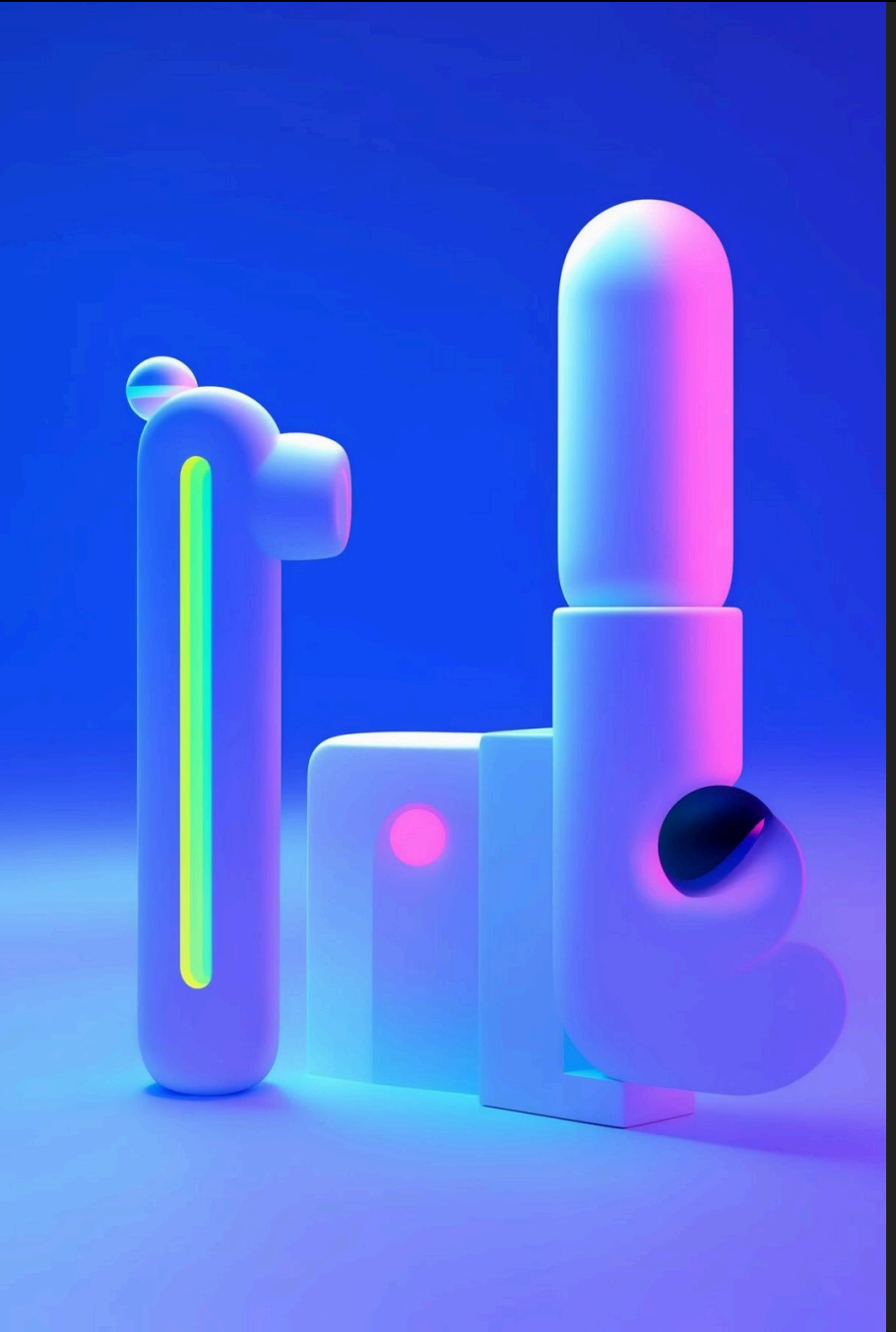


Introduzione alla Programmazione Orientata agli Oggetti

Benvenuti al corso introduttivo sulla Programmazione Orientata agli Oggetti (OOP). Questo percorso didattico è pensato per chi non ha mai programmato ma desidera comprendere uno dei paradigmi più potenti e diffusi nel mondo dello sviluppo software.

La OOP è un approccio alla programmazione che utilizza "oggetti" per modellare entità reali o astratte. Nei prossimi incontri esploreremo i concetti fondamentali di questo paradigma, partendo dalle basi fino ad arrivare a tecniche più avanzate come l'ereditarietà e la composizione.

Questo corso non richiede conoscenze pregresse di programmazione, ma solo curiosità e voglia di imparare. Siete pronti a scoprire come il mondo reale può essere modellato attraverso il codice?

Programmazione Procedurale vs Orientata agli Oggetti

Programmazione Procedurale

Si basa su procedure o funzioni che elaborano dati. I dati e le funzioni sono entità separate. Il programma è una sequenza di istruzioni eseguite in ordine.

Il focus è sulle azioni da eseguire e sul flusso di esecuzione. I dati sono passati alle procedure come parametri.

Programmazione Orientata agli Oggetti

Unisce dati e funzionalità in "oggetti". Gli oggetti sono istanze di "classi" che definiscono struttura e comportamento.

Il focus è sugli oggetti che interagiscono tra loro. I dati e i metodi per manipolarli sono incapsulati insieme all'interno degli oggetti.

La differenza principale tra i due approcci sta nel modo di organizzare il codice. Mentre nella programmazione procedurale il programma è una serie di istruzioni che manipolano dati, nella OOP il programma è una collezione di oggetti che comunicano tra loro attraverso messaggi (chiamate a metodi).

Definizione di una Classe

Cos'è una Classe?

Una classe è un modello o uno schema che definisce le caratteristiche (attributi) e i comportamenti (metodi) che gli oggetti di quel tipo avranno. È come un "progetto" per creare oggetti.

Sintassi in Python

In Python, una classe si definisce con la parola chiave "class" seguita dal nome della classe (in CamelCase per convenzione) e due punti. Il corpo della classe è indentato.

Componenti di una Classe

Una classe contiene attributi (variabili) che rappresentano le caratteristiche e metodi (funzioni) che rappresentano i comportamenti o le azioni che gli oggetti possono eseguire.

Le classi sono fondamentali nella OOP perché forniscono un meccanismo per strutturare i programmi in modo che proprietà e comportamenti siano raggruppati in singole unità. Questo rende il codice più organizzato, riutilizzabile e più facile da mantenere.

Istanziare Oggetti

Definizione della Classe

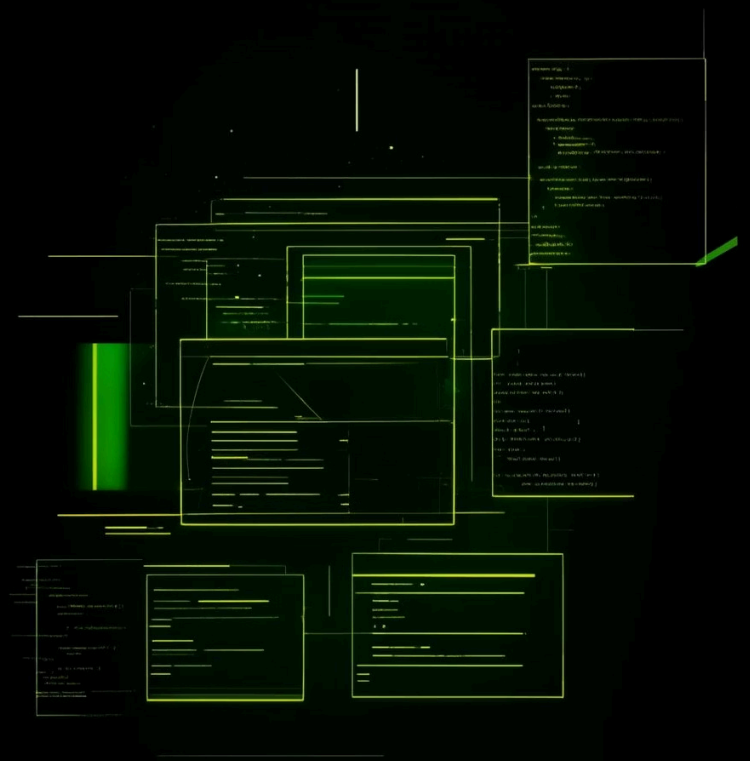
Prima si crea il "progetto" (la classe) con attributi e metodi

Istanziazione

Si crea un oggetto specifico chiamando il nome della classe

Utilizzo dell'Oggetto

Si accede agli attributi e si invocano i metodi dell'oggetto



Istanziare un oggetto significa creare un'istanza concreta di una classe. È come usare uno stampo (la classe) per creare oggetti reali. In Python, questo processo è semplice: basta chiamare il nome della classe come se fosse una funzione.

Per esempio, se abbiamo una classe chiamata **Automobile**, possiamo creare un'istanza scrivendo **mia_auto = Automobile()**. Ora **mia_auto** è un oggetto che possiede tutti gli attributi e i metodi definiti nella classe **Automobile**.

Attributi di Istanza e Metodi

Attributi di Istanza

Sono variabili che appartengono a un singolo oggetto (istanza). Ogni oggetto ha la propria copia di questi attributi, che possono avere valori diversi da oggetto a oggetto.

Metodi

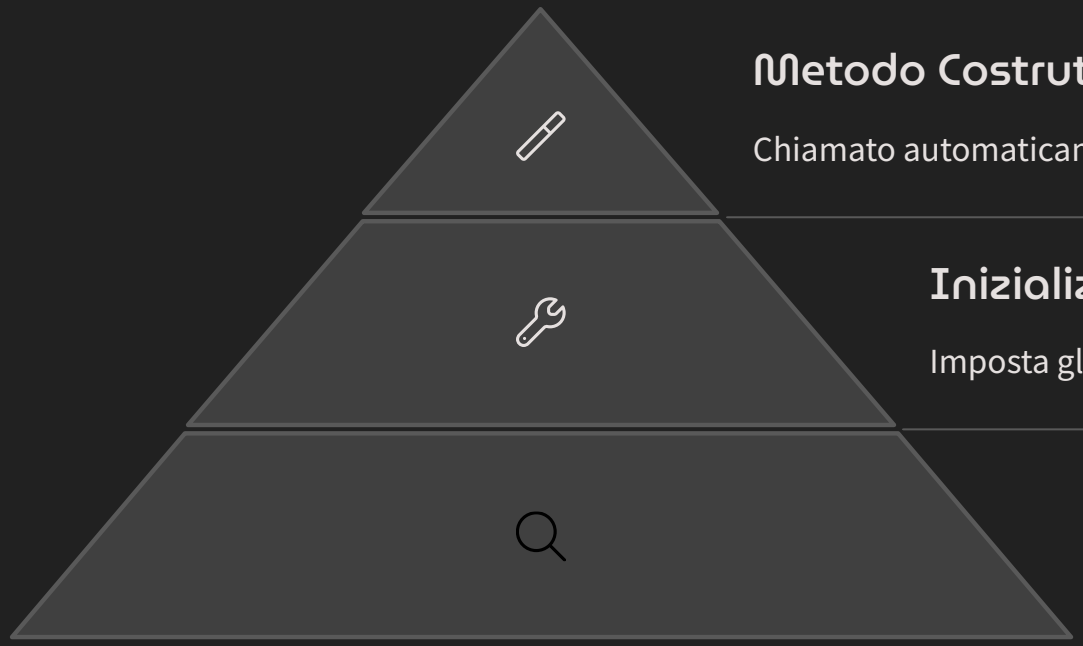
Sono funzioni definite all'interno di una classe che descrivono i comportamenti degli oggetti. I metodi possono accedere e modificare gli attributi dell'oggetto su cui vengono chiamati.

Self

È un parametro speciale che si riferisce all'istanza stessa. Deve essere il primo parametro di ogni metodo di istanza e permette di accedere agli attributi e ad altri metodi dell'oggetto.

Gli attributi di istanza memorizzano dati specifici per ogni oggetto, mentre i metodi definiscono cosa un oggetto può fare. Quando si definisce un metodo, il parametro **self** rappresenta l'istanza su cui il metodo viene chiamato, permettendo di accedere agli attributi di quell'istanza specifica.

Il Metodo __init__()



Metodo Costruttore

Chiamato automaticamente alla creazione dell'oggetto

Inizializzazione

Imposta gli attributi iniziali dell'oggetto

Parametri

Può accettare valori per personalizzare l'oggetto

Il metodo **__init__()** è un metodo speciale in Python, chiamato costruttore. Viene eseguito automaticamente quando si crea una nuova istanza di una classe. Il suo scopo principale è inizializzare gli attributi dell'oggetto con valori specifici.

Ad esempio, se stiamo creando una classe **Persona**, il metodo **__init__()** potrebbe accettare parametri come nome ed età, e assegnarli agli attributi dell'istanza appena creata. Questo garantisce che ogni nuovo oggetto abbia immediatamente tutti gli attributi necessari.

Attributi di Classe vs Attributi di Istanza

Attributi di Classe

Appartengono alla classe stessa e sono condivisi da tutte le istanze. Si definiscono direttamente nella classe, fuori da qualsiasi metodo. Sono utili per valori che dovrebbero essere gli stessi per tutti gli oggetti.

Attributi di Istanza

Appartengono a singole istanze e possono avere valori diversi per ogni oggetto. Si definiscono solitamente nel metodo __init__() usando self. Rappresentano le caratteristiche uniche di ciascun oggetto.

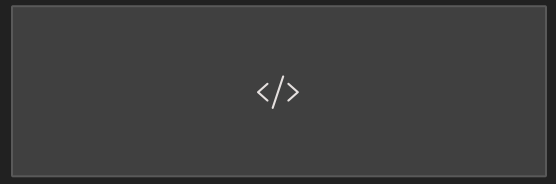
La differenza fondamentale è che gli attributi di classe sono condivisi tra tutte le istanze, mentre gli attributi di istanza sono unici per ogni oggetto. Modificare un attributo di classe influisce su tutte le istanze, mentre modificare un attributo di istanza influisce solo sull'oggetto specifico.

Un esempio pratico: in una classe **Auto**, il numero di ruote potrebbe essere un attributo di classe (tutte le auto hanno 4 ruote), mentre il colore sarebbe un attributo di istanza (ogni auto può avere un colore diverso).

Metodi Speciali

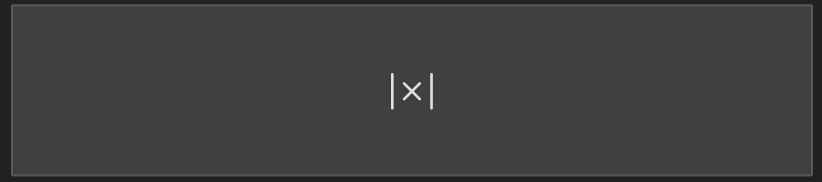
__str__()

Definisce la rappresentazione leggibile dell'oggetto



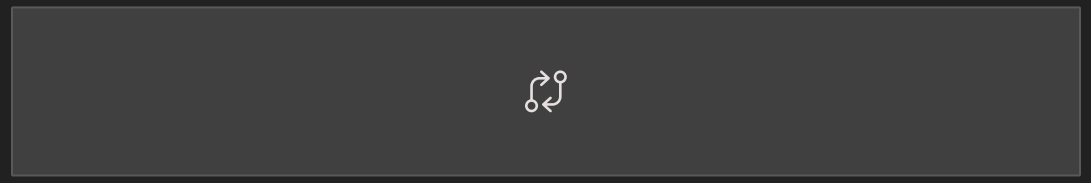
__repr__()

Definisce la rappresentazione tecnica dell'oggetto



Operatori matematici

__add__(), __sub__(), __mul__(), ecc.



Confronti

__eq__(), __lt__(), __gt__(), ecc.

I metodi speciali, anche chiamati "dunder methods" (da "double underscore"), permettono di definire comportamenti specifici per le operazioni integrate di Python. Questi metodi vengono chiamati automaticamente in risposta a determinate operazioni.

Per esempio, il metodo **__str__()** viene chiamato quando si usa la funzione **str()** su un oggetto o quando lo si stampa con **print()**. Il metodo **__add__()** viene chiamato quando si usa l'operatore **+** tra due oggetti. Questi metodi permettono di personalizzare il comportamento degli oggetti nelle operazioni di base.

Incapsulamento: _attributo

Concetto di Incapsulamento

L'incapsulamento è il principio di nascondere i dettagli interni di un oggetto e fornire un'interfaccia controllata per interagire con esso.

Attributi con Underscore

In Python, un attributo preceduto da un underscore singolo (_attributo) è considerato "protetto" per convenzione. Questo indica che l'attributo è pensato per uso interno.

Getter e Setter

Per accedere e modificare attributi protetti, si usano spesso metodi dedicati (getter e setter) che possono includere validazioni e controlli.

L'incapsulamento aiuta a mantenere l'integrità dei dati impedendo accessi diretti che potrebbero compromettere lo stato dell'oggetto. In Python, a differenza di altri linguaggi, l'incapsulamento è più una convenzione che un meccanismo obbligatorio.

Quando si vede un attributo con underscore singolo (_nome), si dovrebbe considerarlo come parte dell'implementazione interna che potrebbe cambiare. L'accesso a tali attributi dovrebbe avvenire attraverso metodi pubblici che forniscono un'interfaccia stabile.

Ereditarietà: Concetti Base

Classe Genitore

Definisce attributi e metodi base

Classe Figlia

Eredita tutte le caratteristiche del genitore

Ridefinizione

Può modificare comportamenti ereditati

Estensione

Aggiunge nuove funzionalità

L'ereditarietà è un concetto fondamentale della OOP che permette di creare nuove classi basate su classi esistenti. La nuova classe (sottoclasse) eredita tutti gli attributi e i metodi della classe genitore (superclasse), e può aggiungerne di nuovi o modificare quelli ereditati.

Questo meccanismo favorisce il riutilizzo del codice e permette di creare gerarchie di classi che modellano relazioni "is-a" (è un). Ad esempio, una classe **Cane** potrebbe ereditare da una classe **Animale**, poiché un cane è un animale.

Classe Genitore e Sottoclasse

Definizione della Classe Base

Si crea una classe che contiene attributi e metodi comuni

Dichiarazione dell'Ereditarietà

Si crea una sottoclasse specificando la classe genitore tra parentesi

Accesso agli Elementi Ereditati

La sottoclasse può utilizzare tutti gli attributi e metodi della classe genitore

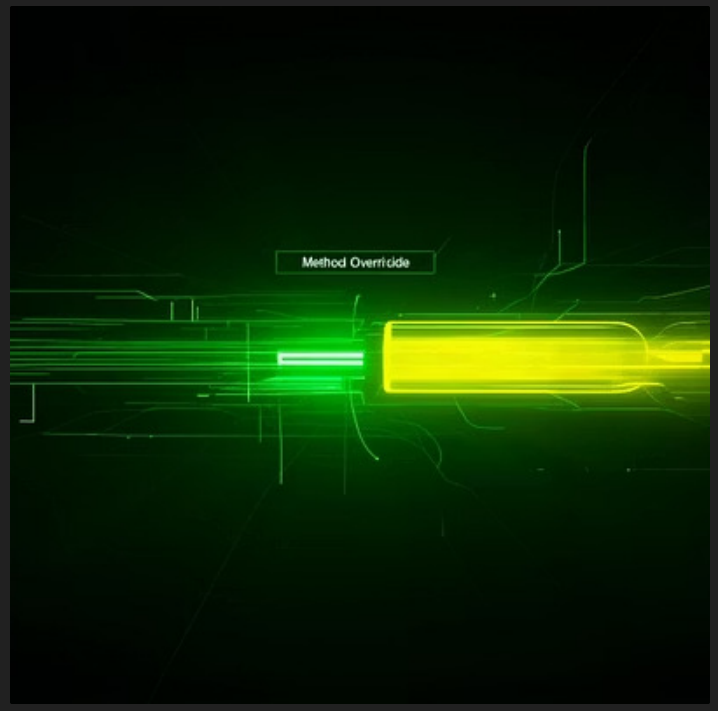
Personalizzazione

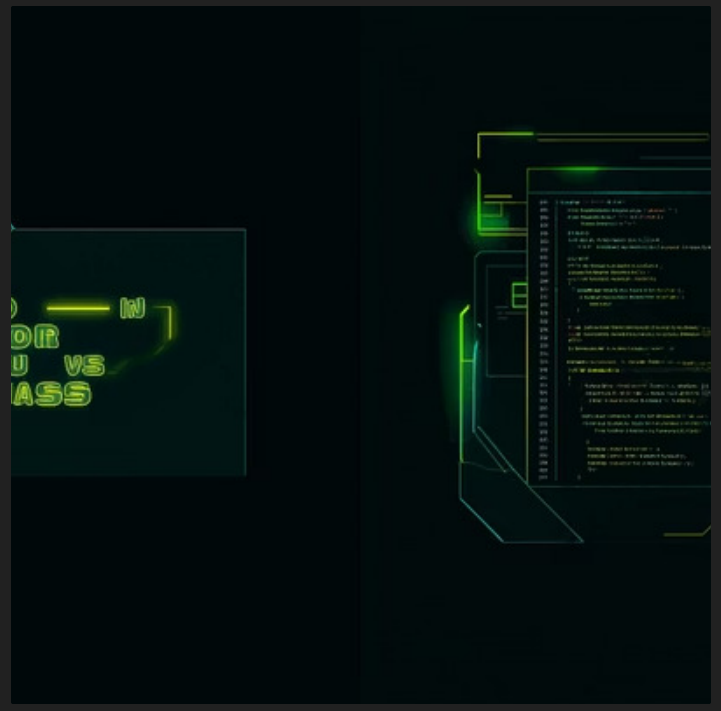
La sottoclasse può aggiungere attributi e metodi specifici

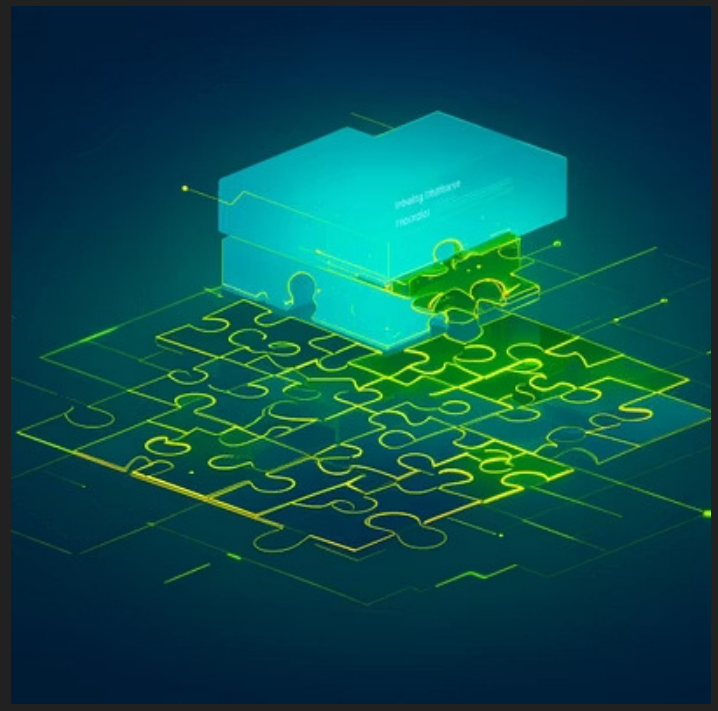
In Python, l'ereditarietà si implementa semplicemente specificando la classe genitore tra parentesi quando si definisce la sottoclasse. Ad esempio: **class Cane(Animale):** crea una classe **Cane** che eredita dalla classe **Animale**.

La sottoclasse eredita automaticamente tutti gli attributi e i metodi della classe genitore, permettendo di riutilizzare il codice esistente. Questo è particolarmente utile quando si hanno più classi che condividono caratteristiche comuni, evitando la duplicazione del codice.

Override di Metodi







Identificare il Metodo da Sovrascrivere

Si individua un metodo della classe genitore che necessita di un comportamento diverso nella sottoclasse.

Ridefinire il Metodo nella Sottoclasse

Si crea un metodo con lo stesso nome e gli stessi parametri nella sottoclasse. Questo metodo sostituirà quello ereditato.

Implementare il Nuovo Comportamento

Si scrive il codice del metodo con la nuova implementazione specifica per la sottoclasse.

L'override (sovrascrittura) dei metodi è una caratteristica potente dell'ereditarietà che permette di modificare il comportamento di un metodo ereditato. Quando una sottoclasse definisce un metodo con lo stesso nome di un metodo della classe genitore, il metodo della sottoclasse prevale (sovrascrive) quello del genitore.

Uso di super()

1

Accesso ai Metodi del Genitore

La funzione super() permette di chiamare metodi della classe genitore dalla sottoclasse, anche dopo averli sovrascritti

2

Estensione del Comportamento

Consente di aggiungere funzionalità al metodo del genitore invece di sostituirlo completamente

3

Costruttori a Catena

Permette di chiamare il costruttore della classe genitore prima di aggiungere inizializzazioni specifiche della sottoclasse

La funzione **super()** è uno strumento fondamentale quando si lavora con l'ereditarietà. Permette a una sottoclasse di accedere ai metodi della classe genitore, anche dopo averli sovrascritti. Questo è particolarmente utile quando si vuole estendere il comportamento di un metodo ereditato invece di sostituirlo completamente.

Un uso comune di **super()** è nel metodo **__init__()** delle sottoclassi. Chiamando **super().__init__()**, la sottoclasse può garantire che l'inizializzazione della classe genitore venga eseguita prima di aggiungere le proprie inizializzazioni specifiche.

Composizione vs Ereditarietà

Ereditarietà (is-a)

Relazione "è un". Una sottoclasse è un tipo specializzato della classe genitore.

Esempio: Un **Cane** è un **Animale**.

Vantaggi: Riutilizzo del codice, rappresentazione naturale di gerarchie tassonomiche.

Svantaggi: Accoppiamento stretto tra classi, potenziali problemi con gerarchie profonde.

Composizione (has-a)

Relazione "ha un". Una classe contiene istanze di altre classi come suoi attributi.

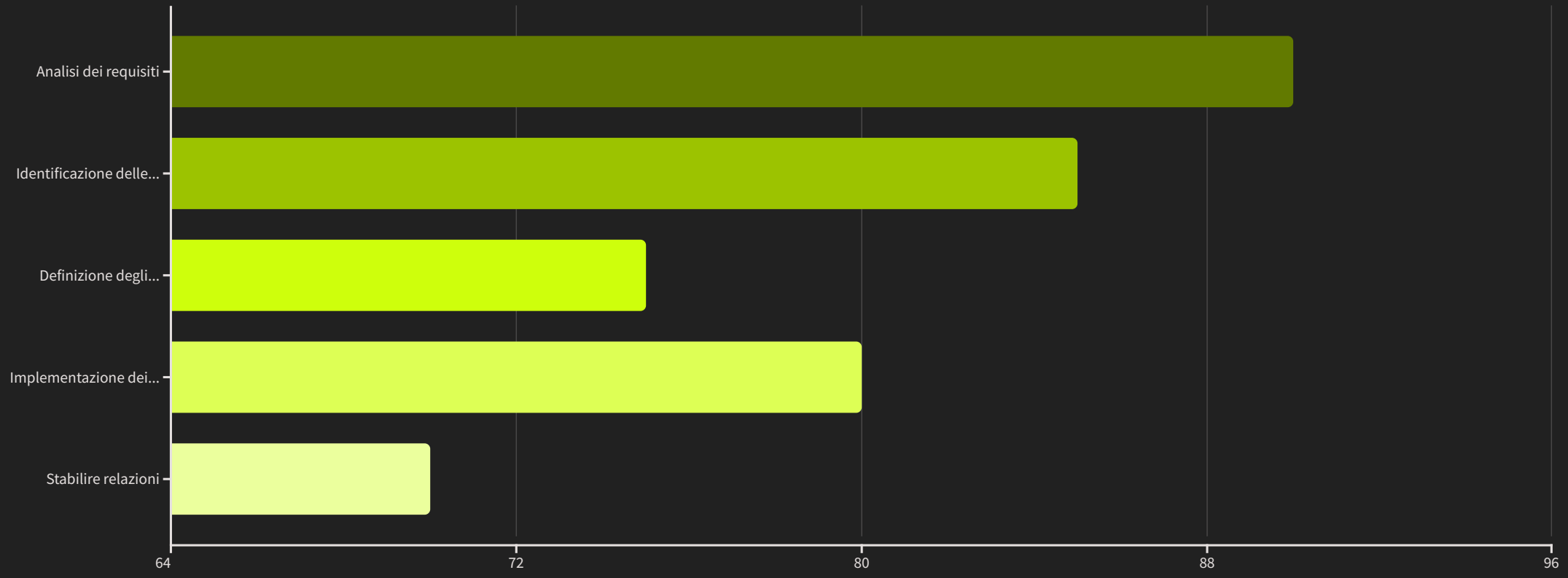
Esempio: Un'**Auto** ha un **Motore**.

Vantaggi: Maggiore flessibilità, accoppiamento più debole, più facile da modificare.

Svantaggi: Può richiedere più codice per delegare funzionalità.

Mentre l'ereditarietà crea una relazione gerarchica dove una classe è un tipo specifico di un'altra, la composizione crea una relazione dove una classe contiene istanze di altre classi. Entrambi sono strumenti potenti, ma servono scopi diversi.

Progettare Classi da Specifiche



Progettare classi efficaci richiede un approccio metodico. Si parte dall'analisi dei requisiti per comprendere il dominio del problema. Si identificano poi le entità principali che diventeranno classi, determinando per ciascuna gli attributi (dati) e i metodi (comportamenti) necessari.

È importante considerare le relazioni tra le classi: ereditarietà per relazioni "is-a" e composizione per relazioni "has-a". Una buona progettazione OOP dovrebbe riflettere la struttura del problema reale, mantenendo le classi coese (focalizzate su un singolo scopo) e poco accoppiate (minimizzando le dipendenze tra classi).