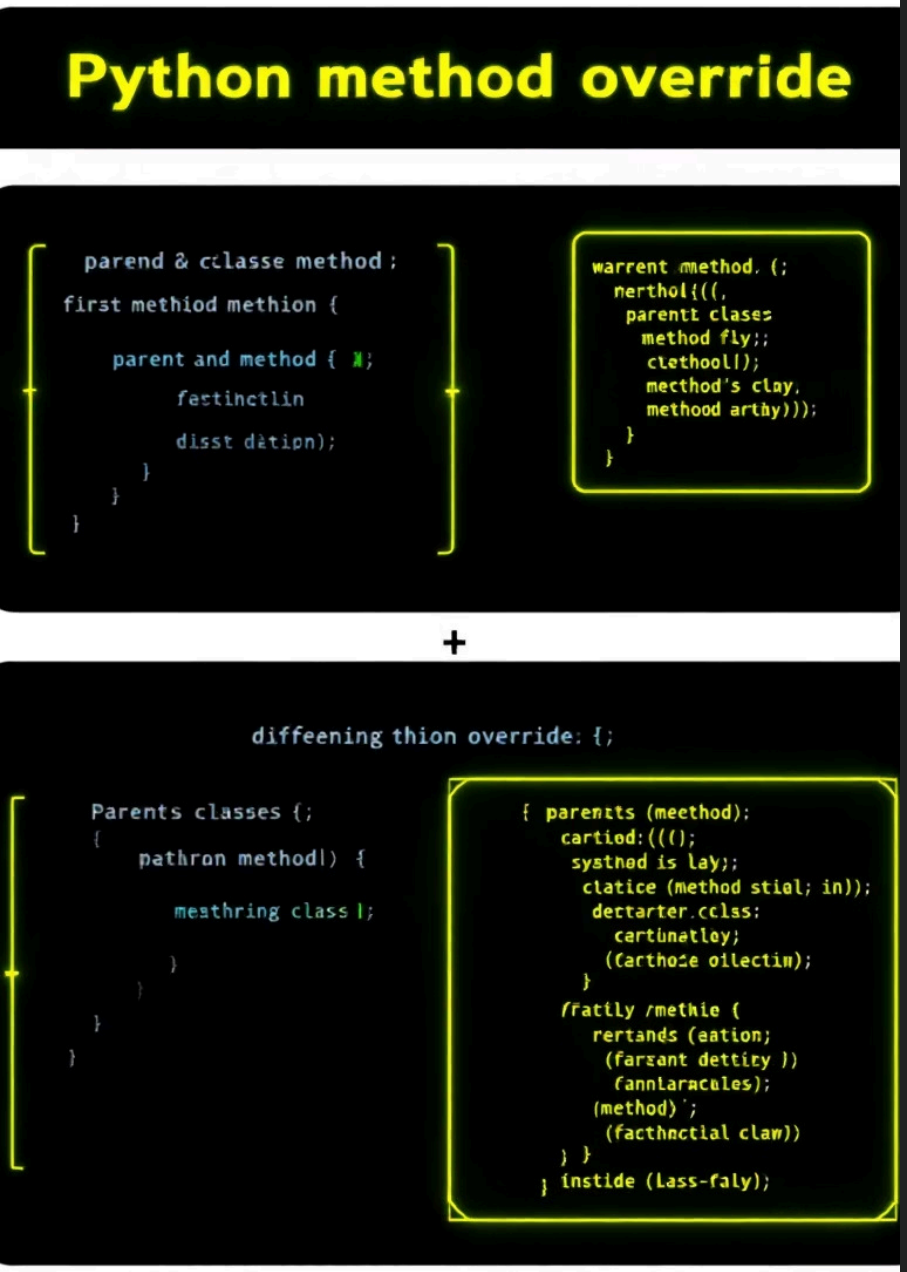
Ereditarietà in Azione

Quando si crea un'istanza di una sottoclasse, questa avrà accesso a tutti gli attributi e metodi definiti sia nella propria classe che nella classe genitore. Questo meccanismo permette di creare sistemi complessi partendo da componenti semplici e riutilizzabili.



Ereditarietà Multipla

Python supporta anche l'ereditarietà multipla, permettendo a una classe di ereditare da più classi genitori. Questo offre grande flessibilità ma richiede attenzione per evitare conflitti tra metodi con lo stesso nome provenienti da diverse classi genitori.



Override dei Metodi

Ridefinizione

Una sottoclasse può implementare una propria versione di un metodo ereditato dalla classe genitore, mantenendo lo stesso nome ma modificando il comportamento.

Estensione

È possibile estendere il comportamento del metodo originale aggiungendo funzionalità specifiche della sottoclasse, preservando parte del comportamento del genitore.

Polimorfismo

L'override permette di usare lo stesso nome di metodo per comportamenti diversi, consentendo di trattare oggetti di classi diverse in modo uniforme.

L'override dei metodi è una tecnica potente che consente di specializzare il comportamento delle sottoclassi. Quando una sottoclasse ridefinisce un metodo ereditato dalla superclasse, il metodo ridefinito viene chiamato automaticamente per le istanze della sottoclasse, mentre il metodo originale continua a essere utilizzato per le istanze della classe genitore.

Utilizzo della Funzione super()

1

Accesso ai Metodi del Genitore

super() fornisce un riferimento alla classe genitore, permettendo di chiamare i suoi metodi dall'interno della sottoclasse.

2

Estensione vs Sostituzione

Consente di estendere il comportamento di un metodo anziché sostituirlo completamente, combinando funzionalità nuove e esistenti.

3

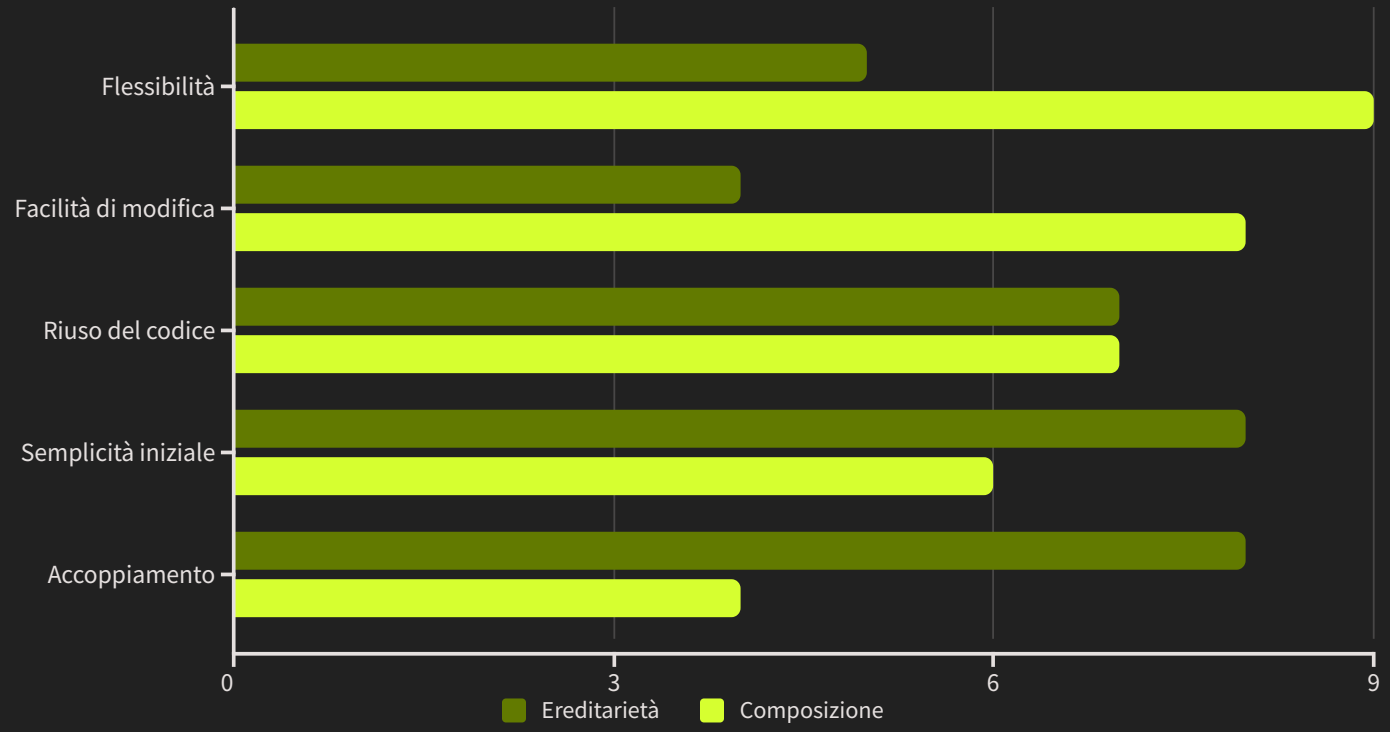
Gestione dell'Ereditarietà Multipla

In caso di ereditarietà multipla, super() segue l'ordine MRO (Method Resolution Order) per determinare quale classe genitore chiamare.

La funzione super() è uno strumento fondamentale per gestire l'ereditarietà in Python. Invece di riferirsi direttamente alla classe genitore per nome, super() offre un approccio più flessibile che funziona correttamente anche in scenari complessi come l'ereditarietà multipla.

È particolarmente utile nel metodo __init__() delle sottoclassi per garantire che l'inizializzazione della classe genitore venga eseguita correttamente prima di aggiungere funzionalità specifiche della sottoclasse.

Composizione vs Ereditarietà



L'ereditarietà ("è un") crea una relazione gerarchica tra le classi, dove una sottoclasse è un tipo specializzato della classe genitore. La composizione ("ha un") invece inserisce istanze di altre classi come attributi, creando relazioni tra oggetti piuttosto che tra classi.

Mentre l'ereditarietà può sembrare più intuitiva inizialmente, la composizione offre maggiore flessibilità e minore accoppiamento. Il principio "preferisci la composizione all'ereditarietà" suggerisce di utilizzare la composizione come approccio predefinito, ricorrendo all'ereditarietà solo quando esiste una chiara relazione "è un tipo di".

Progettare Classi da Specifiche

Analisi dei Requisiti

Identificare le entità del dominio e le loro proprietà. Determinare quali attributi e comportamenti dovrebbero avere le classi in base alle specifiche funzionali.

Modellazione delle Relazioni

Stabilire come le classi si relazionano tra loro. Decidere quali relazioni implementare tramite ereditarietà ("è un") e quali tramite composizione ("ha un").

Implementazione e Raffinamento

Codificare le classi con i loro attributi e metodi. Testare il sistema e raffinare il design in base ai feedback, applicando i principi SOLID per garantire un codice manutenibile.

La progettazione efficace delle classi richiede un equilibrio tra astrazione e praticità. Un buon design OOP dovrebbe riflettere il dominio del problema in modo intuitivo, nascondere i dettagli implementativi non necessari e facilitare l'estensione futura del sistema.

Ricordate che la prima versione raramente è perfetta: la progettazione delle classi è un processo iterativo che migliora con l'esperienza e il feedback derivante dall'uso reale del sistema.