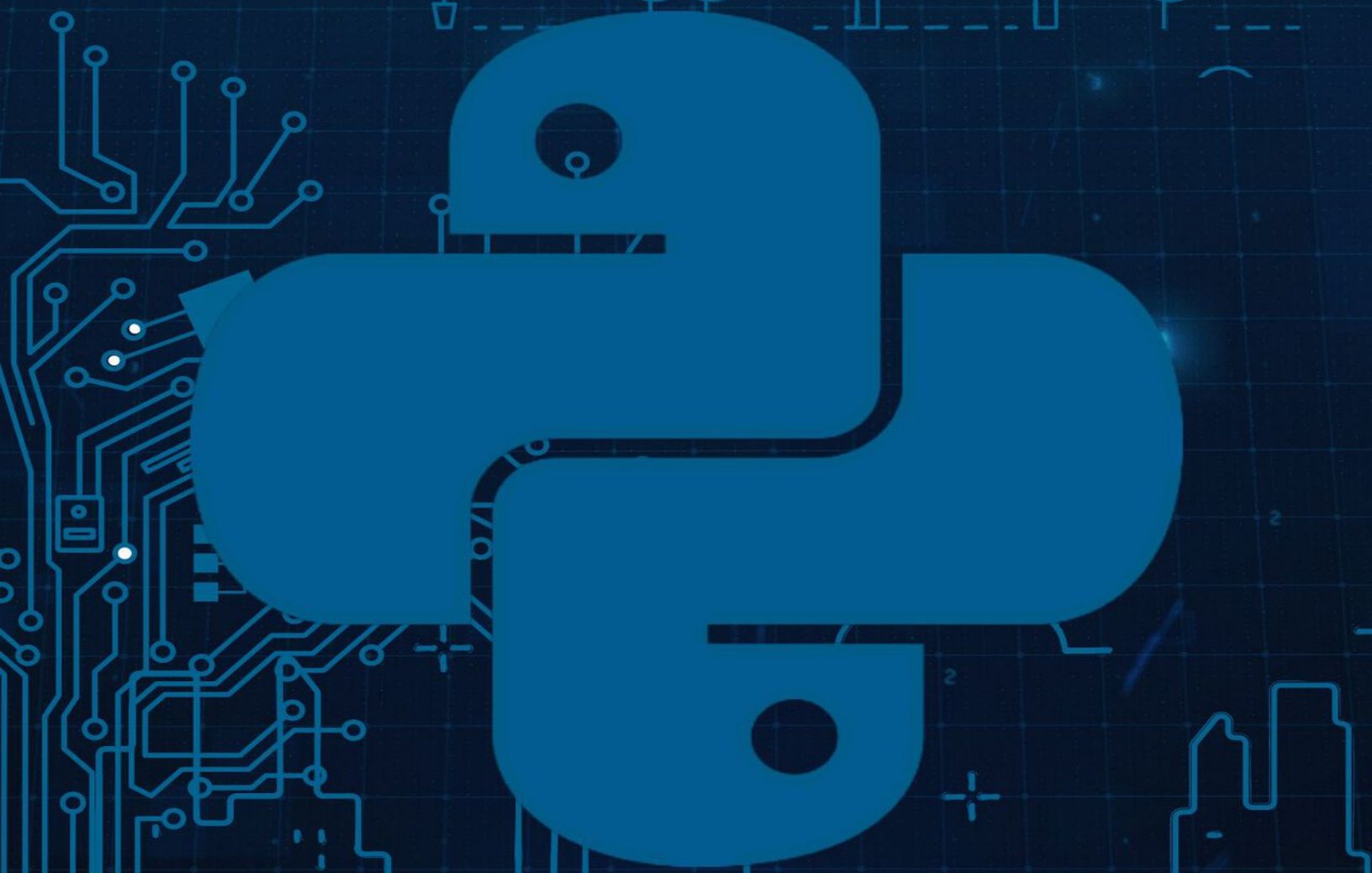
L'ARTE DELLA PROGRAMMAZIONE

PYTHON

Un Percorso Completo per Combinare la Teoria con l'Applicazione Pratica e Diventare un Esperto, Partendo da Zero



Cristian Tesconi

L’ARTE DELLA PROGRAMMAZIONE PYTHON

Un Percorso Completo per Combinare la Teoria
con l'Applicazione Pratica e
Diventare un Esperto,
Partendo da Zero

Cristian Tesconi

Autore

L'autore di questo libro è un ingegnere dell'automazione con una vasta esperienza nel settore automobilistico, dove ha ricoperto una varietà di ruoli che gli hanno fornito una conoscenza approfondita e una vasta competenza nel campo.

Durante la sua carriera, egli ha lavorato allo sviluppo di algoritmi per la guida autonoma, ha sperimentato soluzioni innovative e ha collaborato con team multidisciplinari per creare sistemi avanzati e sicuri.

Inoltre, l'autore ha acquisito una solida esperienza nello sviluppo di applicazioni embedded nell'ambito della telematica. Ha lavorato su progetti che coinvolgono la comunicazione tra veicoli, la gestione dei dati e l'interfacciamento con sistemi esterni. La sua competenza in questo campo lo ha reso consapevole delle sfide e delle opportunità offerte dalla connettività e dalla digitalizzazione nell'industria automobilistica.

Un'altra area di specializzazione riguarda la simulazione di sistemi multi-fisici. Ha sviluppato applicazioni desktop che consentono la modellazione e la simulazione di sistemi, integrando diverse discipline ingegneristiche. La sua esperienza in questo campo lo ha portato a comprendere l'importanza dell'accuratezza e dell'efficienza nella progettazione e nella valutazione di sistemi complessi.

L'autore ha anche contribuito in modo significativo allo sviluppo di soluzioni automatizzate. Ha applicato la sua conoscenza della programmazione e dell'automazione per semplificare processi complessi e migliorare l'efficienza operativa. Ha sviluppato strumenti personalizzati e ha collaborato con team per implementare soluzioni automatizzate in diversi contesti.

Con la combinazione di questa vasta esperienza nel settore automotive, la conoscenza approfondita della programmazione *Python* e l'esperienza pratica nello sviluppo di soluzioni sofisticate, l'autore si impegna a condividere le sue conoscenze e competenze attraverso questo libro. La sua passione per la programmazione e il desiderio di aiutare gli altri nello sviluppo delle loro abilità lo hanno spinto a creare una risorsa completa che guiderà i lettori dal livello base all'avanzato nella programmazione *Python*, fornendo esempi pratici e approfondimenti tecnici.

L'autore spera che questo libro sia uno strumento prezioso per gli appassionati di programmazione *Python*, studenti, professionisti e chiunque sia interessato ad approfondire le proprie competenze nel campo della programmazione.

Prefazione

Benvenuti nell'affascinante mondo della programmazione *Python*! Questo libro è stato concepito come una guida completa per coloro che desiderano imparare a programmare in *Python*, partendo dalle basi fino ad arrivare a livelli avanzati. Con un approccio pratico e orientato all'apprendimento, questo testo ti condurrà attraverso i fondamenti del linguaggio e ti guiderà verso la padronanza delle sue potenti funzionalità.

La scelta di *Python* come linguaggio di programmazione per questo libro è stata dettata dalla sua popolarità e versatilità. Esso è infatti ampiamente utilizzato sia dagli sviluppatori professionisti che dai principianti, grazie alla sua sintassi intuitiva e leggibile, alla vasta gamma di librerie disponibili e alla sua ampia applicazione in diversi settori, come l'intelligenza artificiale, lo sviluppo web, l'automazione di processi e molto altro ancora.

Questo libro è stato strutturato per guidarti attraverso un percorso di apprendimento graduale. Inizieremo dalle nozioni di base della programmazione, come le variabili, gli operatori e i controlli di flusso, per poi approfondire argomenti più avanzati, come le funzioni, le classi e l'elaborazione dei dati. Lungo il percorso, affronteremo esempi pratici che ti aiuteranno a comprendere come applicare i concetti appresi a situazioni reali.

Una delle caratteristiche distintive di questo libro è l'attenzione dedicata agli esempi pratici. Ogni concetto teorico verrà illustrato con codice *Python* reale, in modo da consentirti di sperimentare e acquisire familiarità con le pratiche comuni della programmazione. Sarai incoraggiato a scrivere il codice, eseguirlo e sperimentare con esso per approfondire la tua comprensione e acquisire sicurezza nel tuo percorso di apprendimento.

Tuttavia, la programmazione non è solo una questione di conoscere la sintassi del linguaggio; è anche un'arte che richiede la capacità di risolvere problemi in modo creativo e di scrivere codice pulito ed efficiente. Pertanto, oltre a fornire linee guida e spiegazioni esaustive, questo libro ti insegnerà anche buone pratiche di programmazione, metodi di debug e suggerimenti per migliorare le tue abilità di problem solving.

So che il percorso per diventare un programmatore esperto può sembrare impegnativo, ma sono fiducioso che, con impegno e dedizione, riuscirai a

padroneggiare la programmazione *Python*. Che tu sia un principiante assoluto o un programmatore alle prime armi che desidera consolidare le proprie conoscenze, questo libro è stato progettato per accompagnarti in ogni fase del tuo percorso di apprendimento.

Non vedo l'ora di essere tuo compagno di viaggio in questa avventura nel mondo della programmazione Python. Buon apprendimento e buon divertimento!

Cosa copre questo libro

Questo libro copre una vasta gamma di argomenti per guidarti dai concetti base fino a livelli avanzati.

Ogni capitolo inizia con una spiegazione teorica dei concetti fondamentali, seguita da esempi di codice che illustrano come utilizzare questi concetti nella pratica. Gli esempi sono progettati per essere chiari, comprensibili e facilmente eseguibili. Potete digitare gli esempi nel vostro ambiente di sviluppo *Python* o eseguirli direttamente dalla shell interattiva.

Ti consiglio di leggere attentamente la parte teorica, quindi di studiare e sperimentare con gli esempi di codice forniti. L'apprendimento di *Python* richiede una pratica attiva, quindi ti incoraggio a modificare gli esempi di codice, sperimentare con nuove idee e creare progetti personali.

Spero che il mio approccio combinato di teoria e pratica ti aiuti a padroneggiare Python in modo efficace e divertente!

Struttura del libro:

- ➢ Capitolo 1: Concetti preliminari
- ➢ Capitolo 2: Introduzione a Python
- ➢ Capitolo 3: Concetti base
- ➢ Capitolo 4: Le strutture dati
- ➢ Capitolo 5: La programmazione a oggetti

- ➢ Capitolo 6: Formattazione delle stringhe
- ➢ Capitolo 7: Le espressioni regolari
- ➢ Capitolo 8: La gestione delle eccezioni
- ➢ Capitolo 9: La gestione dei file
- ➢ Capitolo 10: Il debugging
- ➢ Capitolo 11: La generazione della documentazione
- ➢ Capitolo 12: Design patterns
- ➢ Capitolo 13: Concetti avanzati
- ➢ Capitolo 14: Esercizi finali

Utilizzo del codice presente nel libro e scaricamento da GitHub

Questo libro fornisce un prezioso supporto pratico alla tua esperienza di apprendimento. Ogni capitolo contiene esempi di codice *Python* pertinenti agli argomenti trattati. Per accedere a questi esempi, puoi visitare il repository *GitHub* dedicato al libro.

Per scaricare il codice, segui questi semplici passaggi:

- Visita il repository GitHub all'indirizzo https://github.com/GeekyTechGT/L-arte-della-programmazione-Python.
- Naviga tra le diverse cartelle per trovare il codice relativo al capitolo di tuo interesse.
- Clicca sul file desiderato per visualizzare il suo contenuto.
- Puoi copiare il codice manualmente o cliccare sul pulsante "Download" per scaricare l'intero file.

Ti consiglio di utilizzare questi esempi di codice come punto di partenza per i tuoi progetti, sperimentando e apportando modifiche per approfondire la comprensione dei concetti esposti nel libro.

Il repository GitHub sarà aggiornato periodicamente con eventuali correzioni, miglioramenti o nuovi esempi. Ti invito a controllarlo di tanto in tanto per accedere alle ultime versioni del codice.

Spero che l'utilizzo del codice fornito renda il tuo percorso di apprendimento ancora più coinvolgente ed efficace.

Restiamo in contatto

Mi piacerebbe rimanere in contatto con te mentre continui il tuo percorso nella programmazione *Python*. Che tu abbia domande, feedback o semplicemente desideri condividere i tuoi progressi, ci sono diversi modi per metterti in contatto con me:

Email: non esitare a contattarmi via email all'indirizzo *geeky.teck.gt@gmail.com*. Apprezzo il tuo contributo e farò del mio meglio per rispondere alle tue domande e offrire assistenza.

Recensioni del libro: se hai trovato il libro utile e informativo, ti invito a lasciare una recensione su piattaforme di recensioni di libri popolari come *Goodreads* o *Amazon*. Il tuo feedback sincero può aiutare altri aspiranti programmatori *Python* a scoprire e beneficiare di questa risorsa.

Apprezzo sinceramente il tuo supporto: il tuo contributo e il tuo coinvolgimento sono preziosi per me mentre cerco di migliorare e offrirti materiali di apprendimento pertinenti ed efficaci. Non vedo l'ora di rimanere in contatto con te e di assistere alla tua crescita come programmatore *Python* esperto. Buon coding!

Sommario

Capitolo 1: Concetti Preliminari

Cosa si intende con linguaggio di programmazione?

Un linguaggio di programmazione è un insieme di regole, simboli e convenzioni che definiscono le istruzioni utilizzate per scrivere programmi. È un mezzo per comunicare con un computer e fornire una sequenza di istruzioni che specificano le azioni che il computer deve eseguire.

I linguaggi di programmazione sono progettati per essere comprensibili sia dagli esseri umani che dai computer. Gli esseri umani possono utilizzarlo per scrivere codice leggibile, mentre i computer possono interpretare o compilare quel codice per eseguire le istruzioni specificate.

Ci sono diversi tipi di linguaggi di programmazione, tra cui:

Linguaggi di programmazione ad alto livello: sono linguaggi più vicini al linguaggio naturale dell'uomo e sono progettati per essere facilmente comprensibili dagli esseri umani. Alcuni esempi includono *Python*, *Java*, *C*, *C++, JavaScript*, *Rust*, *Ruby*.

Linguaggi di programmazione a basso livello: sono linguaggi più vicini al linguaggio macchina e richiedono una conoscenza più approfondita dell'architettura del computer. Alcuni esempi includono il linguaggio assembly e linguaggio macchina.

Linguaggi di scripting: sono linguaggi utilizzati principalmente per automatizzare compiti ripetitivi. Sono spesso interpretati anziché compilati. Alcuni esempi includono *Python*, *Bash*, *Perl* e *PowerShell*.

Linguaggi di markup: sono linguaggi utilizzati per descrivere la struttura e la formattazione di documenti o dati. Non sono linguaggi di programmazione tradizionali, ma sono utilizzati in contesti come la creazione di pagine web.

Alcuni esempi includono *HTML*, *XML*, *Markdown*.

I linguaggi di programmazione definiscono anche le regole sintattiche e semantiche che determinano come il codice deve essere scritto e interpretato. Ciò include la sintassi per la dichiarazione delle variabili, la definizione delle funzioni, la gestione delle strutture di controllo (come cicli e condizioni) e altro ancora.

In sintesi, un linguaggio di programmazione è uno strumento che permette agli sviluppatori di scrivere istruzioni comprensibili dai computer per eseguire una specifica sequenza di azioni. Ogni linguaggio ha le sue caratteristiche, scopi e sintassi specifiche, ma tutti si basano sul concetto di comunicazione tra umani e macchine.

Linguaggio interpretato VS Linguaggio compilato

Quando si parla di programmazione, uno dei concetti fondamentali riguarda la differenza tra i linguaggi di programmazione interpretati e quelli compilati. Questa distinzione ha un impatto significativo sulla modalità di esecuzione del codice e sulle caratteristiche del linguaggio stesso. In questa sezione esploreremo le differenze tra i linguaggi interpretati e quelli compilati, evidenziandone le caratteristiche principali e le implicazioni per gli sviluppatori.

Linguaggi compilati

I linguaggi di programmazione compilati richiedono una fase preliminare di compilazione prima dell'esecuzione del programma. Durante la fase di compilazione, il codice sorgente viene tradotto in un codice eseguibile specifico per la piattaforma di destinazione. Questo processo di traduzione è gestito da un compilatore, un programma specializzato che analizza il codice sorgente, lo traduce in istruzioni macchina comprensibili dal computer e crea un file eseguibile.

Una volta che il programma è stato compilato con successo, l'eseguibile risultante può essere eseguito ripetutamente senza la necessità di una fase di compilazione ulteriore. Il codice macchina generato è generalmente più efficiente dal punto di vista delle prestazioni rispetto al codice interpretato, poiché è stato ottimizzato durante la fase di compilazione.

Tuttavia, una caratteristica importante dei linguaggi compilati è che il codice compilato è specifico per una piattaforma particolare. Ciò significa che un programma compilato per un sistema operativo specifico e un'architettura di processore non può essere eseguito su un'altra piattaforma senza una nuova compilazione per quella specifica configurazione.

Linguaggi interpretati

A differenza dei linguaggi compilati, i linguaggi di programmazione interpretati non richiedono una fase di compilazione prima dell'esecuzione del programma. Un interprete legge il codice sorgente e lo esegue istruzione per istruzione in tempo reale.

L'interprete analizza il codice sorgente linea per linea, traducendo ciascuna istruzione in codice macchina ed eseguendola immediatamente. Questo processo di interpretazione rende il codice più flessibile, poiché le modifiche possono essere apportate direttamente al codice sorgente senza la necessità di una compilazione preliminare.

Inoltre, i linguaggi interpretati offrono solitamente funzionalità interattive, come un ambiente REPL (Read-Eval-Print Loop), che consente agli sviluppatori di scrivere ed eseguire istruzioni in modo interattivo. Questo rende più facile l'esplorazione e il debug del codice, in quanto gli sviluppatori possono visualizzare immediatamente i risultati delle loro istruzioni.

Tuttavia, a causa dell'interpretazione del codice in tempo reale, i linguaggi interpretati tendono ad essere leggermente più lenti dei linguaggi compilati dal punto di vista delle prestazioni. L'interprete deve analizzare e tradurre il codice ad ogni esecuzione, il che comporta un leggero sovraccarico rispetto all'esecuzione di programma compilato.

Paradigmi di programmazione

Esistono diversi paradigmi di programmazione, ognuno dei quali si basa su una diversa filosofia di progettazione del software e utilizza strumenti e tecniche specifiche. Di seguito riportiamo una breve descrizione dei principali paradigmi.

Programmazione procedurale: è basata sulla definizione di una serie di procedure o funzioni che svolgono specifiche operazioni sulle variabili del programma.

Questo paradigma è basato su una logica sequenziale e strutturata, che permette di organizzare il codice in blocchi di istruzioni ben definiti.
La programmazione procedurale è utilizzata principalmente per lo sviluppo di applicazioni di basso livello, come i driver dei dispositivi hardware o i sistemi operativi.

Programmazione orientata agli oggetti (OOP): si basa sulla definizione di oggetti, che sono entità che racchiudono sia i dati che le operazioni (metodi) che possono essere eseguite su di essi. Gli oggetti sono organizzati in classi, che rappresentano categorie di oggetti con proprietà e comportamenti comuni.

La OOP permette di creare codice modulare, flessibile e facilmente estendibile, ed è utilizzata per lo sviluppo di applicazioni di grandi dimensioni.

Programmazione funzionale: si basa sulla definizione di funzioni che non modificano lo stato del programma, ma che producono sempre lo stesso risultato per un determinato insieme di input. La programmazione funzionale è basata sull'utilizzo di funzioni pure che possono essere composte tra di loro per creare programmi più complessi.

Questo paradigma è utilizzato soprattutto per lo sviluppo di applicazioni che richiedono elaborazioni matematiche o algoritmiche.

Programmazione a eventi: è basata sulla gestione degli eventi generati dal sistema o dall'utente. Gli eventi vengono gestiti

attraverso la definizione di funzioni di callback, che vengono eseguite quando un evento specifico viene generato.

Questo paradigma è utilizzato principalmente per lo sviluppo di applicazioni con interfaccia grafica, come ad esempio i videogiochi o le applicazioni desktop.

Capitolo 2: Introduzione a Python

Perché imparare a programmare in Python?

Python è uno dei linguaggi di programmazione più popolari al mondo. È facile da imparare, ha una sintassi semplice e pulita, ed è utilizzato in molte aree, dall'analisi dei dati alla creazione di software per utenti finali. In questo capitolo, esploreremo le ragioni per cui dovresti considerare di imparare a programmare in *Python*.

- **Python è facile da imparare**: è uno dei linguaggi di programmazione più facili da imparare per i principianti. La sua sintassi è semplice e leggibile, il che significa che il codice scritto in *Python* è facile da comprendere anche per coloro che non hanno familiarità con il linguaggio.
  Inoltre ha una vasta comunità di utenti che forniscono supporto e risorse per aiutare i nuovi programmatori ad imparare.
- **Python è un linguaggio versatile**: è utilizzato in molti campi, tra cui analisi dei dati, intelligenza artificiale, sviluppo web, automazione e molti altri. La sua versatilità implica che, una volta che hai imparato *Python*, puoi utilizzarlo in molte aree differenti, rendendolo un'ottima scelta per coloro che vogliono acquisire una competenza che può essere utilizzata in molte situazioni lavorative.
- **Python è in forte crescita**: è uno dei linguaggi di programmazione in più rapida crescita al mondo. La sua popolarità è in costante aumento e molte grandi aziende utilizzano *Python* per la loro infrastruttura e sviluppo di prodotti. Ciò significa che ci sono molte opportunità di lavoro per i programmatori *Python*, rendendolo un'ottima scelta per coloro che cercano una carriera nella programmazione informatica.
- **Python ha una vasta comunità di utenti**: *Python* ha una delle comunità di utenti più grandi e attive di tutti i linguaggi di programmazione. Ciò significa che ci sono molte risorse disponibili online, come tutorial, forum, comunità di sviluppatori e risorse didattiche.
  Inoltre, la comunità di utenti è molto accogliente e pronta ad aiutare i nuovi arrivati, il che rende più facile per i principianti iniziare a imparare il linguaggio.

- **Python è utilizzato in molte università e corsi di formazione**: *Python* è diventato uno dei linguaggi di programmazione più utilizzati nelle università e nei corsi di formazione in tutto il mondo. Ciò significa che ci sono molte risorse disponibili per imparare *Python*, come libri di testo, corsi online e tutorial. Inoltre, poiché *Python* è così popolare, molti datori di lavoro cercano programmatori *Python*, il che rende più facile trovare lavoro dopo aver completato un corso di formazione o un corso universitario.

In sintesi, imparare a programmare in *Python* può offrire molte opportunità per la tua carriera.

Storia di Python

Python è un linguaggio di programmazione ad alto livello, interpretato, orientato agli oggetti e con una sintassi semplice e intuitiva. Creato nel 1991 da *Guido Van Rossum*, è diventato uno dei linguaggi più popolari al mondo, utilizzato per una vasta gamma di applicazioni.

In questa sezione esploreremo la storia di *Python*, dalla sua creazione alle sue ultime evoluzioni.

La nascita di Python

Python fu creato nel tardo 1991 da Guido Van Rossum, un programmatore olandese, membro del gruppo di ricerca del Centro Nazionale per la Matematica e l'Informatica (CWI) nei Paesi Bassi. *Van Rossum* era insoddisfatto dei linguaggi di programmazione esistenti e decise di crearne uno nuovo che fosse facile da usare e allo stesso tempo potente e versatile.

Nel dicembre 1989, *Van Rossum* iniziò a lavorare su *Python* come un progetto personale. Il nome Python fu scelto in omaggio al programma televisivo inglese *Monty Python's Flying Circus*, che *Van Rossum* apprezzava molto.

Python è stato sviluppato inizialmente come un progetto di svago; tuttavia ben presto *Van Rossu*m si rese conto del suo potenziale come linguaggio di programmazione professionale e nel 1991 egli rilasciò la prima versione pubblica (la versione 0.9.0) su *Usenet*, una rete di discussioni online molto popolare all'epoca.

Il successo di Python

Nel corso degli anni '90, *Python* iniziò a guadagnare popolarità tra i programmatori e gli scienziati informatici. Grazie alla sua sintassi semplice e alla sua facilità d'uso, divenne un'alternativa molto attraente rispetto ad altri linguaggi di programmazione come *C++* e *Java*.

Nel 2000, la Python Software Foundation (PSF) fu creata per gestire lo sviluppo di *Python* e promuovere il suo uso. La PSF è un'organizzazione senza scopo di lucro che si occupa della promozione, del supporto e dello sviluppo di *Python*.

Negli anni successivi, esso divenne sempre più popolare e il numero di librerie e framework disponibili per il linguaggio aumentò in modo significativo; è stato utilizzato per creare una vasta gamma di applicazioni, dalle applicazioni desktop ai giochi, dai siti web alle applicazioni di data science e machine learning.

L'evoluzione di Python

Le versioni di *Python* dalla 1.0 alla 3.0 hanno segnato importanti pietre miliari nella storia del linguaggio, portando significative migliorie e nuove funzionalità al linguaggio.

Vediamo un riassunto delle principali versioni:

- Python 1.0 (gennaio 1994): è stata la prima versione stabile. È stato sviluppato come un linguaggio di programmazione ad alto livello, versatile e facile da imparare. Ha introdotto concetti chiave come le liste, i dizionari e le funzioni ricorsive.
- Python 2.x (distribuito dal 2000 al 2010): la serie di versioni 2.x ha introdotto numerosi miglioramenti rispetto alla versione 1.0, come la gestione delle eccezioni, le espressioni regolari e una maggiore compatibilità con i sistemi operativi. Le versioni più popolari sono state *Python* 2.5, 2.7 e 2.7.18, quest'ultima rilasciata nel 2020 come ultima versione di manutenzione della serie 2.x.
- Python 3.0 (dicembre 2008): ha segnato un importante punto di svolta nella storia del linguaggio. È stato progettato per risolvere diverse problematiche presenti nella serie 2.x e introdurre modifiche significative. Alcuni dei cambiamenti chiave includono la sintassi

migliorata, l'introduzione della divisione float come operazione predefinita, l'eliminazione di funzioni obsolete e la gestione più coerente delle stringhe Unicode. Tuttavia, queste modifiche hanno comportato una rottura di compatibilità con il codice scritto per la serie 2.x.

Le versioni successive di *Python* 3, come *Python* 3.1, 3.2, 3.3 e così via, hanno apportato ulteriori miglioramenti, correzioni di bug e nuove funzionalità al linguaggio.

È importante notare che, nonostante l'introduzione di *Python 3*, molte organizzazioni e progetti hanno continuato a utilizzare la serie 2.x a causa della dipendenza da librerie e codice già esistente. Tuttavia, negli ultimi anni, l'adozione di *Python 3* è cresciuta notevolmente, e la comunità ha incoraggiato la transizione verso *Python 3* per beneficiare delle sue caratteristiche aggiornate e della manutenzione attiva.

Installazione

Python è un linguaggio di programmazione open-source, quindi puoi scaricare e installare l'interprete *Python* gratuitamente sul tuo computer. Esistono diverse versioni disponibili, ma per la maggior parte degli scopi, la versione più recente di *Python 3.x* è la scelta migliore.

Windows

Scaricare il programma di installazione

Per scaricare *Python* su Windows, vai al sito web ufficiale **https://www.python.org/downloads/windows/** e clicca sul pulsante **Download Python X.X.X** (dove "X.X.X" indica la versione più recente di Python). Se hai un sistema operativo a 64-bit, scegli l'opzione di download a 64-bit. Se non sei sicuro della versione del tuo sistema operativo, puoi controllarlo andando su **Start -> Impostazioni -> Sistema -> Informazioni**. La versione del sistema operativo dovrebbe essere elencata sotto **Tipo sistema.**

Installare Python

Una volta scaricato il programma di installazione, fai doppio clic sul file **.exe** e segui le istruzioni per l'installazione. Assicurati di selezionare l'opzione **Add Python X.X.X to PATH** durante l'installazione. Questo

consentirà di accedere facilmente all'interprete di *Python* dalla riga di comando.

Per assicurarti che l'interprete *Python* sia stato installato correttamente, puoi avviare una *Powershell* (**Start -> Windows Powershell**) e digitare quanto segue.

```
C:\Users\{Nome_User}> python --version
Python X.Y.Z
```

Mac

Scaricare il programma di installazione

Per scaricare *Python* su Mac, vai al sito web ufficiale di *Python* https://www.python.org/downloads/windows e clicca sul pulsante **Download Python X.X.X** (dove X.X.X indica la versione più recente di *Python*). Scarica il file **.pkg**.

Installare Python

Fai doppio clic sul file **.pkg** scaricato e segui le istruzioni per l'installazione. Quando ti viene chiesto dove installare *Python*, lascia la posizione predefinita (**/Library/Frameworks/Python.framework**). Assicurati di selezionare l'opzione **Install for all users of this computer** durante l'installazione.

Linux

Installare Python utilizzando il gestore di pacchetti

In molte distribuzioni Linux, *Python* è già preinstallato; se non lo è, puoi installarlo facilmente utilizzando il gestore di pacchetti della tua distribuzione.

Ad esempio, su **Ubuntu**, puoi installare *Python* eseguendo il seguente comando da terminale:

```
sudo apt-get update
sudo apt-get install python3
```

Installare Python manualmente

Se vuoi installare una versione specifica di *Python* o se il tuo gestore di pacchetti non ha l'ultima versione di *Python* disponibile, puoi installarlo manualmente.

Innanzitutto, vai al sito web ufficiale di Python https://www.python.org/downloads/source/.

Nella pagina dei download del codice sorgente, individua la versione di *Python* desiderata e fai clic sul link per scaricare il file tarball (file con estensione **.tar.xz**).

Torna al terminale e sposta il file tarball nella directory desiderata. Ad esempio, puoi utilizzare il comando **cd** per navigare nella directory di tua scelta e il comando **mv** per spostare il file scaricato.

```
cd /path/to/directory
mv /path/to/downloaded/python.tar.xz .
```

Assicurati di sostituire "/path/to/directory" con il percorso della directory in cui desideri posizionare il file. Successivamente estrai il contenuto del file tarball utilizzando il comando **tar**.

```
tar -xf python.tar.xz
```

Una volta estratto, entra nella directory appena creata.

```
cd Python-3.9.5
```

Assicurati di sostituire "3.9.5" con la versione effettiva di *Python* che hai scaricato. Ora puoi configurare, compilare e installare Python utilizzando i seguenti comandi:

```
./configure
make
sudo make install
```

Il comando *configure* verifica le dipendenze e configura il processo di compilazione. Il comando *make* compila il codice sorgente di *Python* e il comando *make install* installa *Python* nel sistema. Potrebbe essere necessario inserire la password di amministratore (sudo password) per eseguire il comando *make install*.

Ambiente di sviluppo

Python è un linguaggio di programmazione interpretato, il che significa che non è necessario compilare il codice prima di eseguirlo.

Sebbene il codice *Python* possa essere eseguito da riga di comando, la maggior parte degli sviluppatori preferisce utilizzare un ambiente di sviluppo integrato (IDE) per scrivere, testare e debuggare il proprio codice. In questa sezione esploreremo alcuni degli IDE più popolari per la programmazione in *Python* e come utilizzarli.

Visual Studio Code

Visual Studio Code (**VS Code**) è un editor di codice sorgente sviluppato da Microsoft. Anche se non è un IDE specifico per Python, offre un'ampia gamma di estensioni per la programmazione in *Python*.

Per utilizzare VS Code per la programmazione in *Python*, è necessario installare l'estensione "Python" di Microsoft dall'apposita sezione degli Add-ons. Dopo l'installazione, puoi creare un nuovo file Python e iniziare a scrivere il tuo codice.

Oltre all'estensione *Python* di Microsoft, VS Code offre anche l'estensione **Python for VSCode** sviluppata da *Don Jayamanne*, che fornisce funzionalità aggiuntive come la possibilità di selezionare un ambiente Python specifico, l'integrazione con il notebook *Jupyter* e molto altro ancora.

Inoltre, VS Code supporta la creazione di ambienti virtuali *Python*, che consentono di isolare i pacchetti e le dipendenze di un progetto in un ambiente separato dal resto del sistema. Questo può aiutare a evitare problemi di compatibilità tra pacchetti e versioni di *Python* diverse.

Per creare un ambiente virtuale Python in VS Code, è necessario installare l'estensione *Python* e poi utilizzare lo shortcut *CTRL + Shift + P* e selezionare il comando *Python: Create Terminal* per aprire un terminale. Nel terminale, eseguire il comando *python3 -m venv <nomeambiente>* per creare un nuovo ambiente virtuale. Successivamente, è possibile selezionare l'ambiente virtuale appena creato utilizzando il comando *Python: Select*

Interpreter e selezionare il percorso all'interprete *Python* all'interno dell'ambiente virtuale.

In conclusione, *Visual Studio Code* è un editor di codice sorgente estremamente versatile che offre una vasta gamma di funzionalità. Grazie alle estensioni disponibili, è possibile personalizzare l'esperienza di sviluppo in modo da adattarla alle proprie esigenze specifiche, offrendo la possibilità di programmare in modo efficiente ed efficace.

PyCharm

PyCharm è un IDE popolare per la programmazione in *Python* sviluppato dalla società di software *JetBrains*. Offre una vasta gamma di funzionalità avanzate, come il debugging interattivo, il refactoring del codice, l'integrazione con il controllo del codice sorgente, la creazione di test unitari e molto altro ancora. *PyCharm* è disponibile in due versioni: Community (gratuita e open source) e Professional (a pagamento).

Per utilizzare *PyCharm*, è necessario scaricarlo e installarlo dal sito ufficiale di *JetBrains*. Una volta installato, puoi creare un nuovo progetto *Python* e iniziare a scrivere il tuo codice. *PyCharm* offre anche un sistema di completamento automatico del codice, che può aiutarti a risparmiare tempo durante la scrittura del codice.

Interprete Python online

La popolarità di *Python* come linguaggio di programmazione versatile e potente ha portato all'espansione di numerose opzioni per utilizzarlo direttamente online. Grazie alla disponibilità di ambienti di sviluppo integrati basati sul web, i programmatori possono scrivere, eseguire e condividere codice *Python* senza dover installare alcun software aggiuntivo sul proprio computer.

Una possibilità è rappresenta da ***Programiz***.

L'utilizzo di *Programiz* come interprete online per Python presenta diversi vantaggi. Innanzitutto, è un'opzione molto accessibile e conveniente, poiché non richiede alcuna installazione o configurazione complessa. Basta accedere al sito web https://www.programiz.com/python-programming/online-compiler/ aprire l'IDE *Python* e iniziare a scrivere il

codice. Questo è particolarmente utile per coloro che desiderano programmare rapidamente senza dover affrontare la configurazione di un ambiente di sviluppo locale.

Un altro vantaggio di utilizzare *Programiz* come interprete online è la sua facilità d'uso. L'interfaccia utente intuitiva e ben progettata offre un'esperienza di programmazione piacevole. È possibile scrivere il codice nella finestra dell'IDE, eseguirlo facendo clic su un pulsante e visualizzare immediatamente i risultati nell'output. Inoltre, l'IDE offre funzionalità di evidenziazione della sintassi, suggerimenti intelligenti e un'ampia documentazione incorporata per aiutare i programmatori a scrivere codice *Python* corretto e efficiente.

Programiz supporta anche molte delle caratteristiche avanzate di *Python*, come l'utilizzo di librerie esterne e la gestione dei moduli. È possibile importare e utilizzare librerie popolari come *NumPy, Pandas e Matplotlib* nel proprio codice. Ciò consente ai programmatori di sperimentare con funzionalità più complesse senza dover preoccuparsi di installare manualmente tali librerie.

Inoltre, *Programiz* offre la possibilità di salvare e condividere il proprio codice. È possibile creare account gratuiti per salvare i progetti e accedere ad essi in qualsiasi momento. Questo è particolarmente utile per i programmatori che desiderano lavorare su progetti *Python* da diverse piattaforme o condividere il proprio codice con colleghi o amici.

La shell di python

La shell di *Python*, o REPL (*Read-Eval-Print Loop*), è un'interfaccia interattiva che consente di eseguire istruzioni Python direttamente dalla riga di comando. Rappresenta uno strumento molto utile per testare rapidamente il codice e per eseguire operazioni di debugging.

Avviare la shell di Python

Per avviare la shell di *Python*, è sufficiente aprire il terminale (digitando nella barra di ricerca *PowerShell* o *cmd* [in ambiente Windows]) del proprio sistema operativo e digitare il comando *python* o python3, a seconda della versione di *Python* installata. Se tutto è andato a buon fine, dovrebbe apparire un prompt simile a questo:

```
PS C:\Users\{nome_user}> python
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41) [MSC v.1929 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Eseguire istruzioni Python

Dopo aver avviato la shell di *Python*, è possibile eseguire istruzioni digitandole direttamente nel prompt e premendo Invio. Ad esempio, digitando *print("Ciao, mondo\!")* seguito da Invio, verrà visualizzato il seguente output:

```
1. >>> print("Ciao, mondo")
2. Ciao, mondo
3. >>>
```

È possibile eseguire qualsiasi tipo di istruzione, come assegnazioni di variabili, definizioni di funzioni, cicli e condizioni. La shell di *Python* esegue le istruzioni una per volta e mostra il risultato immediatamente.

Storia delle istruzioni

La shell di *Python* tiene traccia delle istruzioni eseguite in una storia, che può essere richiamata e modificata. Per accedere alla storia delle istruzioni, è possibile premere il tasto *Freccia Su* per riprodurre l'ultima istruzione eseguita e premere nuovamente il tasto *Freccia Su* per accedere a istruzioni precedenti.

Aiuto in linea

La shell di *Python* offre anche una guida in linea molto utile, che consente di accedere alla documentazione delle funzioni e dei moduli *Python*. Per accedere all'aiuto in linea, digitare *help()* nel prompt e premere Invio. In questo modo verrà visualizzato un elenco di opzioni, come *"modules"* o *"keywords",* che consentono di accedere alla documentazione di moduli e parole chiave.

Vediamo un esempio:

```
C:\Users\${NomeUtente}> python
Python 3.10.4 (tags/v3.10.4:9d38120, Mar 23 2022, 23:13:41)
[MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help()

Welcome to Python 3.10's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.10/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> str
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
```

```
 |  errors defaults to strict.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
```

```
|
 | __iter__(self, /)
|     Implement iter(self).
|
 | __le__(self, value, /)
|     Return self<=value.
|
 | __len__(self, /)
|     Return len(self).
 |
-- More  --
```

Uscita dalla shell di Python

Per uscire dalla shell di *Python*, digitare il comando *exit()* o premere *Ctrl + D*. In questo modo verrà chiusa la shell e si tornerà al prompt del terminale.

In conclusione, la shell di *Python* è uno strumento molto utile per testare rapidamente il codice e per eseguire operazioni di debugging. Conoscerne le sue funzionalità può aiutare a diventare più produttivi e efficienti nella scrittura di codice.

Hello World! in Python

Hello, World! è il classico esempio di programma introduttivo per qualsiasi linguaggio di programmazione. In *Python*, il programma *Hello, World!* è molto semplice e può essere scritto in pochi secondi.

Prima di scrivere il programma *Hello, World!,* è importante capire la sintassi di base del linguaggio. *Python* è un linguaggio molto intuitivo e leggibile, con una sintassi relativamente semplice. Ad esempio, per stampare un messaggio a schermo, si utilizza il comando *print(),* che accetta come argomento il messaggio da stampare, racchiuso tra apici singoli o doppi.

Scrivere il programma "Hello, World!"

Per scrivere il programma *Hello, World!* in *Python*, aprire un editor di testo qualsiasi e digita il seguente codice:

```
print("Hello World!")
```

Questo codice utilizza il comando *print()* per stampare il messaggio *Hello, World!* a schermo. Una volta scritto il codice, salva il file con il nome ***"hello_world.py"*** o un qualsiasi altro nome a piacere, ma con estensione *.py*.

Eseguire il programma Hello, World! in Python

Dopo aver scritto il codice del programma *Hello, World!,* è possibile eseguirlo in diversi modi. Il modo più semplice è aprire la shell di *Python* (vedi la sezione precedente), navigare nella directory (digita il comando *cd nome_cartella*) in cui si trova il file *"hello_world.py"* e digitare il seguente comando:

```
python hello_world.py
```

In questo modo *Python* eseguirà il file *"hello_world.py"* e stamperà il messaggio *Hello, World!* a schermo.

Capitolo 3: Concetti fondamentali

I commenti

I commenti sono porzioni di testo all'interno del codice sorgente che vengono ignorate dall'interprete *Python* durante l'esecuzione del programma.

I commenti vengono utilizzati per aggiungere note, spiegazioni o annotazioni al codice, rendendolo più leggibile e comprensibile per gli sviluppatori e per chiunque lo legga in seguito.

Ci sono due tipi principali di commenti:

- **Commenti in linea**: iniziano con il simbolo # e continuano fino alla fine della riga. Vengono utilizzati per commentare una singola linea di codice. Essi possono essere posizionati all'inizio di una riga o dopo una linea di codice.

```
# Questa è una riga di commento
x = 5  # Questo commento spiega cosa fa questa linea di codice
```

- **Commenti su più righe**: noti anche come commenti multilinea, iniziano e terminano con tre apici (''' o """). Possono estendersi su più righe e vengono utilizzati per commentare blocchi di codice o per scrivere commenti più lunghi.

```
'''
Questo è un commento su più righe.
Può estendersi su più righe
senza il bisogno di utilizzare
il simbolo "#" all inizio di ogni riga.
'''
```

L'utilizzo dei commenti è importante per documentare il codice, spiegare le intenzioni o fornire istruzioni sul funzionamento di parti specifiche del programma. Ecco alcune best practice per l'uso dei commenti:

- Scrivi commenti che siano chiari, concisi e informativi.
- Utilizza commenti per spiegare il "perché" del codice, non solo il "cosa".
- Fornisci informazioni sul ragionamento o sulla logica dietro una determinata implementazione.
- Evita commenti ovvi o ridondanti che non aggiungono valore al codice.
- Mantieni i commenti aggiornati con il codice. Se apporti modifiche al codice, assicurati di aggiornare anche i commenti corrispondenti.
- Evita di commentare codice che è già abbastanza leggibile e auto esplicativo. I commenti dovrebbero essere utilizzati per spiegare concetti complessi o parti del codice che potrebbero non essere immediatamente chiari.

Utilizzando i commenti in modo appropriato, puoi migliorare la leggibilità e la manutenibilità del tuo codice, sia per te che per gli altri sviluppatori che leggeranno o lavoreranno sul tuo codice in futuro.

L'indentazione

In *Python*, l'indentazione si riferisce all'uso di spazi o tabulazioni per spaziare il codice all'interno di un blocco.

Essa viene utilizzata per definire la struttura del codice e per indicare quali istruzioni appartengono a un determinato blocco, come una funzione, un ciclo o un'istruzione condizionale (entreremo nel dettaglio nelle sezioni successive).

A differenza di molti altri linguaggi di programmazione che utilizzano parentesi o parole chiave per delimitare i blocchi di codice, *Python* utilizza l'indentazione come parte della sua sintassi. La quantità di spazi o tabulazioni usate per l'indentazione è significativa e determina il livello di annidamento dei blocchi.

Ecco un esempio che ne illustra l'utilizzo:

```
def saluta():
    print("Ciao!")
    print("Benvenuto nel mondo di Python.")
```

```
saluta()
```

In questo esempio, la funzione *saluta()* è definita con un'indentazione di quattro spazi. Tutte le istruzioni all'interno della funzione, come le due chiamate *print(),* sono indentate con quattro spazi. Questo indica che queste istruzioni appartengono alla funzione *saluta()*.

Inoltre, la chiamata *saluta()* al di fuori della definizione della funzione non ha alcuna indentazione, il che indica che questa chiamata non appartiene alla funzione ma è un'istruzione indipendente.

L'indentazione corretta è essenziale in *Python* perché influenza la semantica e il flusso di controllo del programma. Se l'indentazione non fosse corretta o coerente, il codice potrebbe generare errori di sintassi o comportarsi in modo imprevisto.

Le linee di codice che appartengono allo stesso blocco devono avere la stessa quantità di spazi o tabulazioni all'inizio della riga. Solitamente, l'uso di quattro spazi per l'indentazione è considerato una pratica comune, anche se puoi utilizzare un numero diverso di spazi o tabulazioni a condizione che mantenga la coerenza all'interno del tuo codice.

L'indentazione corretta aiuta a rendere il codice *Python* più leggibile e comprensibile. Inoltre, poiché *Python* si basa sull'indentazione per determinare la struttura del codice, incoraggia la scrittura di codice ben organizzato e favorisce uno stile di programmazione coerente tra i vari sviluppatori.

Variabili e tipi di dati

Le variabili sono uno degli elementi fondamentali della programmazione e vengono utilizzate per memorizzare i dati all'interno di un programma. Una variabile può contenere un valore di qualsiasi tipo, come un numero, una stringa o un oggetto.

Regole per il nome delle variabili

Le best practice per la convenzione dei nomi delle variabili in *Python* sono linee guida che aiutano a creare nomi di variabili chiari, leggibili e coerenti nel codice.

Seguire queste convenzioni consente di migliorare la leggibilità e la comprensibilità del codice, facilitando la collaborazione con altri programmatori e la manutenzione del software nel lungo termine. Di seguito sono riportate alcune delle best practice comuni:

- **Nomi significativi:** i nomi delle variabili dovrebbero essere descrittivi e riflettere il loro scopo e la loro funzione nel contesto del programma. Utilizzare nomi che rendano chiaro il significato delle variabili e che siano leggibili agli altri programmatori.
- **Notazione in stile snake_case**: in *Python*, è comune utilizzare la notazione snake_case. Questo significa che i nomi delle variabili sono composti da parole in minuscolo separate da underscore (_). Ad esempio: *my_variable*, *count_total*, *user_name*.
- **Evitare nomi troppo generici:** evitare nomi di variabili troppo generici come "*data*", "*value*" o "*temp*". Questi nomi non forniscono informazioni specifiche sul contenuto o sull'uso della variabile. È meglio utilizzare nomi più specifici che descrivano il contesto e lo scopo della variabile.
- **Evitare abbreviazioni confuse**: cerca di evitare abbreviazioni che potrebbero essere confuse o poco chiare per gli altri programmatori. È preferibile utilizzare nomi completi e leggibili, anche se un po' più lunghi, che siano esplicativi.
- **Utilizzare nomi in lingua inglese:** è buona pratica utilizzare nomi di variabili in lingua inglese per mantenere la coerenza e facilitare la comprensione da parte di un pubblico più ampio di programmatori. Evita l'uso di caratteri speciali o simboli non standard nei nomi delle variabili.
- **Convenzioni per costanti:** per le costanti, cioè le variabili il cui valore non cambia, solitamente si utilizzano nomi in maiuscolo. Ad esempio: *MAX_VALUE, PI, DEFAULT_COLOR*.
- **Evitare nomi di variabili riservati**: evita di utilizzare nomi di variabili che sono riservati nel linguaggio *Python*, come "*print*", "*if*", "*for*" e così via. L'utilizzo di tali nomi può causare errori o comportamenti imprevisti nel codice.

Rispettare queste best practice contribuisce a creare un codice più leggibile, comprensibile e mantenibile nel tempo. Inoltre, seguire tali convenzioni

aiuta a mantenere la coerenza all'interno di un progetto e facilita la collaborazione tra i membri del team di sviluppo.

Dichiarazione delle variabili

In *Python*, le variabili non devono essere esplicitamente dichiarate prima di poter essere utilizzate. Ciò significa che non è necessario specificare il tipo di dato di una variabile o riservare uno spazio di memoria per essa prima di assegnarle un valore.

Quando si assegna un valore a una variabile per la prima volta, *Python* riconosce automaticamente il tipo di dato in base al valore assegnato.

```
age  = 25      # La variabile "age" viene assegnata con il # valore intero 25
name = "John"  # La variabile "name" viene assegnata con il # valore stringa "John"
```

In questo esempio, le variabili *age* e *name* vengono create e inizializzate con i valori specificati. Non c'è bisogno di dichiarare esplicitamente il tipo di dato delle variabili come *int* o *str*: *Python* determina automaticamente il tipo di dato in base al valore assegnato.

Questa caratteristica è possibile grazie alla sua natura di linguaggio di programmazione dinamico e fortemente tipizzato. In un linguaggio di programmazione staticamente tipizzato, come C/C++ o Java, è necessario dichiarare esplicitamente il tipo di dato delle variabili prima di poterle utilizzare.

L'assenza della dichiarazione esplicita delle variabili semplifica la scrittura del codice in *Python*. Tuttavia, può anche comportare una maggiore attenzione nella gestione delle variabili, in quanto è possibile assegnare valori di tipi diversi alla stessa variabile in momenti diversi. Pertanto, è importante essere consapevoli dei tipi di dati utilizzati e delle operazioni eseguite sulle variabili per evitare errori di tipo durante l'esecuzione del programma.

Tipi di dati delle variabili

In Python, le variabili possono contenere dati di diversi tipi.

I tipi di dati comuni includono:

- **integer:** numeri interi senza parte decimale. Esempi di numeri interi includono 1, 2, 3, -4, -10.
- **float:** numeri con la parte decimale. Esempi di numeri in virgola mobile includono 1.0, 2.5, -3.14.
- **string:** sequenze di caratteri racchiuse tra apici. Ad esempio, "ciao", "Hello World", "123".
- **boolean:** valori che possono essere *True* o *False*.
- **liste**: raccolta di valori, anche di tipi diversi, racchiusi tra parentesi quadre e separati da virgole.
- **tuple**: come le liste, ma i valori racchiusi tra parentesi tonde.
- **dict**: una struttura dati che permette di memorizzare coppie chiave-valore.

Assegnazione multipla delle variabili

In *Python*, è possibile assegnare il valore a più variabili contemporaneamente utilizzando l'assegnazione multipla. Ad esempio, per assegnare i valori 1, 2 e 3 a tre diverse variabili *a*, *b* e *c*, si può scrivere:

```
a, b, c = 1, 2, 3
```

Conversione dei tipi di dati

In *Python*, è possibile convertire un tipo di dato in un altro utilizzando le funzioni di conversione *built-in* (vedi sezione \ref{cap:funzioni_built_in}).

Questo è utile quando si ha bisogno di convertire un valore da un tipo di dato a un altro per poterlo utilizzare in un'operazione specifica.

Conversione di interi e floating point

Per convertire un numero in un intero, si può utilizzare la funzione *int()*.

```
x = int(3.14)
```

In questo esempio, la funzione *int()* converte il valore decimale 3.14 in un intero 3.

Per convertire un numero in un floating point, si può utilizzare la funzione *float()*.

```
y = float(5)
```

In questo esempio, la funzione *float()* converte il valore 5 in un floating point 5.0.

Conversione di stringhe

Per convertire una stringa in un intero, si può utilizzare la funzione *int()*.

```
s = "10"
x = int(s)
```

In questo esempio, la funzione *int()* converte la stringa "10" in un intero 10.

Per convertire una stringa in un floating point, si può utilizzare la funzione *float()*.

```
s = "3.14"
y = float(s)
```

In questo esempio, la funzione *float()* converte la stringa "3.14" in un floating point 3.14.

Conversione di bool

Per convertire un valore in un booleano, si può utilizzare la funzione *bool()*. Qualsiasi valore diverso da 0 o vuoto viene convertito in *True*, mentre 0 o valore vuoto viene convertito in *False*.

```
x = bool(1)
y = bool(0)
```

In questo esempio, *x* viene convertito in *True* mentre *y* viene convertito in *False*.

Conversione di liste, tuple e dizionari

Per convertire una lista in una tupla, si può utilizzare la funzione *tuple()*.

```
a = [1, 2, 3]
b = tuple(a)
```

In questo esempio, la funzione *tuple()* converte la lista [1, 2, 3] in una tupla (1, 2, 3).

Per convertire una tupla in una lista, si può utilizzare la funzione *list()*.

```
a = (1, 2, 3)
b = list(a)
```

In questo esempio, la funzione *list()* converte la tupla (1, 2, 3) in una lista [1, 2, 3].

Per convertire un dizionario in una lista di tuple, si può utilizzare il metodo *items()* del dizionario e la funzione *list()*.

```
a = {'nome': 'Mario', 'cognome': 'Rossi', 'eta': 30}
b = list(a.items())   #output: [('nome', 'Mario'), ('cognome', 'Rossi'), ('eta', 30)]
```

Operatori ed espressioni

In *Python*, gli operatori sono simboli speciali utilizzati per eseguire operazioni su uno o più valori. Ad esempio, l'operatore aritmetico "+" viene utilizzato per eseguire l'addizione tra due valori. Ecco una lista degli operatori più comuni ed utilizzati:

- **Operatori aritmetici + - * / \% ** // : vengono utilizzati per eseguire operazioni matematiche su numeri.

    - Il simbolo + viene utilizzato per eseguire l'addizione tra due numeri.
    - Il simbolo - viene utilizzato per eseguire la sottrazione tra due numeri.
    - Il simbolo * viene utilizzato per eseguire la moltiplicazione tra due numeri.
    - Il simbolo / viene utilizzato per eseguire la divisione tra due numeri.
    - Il simbolo **% viene utilizzato per calcolare il resto di una divisione.
    - Il simbolo ** viene utilizzato per eseguire l'elevamento a potenza.
    - Il simbolo // viene utilizzato per eseguire la divisione intera.

```
a = 10
b = 5

print(a + b)   # output: 15
print(a - b)   # output: 5
print(a * b)   # output: 50
print(a / b)   # output: 2.0
print(a % b)   # output: 0
```

```
print(a ** 2)  # output: 100
print(a // 3)  # output: 3
```

➢ **Operatori di confronto** == != > < >= <= : vengono utilizzati per confrontare due valori e restituire un valore booleano (*True* o *False*).

- Il simbolo == viene utilizzato per verificare se due valori sono uguali.
- Il simbolo **!=** viene utilizzato per verificare se due valori sono diversi.
- Il simbolo > viene utilizzato per verificare se il primo valore è maggiore del secondo.
- Il simbolo < viene utilizzato per verificare se il primo valore è minore del secondo.
- Il simbolo >= viene utilizzato per verificare se il primo valore è maggiore o uguale al secondo.
- Il simbolo <= viene utilizzato per verificare se il primo valore è minore o uguale al secondo.

```
a = 10
b = 5

print(a == b) # output: False
print(a != b) # output: True
print(a > b) # output: True
print(a < b) # output: False
print(a >= b) # output: True
print(a <= b) # output: False
```

➢ **Operatori logici**: **and, or, not**: sono utilizzati per combinare espressioni booleane (un'istruzione che viene valutata come vera o falsa) e produrre un risultato booleano (*True* o *False*).

- L'operatore ***and*** restituisce *True* solo se entrambe le espressioni booleane sono *True*.

```
a = 5
b = 10
```

```
c = 15

print(a < b and b < c) # output: True
print(a < b and c < b) # output: False
```

Nel primo esempio, l'espressione $a < b$ è *True* e l'espressione $b < c$ è *True*, quindi l'operatore *and* restituisce *True*.

Nel secondo esempio, l'espressione $a < b$ è *True*, ma l'espressione $c < b$ è *False*, quindi l'operatore *and* restituisce *False*.

- L'operatore ***or*** restituisce *True* se almeno una delle espressioni booleane è *True*.

```
a = 5
b = 10
c = 15

print(a < b or b < c) # output: True
print(a > b or c < b) # output: False
```

Nel primo esempio, l'espressione $a < b$ è *True*, quindi l'operatore *or* restituisce *True* senza valutare l'espressione $b < c$.

Nel secondo esempio, entrambe le espressioni $a > b$ e $c < b$ sono *False*, quindi l'operatore *or* restituisce *False*.

- L'operatore ***not*** restituisce l'inverso dell'espressione booleana.

```
a = 5
b = 10

print(not a < b) # output: False
print(not a > b) # output: True
```

Nel primo esempio, l'espressione $a < b$ è *True*, ma l'operatore *not* la inverte in *False*.

Nel secondo esempio, l'espressione $a > b$ è *False*, quindi l'operatore *not* la inverte in *True*.

Strutture di controllo

Le strutture di controllo consentono di eseguire un blocco di codice solo se una condizione specificata è vera, di eseguire un blocco di codice ripetutamente fino a quando una condizione diventa falsa, oppure di eseguire un blocco di codice in alternativa a un altro.

Le tre principali strutture di controllo in *Python* sono:

- **If-else statement**
- **While loop**
- **For loop**

If-else

L' *if-else* è utilizzato per eseguire un blocco di codice solo se una condizione specificata è vera, altrimenti esegue un altro blocco. La sintassi è la seguente:

```
if condizione:
    # esegui questo blocco di codice se la condizione è vera
else:
    # esegui questo blocco di codice se la condizione è falsa
```

Esempio:

```
x = 5

if x > 10:
   print("x è maggiore di 10")
else:
   print("x è minore o uguale a 10")
```

While loop

Il ciclo *while* è utilizzato per eseguire un blocco di codice ripetutamente fino a quando una condizione diventa falsa.

La sintassi è la seguente:

```
while condizione:
    # esegui questo blocco di codice finché la condizione è vera
```

Esempio:

```
x = 0

while x < 5:
    print(x)
    x += 1    # incrementa x di 1
```

Condizione di uscita

All'interno di un ciclo *while* puoi utilizzare la dichiarazione *break* per interrompere immediatamente l'esecuzione del ciclo e uscire da esso. Quando viene incontrata la dichiarazione *break*, il controllo del programma viene trasferito all'istruzione immediatamente successiva al ciclo.

```
numero = 0

while numero < 10:
    print(numero)
    if numero == 5:
        break
    numero += 1

print("Ciclo terminato.")
```

Nell'esempio sopra, il ciclo *while* viene eseguito finché la variabile *numero* è inferiore a 10. Ad ogni iterazione, viene stampato il valore corrente di *numero* e poi viene incrementato di 1 con l'operatore di incremento +=

Quando *numero* raggiunge il valore 5, viene eseguita la dichiarazione *break*, interrompendo il ciclo. Di conseguenza, l'istruzione *print(numero)* per i numeri successivi a 5 non viene eseguita.

Dopo l'uscita dal ciclo, viene eseguita l'istruzione *print("Ciclo terminato."),* che rappresenta l'istruzione successiva al ciclo *while*.

L'utilizzo della dichiarazione *break* può essere utile per terminare il ciclo in base a una specifica condizione, evitando così l'esecuzione di ulteriori iterazioni quando non è più necessario.

For loop

Il ciclo *for* è utilizzato per eseguire un blocco di codice per ogni elemento di una sequenza. La sintassi è la seguente:

```
for variabile in sequenza:
    # esegui questo blocco di codice per ogni elemento della sequenza
```

Esempio:

```
frutta = ["mela", "banana", "fragola"]

for frutto in frutta:
  print(frutto)
```

Condizione di uscita

All'interno di un ciclo *for* è possibile utilizzare la dichiarazione *break* per interrompere immediatamente l'esecuzione del ciclo e uscire da esso. Quando viene incontrata suddetta dichiarazione, il controllo del programma viene trasferito all'istruzione immediatamente successiva al ciclo.

L'uso della dichiarazione *break* è utile quando si desidera interrompere un ciclo *for* in base a una determinata condizione. Ad esempio, potresti voler cercare un elemento specifico all'interno di una lista e, una volta trovato, non hai più bisogno di eseguire il resto del ciclo.

```
numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for numero in numeri:
  if numero == 5:
    break
  print(numero)

print("Ciclo terminato.")
```

Nell'esempio sopra, il ciclo *for* viene iterato su una lista di *numeri* da 1 a 10. Quando il valore di *numero* diventa 5, viene eseguita la dichiarazione *break*, interrompendo il ciclo. Di conseguenza, l'istruzione *print(numero)* per i numeri successivi a 5 non viene eseguita.

Dopo l'uscita dal ciclo, viene eseguita l'istruzione *print("Ciclo terminato."),* poiché rappresenta l'istruzione successiva al ciclo *for*.

È importante notare che la dichiarazione *break* interrompe solo il ciclo più interno in cui viene utilizzata, per cui se si utilizzano cicli annidati, la dichiarazione *break* interromperà solo il ciclo in cui viene dichiarata.

Continuazione all'istruzione successiva

All'interno di un ciclo *for* è possibile utilizzare la dichiarazione *continue* per passare all'iterazione successiva, ignorando le istruzioni rimanenti all'interno dell'iterazione corrente. Ciò consente di saltare condizionalmente alcune iterazioni del ciclo senza terminarlo completamente.

L'uso della dichiarazione *continue* è utile quando si desidera escludere specifici elementi o condizioni dal ciclo e passare direttamente all'iterazione successiva. Ad esempio, potresti voler escludere un valore particolare da un calcolo o da un'elaborazione all'interno del ciclo.

```
numeri = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for numero in numeri:
    if numero % 2 == 0:
        continue
    print(numero)

print("Ciclo terminato.")
```

Nell'esempio sopra, il ciclo *for* viene iterato su una lista di *numeri* da 1 a 10. Quando il valore di numero è pari (cioè il resto della divisione per 2 è 0), viene eseguita la dichiarazione *continue*, passando direttamente all'iterazione successiva senza eseguire le istruzioni rimanenti all'interno dell'iterazione corrente.

Di conseguenza, l'istruzione *print(numero)* viene eseguita solo per i numeri dispari, escludendo i numeri pari.

Dopo l'esecuzione del ciclo, viene eseguita l'istruzione *print("Ciclo terminato."),* poiché rappresenta l'istruzione successiva al ciclo *for*.

Funzioni

Le funzioni sono blocchi di codice riutilizzabili che eseguono un'azione specifica. Esse consentono di scrivere codice modulare e di suddividere il programma in parti più piccole e gestibili.

Le funzioni built-in

Le funzioni *built-in* di *Python* sono funzioni predefinite che sono incluse nel linguaggio e possono essere utilizzate direttamente senza la necessità di importare moduli aggiuntivi.

Queste funzioni forniscono una vasta gamma di funzionalità utili che possono essere utilizzate per eseguire operazioni comuni o manipolare dati.

Nelle sezioni precedenti abbiamo avuto modo di utilizzare alcune di queste funzioni, come quelle utilizzate nella sezione relativa alle conversioni di tipi.

Ecco alcuni esempi di funzioni *built-in*:

- **print()**: utilizzata per stampare un valore sulla console o in un file di output.
- **len()**: restituisce la lunghezza di un oggetto, come una stringa, una lista o una tupla.
- **type()**: restituisce il tipo di un oggetto. Ad esempio, può essere utilizzato per determinare se una variabile è di tipo intero, stringa o lista.
- **int(), float(), str()**: utilizzate per convertire un valore in un intero, in un valore in virgola mobile o in una stringa, rispettivamente.
- **input()**: permette di acquisire l'input dell'utente da tastiera.
- **range()**: restituisce una sequenza di numeri da un valore iniziale a un valore finale, con un incremento specificato.

- **sum()**: calcola la somma di tutti gli elementi in una sequenza, come una lista o una tupla.
- **max()**, **min()**: restituiscono il valore massimo o minimo in una sequenza di elementi.
- **sorted()**: restituisce una nuova lista ordinata degli elementi di una sequenza.
- **abs()**: restituisce il valore assoluto di un numero.

Questi sono solo alcuni esempi delle molte funzioni *built-in* disponibili. La documentazione ufficiale fornisce una lista completa di queste funzioni e offre dettagli su come utilizzarle correttamente.

Definizione di una funzione

Per definire una funzione, si utilizza la parola chiave *def*, seguita dal nome della funzione e dalla lista dei parametri racchiusa tra parentesi tonde. La definizione termina con i due punti (:).

Il blocco di codice che costituisce la funzione viene indentato rispetto alla riga della definizione.

```
def saluta(nome, età):
    print("Ciao", nome, "!")
    print("Hai", età, "anni.")
```

In questo esempio, abbiamo definito una funzione *saluta*() che prende in input due parametri *nome* e *età* e stampa due messaggi utilizzando i valori dei parametri.

Chiamata di una funzione

Per chiamare una funzione si utilizza il nome della funzione seguito dalla lista degli argomenti racchiusa tra parentesi tonde. Gli argomenti possono essere valori letterali o variabili.

Puoi passare gli argomenti posizionalmente, in base all'ordine dei parametri, o specificandoli esplicitamente con il nome del parametro.

```
saluta("Alice", 25)  # Passaggio argomenti in base # all'ordine dei paramentri
```

```
saluta(età=25, nome="Alice")  # Passaggio argomenti in base # al nome del parametro
```

Valori di ritorno

Una funzione può restituire un valore utilizzando la parola chiave *return*.

Il valore restituito può essere utilizzato come input per altre funzioni o assegnato a una variabile.

```
def calcola_area(base, altezza):
    area = base * altezza
    return area

# Chiamata alla funzione passando gli argomenti posizionalmente
risultato1 = calcola_area(5, 10)

# Chiamata alla funzione specificando esplicitamente i nomi dei parametri
risultato2 = calcola_area(base=7, altezza=3)
```

Nell'esempio sopra, abbiamo la funzione *calcola_area()* che accetta due parametri: *base* e *altezza*. Possiamo chiamare la funzione passando gli argomenti posizionalmente, come *calcola_area(5, 10),* o specificando esplicitamente i nomi dei parametri, come *calcola_area(base=7, altezza=3)*.

In entrambi i casi, gli argomenti passati corrispondono ai parametri definiti nella funzione e vengono utilizzati all'interno del corpo della funzione per eseguire il calcolo dell'area.

Argomenti predefiniti

In *Python*, è possibile definire argomenti predefiniti per una funzione.

Gli argomenti predefiniti sono valori che vengono utilizzati se l'utente non fornisce un valore per quel parametro.

È importante specificare che gli argomenti predefiniti devono essere specificati dopo gli argomenti non predefiniti nella definizione della funzione. Altrimenti, si verificheranno errori di sintassi.

```
def saluta(nome="utente"):
    print("Ciao", nome)

saluta()        #output: Ciao utente
saluta("Marco")  #output: Ciao Marco

# Casistica con ERRORE
def saluta(saluto="Ciao", nome):
    print(saluto, nome)

saluta('Luca')
"""
Ottengo il seguente errore:
    def saluta(saluto="Ciao", nome):
                              ^^^^
SyntaxError: non-default argument follows default argument
```

In questo esempio, abbiamo definito una funzione *saluta()* con un argomento predefinito *nome* impostato su "utente".

Se non viene fornito alcun argomento, la funzione utilizza il valore predefinito. Altrimenti, utilizza il valore fornito dall'utente.

Regole per la creazione di funzioni

Ecco alcune best practice da seguire nella creazione di funzioni:

- **Nomi descrittivi**: utilizza nomi descrittivi per le tue funzioni che riflettano chiaramente ciò che fanno. Un nome appropriato può rendere il codice più leggibile e comprensibile.
- **Docstring**: aggiungi una *docstring* alle tue funzioni per spiegare ciò che fanno e come vengono utilizzate. La docstring dovrebbe descrivere il comportamento della funzione, i parametri che accetta, il valore di ritorno e qualsiasi altra informazione rilevante. Una buona documentazione aiuta altri sviluppatori a capire il funzionamento delle tue funzioni.

```
def calcola_area_rettangolo(base, altezza):
    """
    Calcola l'area di un rettangolo dati la base e l'altezza.

    Args:
        base (float): La base del rettangolo.
        altezza (float): L'altezza del rettangolo.

    Returns:
            float: L'area del rettangolo calcolata come
              prodotto di base e altezza.

    Raises:
        ValueError: Se la base o l'altezza sono negative o nulle.

    """

    if base <= 0 or altezza <= 0:
        raise ValueError("La base e l'altezza devono \
                        essere valori positivi.")

    area = base * altezza
    return area
```

- **Funzioni atomiche e modulari**: scrivi funzioni che siano atomiche e svolgano un singolo compito specifico. Le funzioni troppo lunghe e complesse possono essere difficili da comprendere e da testare. Se una funzione sta diventando troppo grande, considera di suddividerla in funzioni più piccole e riutilizzabili.
- **Parametri e ritorno**: limita il numero di parametri di una funzione per evitare di complicare l'interfaccia. Se una funzione richiedesse molti parametri, potrebbe essere opportuno raggrupparli in una struttura dati composta come un dizionario o un oggetto. Inoltre, assicurati che la funzione restituisca un risultato coerente e ben definito. Se una

funzione non ha un valore di ritorno, usa la parola chiave *None* per indicare l'assenza di un valore significativo.

- **Gestione degli errori**: gestisci gli errori e le eccezioni all'interno delle tue funzioni in modo appropriato. Utilizza blocchi *try-except* (vedi Capitolo 7: Le eccezioni) per catturare e gestire le eccezioni specifiche. Fornisci messaggi di errore significativi per aiutare gli utenti a capire cosa è andato storto.
- **Evita effetti collaterali**: cerca di scrivere funzioni che abbiano effetti collaterali minimi o nulli. Ciò significa che una funzione dovrebbe fare il suo lavoro senza modificare lo stato esterno o causare cambiamenti indesiderati nell'ambiente di esecuzione. Ciò rende le funzioni più prevedibili e facili da testare.
- **Testabilità**: scrivi le tue funzioni in modo che siano facilmente testabili. Isola la logica di business dal codice di interfaccia utente o dal codice di I/O (input/output), in modo da poter testare la logica di business in modo indipendente. Utilizza framework di testing come *pytest* o *unittest* per creare test automatizzati per le tue funzioni.
- **Convenzioni di stile**: segui le convenzioni di stile di *Python*, come quelle definite nelle *PEP 8 (Python Enhancement Proposals),* per rendere il tuo codice coerente e facilmente leggibile. Utilizza l'indentazione corretta, nomi di funzioni in minuscolo con parole separate da underscore e rispetta le linee guida generali di formattazione del codice.
- **Riutilizzabilità**: scrivi le tue funzioni in modo che possano essere riutilizzate in diverse parti del tuo codice o anche in progetti futuri. Cerca di isolare la logica specifica all'interno delle funzioni, in modo che possano essere applicate a situazioni simili in modo più generico.

L'istruzione pass

L'istruzione *pass* è un segnaposto vuoto. Viene utilizzata quando è richiesto un blocco di codice sintatticamente, ma non è necessario eseguire alcuna operazione al suo interno. Può essere considerata come un segnaposto temporaneo per il codice che verrà implementato in seguito.

L'uso più comune di questa istruzione è all'interno delle definizioni di funzioni o classi quando si desidera specificare la struttura senza

implementare ancora il corpo della funzione o della classe.

```
def mia_funzione():
    pass
```

In questo esempio, la funzione *mia_funzione()* non ha alcuna implementazione. Tuttavia, la presenza di *pass* consente di definire correttamente la funzione senza generare errori di sintassi. Questo è utile quando si lavora su un progetto in cui si vuole definire un'interfaccia o una struttura generale prima di implementare i dettagli specifici della funzione.

Un altro caso d'uso comune di *pass* è all'interno di istruzioni di controllo come *if, while* o *for* in cui è richiesto un blocco di codice, ma non si desidera eseguire alcuna operazione al suo interno.

```
condizione = True

if condizione:
    pass
else:
    print("")
    # Altre istruzioni qui
```

In questo caso, l'istruzione *pass* viene utilizzata come segnaposto nel blocco *if*, indicando che non sono necessarie operazioni specifiche al momento. Tuttavia, si può aggiungere il codice necessario in seguito senza dover modificare la struttura condizionale.

È importante notare che l'istruzione *pass* non fa nulla in termini di esecuzione del codice. È semplicemente una dichiarazione vuota che aiuta a mantenere la coerenza sintattica del programma. Quando si utilizza *pass*, è necessario assicurarsi di implementare successivamente il codice corretto in modo da ottenere il comportamento desiderato.

Moduli e librerie

I moduli sono file che contengono definizioni di funzioni, classi(vedi Capitolo 5 : La Programmazione a Oggetti) e variabili.

Essi consentono di organizzare il codice in modo logico e di suddividere il programma in parti più piccole e gestibili. In questa sezione vedremo come creare e utilizzare i moduli in *Python*, nonché le librerie di terze parti disponibili.

Creazione di un modulo

Per creare un modulo, basta scrivere il codice che si desidera condividere e salvarlo con l'estensione *.py*. Per utilizzare le definizioni contenute nel modulo in un altro file, è possibile utilizzare la parola chiave *import*. Ad esempio, supponiamo di voler creare un modulo chiamato *mio_modulo.py* contenente una funzione *saluta()*:

```
def saluta(nome):
    print("Ciao", nome)
```

Per utilizzare la funzione *saluta()* in un altro file, possiamo importare il modulo utilizzando la parola chiave *import*:

```
import mio_modulo

mio_modulo.saluta("Marco")
```

In questo esempio, stiamo importando il modulo *mio_modulo* e chiamando la funzione *saluta* utilizzando la notazione con il punto.

Importazione selettiva

È anche possibile importare solo alcune definizioni da un modulo utilizzando la sintassi *from ... import ....*

Ad esempio, se vogliamo importare solo la funzione *saluta* dal modulo *mio_modulo*:

```
from mio_modulo import saluta

saluta("Marco")
```

In questo esempio, stiamo importando solo la funzione saluta() dal modulo *mio_modulo*, senza dover usare la notazione con il punto.

Organizzazione dei moduli in pacchetti

Quando un'applicazione diventa più complessa e richiede un insieme di moduli correlati, è utile organizzare i moduli in pacchetti. Un pacchetto è semplicemente una cartella che contiene uno o più moduli *Python*.

Ad esempio, supponiamo di avere un pacchetto chiamato *mio_pacchetto* che contiene due moduli: *modulo1.py* e *modulo2.py*. La struttura del pacchetto sarà la seguente:

```
mio_pacchetto/
    __init__.py
    modulo1.py
    modulo2.py
```

La direttiva init.py

All'interno di un pacchetto, è possibile creare un file chiamato *init.py*. Questo file speciale viene eseguito automaticamente quando il pacchetto viene importato. È possibile utilizzare questo file per eseguire alcune operazioni di inizializzazione o per specificare quali definizioni sono importate quando si importa il pacchetto.

Ad esempio, supponiamo che nel nostro pacchetto *mio_pacchetto* vogliamo esportare solo la funzione *saluta()* dal *modulo1*.

Possiamo modificare il file init.py come segue:

```
# File: __init__.py
from .modulo1 import saluta
```

Ora, quando importiamo il pacchetto *mio_pacchetto*, avremo accesso solo alla funzione *saluta()*:

```
import mio_pacchetto

mio_pacchetto.saluta("Mario")
```

Librerie di terze parti

In *Python*, ci sono numerose librerie di terze parti disponibili che forniscono funzionalità aggiuntive al linguaggio. Le librerie possono essere installate utilizzando il package manager di *Python*, *pip* .

Pip: python install packages

pip è uno strumento di gestione dei pacchetti che facilita l'installazione e la gestione di librerie e moduli aggiuntivi. Il nome *pip* è un acronimo per *pip installs packages* (pip installa pacchetti).

Ecco alcune informazioni chiave su *pip*:

- **Installazione dei pacchetti**: pip consente di installare pacchetti esterni da PyPI (Python Package Index) e da altre origini. *PyPI* è il repository ufficiale delle librerie *Python*, che contiene migliaia di pacchetti pronti per l'uso. Puoi utilizzare il comando *pip install <nome_pacchetto>* per installare un pacchetto specifico.
- **Gestione delle dipendenze**: *pip* gestisce automaticamente le dipendenze tra i pacchetti. Quando installi un pacchetto, *pip* verificherà se ci sono altre librerie o moduli richiesti e li installerà automaticamente se necessario. Questo semplifica notevolmente la gestione delle dipendenze del tuo progetto.
- **Aggiornamento dei pacchetti**: pip consente di aggiornare i pacchetti installati alla versione più recente disponibile. Puoi utilizzare il comando *pip install --upgrade <nome_pacchetto>* per aggiornare un pacchetto specifico o *pip install --upgrade* per aggiornare tutti i pacchetti installati.
- **Disinstallazione dei pacchetti**: pip consente anche di disinstallare pacchetti che non sono più necessari. Puoi utilizzare il comando *pip uninstall <nome_pacchetto>* per rimuovere un pacchetto specifico.
- **Requisiti di progetto**: *pip* consente di specificare le dipendenze di un progetto in un file chiamato *requirements.txt.* Questo file elenca tutti i pacchetti necessari per eseguire il tuo progetto e le rispettive versioni. Puoi quindi utilizzare il comando *pip install -r requirements.txt* per installare tutti i pacchetti elencati nel file.

  Ecco un esempio di un file *requirements.txt* che può essere utilizzato per specificare le dipendenze di un progetto:

```
requests==2.25.1
```

```
numpy>=1.19.5
pandas==1.2.4
matplotlib==3.3.4
```

Nell'esempio sopra, ogni riga rappresenta una dipendenza specifica con la seguente sintassi:

```
nome_dipendenza==versione
```

- ➢ **nome_dipendenza**: è il nome del pacchetto o della libreria di cui il progetto ha bisogno.
- ➢ **versione**: specifica la versione esatta della dipendenza richiesta. Può anche includere operatori di confronto come >=, <=, >, <, ~=, a seconda delle esigenze del progetto.

❖ **Personalizzazione dell'ambiente**: *pip* può essere utilizzato con vari flag e opzioni per personalizzare l'installazione dei pacchetti. Ad esempio, è possibile specificare una versione specifica di un pacchetto, installare pacchetti da file locali, installare pacchetti in una directory personalizzata, ecc.

Esempio di installazione e utilizzo di una libreria esterna

Ad esempio, per installare la libreria *NumPy* (Numerical Python, libreria molto popolare utilizzata per eseguire calcoli scientifici e matematici complessi):

```
pip install numpy
```

Dopo l'installazione, è possibile utilizzare le definizioni della libreria in un programma *Python* utilizzando l'importazione normale o selettiva.

Ad esempio, se vogliamo utilizzare la funzione *dot* per calcolare il prodotto scalare di due vettori:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.dot(a, b)
```

```
print(c)
```

In questo esempio, stiamo importando la libreria *numpy* utilizzando l'alias *np*, e utilizziamo la funzione *dot* per calcolare il prodotto scalare di due vettori.

Input/output

L'input e l'output si riferiscono rispettivamente all'acquisizione e alla visualizzazione dei dati. Questo processo è essenziale in qualsiasi applicazione, poiché permette agli utenti di interagire con il programma e di visualizzare i risultati.

In *Python*, l'input può essere acquisito da diverse fonti, tra cui la tastiera, un file di testo o un database. La funzione *input()* consente di acquisire l'input dall'utente attraverso la tastiera. Ad esempio, il seguente codice richiede all'utente di inserire il proprio nome e lo salva in una variabile:

```
nome = input("Inserisci il tuo nome: ")
```

L'output può essere visualizzato sulla console o scritto su un file o su un altro dispositivo di output. La funzione *print()* è la principale utilizzata per visualizzare l'output sulla console. Ad esempio, il seguente codice stampa una stringa sulla console:

```
print("Ciao, mondo!")
```

Esercizi

Esercizio 1

Scrivi 3 funzioni che svolgono i seguenti compiti:

- La prima funzione, stampa_numeri(n), accetta un parametro n e stampa i numeri da 1 a n utilizzando un ciclo while. Esegui la funzione stampa_numeri(5) e riporta l'output risultante.

- La seconda funzione, stampa_numeri_pari(n), accetta un parametro n e stampa solo i numeri pari da 1 a n utilizzando un ciclo for e un'istruzione if-else. Esegui la funzione stampa_numeri_pari(10) e riporta l'output risultante.
- La terza funzione, calcola_fattoriale(n), accetta un parametro n e calcola il fattoriale del numero n utilizzando un ciclo for. Restituisce il risultato del calcolo del fattoriale. Esegui la funzione calcola_fattoriale(6) e riporta il valore restituito.

```
def stampa_numeri(n):
    """
    Stampa i numeri da 1 a n.

    Args:
        n (int): Limite superiore dei numeri da stampare.
    """
    print("Numeri da 1 a", n)
    i = 1
    while i <= n:
        print(i)
        i += 1

def stampa_numeri_pari(n):
    """
    Stampa i numeri pari da 1 a n.

    Args:
        n (int): Limite superiore dei numeri da stampare.
    """
    print("Numeri pari da 1 a", n)
    for i in range(1, n+1):
        if i % 2 == 0:
            print(i)

def calcola_fattoriale(n):
```

```
    """
    Calcola il fattoriale di un numero.

    Args:
        n (int): Numero intero positivo.

    Returns:
        int: Il fattoriale del numero n.
    """
    fattoriale = 1
    for i in range(1, n+1):
        fattoriale *= i
    return fattoriale

# Esempio di utilizzo delle funzioni

# Utilizzo del ciclo while
stampa_numeri(5)
print()

# Utilizzo del ciclo for e dell'istruzione if-else
stampa_numeri_pari(10)
print()

# Utilizzo del ciclo for e calcolo del fattoriale
numero = 6
risultato = calcola_fattoriale(numero)
print("Il fattoriale di", numero, "è", risultato)
```

Esercizio 2

Scrivi 2 funzioni che svolgano i seguenti compiti:

- La prima funzione, somma_numeri_pari(n), accetta un parametro n e calcola la somma dei numeri pari da 1 a n utilizzando un ciclo while e un'istruzione condizionale. Esegui la funzione somma_numeri_pari(10) e riporta il valore restituito.
- La seconda funzione, conta_vocali(parola), accetta una parola come parametro e conta il numero di vocali presenti nella parola utilizzando un ciclo for, un'istruzione condizionale e un'operazione di confronto tra caratteri. Esegui la funzione conta_vocali("OpenAI") e riporta il valore restituito.

```
def somma_numeri_pari(n):
    """
    Calcola la somma dei numeri pari da 1 a n.

    Args:
        n (int): Limite superiore dei numeri da sommare.

    Returns:
        int: La somma dei numeri pari da 1 a n.
    """
    somma = 0
    i = 1
    while i <= n:
        if i % 2 == 0:
            somma += i
        i += 1
    return somma

def conta_vocali(parola):
    """
    Conta il numero di vocali in una parola.

    Args:
        parola (str): Parola di cui contare le vocali.
```

```
    Returns:
        int: Il numero di vocali nella parola.
    """
    vocali = 'aeiouAEIOU'
    conteggio = 0
    for lettera in parola:
        if lettera in vocali:
            conteggio += 1
    return conteggio

# Esempio di utilizzo delle funzioni

# Task 1: Calcolare la somma dei numeri pari
numero_limite = 10
risultato_somma = somma_numeri_pari(numero_limite)
print(f"La somma dei numeri pari da 1 a {numero_limite} è: {risultato_somma}")

# Task 2: Contare il numero di vocali
parola_input = "OpenAI"
risultato_conteggio = conta_vocali(parola_input)
print(f"Il numero di vocali nella parola '{parola_input}' è: {risultato_conteggio}")
```

Capitolo 4: Le Strutture Dati

Le strutture dati sono un elemento fondamentale nella programmazione, e *Python* ne offre una vasta gamma integrate nel linguaggio, che possono essere utilizzate per risolvere una varietà di problemi. Le più comuni sono le liste, le tuple, i dizionari e gli insiemi. Vediamole insieme nelle loro caratteristiche fondamentali.

Le liste

Le liste sono una delle strutture dati più utilizzate.

Una lista è un'insieme ordinato di elementi, dove ogni elemento può essere di qualsiasi tipo di dati.

Le liste sono oggetti mutabili, il che significa che possiamo aggiungere, rimuovere o modificare gli elementi dopo la loro creazione.

Gli elementi di una lista sono indicizzati, il che implica che ogni elemento ha un indice univoco che può essere utilizzato per accedere all'elemento stesso.

Di seguito vediamo le principali operazioni.

- **Creazione di una lista**

  Per creare una lista, si utilizzano le parentesi quadre []. I suoi elementi sono separati da virgole. Ad esempio, per creare una lista di numeri interi, possiamo scrivere:

```
numeri = [1, 2, 3, 4, 5]
```

  Possiamo anche creare una lista di stringhe:

```
frutta = ["mela", "banana", "arancia", "kiwi"]
```

  E' possibile anche creare liste che contengono elementi di tipi misti:

```
list_mista = [1, "due", 3.0, [4, 5], {"sei": 6}]
```

- **Accesso agli elementi di una lista**

Possiamo accedere agli elementi di una lista utilizzando l'indice dell'elemento desiderato all'interno delle parentesi quadre. Gli indici delle liste in *Python* partono da zero, il che significa che il primo elemento ha indice 0, il secondo ha indice 1 e così via. Ad esempio, per accedere al primo elemento della lista *numeri* precedentemente definita, possiamo scrivere:

```
print(numeri[0]) #output: 1
```

Possiamo anche utilizzare gli indici negativi per accedere agli elementi della lista partendo dalla fine. Ad esempio, per accedere all'ultimo elemento della lista *frutta*, possiamo scrivere:

```
print(frutta[-1])   #output: Kiwi
```

❖ Aggiungere elementi a una lista

Possiamo aggiungere un elemento ad una lista usando il metodo *append()* o *extend()*. Il metodo *append()* ci permette di aggiungere un elemento alla fine della lista, mentre il metodo *extend()* ci permette di estendere la lista aggiungendo una serie di elementi.

```
lista = [1, 2, 3]

# Aggiungiamo un elemento alla fine usando append()
lista.append(4)
print(lista) # Output: [1, 2, 3, 4]

# Aggiungiamo più elementi alla fine usando extend()
lista.extend([5, 6, 7])
print(lista) # Output: [1, 2, 3, 4, 5, 6, 7]
```

❖ Rimuovere elementi da una lista

Possiamo rimuovere un elemento da una lista usando il metodo *remove()* o *pop()*.

Il metodo *remove()* ci permette di rimuovere un elemento specifico, mentre il metodo *pop()* consente di rimuovere un elemento usando l'indice.

```
lista = [1, 2, 3, 4, 5]

# Rimuoviamo l'elemento 3 dalla lista usando remove()
lista.remove(3)
print(lista) # Output: [1, 2, 4, 5]

# Rimuoviamo l'elemento con indice 2 dalla lista usando pop()
lista.pop(2)
print(lista) # Output: [1, 2, 5]
```

Quando si utilizza il metodo *remove()* bisogna essere certi che l'elemento che si vuole rimuovere sia presente nella lista.

Vediamo come gestire un eventuale assenza dell'elemento.

```
lista = [1, 2, 3, 4, 5]

elemento_da_rimuovere = 6

try:
    lista.remove(elemento_da_rimuovere)
except ValueError:
    print("Errore: l'elemento non è presente nella lista.")
```

In questo esempio ho introdotto la gestione delle eccezioni, argomento che verrà trattato in seguito (vedi Capitolo 7: Le eccezioni). Per adesso ti basta sapere che certe casistiche vanno gestite per scrivere un codice funzionante in qualsiasi circostanza.

❖ Ordinare una lista

Possiamo ordinare gli elementi di una lista usando il metodo *sort()*.

```
lista = [5, 2, 8, 1, 9]
```

```
# Ordiniamo la lista in ordine decrescente
lista.sort(reverse=True)
print(lista) #output: [9, 8, 5, 2, 1]

# Ordiniamo la lista in ordine crescente
lista.sort(reverse=False)
print(lista) #output: [1, 2, 5, 8, 9]
```

- **Unire due liste**

Possiamo unire due liste usando l'operatore +.

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]

# Uniamo le due liste usando l'operatore +
lista3 = lista1 + lista2
print(lista3) # Output: [1, 2, 3, 4, 5, 6]
```

- **Ottenere la lunghezza di una lista**

Possiamo ottenere la lunghezza di una lista usando la funzione *len()*.

```
lista = [1, 2, 3, 4, 5]

# Otteniamo la lunghezza della lista usando len()
lunghezza = len(lista)
print(lunghezza) # Output: 5
```

Questi sono solo alcuni esempi delle operazioni che possiamo eseguire; tuttavia, esistono molte altre operazioni e metodi che possono essere utilizzati per manipolare e analizzare le liste.

List comprehension

Una *list comprehension* è un costrutto sintattico che consente di creare una nuova lista in modo conciso e leggibile. Consiste nell'esprimere la definizione di una lista in una singola riga di codice, utilizzando una combinazione di espressioni, cicli e condizioni.

La sintassi di base è la seguente:

```
nuova_list = [espressione for elemento in sequenza if condizione]
```

Nell'espressione, si specifica cosa deve essere incluso nella nuova lista. La variabile "elemento" rappresenta ciascun elemento della sequenza, che può essere una lista, una tupla, un range o qualsiasi altro oggetto iterabile. La condizione, facoltativa, filtra gli elementi sulla base di una condizione specifica.

```
numeri = [1, 2, 3, 4, 5]

# Creazione di una nuova lista che contiene il quadrato di ogni numero
numeri_quadrati = [x**2 for x in numeri]
print(numeri_quadrati)  # Output: [1, 4, 9, 16, 25]

# Creazione di una nuova lista che contiene solo i numeri pari
numeri_pari = [x for x in numeri if x % 2 == 0]
print(numeri_pari)  # Output: [2, 4]
```

Il metodo range

Il metodo *range()* è una funzione integrata in *Python* che viene utilizzata per generare una sequenza di numeri. Questa sequenza è comunemente utilizzata per iterare attraverso un loop o per indicizzare elementi in una struttura dati come una lista.

La sintassi generale è la seguente:

```
range([start], stop, [step])
```

I parametri di *range()* sono:

- ➢ start (opzionale): il valore di partenza della sequenza. Se non specificato, di default è 0.
- ➢ stop (obbligatorio): il valore di fine della sequenza. La sequenza generata si ferma prima di raggiungere questo valore.
- ➢ step (opzionale): l'incremento tra i numeri successivi nella sequenza. Se non specificato, di default è 1.

Vediamo alcuni esempi:

```
for i in range(10):
    print(i)
# output: 0 1 2 3 4 5 6 7 8 9
```

```
for i in range(1, 11, 2):
    print(i)
# output: 1 3 5 7 9

numbers = list(range(1, 6))
print(numbers)
# output [1, 2, 3, 4, 5]
```

Conversione di una stringa in una lista

Una delle operazioni più comuni nel lavoro con le stringhe è la suddivisione della stringa in una lista di elementi. *Python* fornisce il metodo *split* che ci consente di fare proprio questo.

La sintassi generale per utilizzare il comando *split* è la seguente:

```
stringa.split(separatore, massimo)
```

- **stringa** rappresenta la stringa che desideriamo suddividere.
- **separatore** è il carattere (o la sequenza di caratteri) utilizzato come delimitatore per suddividere la stringa. Questo argomento è opzionale e se non viene fornito, verrà utilizzato uno spazio come separatore predefinito.

- **massimo** è un argomento opzionale che specifica il numero massimo di suddivisioni che vogliamo eseguire sulla stringa. Se fornito, il risultato sarà una lista con un massimo di massimo + 1 elementi.

Ecco un esempio pratico:

```
stringa = "Questo è un esempio di stringa"
lista = stringa.split()
print(lista)
#output: ['Questo', 'è', 'un', 'esempio', 'di', 'stringa']
```

Nell'esempio sopra, abbiamo utilizzato il metodo *split* senza fornire alcun separatore specifico, quindi è stato utilizzato uno spazio come separatore predefinito. La stringa è stata suddivisa in una lista di parole, e il risultato è stato assegnato alla variabile lista.

È anche possibile specificare un separatore personalizzato come argomento del metodo split. Ad esempio:

```
stringa = "10-05-2023"
lista = stringa.split("-")
print(lista)
#output: ['10', '05', '2023']
```

In questo esempio, abbiamo utilizzato il carattere "-" come separatore. La stringa è stata suddivisa in una lista di tre elementi, corrispondenti ai tre numeri separati da "-".

Si ricorda che il comando split restituisce una lista di elementi e, quindi, è possibile assegnare il risultato a una variabile per ulteriori manipolazioni o analisi.

Le tuple

Le tuple sono una struttura di dati immutabile. Questo significa che, una volta create, non possono essere modificate. Sono molto utili quando si

desidera memorizzare un insieme di valori che non devono essere alterati.

❖ Creazione di una tupla

Per creare una tupla, è possibile utilizzare le parentesi tonde o la funzione *tuple()*.

```
# Creazione di una tupla con parentesi tonde
tupla1 = (1, 2, 3, 4, 5)

# Creazione di una tupla con la funzione tuple()
tupla2 = tuple(['a', 'b', 'c'])
```

❖ Accesso agli elementi di una tupla

È possibile accedere agli elementi di una tupla utilizzando l'operatore di indicizzazione.

L'indicizzazione inizia da 0 per il primo elemento.

```
tupla = ('a', 'b', 'c', 'd', 'e')

print(tupla[0])  # Output: 'a'

print(tupla[2])  # Output: 'c'
print(tupla[-1]) # Output: 'e' (accesso all'ultimo elemento)
```

❖ Iterazione attraverso una tupla

È possibile iterare attraverso una tupla utilizzando un ciclo *for*.

```
tupla = ('apple', 'banana', 'cherry')

for elemento in tupla:
    print(elemento)

# Output:
# apple
# banana
```

```
# cherry
```

❖ Slicing di una tupla

Lo slicing permette di ottenere una porzione della tupla.

```
tupla = (1, 2, 3, 4, 5)

sottotupla = tupla[1:4]
print(sottotupla)  # Output: (2, 3, 4)
```

❖ Lunghezza di una tupla

È possibile ottenere la lunghezza di una tupla utilizzando la funzione *len()*.

```
tupla = ('a', 'b', 'c', 'd', 'e')

lunghezza = len(tupla)
print(lunghezza)  # Output: 5
```

❖ Tuple annidate

È possibile creare tuple annidate, ovvero tuple all'interno di tuple.

```
tupla_esterna = (1, 2, 3)
tupla_interna = ('a', 'b', 'c')
tupla_annidata = (tupla_esterna, tupla_interna)

print(tupla_annidata)  # Output: ((1, 2, 3), ('a', 'b', 'c'))
```

❖ Assegnamento multi-valore con le tuple

È possibile assegnare i valori di una tupla a più variabili contemporaneamente.

```
tupla = (10, 20, 30)
```

```
a, b, c = tupla

print(a)  # Output: 10
print(b)  # Output: 20
print(c)  # Output: 30
```

In questo esempio, i valori della tupla vengono assegnati alle variabili a, b e c rispettivamente. Questo tipo di assegnamento multi-valore è molto utile quando si desidera decomporre una tupla in diverse variabili.

I dizionari

I dizionari sono una struttura dati flessibile che permette di associare coppie chiave-valore. A differenza delle liste e delle tuple, i dizionari non sono ordinati e gli elementi vengono recuperati tramite la chiave anziché l'indice.

- **Creazione di un dizionario**

  Per creare un dizionario, è possibile utilizzare le parentesi graffe {} o la funzione *dict()*.

```
# Creazione di un dizionario con le parentesi graffe
dizionario = {
    "nome": "Mario",
    "cognome": "Rossi",
    "età": 30,
    "città": "Roma"
}

print(dizionario)
#output: {'nome': 'Mario', 'cognome': 'Rossi', 'età': 30, \
#        'città': 'Roma'}

# Creazione di un dizionario con la funzione dict()
chiavi = ["nome", "cognome", "età"]
valori = ["Mario", "Rossi", 30]
```

```
# Utilizzo la funzione built-in zip() per combinare le due
# sequenze in coppie chiave-valore.
dizionario = dict(zip(chiavi, valori))
print(dizionario)
#output: {'nome': 'Mario', 'cognome': 'Rossi', 'età': 30}
```

❖ Accesso agli elementi di un dizionario

È possibile accedere agli elementi di un dizionario utilizzando la chiave come indice.

```
dizionario = {'nome': 'Alice', 'età': 25, 'città': 'Roma'}

print(dizionario['nome'])  # Output: 'Alice'
print(dizionario['età'])   # Output: 25
print(dizionario['città']) # Output: 'Roma'
```

❖ Modifica degli elementi di un dizionario

È possibile modificare gli elementi di un dizionario assegnando un nuovo valore alla chiave desiderata.

```
dizionario = {'nome': 'Alice', 'età': 25, 'città': 'Roma'}

dizionario['età'] = 26

print(dizionario)
#output: {'nome': 'Alice', 'età': 26, 'città': 'Roma'}
```

❖ Aggiunta di nuovi elementi a un dizionario

È possibile aggiungere nuove coppie chiave-valore a un dizionario assegnando un valore a una nuova chiave.

```
dizionario = {'nome': 'Alice', 'età': 25}

dizionario['città'] = 'Roma'
```

```
print(dizionario)
# Output: {'nome': 'Alice', 'età': 25, 'città': 'Roma'}
```

❖ Rimozione di elementi da un dizionario

È possibile rimuovere un elemento da un dizionario utilizzando l'operatore del} seguito dalla chiave dell'elemento da eliminare.

```
dizionario = {'nome': 'Alice', 'età': 25, 'città': 'Roma'}

del dizionario['età']

print(dizionario)  # Output: {'nome': 'Alice', 'città': 'Roma'}
```

❖ Iterazione attraverso un dizionario

È possibile iterare attraverso un dizionario utilizzando un ciclo *for*. Durante l'iterazione, si otterranno le chiavi del dizionario.

```
dizionario = {'nome': 'Alice', 'età': 25, 'città': 'Roma'}

for chiave in dizionario:
    valore = dizionario[chiave]
    print(chiave, valore)

# Output:
# nome Alice
# età 25
# città Roma
```

In questo esempio, abbiamo utilizzato un ciclo *for* per iterare attraverso il dizionario. Durante ogni iterazione, abbiamo ottenuto una chiave e abbiamo utilizzato quella chiave per accedere al valore corrispondente nel dizionario.

❖ Verifica dell'esistenza di una chiave in un dizionario

È possibile verificare se una chiave esiste in un dizionario utilizzando l'operatore *in*.

```
dizionario = {'nome': 'Alice', 'età': 25, 'città': 'Roma'}

if 'nome' in dizionario:
    print("La chiave 'nome' esiste nel dizionario.")
else:
    print("La chiave 'nome' non esiste nel dizionario.")

# Output: La chiave 'nome' esiste nel dizionario.
```

❖ Metodo keys(), values() e items()

Esistono tre metodi utili per ottenere rispettivamente le chiavi, i valori o le coppie chiave-valore di un dizionario.

```
dizionario = {'nome': 'Alice', 'età': 25, 'città': 'Roma'}

chiavi = dizionario.keys()
valori = dizionario.values()
coppie = dizionario.items()

print(chiavi)  # Output: dict_keys(['nome', 'età', 'città'])
print(valori)  # Output: dict_values(['Alice', 25, 'Roma'])
print(coppie)  # Output: dict_items([('nome', 'Alice'), ('età', 25),
               #                     ('città', 'Roma')])
```

Gli insiemi

Gli insiemi sono una collezione di elementi unici e non ordinati. Sono utili quando si desidera lavorare con una collezione di elementi senza duplicati e senza la necessità di mantenere un ordine specifico.

❖ Creazione di un insieme

Per creare un insieme è possibile utilizzare le parentesi graffe {} o la funzione *set()*.

```
# Creazione di un insieme con le parentesi graffe
insieme1 = {1, 2, 3, 4, 5}

# Creazione di un insieme con la funzione set()
insieme2 = set([1, 2, 3, 4, 5])
```

❖ Aggiunta di elementi a un insieme

È possibile aggiungere elementi a un insieme utilizzando il metodo *add()*.

```
insieme = {1, 2, 3}

insieme.add(4)

print(insieme)  # Output: {1, 2, 3, 4}
```

❖ Rimozione di elementi da un insieme

È possibile rimuovere un elemento da un insieme utilizzando il metodo *remove()* o il metodo *discard()*.

```
insieme = {1, 2, 3, 4}

insieme.remove(3)

print(insieme)  # Output: {1, 2, 4}

insieme.discard(2)

print(insieme)  # Output: {1, 4}
```

La differenza tra *remove()* e *discard()* è che *remove()* genererà un'eccezione se l'elemento non è presente nell'insieme (come visto in precedenza nella sezione sulle liste), mentre *discard()* non genererà alcuna eccezione.

```
insieme = {1, 2, 3, 4, 5}

try:
```

```
    # Tentativo di rimuovere l'elemento 6 dall'insieme
    insieme.remove(6)
except KeyError:
    print("L'elemento non esiste nell'insieme!")

print(insieme)

# Tentativo di rimuovere l'elemento 6 dall'insieme
# Non verrà sollevata nessuna eccezione.
insieme.discard(6)
```

❖ Operazioni sugli insiemi

Gli insiemi supportano diverse operazioni comuni come l'unione, l'intersezione e la differenza.

```
insieme1 = {1, 2, 3}
insieme2 = {2, 3, 4}

# Unione di due insiemi
unione = insieme1.union(insieme2)
print(unione)  # Output: {1, 2, 3, 4}

# Intersezione di due insiemi
intersezione = insieme1.intersection(insieme2)
print(intersezione)  # Output: {2, 3}

# Differenza tra due insiemi
differenza = insieme1.difference(insieme2)
print(differenza)  # Output: {1}
```

❖ Verifica dell'appartenenza di un elemento a un insieme

È possibile verificare se un elemento appartiene a un insieme utilizzando l'operatore *in*.

```
insieme = {1, 2, 3, 4, 5}
```

```
if 3 in insieme:
    print("3 appartiene all'insieme.")

# Output: 3 appartiene all'insieme.
```

- **Iterazione attraverso un insieme**

  È possibile iterare attraverso gli elementi di un insieme utilizzando un ciclo *for*. Durante l'iterazione, si otterranno gli elementi dell'insieme senza un ordine specifico.

```
insieme = {1, 2, 3, 4, 5}

for elemento in insieme:
    print(elemento)

# Output:
# 1
# 2
# 3
# 4
# 5
```

Esercizi

Esercizio 1

Scrivi 2 funzioni che permettano di eseguire i seguenti compiti:

- La prima funzione, calcola_media(lista_numeri), accetta una lista di numeri come parametro e calcola la media di questi numeri utilizzando la funzione sum() e operazioni aritmetiche. Esegui la funzione calcola_media([10, 20, 30, 40, 50]) e riporta il valore restituito.
- La seconda funzione, trova_parole_comuni(testo1, testo2), accetta due testi come parametri e trova le parole comuni tra i due testi utilizzando le operazioni con stringhe, set e metodi come split() e intersection(). Esegui la funzione trova_parole_comuni("Questo è un

esempio di testo.", "Questo è un altro esempio di testo.") e riporta l'insieme delle parole comuni.

```
def calcola_media(lista_numeri):
    """
    Calcola la media dei numeri nella lista.

    Args:
        lista_numeri (list): Lista di numeri.

    Returns:
        float: La media dei numeri nella lista.
    """
    somma = sum(lista_numeri)
    media = somma / len(lista_numeri)
    return media

def trova_parole_comuni(testo1, testo2):
    """
    Trova le parole comuni tra due testi.

    Args:
        testo1 (str): Primo testo.
        testo2 (str): Secondo testo.

    Returns:
        set: Insieme delle parole comuni tra i due testi.
    """
    parole1 = set(testo1.split())
    parole2 = set(testo2.split())
    parole_comuni = parole1.intersection(parole2)
    return parole_comuni

# Esempio di utilizzo delle funzioni
```

```python
# Task 1: Calcolare la media di una lista di numeri
lista_numeri = [10, 20, 30, 40, 50]
media = calcola_media(lista_numeri)
print(f"La media dei numeri nella lista {lista_numeri} è: {media}")

# Task 2: Trovare le parole comuni tra due testi
testo1 = "Questo è un esempio di testo."
testo2 = "Questo è un altro esempio di testo."
parole_comuni = trova_parole_comuni(testo1, testo2)
print(f"Le parole comuni tra i due testi sono: {parole_comuni}")
```

Esercizio 2

Scrivi 5 funzioni che permettano di svolgere i seguenti compiti:

- La funzione genera_dizionario_studenti(lista_studenti) genera un dizionario degli studenti a partire da una lista di studenti, inizializzando i loro voti a 0. Esegui la funzione con una lista di studenti specificata e riporta il dizionario degli studenti risultante.
- La funzione registra_voto(dizionario_studenti, studente, voto) registra il voto di uno studente nel dizionario degli studenti. Esegui la funzione per registrare i voti degli studenti specificati nel dizionario degli studenti.
- La funzione calcola_media_voti(dizionario_studenti) calcola la media dei voti degli studenti nel dizionario degli studenti. Esegui la funzione per calcolare la media dei voti degli studenti e riporta il valore risultante.
- La funzione trova_studenti_con_voto_alto(dizionario_studenti, soglia) trova gli studenti con un voto superiore a una soglia specificata. Esegui la funzione per trovare gli studenti con un voto superiore alla soglia specificata e riporta l'insieme degli studenti risultante.
- La funzione conta_studenti_per_voto(dizionario_studenti) conta il numero di studenti per ogni voto nel dizionario degli studenti. Esegui la funzione per contare il numero di studenti per ogni voto e riporta il dizionario risultante con il conteggio degli studenti per ogni voto.

```
def genera_dizionario_studenti(lista_studenti):
    """
    Genera un dizionario degli studenti a partire da una lista di studenti.

    Args:
        lista_studenti (list): Lista degli studenti.

    Returns:
        dict: Dizionario degli studenti con i rispettivi voti inizializzati a 0.
    """
    dizionario_studenti = {}
    for studente in lista_studenti:
        dizionario_studenti[studente] = 0
    return dizionario_studenti

def registra_voto(dizionario_studenti, studente, voto):
    """
    Registra il voto di uno studente nel dizionario degli studenti.

    Args:
        dizionario_studenti (dict): Dizionario degli studenti con i rispettivi voti.
        studente (str): Nome dello studente.
        voto (int): Voto dello studente.
    """
    dizionario_studenti[studente] = voto

def calcola_media_voti(dizionario_studenti):
    """
    Calcola la media dei voti degli studenti nel dizionario degli studenti.

    Args:
        dizionario_studenti (dict): Dizionario degli studenti con i rispettivi voti.

    Returns:
```

```
    float: La media dei voti degli studenti.
    """
    voti = list(dizionario_studenti.values())
    media = sum(voti) / len(voti)
    return media

def trova_studenti_con_voto_alto(dizionario_studenti, soglia):
    """
    Trova gli studenti con un voto superiore alla soglia specificata.

    Args:
        dizionario_studenti (dict): Dizionario degli studenti con i rispettivi voti.
        soglia (int): Soglia del voto.

    Returns:
        set: Insieme degli studenti con un voto superiore alla soglia.
    """
    studenti_con_voto_alto = set()
    for studente, voto in dizionario_studenti.items():
        if voto > soglia:
            studenti_con_voto_alto.add(studente)
    return studenti_con_voto_alto

def conta_studenti_per_voto(dizionario_studenti):
    """
    Conta il numero di studenti per ogni voto nel dizionario degli studenti.

    Args:
        dizionario_studenti (dict): Dizionario degli studenti con i rispettivi voti.

    Returns:
        dict: Dizionario con il conteggio degli studenti per ogni voto.
    """
    conteggio_studenti_per_voto = {}
```

```
    for voto in dizionario_studenti.values():
        if voto in conteggio_studenti_per_voto:
            conteggio_studenti_per_voto[voto] += 1
        else:
            conteggio_studenti_per_voto[voto] = 1
    return conteggio_studenti_per_voto

# Esempio di utilizzo delle funzioni

# Task 1: Generare un dizionario degli studenti
lista_studenti = ["Alice", "Bob", "Charlie"]
dizionario_studenti = genera_dizionario_studenti(lista_studenti)
print("Dizionario degli studenti:", dizionario_studenti)
print()

# Task 2: Registare i voti degli studenti
registra_voto(dizionario_studenti, "Alice", 85)
registra_voto(dizionario_studenti, "Bob", 92)
registra_voto(dizionario_studenti, "Charlie", 78)
print("Dizionario degli studenti dopo aver registrato i voti:", dizionario_studenti)
print()

# Task 3: Calcolare la media dei voti degli studenti
media_voti = calcola_media_voti(dizionario_studenti)
print("Media dei voti degli studenti:", media_voti)
print()

# Task 4: Trovare gli studenti con un voto alto
soglia_voto_alto = 80
studenti_voto_alto = trova_studenti_con_voto_alto(dizionario_studenti, soglia_voto_alto)
print(f"Gli studenti con un voto superiore a {soglia_voto_alto} sono:", studenti_voto_alto)
print()

# Task 5: Contare gli studenti per ogni voto
conteggio_studenti_per_voto = conta_studenti_per_voto(dizionario_studenti)
```

```
print("Conteggio degli studenti per ogni voto:")
for voto, conteggio in conteggio_studenti_per_voto.items():
    print(f"Voto: {voto}, Numero di studenti: {conteggio}")
```

Capitolo 5 : La Programmazione a Oggetti

Concetti fondamentali

La programmazione a oggetti è un paradigma di programmazione che si basa sull'utilizzo di oggetti e classi per organizzare e strutturare il codice. Questo approccio consente di raggruppare dati e funzionalità correlate all'interno di un'unica entità, l'oggetto, e di definire un modello astratto della realtà attraverso le classi.

Vediamo insieme i concetti fondamentali.

- **Classe:** una classe è un modello astratto che definisce la struttura di un oggetto. Essa definisce gli attributi e i metodi, ovvero le caratteristiche e le funzionalità dell'oggetto, rispettivamente.
- **Oggetto:** un oggetto è un'istanza di una classe, ovvero un esemplare concreto che contiene i valori degli attributi e che può eseguire i metodi definiti nella classe.
- **Incapsulamento:** l'incapsulamento è il meccanismo che consente di nascondere l'implementazione interna di un oggetto. Ciò significa che l'accesso agli attributi e ai metodi di un oggetto può essere controllato e limitato, rendendo il codice più sicuro e meno soggetto ad errori causati da manipolazioni indesiderate.
- **Ereditarietà:** l'ereditarietà consente di definire una nuova classe a partire da una classe esistente. La nuova classe, detta classe derivata o sottoclasse, eredita gli attributi e i metodi della classe base o superclasse e può definire nuovi attributi e metodi o ridefinire quelli ereditati.
- **Polimorfismo**: il polimorfismo consente di utilizzare oggetti di diverse classi in modo intercambiabile, purché implementino la stessa interfaccia o metodi con la stessa firma. Ciò significa che un metodo può essere applicato a oggetti di diverse classi senza dover conoscere la specifica classe di ogni oggetto.
- **Attributi di classe e di istanza**: gli attributi possono essere di due tipi: attributi di classe e attributi di istanza. Gli attributi di classe sono

definiti nella classe stessa e sono condivisi da tutte le istanze della classe. Gli attributi di istanza, invece, sono definiti per ogni singola istanza dell'oggetto e possono essere diversi per ciascuna istanza.

Vantaggi della programmazione orientata agli oggetti

La programmazione orientata agli oggetti (OOP - Object Oriented Programming) offre numerosi vantaggi rispetto ad altre metodologie di programmazione. Vediamoli insieme.

- **Modularità**: l'OOP promuove la modularità del codice, permettendo di suddividere un sistema complesso in parti più semplici e facilmente gestibili. In questo modo, è possibile dividere un progetto in più classi, ognuna delle quali svolge una specifica funzione. Questo consente di lavorare sul codice in modo più semplice e di isolare i problemi quando si verificano errori.
- **Riusabilità**: l'OOP rende il codice più riutilizzabile, poiché le classi e gli oggetti possono essere utilizzati in diversi contesti. Ad esempio, una classe che rappresenta un database potrebbe essere utilizzata in diversi progetti, senza dover essere riscritta ogni volta.
- **Estensibilità**: l'OOP consente di estendere facilmente il codice esistente. Ad esempio, una classe può essere derivata da un'altra classe (ereditarietà), permettendo di creare una nuova classe con funzionalità aggiuntive basate sulla classe originale. In questo modo, è possibile modificare il comportamento di una classe senza dover riscrivere tutto il codice.

In generale, la programmazione orientata agli oggetti offre una maggiore flessibilità, modularità, sicurezza e riutilizzabilità del codice rispetto ad altre metodologie di programmazione. Inoltre, consente di lavorare in modo più intuitivo e naturale, poiché si basa sui concetti di oggetti del mondo reale.

Definizione di classe e creazione di oggetti

Le classi ci permettono di definire le proprietà e i comportamenti degli oggetti in modo modulare e riutilizzabile. Inoltre, ci permettono di organizzare e strutturare il codice in modo logico e gerarchico.

Per esempio, immaginiamo di avere una classe *Automobile*. Questa classe potrebbe definire gli attributi dell'automobile, come il modello, la marca, il numero di porte, il tipo di motore, ecc.

Potrebbe anche definire i metodi dell'automobile, come accendere il motore, accelerare, frenare, cambiare marcia, ecc.

Una volta definita la classe, possiamo crearne delle istanze. Un'istanza è un esemplare di una classe, o in altre parole, un oggetto specifico che ha gli attributi e i metodi definiti nella classe.

Per creare un'istanza di una classe in *Python*, dobbiamo utilizzare il nome della classe seguita da parentesi tonde, all'interno delle quali specificare i valori degli attributi dell'oggetto.

Ad esempio, se la classe *Automobile* ha gli attributi *marca* e *modello*, possiamo creare un'istanza di quella classe specificando la marca e il modello dell'automobile:

```
mia_auto = Automobile(marca="Ford", modello="Mustang")
```

Una volta creata l'istanza, possiamo accedere agli attributi dell'oggetto utilizzando la sintassi *nome_istanza.nome_attributo*.

Se vogliamo accedere alla marca dell'automobile, possiamo utilizzare:

```
print(mia_auto.marca)  #output: Ford
```

Possiamo anche utilizzare i metodi dell'oggetto utilizzando la sintassi *nome_istanza.nome_metodo()*.

Se vogliamo accendere il motore dell'automobile, possiamo utilizzare:

```
mia_auto.accendere_motore()
```

Ecco l'esempio di classe *Automobile*, sopra citato, nel dettaglio:

```
class Automobile:
    def __init__(self, marca, modello, anno, colore, velocita_max):
        self.marca = marca
        self.modello = modello
        self.anno = anno
```

```
        self.colore = colore
        self.velocita_max = velocita_max
        self.velocita_attuale = 0

    def accelerare(self, velocita):
        if self.velocita_attuale + velocita > self.velocita_max:
            self.velocita_attuale = self.velocita_max
        else:
            self.velocita_attuale += velocita

    def frenare(self, velocita):
        if self.velocita_attuale - velocita < 0:
            self.velocita_attuale = 0
        else:
            self.velocita_attuale -= velocita

    def accendere_motore(self):
        print(f"{self.marca} {self.modello}: Motore acceso.")

    def spegnere_motore(self):
        print(f"{self.marca} {self.modello}: Motore spento.")
```

In questo esempio, abbiamo definito una classe chiamata *Automobile*. La classe ha cinque attributi: *marca*, *modello*, *anno*, *colore* e *velocita_max*, che rappresentano rispettivamente la marca dell'automobile, il modello, l'anno di produzione, il colore e la velocità massima che l'automobile può raggiungere. L'attributo *velocita_attuale* rappresenta la velocità attuale dell'automobile e viene inizializzato a 0.

La classe ha anche quattro metodi: *accelerare*, *frenare*, *accendere_motore* e *spegnere_motore*.

Il metodo *accelerare* aumenta la velocità dell'automobile di un certo valore (specificato come parametro), a condizione che la velocità risultante non superi la velocità massima dell'automobile.

Il metodo *frenare* diminuisce la velocità dell'automobile di un certo valore (specificato come parametro), a condizione che la velocità risultante non sia

inferiore a 0.

Il metodo *accendere_motore* stampa un messaggio che indica che il motore dell'automobile è stato acceso, mentre il metodo *spegnere_motore* stampa un messaggio che indica che il motore dell'automobile è stato spento.

Il metodo speciale *__init__* è un metodo costruttore che viene chiamato quando viene creata un'istanza della classe. Il costruttore inizializza gli attributi dell'oggetto con i valori specificati come parametri.

Possiamo creare un'istanza di questa classe utilizzando la sintassi seguente:

```
mia_auto = Automobile(  marca="Fiat", modello="500", \
                    anno=2020,  colore="rosso", velocita_max=200)
```

Questa istruzione crea un'istanza della classe *Automobile* con i valori specificati per gli attributi.

La parola chiave *self*

In *Python*, *self* è un termine convenzionale utilizzato come primo parametro nei metodi di una classe. Rappresenta l'istanza corrente della classe e viene utilizzato per accedere agli attributi e ai metodi dell'istanza stessa.

Quando chiami un metodo di un'istanza di classe, *Python* automaticamente passa l'istanza stessa come primo argomento al metodo.

Se cerchi di chiamare un metodo di un'istanza di una classe senza utilizzare il parametro *self* nel metodo, si verificherà un errore durante l'esecuzione del codice.

Senza il parametro *self*, il metodo non saprà a quale istanza fare riferimento e non sarà in grado di accedere agli attributi o ai metodi specifici dell'istanza. *Python* si aspetta che il primo parametro di un metodo di istanza sia *self* per consentire il corretto collegamento tra l'istanza e il metodo.

Ecco un esempio che mostra l'errore che si verifica quando si cerca di chiamare un metodo senza il parametro *self*:

```
class MiaClasse:
```

```
    def saluta():
        print("Ciao!")

istanza = MiaClasse()
istanza.saluta()

"""
Ottengo il seguente errore:
TypeError: MiaClasse.saluta() takes 0 positional arguments but 1 was given
"""
```

Attributi di classe e di istanza

Ogni oggetto appartiene a una classe e ogni classe definisce una serie di attributi e metodi che possono essere utilizzati dagli oggetti appartenenti a quella classe. Gli attributi possono essere di due tipi: di classe e di istanza.

Attributi di classe

Gli attributi di classe sono quelli definiti all'interno della classe stessa e sono condivisi da tutti gli oggetti appartenenti a quella classe.

Questo significa che tutti gli oggetti creati da quella classe condivideranno gli stessi valori degli attributi di classe.

Per definire un attributo di classe in *Python*, basta definirlo all'interno della classe, ma all'esterno di qualsiasi metodo; per inizializzarlo si deve utilizzare la sintassi *nome_classe.nome_attributo*.

Ad esempio, consideriamo la classe *Automobile* definita precedentemente e aggiungiamo un attributo di classe *num_ruote*:

```
class Automobile:
    # Attributi di classe
    num_ruote = 4
    car_count = 0

    def __init__(self, marca, modello, anno, colore, velocita_max):
        self.marca = marca
        self.modello = modello
```

```
        self.anno = anno
        self.colore = colore
        self.velocita_max = velocita_max
        self.velocita_attuale = 0
        Automobile.car_count += 1

    def accelerare(self, velocita):
        if self.velocita_attuale + velocita > self.velocita_max:
            self.velocita_attuale = self.velocita_max
        else:
            self.velocita_attuale += velocita

    def frenare(self, velocita):
        if self.velocita_attuale - velocita < 0:
            self.velocita_attuale = 0
        else:
            self.velocita_attuale -= velocita

    def accendere_motore(self):
        print(f"{self.marca} {self.modello}: Motore acceso.")

    def spegnere_motore(self):
        print(f"{self.marca} {self.modello}: Motore spento."):


# Creazione di istanze della classe Automobile
auto1 = Automobile( marca="Fiat", modello="500", \
                anno=2020,  colore="rosso", velocita_max=200)
auto2 = Automobile( marca="Fiat", modello="Tipo", \
                anno=2020,  colore="bianco", velocita_max=220)

# Accesso agli attributi di classe
print("Numero di automobili create:", Automobile.car_count)
```

Gli attributi di classe sono utili quando si desidera mantenere informazioni condivise tra tutte le istanze di una classe. Possono essere utilizzati per

tenere traccia di dati comuni, configurazioni, contatori o qualsiasi altra informazione che deve essere condivisa tra tutte le istanze.

Attributi di istanza

Gli attributi di istanza, invece, sono specifici per ogni oggetto creato a partire da quella classe.

Per definire un attributo di istanza, si utilizza la sintassi *self.nome_attributo* all'interno del metodo *__init__* della classe. La keyword *self* si riferisce all'oggetto appena creato e serve a distinguere l'attributo di istanza da quello di classe.

Ad esempio, possiamo definire la classe *Auto* che ha un attributo di istanza *marca*:

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

In questo caso, quando creiamo un oggetto di tipo *Auto*, dobbiamo specificare il valore dell'attributo *marca*:

```
auto1 = Auto('Fiat')
```

Conclusione

Per accedere agli attributi di classe e di istanza, si utilizza la stessa sintassi del punto, ma con la differenza che per gli attributi di classe si utilizza il nome della classe invece che quello dell'oggetto.

```
print(Veicolo.num_ruote)    # output: 4
print(auto1.marca)          # output: Fiat
```

Un vantaggio dell'utilizzo degli attributi di classe è che permettono di evitare la duplicazione di dati, in quanto tutti gli oggetti condividono lo stesso valore dell'attributo di classe.

Gli attributi di istanza, invece, sono utili per definire le caratteristiche specifiche di ogni oggetto e per tenere traccia del loro stato interno. Inoltre, gli attributi di istanza possono essere utilizzati per definire le proprietà

dell'oggetto e per fornire un'interfaccia con l'utente più intuitiva e comprensibile.

Metodi di classe e di istanza

I metodi di classe e i metodi di istanza sono due tipi di metodi che possono essere definiti all'interno di una classe. Entrambi sono utilizzati per definire il comportamento degli oggetti di una classe, ma ci sono alcune differenze fondamentali tra di essi.

Metodi di istanza

I metodi di istanza sono metodi che operano su un'istanza della classe. In altre parole, un metodo di istanza può accedere e manipolare gli attributi di istanza dell'oggetto. Sono definiti utilizzando la parola chiave *def*, come qualsiasi altra funzione in *Python*, ma vengono definiti all'interno della classe.

```
class Persona:
    def __init__(self, nome, eta):
        self.nome = nome
        self.eta = eta

    def saluta(self):
        print(f"Ciao, sono {self.nome} e ho {self.eta} anni")
```

In questo esempio, abbiamo definito la classe *Persona* con due attributi di istanza (*nome* ed *età*) e un metodo di istanza chiamato *saluta()*. Quest'ultimo accede agli attributi di istanza *nome* e *età* dell'oggetto e li utilizza per generare un messaggio di saluto.

Per utilizzare un metodo di istanza, è necessario prima creare un'istanza della classe e quindi chiamare il metodo sull'istanza.

```
p = Persona("Mario", 30)
p.saluta()  # Output: Ciao, sono Mario e ho 30 anni
```

Metodi di classe

I metodi di classe sono metodi che operano sulla classe stessa, anziché su una specifica istanza della classe. Essi sono definiti utilizzando il decoratore (vediamo di cosa si tratta nella sezione Che cosa sono i decoratori?) *@classmethod*, seguito dalla definizione del metodo.

```
class Persona:
    # Attributo di classe
    popolazione = 0

    def __init__(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome
        Persona.popolazione += 1

    def nome_completo(self):
        return f"{self.nome} {self.cognome}"

    @classmethod
    def resetta_popolazione(cls):
        cls.popolazione = 0

    @classmethod
    def da_stringa(cls, stringa):
        nome, cognome = stringa.split()
        return cls(nome, cognome)
```

```
# Creazione di istanze della classe Persona
persona1 = Persona("Mario", "Rossi")
persona2 = Persona("Luca", "Bianchi")

# Accesso all'attributo di classe
print("Popolazione:", Persona.popolazione)

# Utilizzo del metodo di classe per reimpostare la popolazione
Persona.resetta_popolazione()
```

```
print("Popolazione dopo il reset:", Persona.popolazione)

# Utilizzo del metodo di classe per creare un'istanza dalla stringa
persona3 = Persona.da_stringa("Giuseppe Verdi")
print("Nome completo della persona3:", persona3.nome_completo())
```

In questo esempio, abbiamo definito la classe *Persona*. All'interno della classe, abbiamo l'attributo di classe *popolazione*, che tiene traccia del numero totale di persone create.

Il metodo *__init__* viene chiamato quando viene creata un'istanza della classe *Persona*: prende due parametri, *nome* e *cognome*, e inizializza gli attributi di istanza *self.nome* e *self.cognome* con i valori forniti. Inoltre, incrementa l'attributo di classe *popolazione* di 1 per tener conto della nuova persona creata.

Il metodo *nome_completo* restituisce il nome completo della persona concatenando *self.nome* e *self.cognome*.

Abbiamo anche definito due metodi di classe utilizzando l'annotazione *@classmethod*. Il metodo *resetta_popolazione* reimposta l'attributo di classe *popolazione* a 0.

Utilizziamo il decoratore *@classmethod* per indicare che il metodo è un metodo di classe e utilizziamo l'argomento *cls* per fare riferimento alla classe stessa.

Il metodo *da_stringa* è un altro metodo di classe che prende una stringa nel formato "nome cognome" e crea un'istanza di *Persona* con i valori estratti dalla stringa.

Esso restituisce l'istanza appena creata utilizzando *cls(nome, cognome)*.

Nell'esempio di utilizzo, creiamo due istanze della classe *Persona* chiamate *persona1* e *persona2* passando i valori appropriati per *nome* e *cognome*. Accediamo all'attributo di classe *popolazione* utilizzando il nome della classe *Persona* e lo stampiamo per visualizzare il numero totale di persone create.

Successivamente, utilizziamo il metodo di classe *resetta_popolazione* per reimpostare l'attributo di classe *popolazione* a 0. Questa chiamata al metodo

resetta_popolazione ci consente di azzerare il conteggio della popolazione, fornendo un modo semplice per ripartire da zero.

Infatti, dopo aver chiamato il metodo *resetta_popolazione*, possiamo accedere nuovamente all'attributo di classe *popolazione* utilizzando il nome della classe *Persona* e constatare che il valore è stato effettivamente reimpostato a 0.

Questo è particolarmente utile in situazioni in cui desideriamo ripulire lo stato della classe o azzerare un contatore o una variabile di riferimento comune a tutte le istanze.

L'utilizzo di metodi di classe come *resetta_popolazione* ci offre una soluzione pulita e strutturata per effettuare operazioni specifiche sulla classe nel suo complesso, consentendo una maggiore flessibilità e facilitando la manutenzione del codice.

Ricorda che i metodi di classe possono essere definiti utilizzando il decoratore *@classmethod* sopra la loro definizione e prendono come primo parametro *cls*, che fa riferimento alla classe stessa anziché all'istanza.

L'incapsulamento

L'incapsulamento è un concetto di programmazione orientata agli oggetti che consente di nascondere i dettagli interni di una classe e proteggere gli attributi e i metodi dalla manipolazione indesiderata da parte di altre parti del programma.

L'incapsulamento viene raggiunto utilizzando la convenzione di denominazione degli attributi e dei metodi con un prefisso di accesso (_ o __).

Vediamo di seguito la differenza fra incapsulamento pubblico, protetto e privato.

Incapsulamento pubblico

L'incapsulamento pubblico è il livello predefinito di accesso agli attributi e ai metodi di una classe. Gli attributi e i metodi pubblici possono essere acceduti e utilizzati sia all'interno che dall'esterno della classe.

```
class MiaClasse:
```

```
    def __init__(self):
        self.attributo_pubblico = "Sono un attributo pubblico"

    def metodo_pubblico(self):
        print("Sono un metodo pubblico")
```

```
istanza = MiaClasse()
print(istanza.attributo_pubblico)  # Accesso consentito
istanza.metodo_pubblico()  # Accesso consentito
```

Nell'esempio sopra, abbiamo definito la classe *MiaClasse* con un attributo pubblico *attributo_pubblico* e un metodo pubblico *metodo_pubblico*. Gli attributi pubblici possono essere accessibili e modificati dall'esterno della classe, così come i metodi pubblici possono essere chiamati dall'esterno.

L'incapsulamento pubblico in *Python* è il livello di default e non richiede l'utilizzo di prefissi particolari o convenzioni. Gli attributi e i metodi dichiarati senza prefissi di accesso particolari vengono considerati pubblici.

Tuttavia, è importante notare che l'incapsulamento pubblico offre un accesso diretto e aperto agli attributi e ai metodi. Ciò significa che non ci sono restrizioni o controlli che impediscono la manipolazione indesiderata dei dati. È responsabilità dello sviluppatore utilizzare l'incapsulamento pubblico in modo appropriato e garantire che gli attributi e i metodi pubblici siano utilizzati correttamente e coerentemente nel codice.

Incapsulamento protetto

L'incapsulamento protetto si ottiene utilizzando un singolo underscore _ come prefisso per gli attributi e i metodi che si desidera trattare come protetti. Questa convenzione indica agli sviluppatori che gli attributi e i metodi sono destinati ad essere utilizzati internamente alla classe o dalle sottoclassi, ma non dovrebbero essere accessibili direttamente dall'esterno della classe.

```
class MiaClasse:
    def __init__(self):
        self._attributo_protetto = "Sono un attributo protetto"
```

```
    def _metodo_protetto(self):
        print("Sono un metodo protetto")
```

```
istanza = MiaClasse()
print(istanza._attributo_protetto)  # Accesso consentito
istanza._metodo_protetto()  # Accesso consentito
```

Nell'esempio sopra, abbiamo definito la solita classe *MiaClasse* con un attributo protetto *_attributo_protetto* e un metodo protetto *_metodo_protetto*. Utilizzando il singolo underscore (_) come prefisso, indichiamo che questi elementi sono protetti e non dovrebbero essere accessibili direttamente dall'esterno della classe.

Tuttavia, è importante notare che l'incapsulamento protetto si basa principalmente su una convenzione e sulla responsabilità del programmatore di rispettare l'accesso corretto. Non ci sono restrizioni o controlli rigorosi che impediscono l'accesso diretto agli attributi o ai metodi protetti. Pertanto, è possibile accedere e modificare gli attributi protetti dall'esterno della classe, sebbene sia considerato una pratica non consigliata.

L'incapsulamento protetto fornisce un livello di indicazione e suggerisce agli sviluppatori che certi elementi sono destinati ad essere utilizzati internamente alla classe o dalle sottoclassi. È importante seguire questa convenzione per mantenere la coerenza e la leggibilità del codice e per evitare manipolazioni indesiderate degli attributi o dei metodi protetti.

Incapsulamento privato

L'incapsulamento privato si ottiene utilizzando un doppio underscore __ come prefisso per gli attributi e i metodi che si desidera trattare come privati. Questa convenzione indica che gli attributi e i metodi sono destinati ad essere utilizzati solo internamente alla classe e non dovrebbero essere accessibili dall'esterno o dalle sottoclassi.

```
class MiaClasse:
    def __init__(self):
        self.__attributo_privato = "Sono un attributo privato"
```

```
    def __metodo_privato(self):
        print("Sono un metodo privato")

    def usa_attributi_e_metodi_privati(self):
        print(self.__attributo_privato)
        self.__metodo_privato()
```

```
istanza = MiaClasse()
print(istanza.__attributo_privato)  # Genera un AttributeError
istanza.__metodo_privato()  # Genera un AttributeError
istanza.usa_attributi_e_metodi_privati()
# Accesso consentito all'interno della classe
```

Nell'esempio sopra, abbiamo definito la solita classe *MiaClasse* la quale però adesso presenta un attributo privato *__attributo_privato* e un metodo privato *__metodo_privato*. Utilizzando il doppio underscore come prefisso, indichiamo che questi elementi sono privati e non dovrebbero essere accessibili dall'esterno della classe o dalle sottoclassi.

Quando si cerca di accedere agli attributi o ai metodi privati dall'esterno della classe, viene generato un errore di tipo *AttributeError*. Tuttavia, è possibile accedere e utilizzare gli attributi e i metodi privati all'interno della stessa classe.

Sebbene sia possibile accedere agli attributi e ai metodi privati utilizzando il nome mangled, ad esempio *_Classe__attributo_privato*, questa pratica non è consigliata e viola l'incapsulamento e l'intento del codice.

```
istanza = MiaClasse()
# Accesso all'attributo privato utilizzando il name mangling
print(istanza._MiaClasse__attributo_privato)
# Accesso al metodo privato utilizzando il name mangling
istanza._MiaClasse__metodo_privato()
```

L'incapsulamento privato fornisce una protezione più forte rispetto all'incapsulamento protetto, limitando l'accesso agli attributi e ai metodi solo all'interno della classe stessa. È importante utilizzarlo quando si

desidera nascondere completamente l'implementazione interna di una classe e proteggere i dati sensibili da accessi indesiderati.

Ereditarietà e polimorfismo

Ereditarietà

L'ereditarietà è un concetto fondamentale della programmazione ad oggetti che consente di definire una nuova classe che è una versione specializzata di una classe esistente. Essa viene implementata attraverso la creazione di una classe "figlia" che eredita gli attributi e i metodi da una classe "genitore" esistente.

Definizione di una classe figlia

Per definire una classe figlia, si utilizza la sintassi seguente:

```
class ClasseFiglia(ClasseGenitore):
    # definizione degli attributi e dei metodi della classe figlia
```

In questa sintassi, *ClasseFiglia* è il nome della classe figlia, mentre *ClasseGenitore* è il nome della classe genitore da cui la classe figlia eredita gli attributi e i metodi.

```
# Classe genitore
class Animale:
    def __init__(self, nome, eta):
        self.nome = nome
        self.età = età

    def fare_rumore(self):
        print("L'animale sta facendo rumore.")

# Classe figlia
class Cane(Animale):
    def __init__(self, nome, eta, razza):
        super().__init__(nome, eta)
        self.razza = razza
```

```
    def fare_rumore(self):
        print("Il cane sta abbaiando.")
```

In questo esempio, abbiamo definito due classi: *Animale* e *Cane*. La classe *Animale* ha due attributi: *nome* ed *età*, e un metodo chiamato *fare_rumore* che stampa un messaggio che indica che l'animale sta facendo rumore.

La classe *Cane* è una classe figlia di *Animale* e quindi eredita gli attributi e i metodi di quest'ultima. Tuttavia, *Cane* ha anche un nuovo attributo chiamato *razza*. Il metodo *fare_rumore* della classe *Cane* è stato sovrascritto per stampare un messaggio che indica che il cane sta abbaiando.

```
mio_cane = Cane(nome="Fido", eta=3, razza="Labrador")

print(mio_cane.nome)   #output: Fido
```

Il metodo super()

Nella classe figlia *Cane*, abbiamo definito un metodo costruttore *__init__* che ha tre parametri: *nome*, *età* e *razza*.

Tuttavia, abbiamo anche chiamato il metodo *__init__* della classe genitore utilizzando il metodo *super()*. Questo è importante perché ci assicura che gli attributi della classe genitore vengano inizializzati correttamente, anche se non li abbiamo esplicitamente definiti nel metodo costruttore della classe figlia.

Dunque, *super()* consente di accedere e invocare metodi definiti nella classe genitore da una classe figlia.

Nell'esempio sopra, la chiamata *super().__init__(nome, età)* chiama il metodo *__init__* della classe genitore con i parametri *nome* ed *età*, inizializzando così gli attributi "*nome*" ed *età* della classe figlia.

Ereditarietà multipla

L'ereditarietà multipla consente a una classe di ereditare le caratteristiche da più classi padre.

Quando una classe eredita da più classi padre, può accedere e utilizzare gli attributi e i metodi definiti in ciascuna delle classi padre. Ciò offre

maggiore flessibilità nella progettazione del software, consentendo di creare gerarchie di classi più complesse e di riusare il codice in modo efficiente.

Per implementare l'ereditarietà multipla in *Python*, una classe figlia può essere definita elencando più classi padre separate da virgole all'interno delle parentesi tonde dopo il nome della classe figlia.

```
class ClasseFiglia(ClassePadre1, ClassePadre2, ...):
    # Definizioni della classe figlia
```

Vantaggi dell'ereditarietà

L'utilizzo dell'ereditarietà offre numerosi vantaggi, tra cui:

- **Riduzione del codice ripetitivo**: ereditando gli attributi e i metodi da una classe genitore, possiamo evitare di doverli definire nuovamente nella classe figlia, riducendo così il codice ripetitivo.
- **Maggiore flessibilità**: la classe figlia può estendere e modificare il comportamento della classe genitore. Ad esempio, nella classe *Cane* dell'esempio sopra, abbiamo sovrascritto il metodo *fare_rumore* della classe *Animale* per fornire un comportamento specifico per il cane.
- **Migliore organizzazione del codice**: utilizzando l'ereditarietà, possiamo organizzare il nostro codice in una gerarchia di classi, in cui le classi figlie sono specializzazioni delle classi genitore.

Polimorfismo

Il polimorfismo è un concetto di programmazione orientata agli oggetti che si riferisce alla capacità di un oggetto di assumere più forme o comportamenti.

In generale, il polimorfismo consente di scrivere codice che può essere applicato a oggetti di diverse classi in modo trasparente, senza preoccuparsi del tipo specifico dell'oggetto.

Il polimorfismo viene realizzato attraverso l'uso di metodi polimorfici i quali sono metodi definiti all'interno di classi diverse, ma che hanno la stessa firma, ovvero lo stesso nome e gli stessi parametri. Questi metodi

possono essere implementati in modo diverso nelle varie classi, consentendo agli oggetti di comportarsi in modo specifico a seconda del tipo di oggetto.

Un esempio comune di polimorfismo è dato dal concetto di "metodo di interfaccia". Una classe base può definire un metodo di interfaccia senza fornire un'implementazione concreta, mentre le classi derivate possono fornire implementazioni specifiche del metodo. Quando si utilizza un oggetto di una classe derivata, è possibile chiamare il metodo di interfaccia senza preoccuparsi dell'implementazione specifica sottostante.

Ecco un esempio di come il polimorfismo può essere utilizzato in *Python*:

```
class Forma:
    def calcola_area(self):
        pass

class Rettangolo(Forma):
    def __init__(self, base, altezza):
        self.base = base
        self.altezza = altezza

    def calcola_area(self):
        return self.base * self.altezza

class Cerchio(Forma):
    def __init__(self, raggio):
        self.raggio = raggio

    def calcola_area(self):
        return 3.14159 * self.raggio**2

def stampa_area(forma):
    print("L'area è:", forma.calcola_area())

rettangolo = Rettangolo(5, 3)
cerchio = Cerchio(4)
```

```
stampa_area(rettangolo)  # Stampa "L'area è: 15"
stampa_area(cerchio)    # Stampa "L'area è: 50.26544"
```

Nell'esempio sopra, abbiamo definito una classe base chiamata *Forma* con un metodo polimorfico chiamato *calcola_area()*. Le classi derivate *Rettangolo* e *Cerchio* forniscono le rispettive implementazioni del metodo *calcola_area()*. La funzione *stampa_area()* prende un oggetto di tipo *Forma* come argomento e chiama il metodo *calcola_area()* su di esso senza preoccuparsi del tipo specifico dell'oggetto.

Quando chiamiamo *stampa_area(rettangolo),* il metodo *calcola_area()* della classe *Rettangolo* viene eseguito, restituendo l'area del rettangolo.

Allo stesso modo, quando chiamiamo *stampa_area(cerchio),* il metodo *calcola_area()* della classe *Cerchio* viene eseguito, restituendo l'area del cerchio. In entrambi i casi, nonostante l'utilizzo di oggetti di classi diverse, il polimorfismo ci consente di chiamare il metodo *calcola_area()* in modo coerente e ottenere il risultato desiderato.

Vantaggi del polimorfismo

Il polimorfismo offre numerosi vantaggi.

Alcuni di questi sono:

- **Flessibilità**: il polimorfismo consente di scrivere codice più flessibile e riutilizzabile, in quanto gli oggetti possono essere trattati allo stesso modo indipendentemente dalla loro classe effettiva.
- **Manutenibilità**: il polimorfismo rende il codice più facile da mantenere e aggiornare, poiché il comportamento degli oggetti può essere modificato semplicemente modificando le sottoclassi (classi derivate).
- **Estensibilità**: il polimorfismo consente di estendere facilmente la funzionalità del codice aggiungendo nuove sottoclassi che implementano i metodi richiesti. Ciò significa che possiamo aggiungere nuovi tipi di oggetti senza dover modificare il codice esistente.

Ad esempio, se volessimo aggiungere una nuova classe *Triangolo* alla gerarchia di classi precedente, potremmo farlo come segue:

```
class Triangolo(Forma):
```

```
    def __init__(self, base, altezza):
        self.base = base
        self.altezza = altezza

    def calcola_area(self):
        return (self.base * self.altezza) / 2
```

In questo esempio, abbiamo definito una nuova classe *Triangolo* che eredita da *Forma*. La classe ha un costruttore che prende la base e l'altezza del triangolo e un metodo *calcola_area()* che calcola l'area del triangolo. Poiché *Triangolo* eredita da *Forma*, ha anche il metodo *calcola_area(),* che ora implementa il calcolo dell'area del triangolo.

Ora possiamo creare un oggetto *Triangolo* e chiamare il metodo *stampa_area* su di esso, ottenendo il risultato appropriato:

```
triangolo = Triangolo(4, 8)
stampa_area(triangolo)    # Stampa "L'area è: 16"
```

Come puoi vedere, il polimorfismo ci ha permesso di estendere facilmente la gerarchia di classi esistente aggiungendo una nuova classe senza dover modificare il codice esistente.

I metodi getter e setter

In molti linguaggi di programmazione, incluso *Python*, è comune utilizzare metodi getter e setter per accedere e modificare gli attributi di un oggetto. Questi metodi forniscono un'interfaccia controllata per leggere e scrivere i valori degli attributi, consentendo un migliore controllo sull'accesso ai dati.

Questi metodi vengono utilizzati soprattutto quando si desidera imporre controlli aggiuntivi o eseguire operazioni speciali quando si accede o si modifica un attributo. Ad esempio, potresti voler verificare la validità di un valore prima di impostarlo o eseguire azioni specifiche ogni volta che un attributo viene letto o modificato.

```
class Persona:
    def __init__(self, nome, età):
        self._nome = nome
        self._età = età
```

```
    def get_nome(self):
        return self._nome

    def set_nome(self, nuovo_nome):
        self._nome = nuovo_nome

    def get_età(self):
        return self._età

    def set_età(self, nuova_età):
        if nuova_età > 0:
            self._età = nuova_età

# Creazione di un'istanza della classe Persona
persona = Persona("Mario", 30)

# Utilizzo dei metodi getter e setter
print(persona.get_nome())  # Output: "Mario"
persona.set_nome("Luigi")
print(persona.get_nome())  # Output: "Luigi"

print(persona.get_età())  # Output: 30
persona.set_età(35)
print(persona.get_età())  # Output: 35
persona.set_età(-5)  # La chiamata a set_età() non modificherà \
                     # l'età perché il valore è negativo
print(persona.get_età())  # Output: 35
```

I metodi speciali

I metodi speciali, noti anche come metodi magici, sono una parte importante della programmazione in *Python*. Sono dei metodi predefiniti che ci permettono di definire il comportamento di un oggetto in determinate

situazioni. In questa sezione esploreremo i metodi speciali più comuni e come usarli nelle nostre classi.

Innanzitutto, tutti i metodi speciali sono identificati dal doppio underscore ("__") sia all'inizio che alla fine del loro nome.

Ad esempio, il metodo speciale per la definizione di un costruttore è *__init__*, mentre il metodo speciale per la rappresentazione testuale di un oggetto è *str*.

Esempi di metodi speciali comuni (str, repr, etc.)

Ecco alcuni dei metodi speciali più comuni e come usarli:

- **__new__**: questo metodo viene chiamato per creare una nuova istanza di una classe. È responsabile per l'allocazione della memoria e la creazione dell'oggetto. Esso viene chiamato prima del metodo *__init__*.

  Viene chiamato esplicitamente dalla creazione dell'istanza della classe, ad esempio tramite l'operatore ().

  La ragione principale per cui il metodo *__new__* non deve essere chiamato direttamente è che la sua chiamata è gestita internamente dal linguaggio. La chiamata diretta a *__new__* potrebbe causare comportamenti imprevisti o errori nel processo di creazione delle istanze.

  In generale, il metodo *__new__* viene implementato solo in casi specifici in cui è necessario un controllo avanzato sulla creazione delle istanze, ad esempio quando si desidera sottoporre l'istanza a una logica personalizzata.

  La firma del metodo *__new__* è la seguente:

```
def __new__(cls, *args, **kwargs):
    # Logica di creazione dell'oggetto
    return super().__new__(cls)
```

  Il primo argomento, *cls*, rappresenta la classe stessa. Gli eventuali argomenti successivi (*args* e *kwargs*) sono gli argomenti passati al costruttore della classe.

Il metodo *__new__* restituisce un'istanza dell'oggetto creato. Di solito, viene chiamato il metodo *__new__* della superclasse utilizzando *super().__new__(cls)* per garantire che l'oggetto venga creato correttamente.

- **__init__**: questo metodo viene chiamato quando viene creato un nuovo oggetto della classe ed è responsabile dell'inizializzazione dei suoi attributi.
- **__call__**: quando una classe definisce questo metodo, le istanze di quella classe possono essere chiamate come se fossero funzioni.

Questo aspetto può essere utile in vari scenari, come nel caso di oggetti che devono mantenere uno stato interno o nel caso di oggetti che implementano funzionalità specifiche che possono essere richiamate attraverso la chiamata come funzioni.

```
class Contatore:
    def __init__(self):
        self.count = 0

    def __call__(self):
        self.count += 1
        print(f"Il contatore è stato chiamato {self.count} volte.")

# Creazione di un'istanza della classe Contatore
counter = Contatore()

# Chiamata dell'istanza come se fosse una funzione
counter()  # Output: Il contatore è stato chiamato 1 volta.
counter()  # Output: Il contatore è stato chiamato 2 volte.
counter()  # Output: Il contatore è stato chiamato 3 volte.
```

- **__str__**: questo metodo consente di definire la rappresentazione in forma di stringa dell'oggetto.
- **__repr__**: questo metodo consente di definire la rappresentazione dell'oggetto utilizzata per la sua rappresentazione formale.
- **__eq__**: questo metodo consente di definire l'uguaglianza tra oggetti.

- **__ne__**: questo metodo consente di definire la disuguaglianza tra oggetti.
- **__lt__**: questo metodo consente di definire l'operatore "minore di" tra oggetti.
- **__gt__**: questo metodo consente di definire l'operatore "maggiore di" tra oggetti.
- **__le__**: questo metodo consente di definire l'operatore "minore o uguale a" tra oggetti.
- **__ge__**: questo metodo consente di definire l'operatore "maggiore o uguale a" tra oggetti.
- **__len__**: questo metodo consente di definire la lunghezza dell'oggetto.
- **__getitem__**: questo metodo consente di accedere agli elementi dell'oggetto tramite la notazione delle parentesi quadre.
- **__setitem__**: questo metodo consente di impostare i valori degli elementi dell'oggetto tramite la notazione delle parentesi quadre.

L'utilizzo dei metodi speciali è fondamentale poiché permette di personalizzare il comportamento degli oggetti in modo molto preciso.

Ad esempio, se volessimo definire una classe che rappresenta una matrice, possiamo definire i metodi speciali __add__, __sub__, __mul__ per definire come avviene l'addizione, la sottrazione e la moltiplicazione tra matrici. In questo modo, possiamo creare oggetti della nostra classe e utilizzare gli operatori matematici come se fossero stati definiti nativamente in *Python*.

Implementazione di metodi speciali personalizzati

I metodi speciali possono essere personalizzati per le classi definite dall'utente per specificare il comportamento di determinate operazioni su oggetti. In questa sezione vedremo come implementarne alcuni dei piu utilizzati.

- **__str__(self)**: è uno dei metodi speciali più utilizzati. Esso ci consente di definire come un oggetto debba essere rappresentato come stringa. Questo metodo viene richiamato automaticamente quando viene chiamata la funzione *str()* sull'oggetto, oppure quando l'oggetto viene stampato con la funzione *print()*.

```python
class Persona:
    def __init__(self, nome, cognome, eta):
        self.nome = nome
        self.cognome = cognome
        self.eta = eta

    def __str__(self):
        return f"{self.nome} {self.cognome}, {self.eta} anni"

p = Persona("Mario", "Rossi", 30)
print(p)  # output: "Mario Rossi, 30 anni"
```

In questo esempio abbiamo definito una classe *Persona* con gli attributi *nome*, *cognome* ed *età*. Abbiamo anche definito il metodo speciale *__str__* che restituisce una stringa formattata con il nome, cognome e età della persona. Infine abbiamo creato un'istanza della classe *Persona* e l'abbiamo stampata utilizzando la funzione *print()*. Grazie al metodo *__str__*, l'oggetto viene stampato come una stringa nel formato desiderato.

Notare che il metodo *__str__* deve sempre restituire una stringa. Se non viene definito, verrà utilizzato il valore di default, che restituisce l'indirizzo di memoria dell'oggetto come stringa. Definire il metodo *__str__* ci permette quindi di personalizzare la rappresentazione dell'oggetto in modo da renderlo più comprensibile e leggibile.

- **__repr__(self)**: ci consente di definire una rappresentazione "ufficiale" dell'oggetto, ovvero una stringa che può essere utilizzata per ricostruire l'oggetto stesso. Questo metodo viene chiamato quando viene invocata la funzione *repr()* sull'oggetto o quando l'oggetto viene visualizzato direttamente nella console di *Python*.

  La sua utilità principale è fornire una rappresentazione testuale dell'oggetto che sia utile per il debugging e per comprendere meglio la sua struttura. La stringa restituita dal metodo *__repr__* dovrebbe essere univoca e contenere informazioni significative sull'oggetto, in modo che sia possibile capire come è stato creato o quali sono i suoi attributi principali.

```
class Persona:
    def __init__(self, nome, cognome, eta):
        self.nome = nome
        self.cognome = cognome
        self.eta = eta

    def __str__(self):
        return f"{self.nome} {self.cognome}, {self.eta} anni"

    def __repr__(self):
        return f"Persona('{self.nome}', '{self.cognome}', \
                    {self.eta})"

p = Persona("Mario", "Rossi", 30)
print(repr(p))  # stampa "Persona('Mario', 'Rossi', 30)"

p1 = repr(p)
print(p1)     # stampa "Persona('Mario', 'Rossi', 30)"
```

In questo esempio abbiamo definito la stessa classe *Persona* dell'esempio precedente, ma abbiamo aggiunto il metodo speciale *__repr__*. Questo metodo restituisce una stringa che rappresenta l'oggetto in modo tale da poter essere utilizzata per crearne una copia identica. Definire il metodo *__repr__* ci permette quindi di avere una rappresentazione "ufficiale" dell'oggetto che possiamo utilizzare per creare copie identiche, debuggare il codice e migliorare la comprensione del funzionamento dell'oggetto.

- **__eq__(self, other)**: Il metodo *__eq__* consente di definire il comportamento dell'operatore di confronto "uguale a" (==) per le istanze della nostra classe. Questo metodo viene chiamato quando si utilizza l'operatore di confronto == per confrontare due oggetti della nostra classe.

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

```
        self.y = y

    def __eq__(self, other):
        if isinstance(other, Punto):
            return self.x == other.x and self.y == other.y
        return False

p1 = Punto(1, 2)
p2 = Punto(1, 2)
p3 = Punto(3, 4)

print(p1 == p2)  # stampa "True"
print(p1 == p3)  # stampa "False"
```

In questo esempio abbiamo definito la classe *Punto*, che rappresenta un punto nel piano cartesiano. Abbiamo poi definito il metodo speciale *__eq__*, che confronta l'oggetto corrente con un altro oggetto passato come parametro (*other*). Se l'oggetto *other* è un'istanza della classe *Punto* e le sue coordinate *x* e *y* sono uguali a quelle dell'oggetto corrente, il metodo restituisce *True*. In caso contrario, restituisce *False*. Infine, abbiamo creato tre istanze della classe *Punto* e abbiamo utilizzato l'operatore di confronto == per confrontare i primi due oggetti (che hanno le stesse coordinate) e il primo e il terzo oggetto (che hanno coordinate diverse).

Notare che il metodo *__eq__* deve sempre restituire un valore booleano (*True* o *False*) e deve gestire correttamente il caso in cui l'oggetto passato come parametro non sia un'istanza della nostra classe (in questo caso, restituisce sempre *False*). Inoltre, se non viene definito il metodo *__eq__*, l'operatore di confronto == utilizzerà l'operatore di confronto *is*, che confronta gli oggetti in base all'indirizzo di memoria, ovvero verificherà che siano lo stesso oggetto. Definire il metodo *__eq__* ci permette quindi di personalizzare il comportamento dell'operatore di confronto == per le istanze della nostra classe, rendendo il nostro codice più chiaro e flessibile.

Esercizi

Esercizio 1: negozio di libri

Immaginiamo di dover creare un programma per gestire un negozio di libri. Utilizzeremo diverse strutture dati per gestire l'inventario, le informazioni dei libri e le operazioni di vendita. Sfrutteremo anche la programmazione a oggetti per creare classi e oggetti che rappresentano i libri.

```
class Libro:
    """
    Classe per rappresentare un libro.
    """

    def __init__(self, titolo, autore, anno_pubblicazione):
        """
        Inizializza un nuovo oggetto Libro con il titolo, l'autore e l'anno di pubblicazione specificati.

        Args:
            titolo (str): Il titolo del libro.
            autore (str): L'autore del libro.
            anno_pubblicazione (int): L'anno di pubblicazione del libro.
        """
        self.titolo = titolo
        self.autore = autore
        self.anno_pubblicazione = anno_pubblicazione

    def __str__(self):
        """
        Restituisce una rappresentazione in formato stringa del libro.

        Returns:
            str: La stringa che rappresenta il libro nel formato "Titolo di Autore (Anno di pubblicazione)".
        """
        return f"{self.titolo} di {self.autore} ({self.anno_pubblicazione})"
```

```
inventario = []
```

```
dizionario_libri = {}
insieme_autori = set()
coda_vendite = []

# Aggiungiamo alcuni libri all'inventario
libro1 = Libro("Il signore degli anelli", "J.R.R. Tolkien", 1954)
libro2 = Libro("Harry Potter e la pietra filosofale", \
            "J.K. Rowling", 1997)
libro3 = Libro("1984", "George Orwell", 1949)

inventario.append(libro1)
inventario.append(libro2)
inventario.append(libro3)

# Aggiungiamo i libri al dizionario dei libri utilizzando
# il titolo come chiave
dizionario_libri[libro1.titolo] = libro1
dizionario_libri[libro2.titolo] = libro2
dizionario_libri[libro3.titolo] = libro3

# Aggiungiamo gli autori all'insieme degli autori
insieme_autori.add(libro1.autore)
insieme_autori.add(libro2.autore)
insieme_autori.add(libro3.autore)

# Aggiungiamo una vendita alla coda delle vendite
coda_vendite.append(libro1)
coda_vendite.append(libro2)

# Stampiamo l'inventario dei libri
print("Inventario:")
for libro in inventario:
    print(libro)

# Stampiamo i libri nel dizionario dei libri
```

```
print("Dizionario dei libri:")
for libro in dizionario_libri.values():
    print(libro)

# Stampiamo gli autori
print("Autori:")
for autore in insieme_autori:
    print(autore)

# Effettuiamo la vendita di un libro dalla coda delle vendite
libro_venduto = coda_vendite.pop(0)
print("Libro venduto:", libro_venduto)

# Aggiorniamo l'inventario dopo la vendita
inventario.remove(libro_venduto)

# Verifichiamo l'aggiornamento dell'inventario
print("Nuovo inventario:")
for libro in inventario:
    print(libro)
```

Esercizio 2: azienda di trasporto

Immaginiamo di dover creare un programma per gestire un'azienda di trasporto. L'azienda gestisce una flotta di veicoli di diversi tipi, tra cui autobus, camion e auto. Dovremo tenere traccia dei veicoli, gestire le loro caratteristiche e le operazioni di manutenzione. Utilizzeremo le strutture dati e la programmazione a oggetti per organizzare e gestire efficacemente i dati.

```
class Veicolo:
    """
    Classe per rappresentare un veicolo.
    """

    def __init__(self, targa, marca, modello, anno, tipo):
```

```
        """
        Inizializza un nuovo oggetto Veicolo con la targa, la marca, il modello, l'anno di produzione e il tipo specificati.

        Args:
            targa (str): La targa del veicolo.
            marca (str): La marca del veicolo.
            modello (str): Il modello del veicolo.
            anno (int): L'anno di produzione del veicolo.
            tipo (str): Il tipo di veicolo.
        """
        self.targa = targa
        self.marca = marca
        self.modello = modello
        self.anno = anno
        self.tipo = tipo

    def informazioni(self):
        """
        Restituisce una stringa con le informazioni del veicolo.

        Returns:
            str: Le informazioni del veicolo nel formato "Marca Modello (Anno), Tipo: TipoVeicolo".
        """
        return f"{self.marca} {self.modello} ({self.anno}), Tipo: {self.tipo}"

class Flotta:
    """
    Classe per rappresentare una flotta di veicoli.
    """

    def __init__(self):
        """
        Inizializza un nuovo oggetto Flotta con una lista vuota di veicoli.
```

```
        """
        self.veicoli = []

    def aggiungi_veicolo(self, veicolo):
        """
        Aggiunge un veicolo alla flotta.

        Args:
            veicolo (Veicolo): Il veicolo da aggiungere alla flotta.
        """
        self.veicoli.append(veicolo)

    def rimuovi_veicolo(self, targa):
        """
        Rimuove un veicolo dalla flotta in base alla targa.

        Args:
            targa (str): La targa del veicolo da rimuovere.
        """
        for veicolo in self.veicoli:
            if veicolo.targa == targa:
                self.veicoli.remove(veicolo)
                break

    def elenco_veicoli(self):
        """
        Stampa l'elenco dei veicoli nella flotta, mostrando le informazioni di ciascun veicolo.
        """
        for veicolo in self.veicoli:
            print(veicolo.informazioni())
```

```
# Creiamo la flotta di veicoli
flotta = Flotta()
```

```
# Aggiungiamo alcuni veicoli alla flotta
autobus1 = Veicolo("AB123CD", "Mercedes", "Tourismo", 2015, "Autobus")
camion1 = Veicolo("XY789ZA", "Scania", "R500", 2018, "Camion")
auto1 = Veicolo("EF456GH", "Toyota", "Corolla", 2020, "Auto")

flotta.aggiungi_veicolo(autobus1)
flotta.aggiungi_veicolo(camion1)
flotta.aggiungi_veicolo(auto1)

# Stampiamo l'elenco dei veicoli nella flotta
print("Elenco veicoli:")
flotta.elenco_veicoli()

# Rimuoviamo un veicolo dalla flotta
flotta.rimuovi_veicolo("XY789ZA")

# Stampiamo nuovamente l'elenco dei veicoli nella flotta
print("Nuovo elenco veicoli:")
flotta.elenco_veicoli()
```

Esercizio 3: negozio

L'obiettivo di questo esercizio è creare un sistema semplificato per gestire un negozio e le interazioni con i clienti. Il negozio è rappresentato dalla classe Negozio, che tiene traccia dell'inventario dei prodotti disponibili. I clienti sono rappresentati dalla classe Cliente, che ha un carrello per aggiungere e rimuovere prodotti.

L'esercizio prevede l'implementazione di diverse funzionalità. Per il negozio, è necessario aggiungere prodotti all'inventario, rimuoverli, e calcolare il totale delle vendite effettuate. Per i clienti, è necessario aggiungere prodotti al carrello, rimuoverli, e calcolare il totale dell'acquisto.

Il codice fornisce un esempio di come utilizzare classi, oggetti, ereditarietà, metodi getter e setter, e strutture dati (come dizionari) in *Python* per creare un sistema di gestione di un negozio. L'aggiunta del calcolo del prezzo

unitario e del calcolo dei totali offre un'esplorazione più approfondita di come le classi possono essere utilizzate per gestire dati e funzionalità correlate al prezzo.

```
class Negozio:
    """
    Classe per rappresentare un negozio.
    """

    def __init__(self, nome, categoria):
        """
        Inizializza un nuovo negozio con il nome e la categoria specificati.
        """
        self._nome = nome
        self._categoria = categoria
        self._inventario = {}

    def aggiungi_prodotto(self, nome, quantita, prezzo_unitario):
        """
        Aggiunge un prodotto all'inventario del negozio con il nome, la quantità e il prezzo unitario specificati.
        """
        if nome in self._inventario:
            self._inventario[nome]["quantita"] += quantita
        else:
            self._inventario[nome] = {"quantita": quantita, "prezzo_unitario": prezzo_unitario}

    def rimuovi_prodotto(self, nome, quantita):
        """
        Rimuove una quantità specificata di un prodotto dall'inventario del negozio.
        """
        if nome in self._inventario:
            if self._inventario[nome]["quantita"] >= quantita:
                self._inventario[nome]["quantita"] -= quantita
                if self._inventario[nome]["quantita"] == 0:
```

```
                del self._inventario[nome]
            else:
                print("Quantità insufficiente disponibile per rimuovere il prodotto.")
        else:
            print("Il prodotto non è presente nell'inventario.")

    def ottieni_inventario(self):
        """
        Restituisce l'inventario del negozio.
        """
        return self._inventario

    def calcola_totale_vendite(self):
        """
        Calcola il totale delle vendite effettuate nel negozio.
        """
        totale = 0
        for prodotto, dettagli in self._inventario.items():
            totale += dettagli["quantita"] * dettagli["prezzo_unitario"]
        return totale
```

```
class Cliente:
    """
    Classe per rappresentare un cliente di un negozio.
    """

    def __init__(self, nome):
        """
        Inizializza un nuovo cliente con il nome specificato.
        """
        self._nome = nome
        self._carrello = {}

    def aggiungi_al_carrello(self, nome_prodotto, quantita):
```

```
    """
    Aggiunge un prodotto al carrello del cliente con il nome e la quantità specificati.
    """
    if nome_prodotto in self._carrello:
        self._carrello[nome_prodotto] += quantita
    else:
        self._carrello[nome_prodotto] = quantita

def rimuovi_dal_carrello(self, nome_prodotto, quantita):
    """
    Rimuove una quantità specificata di un prodotto dal carrello del cliente.
    """
    if nome_prodotto in self._carrello:
        if self._carrello[nome_prodotto] >= quantita:
            self._carrello[nome_prodotto] -= quantita
            if self._carrello[nome_prodotto] == 0:
                del self._carrello[nome_prodotto]
        else:
            print("Quantità insufficiente disponibile per rimuovere il prodotto dal carrello.")
    else:
        print("Il prodotto non è presente nel carrello.")

def ottieni_carrello(self):
    """
    Restituisce il carrello del cliente.
    """
    return self._carrello

def calcola_totale_acquisto(self, negozio):
    """
    Calcola il totale dell'acquisto del cliente presso il negozio specificato.
    """
    totale = 0
    for prodotto, quantita in self._carrello.items():
```

```
        if prodotto in negozio.ottieni_inventario():
            prezzo_unitario = negozio.ottieni_inventario()[prodotto]["prezzo_unitario"]
            totale += quantita * prezzo_unitario
        else:
            print(f"Il prodotto '{prodotto}' non è disponibile nel negozio.")
    return totale
```

```
def main():
    # Creazione di un'istanza del negozio
    mio_negozio = Negozio("Supermercato", "Alimentari")

    # Aggiunta di prodotti all'inventario del negozio con prezzo unitario
    mio_negozio.aggiungi_prodotto("Pasta", 10, 2.5)
    mio_negozio.aggiungi_prodotto("Riso", 5, 3.0)
    mio_negozio.aggiungi_prodotto("Latte", 8, 1.5)

    # Creazione di un'istanza del cliente
    io = Cliente("Mario")

    # Aggiunta di prodotti al carrello del cliente
    io.aggiungi_al_carrello("Pasta", 2)
    io.aggiungi_al_carrello("Latte", 3)

    # Rimozione di prodotti dal carrello del cliente
    io.rimuovi_dal_carrello("Pasta", 1)

    # Stampa dell'inventario del negozio
    inventario_negozio = mio_negozio.ottieni_inventario()
    print("Inventario del negozio:")
    for prodotto, dettagli in inventario_negozio.items():
        print(f"{prodotto}: {dettagli['quantita']}, Prezzo Unitario: {dettagli['prezzo_unitario']}")

    # Stampa del carrello del cliente
    carrello_cliente = io.ottieni_carrello()
```

```
    print("Carrello del cliente:")
    for prodotto, quantita in carrello_cliente.items():
        print(f"{prodotto}: {quantita}")

    # Calcolo del totale delle vendite del negozio
    totale_vendite = mio_negozio.calcola_totale_vendite()
    print(f"Totale delle vendite del negozio: {totale_vendite}")

    # Calcolo del totale dell'acquisto del cliente
    totale_acquisto = io.calcola_totale_acquisto(mio_negozio)
    print(f"Totale dell'acquisto del cliente: {totale_acquisto}")
```

```
if __name__ == "__main__":
    main()
```

Esercizio 4: cifrario

L'obiettivo di questo esercizio è creare una classe Cifrario che crittografi e decrittografi un messaggio utilizzando un algoritmo di sostituzione. L'algoritmo sostituisce i caratteri del messaggio originale con altri caratteri in base a una chiave specificata.

La classe Cifrario ha i seguenti obiettivi:

- Fornire un'interfaccia per crittografare un messaggio utilizzando un algoritmo di sostituzione.
- Fornire un'interfaccia per decrittografare un messaggio crittografato utilizzando la stessa chiave di crittografia.
- Consentire l'utilizzo di una chiave personalizzata per l'algoritmo di sostituzione, ma fornire anche una chiave predefinita.
- Consentire la modifica della chiave predefinita per tutti gli oggetti Cifrario.
- Proteggere gli attributi e i metodi sensibili utilizzando attributi e metodi privati o protetti.

L'esercizio richiede l'implementazione di diverse funzionalità, tra cui l'inizializzazione di un oggetto Cifrario con una chiave specificata o la chiave predefinita, la crittografia di un messaggio utilizzando l'algoritmo di sostituzione, la decrittografia di un messaggio cifrato utilizzando la stessa chiave, e la possibilità di impostare una nuova chiave predefinita per tutti gli oggetti Cifrario.

```
class Cifrario:
    """
    Classe per crittografare e decrittografare un messaggio utilizzando un algoritmo di sostituzione.
    """

    chiave_predefinita = "qwertyuiopasdfghjklzxcvbnm"

    def __init__(self, chiave=None):
        """
        Inizializza un nuovo oggetto Cifrario con la chiave specificata o utilizza la chiave predefinita.

        Args:
            chiave (str, optional): La chiave per l'algoritmo di sostituzione. Se non fornita, viene utilizzata la chiave predefinita.
        """
        if chiave is not None:
            self._chiave = chiave
        else:
            self._chiave = Cifrario.chiave_predefinita
        self.__alfabeto = "abcdefghijklmnopqrstuvwxyz"

    @classmethod
    def imposta_chiave_predefinita(cls, nuova_chiave):
        """
        Imposta la chiave predefinita per tutti gli oggetti Cifrario.

        Args:
            nuova_chiave (str): La nuova chiave predefinita.
```

```
    """
    cls.chiave_predefinita = nuova_chiave

@staticmethod
def __applica_sostituzione(carattere, chiave):
    """
    Applica la sostituzione utilizzando la chiave sull'alfabeto.

    Args:
        carattere (str): Il carattere da sostituire.
        chiave (str): La chiave per l'algoritmo di sostituzione.

    Returns:
        str: Il carattere cifrato o originale.
    """
    alfabeto = "abcdefghijklmnopqrstuvwxyz"
    if carattere in alfabeto:
        indice = alfabeto.index(carattere)
        carattere_cifrato = chiave[indice]
        return carattere_cifrato
    else:
        return carattere

def crittografa(self, messaggio):
    """
    Crittografa il messaggio utilizzando l'algoritmo di sostituzione.

    Args:
        messaggio (str): Il messaggio da crittografare.

    Returns:
        str: Il messaggio crittografato.
    """
    messaggio_cifrato = ""
```

```
        for carattere in messaggio:
            carattere_cifrato = self.__applica_sostituzione(carattere.lower(), self._chiave)
            messaggio_cifrato += carattere_cifrato
        return messaggio_cifrato

    def decrittografa(self, messaggio_cifrato):
        """
        Decrittografa il messaggio cifrato utilizzando l'algoritmo di sostituzione.

        Args:
            messaggio_cifrato (str): Il messaggio cifrato da decrittografare.

        Returns:
            str: Il messaggio originale decrittografato.
        """
        messaggio_originale = ""
        for carattere in messaggio_cifrato:
            carattere_originale = self.__applica_sostituzione(carattere.lower(), self._chiave)
            messaggio_originale += carattere_originale
        return messaggio_originale
```

```
def main():
    # Creazione di un'istanza del cifrario con una chiave specifica
    cifrario = Cifrario("qwertyuiopasdfghjklzxcvbnm")

    # Crittografa un messaggio
    messaggio = "ciao, come stai?"
    messaggio_cifrato = cifrario.crittografa(messaggio)
    print("Messaggio cifrato:", messaggio_cifrato)  # Stampa: "ompm, yodk fyop?"

    # Decrittografa il messaggio cifrato
    messaggio_decifrato = cifrario.decrittografa(messaggio_cifrato)
    print("Messaggio decifrato:", messaggio_decifrato)  # Stampa: "ciao, come stai?"
```

```
    # Imposta una nuova chiave predefinita per tutti gli oggetti Cifrario
    Cifrario.imposta_chiave_predefinita("zyxwvutsrqponmlkjihgfedcba")

    # Crea un nuovo oggetto Cifrario senza specificare la chiave
    nuovo_cifrario = Cifrario()

    # Crittografa un messaggio utilizzando la nuova chiave predefinita
    messaggio_cifrato_nuovo_cifrario = nuovo_cifrario.crittografa(messaggio)
    print("Messaggio cifrato (nuovo cifrario):", messaggio_cifrato_nuovo_cifrario)

if __name__ == "__main__":
    main()
```

Capitolo 6: La Formattazione delle stringhe

La formattazione delle stringhe è un'operazione comune nella programmazione *Python* per manipolare e presentare i dati in modo leggibile e personalizzato. In questo capitolo, esploreremo le diverse tecniche e approcci disponibili.

È importante scegliere l'opzione più adatta in base alle esigenze del tuo progetto, tenendo conto di leggibilità, flessibilità e prestazioni.

Le stringhe raw

Le stringhe raw, spesso indicate con il prefisso *"r"*, sono un tipo speciale di stringhe in *Python* che trattano gli *escape characters* in modo letterale anziché come sequenze di escape. Ciò significa che i caratteri preceduti dal backslash *"\"* sono considerati come caratteri normali anziché essere interpretati come sequenze speciali.

Le *raw strings* sono utili in diverse situazioni, come quando si lavora con espressioni regolari, percorsi di file di sistema operativo o testi che contengono molti caratteri speciali.

Ecco alcuni punti chiave sulle *raw strings*:

- Una stringa raw viene creata aggiungendo il prefisso "r" (o "R") all'inizio della stringa. Ad esempio: *stringa_raw = r"Ciao\nMondo"*.
- Gli escape characters come \n, \r, \t, \\, \", ecc., all'interno di una raw string vengono trattati come caratteri normali e non come sequenze di escape.
- I backslash che non precedono un carattere di escape vengono considerati come caratteri letterali all'interno di una raw string.
- Le raw strings possono semplificare la scrittura di percorsi di file di sistema operativo, in quanto non è necessario utilizzare doppi backslash per separare i componenti del percorso. Ad esempio: *percorso_file = r"C:\Programmi\MieiFile\file.txt"*.
- Le raw strings sono frequentemente utilizzate quando si lavora con espressioni regolari (vedi Capitolo 7: Le espressioni regolari), in quanto semplificano la scrittura di pattern che contengono molti caratteri speciali senza dover utilizzare doppi backslash.
- Le raw strings non influenzano le operazioni di concatenazione o formattazione delle stringhe. Possono essere combinate con altre stringhe e utilizzate in metodi di formattazione come format() senza alcuna differenza.

Ecco un esempio che mostra l'utilizzo delle raw strings:

```
percorso_file = r"C:\Programmi\MieiFile\file.txt"
espressione_regolare = r"\d{3}-\d{3}-\d{4}"
```

Nell'esempio sopra, la variabile *percorso_file* contiene un percorso di file di sistema operativo rappresentato come raw string. Ciò permette di scrivere il percorso senza dover utilizzare doppi backslash come caratteri di escape.

La variabile *espressione_regolare* contiene un pattern per un numero di telefono scritto come raw string. In questo modo, non è necessario utilizzare doppi backslash per i metacaratteri dell'espressione regolare come \d.

Le raw strings semplificano la gestione di caratteri speciali e sequenze di escape all'interno delle stringhe, rendendo il codice più leggibile e meno soggetto a errori quando si lavora con testi che contengono molti caratteri speciali o pattern complessi.

Metodo format

Il metodo *format()* è uno dei principali strumenti per formattare le stringhe. Consente di incorporare valori all'interno di una stringa utilizzando il segnaposto {}.

```
name = "Alice"
age = 25
message = "Ciao, mi chiamo {} e ho {} anni.".format(name, age)
print(message) #Output: "Ciao, mi chiamo Alice e ho 25 anni."
```

È anche possibile specificare formati più avanzati per i valori, come ad esempio la larghezza del campo, il numero di decimali per i numeri, l'allineamento, l'indicizzazione e ripetizione.

Ecco un esempio di utilizzo del metodo *format()* per formattare un numero con un numero specifico di decimali:

```
pi = 3.14159265359


# Formattazione con 2 decimali
formatted_pi = "Valore di pi: {:.2f}".format(pi)
print(formatted_pi)  #output: Valore di pi: 3.14
```

```
# Formattazione con 4 decimali
formatted_pi = "Valore di pi: {:.4f}".format(pi)
print(formatted_pi)  #output: Valore di pi: 3.1416
```

Veidiamo un altro esempio di utilizzo del metodo *format()* con l'allineamento dei valori:

```
name = "Alice"
age = 25

# Allineamento a sinistra
print("Nome: {:<10} Età: {}".format(name, age))

# Allineamento a destra
print("Nome: {:>10} Età: {}".format(name, age))

# Allineamento centrato
print("Nome: {:^10} Età: {}".format(name, age))
```

Output:

```
Nome: Alice      Età: 25
Nome:      Alice Età: 25
Nome:   Alice    Età: 25
```

Nell'esempio sopra, abbiamo utilizzato la sintassi:

- {:<10} allinea il valore a sinistra all'interno di un campo di larghezza 10.
- {:>10} allinea il valore a destra all'interno di un campo di larghezza 10.
- {:^10} allinea il valore al centro all'interno di un campo di larghezza 10.

È possibile modificare la larghezza del campo in base alle proprie esigenze. Questo tipo di allineamento può essere utile per organizzare e presentare in

modo uniforme i dati nelle stringhe formattate.

Come ultimo esempio mostriamo l'utilizzo dell'indicizzazione e della ripetizione.

```
name = "Alice"
age = 25

# Indicizzazione degli argomenti
message = "Il mio nome è {0} e ho {1} anni.".format(name, age)
print(message)  #output: Il mio nome è Alice e ho 25 anni.

# Ripetizione degli argomenti
message = "Il mio nome è {0} e ho {1} anni. {0} è un bel nome!".format(\
name, age)
print(message)
#output: Il mio nome è Alice e ho 25 anni. Alice è un bel nome!
```

Abbiamo utilizzato le indicizzazioni {0} e {1} all'interno della stringa di formattazione per indicare la posizione degli argomenti nell'ordine in cui sono passati al metodo *format()*. Questo ci consente di controllare l'ordine in cui i valori vengono inseriti nella stringa.

Inoltre, abbiamo ripetuto l'argomento {0} nella seconda stringa di formattazione per mostrarne l'effetto di ripetizione.

L'utilizzo dell'indicizzazione e della ripetizione può essere utile quando si desidera controllare l'ordine dei valori o quando si desidera ripetere lo stesso valore più volte all'interno della stringa formattata.

F-strings (Formatted String Literals)

Le *F-strings* sono una nuova sintassi introdotta in *Python 3.6* per semplificare la formattazione delle stringhe. Consentono di incorporare direttamente le variabili all'interno delle stringhe precedute dal prefisso 'f'.

```
name = "Bob"
age = 30
message = f"Ciao, mi chiamo {name} e ho {age} anni."
```

```
print(message)  # Output: "Ciao, mi chiamo Bob e ho 30 anni."
```

Le *F-strings* supportano anche l'utilizzo di espressioni *Python* all'interno delle parentesi graffe, consentendo di eseguire calcoli o manipolazioni di dati complessi direttamente nella stringa formattata.

```
name = "Alice"
age = 25

# Calcoli all'interno delle parentesi
message = f"Il mio nome è {name.upper()} e tra 10 anni \
avrò {age + 10} anni."
print(message)  #output: Il mio nome è ALICE e tra 10 anni avrò 35 anni.

# Chiamate di funzioni all'interno delle parentesi
def double(x):
    return x * 2

number = 5
result = f"Il doppio di {number} è {double(number)}."
print(result)  #output: Il doppio di 5 è 10.
```

Operatore di concatenazione e conversione di stringhe

L'operatore di concatenazione delle stringhe (+) può essere utilizzato per unire più stringhe. Tuttavia, se si desidera concatenare una stringa con un valore non stringa, è necessario convertire esplicitamente il valore in una stringa utilizzando la funzione *str()*.

```
name = "Carol"
age = 35
message = "Ciao, mi chiamo " + name + " e ho " + str(age) + " anni."
print(message)  # Output: "Ciao, mi chiamo Carol e ho 35 anni."
```

L'operatore di concatenazione delle stringhe può essere meno flessibile e meno leggibile rispetto alle altre tecniche di formattazione, ma può essere utile in determinate situazioni.

Metodo join

Il metodo *join()* è utile per concatenare elementi di una lista in una singola stringa. Può essere combinato con una *list comprehension* (vedi List comprehension) per formattare una lista di valori in modo personalizzato.

```
fruits = ["mela", "banana", "arancia"]
formatted_fruits = ", ".join([f"Ho una {fruit}" for fruit in fruits])
print(formatted_fruits)
# Output: "Ho una mela, Ho una banana, Ho un'arancia"
```

Questo metodo è particolarmente utile quando si desidera formattare una lista di elementi con uno schema comune.

Modulo string.Template

Il modulo *string.Template* fornisce una modalità di formattazione delle stringhe basata su modelli predefiniti. È utile quando si desidera consentire agli utenti di personalizzare le stringhe di output. Ecco un esempio:

```
from string import Template

name = "Eva"
age = 50

template = Template("Ciao, mi chiamo $name e ho $age anni.")
message = template.substitute(name=name, age=age)
print(message)  # Output: "Ciao, mi chiamo Eva e ho 50 anni."
```

Vediamo un altro esempio:

```
from string import Template

# Template con sostituzione di lista
template1 = Template("Ciao, sono $name e mi piacciono: $hobbies")
```

```
name = "Alice"
hobbies = ["pittura", "musica", "giardinaggio"]
result1 = template1.substitute(name=name, hobbies=", ".join(hobbies))
print(result1)
# output: Ciao, sono Alice e mi piacciono: pittura, musica, giardinaggio

# Template con sostituzione di dizionario
template2 = Template("Il mio nome è $name e sono di $city")
info = {"name": "Bob", "city": "New York"}
result2 = template2.substitute(info)
print(result2) #output: Il mio nome è Bob e sono di New York
```

Questo modulo offre un'alternativa per la formattazione delle stringhe, specialmente quando è necessario consentire una maggiore personalizzazione da parte dell'utente.

Formattazione delle date e degli orari

Per formattare le date e gli orari, è possibile utilizzare il modulo *datetime* e il suo metodo *strftime()*. Questo metodo consente di specificare formati personalizzati per rappresentare le date e gli orari.

```
from datetime import datetime

adesso = datetime.now()
print(adesso.strftime("%Y-%m-%d %H:%M:%S"))
# Output: 2022-01-01 15:30:45
```

Vediamo un altro esempio:

```
import datetime

# Data corrente
data_corrente = datetime.date.today()

# Formattazione della data
data_formattata = data_corrente.strftime("%d/%m/%Y")
```

```
print("Data formattata:", data_formattata)

# Ora corrente
ora_corrente = datetime.datetime.now().time()

# Formattazione dell'ora
ora_formattata = ora_corrente.strftime("%H:%M:%S")
print("Ora formattata:", ora_formattata)
```

In questo esempio, per la formattazione della data, utilizziamo il formato *%d/%m/%Y* che rappresenta il giorno, il mese e l'anno separati da /. Ad esempio, l'output potrebbe essere "13/06/2023" per il 13 giugno 2023.

Per la formattazione dell'ora, utilizziamo il formato %H:%M:%S che rappresenta l'ora, i minuti e i secondi separati da :. Ad esempio, l'output potrebbe essere "16:30:45" per le 16:30:45.

Puoi personalizzare il formato di formattazione utilizzando i diversi simboli disponibili nella documentazione ufficiale di *Python* per il modulo *datetime:*

https://docs.python.org/3/library/datetime.html\#strftime-strptime-behavior

Capitolo 7: Le espressioni regolari

Introduzione alle espressioni regolari

Le espressioni regolari sono sequenze di caratteri che definiscono modelli di ricerca all'interno di un testo. Sono ampiamente utilizzate nella programmazione per la ricerca e la manipolazione efficiente di testi complessi.

Le espressioni regolari consentono di specificare criteri di corrispondenza per identificare parti specifiche di un testo, come parole, frasi, numeri, formati di data e altro ancora. La loro utilità nella ricerca e manipolazione di testi è notevole.

Possono essere utilizzate per diverse operazioni, come la verifica di un formato specifico per un campo dati, la validazione di indirizzi e-mail, la ricerca di parole chiave in un documento o l'estrarre informazioni strutturate da un testo non strutturato.

La loro flessibilità e potenza consentono di gestire complessi schemi di ricerca, inclusi caratteri speciali, quantificatori, gruppi di cattura e altro ancora. Sono strumenti fondamentali per programmatori, data scientist, web developer e altri professionisti che lavorano con dati e testi.

Sono supportate in molti linguaggi di programmazione, incluso *Python* con il modulo **re**.

In questa sezione del libro, si esplorerà l'importanza delle espressioni regolari nella ricerca e manipolazione di testi, fornendo esempi pratici e spiegando le varie funzionalità offerte dal modulo *re* di *Python*. Ciò consentirà ai lettori di acquisire le competenze necessarie per utilizzare efficacemente le espressioni regolari nella programmazione *Python* e sfruttarne appieno il loro potenziale nella gestione di testi complessi.

Descrizione del modulo re di Python

Il modulo re di *Python* è una libreria standard potente e versatile che offre supporto completo per le espressioni regolari. Esso fornisce una vasta gamma di funzionalità per la manipolazione e la gestione di pattern di testo complessi.

Con il modulo *re*, è possibile compilare ed utilizzare espressioni regolari in modo efficiente. La compilazione di un'espressione regolare viene effettuata utilizzando la funzione *re.compile(),* che converte la stringa dell'espressione regolare in un oggetto pattern. Una volta compilato, l'oggetto pattern può essere utilizzato per eseguire varie operazioni sul testo.

Il modulo *re* offre diverse funzioni utili per lavorare con espressioni regolari. Ad esempio, la funzione *search()* consente di cercare la prima occorrenza del pattern all'interno di una stringa di testo, mentre la funzione *match()* effettua una corrispondenza solo all'inizio della stringa. Inoltre, la funzione *findall()* restituisce una lista di tutte le corrispondenze trovate, mentre la funzione f*inditer()* restituisce un iteratore per scorrere tutte le corrispondenze.

Oltre alle funzioni di ricerca, il modulo *re* offre anche funzionalità per la sostituzione del testo. La funzione *sub()* consente di sostituire le corrispondenze trovate con un'altra stringa o con il risultato di una funzione specificata.

Il modulo *re* supporta anche diverse opzioni per modificare il comportamento delle espressioni regolari, come l'uso dei flag per gestire maiuscole/minuscole, la modalità multilinea e altro ancora.

In questa sezione del libro, verrà esaminato in dettaglio il modulo *re* di *Python*. Saranno presentate le funzioni principali offerte dal modulo, fornendo esempi pratici per illustrarne l'utilizzo corretto. I lettori saranno in grado di comprendere appieno il funzionamento del modulo e saranno in grado di utilizzare con successo le espressioni regolari nel loro codice *Python* per la manipolazione e la ricerca avanzata di testi.

Sintassi delle espressioni regolari:

La sintassi delle espressioni regolari offre una serie di caratteristiche fondamentali che consentono di costruire pattern di ricerca complessi. Queste caratteristiche includono metacaratteri speciali, modelli di quantificatori, classi di caratteri, gruppi di cattura e ancoraggi.

I *metacaratteri speciali* sono caratteri con un significato speciale nelle espressioni regolari. Ad esempio, il punto **.** rappresenta qualsiasi carattere,

mentre *****, + e **?** indicano rispettivamente zero o più, uno o più, e zero o una occorrenza del carattere o gruppo precedente.

I modelli di quantificatori consentono di specificare la quantità di occorrenze di un carattere o gruppo. Ad esempio, *{n}* indica esattamente n occorrenze, *{n,}* indica almeno n occorrenze e *{n,m}* indica da *n* a *m* occorrenze.

Le classi di caratteri definiscono un insieme di caratteri tra cui cercare una corrispondenza. Ad esempio, *[abc]* corrisponde a uno qualsiasi dei caratteri *a*, *b* o *c*, mentre *[^abc]* corrisponde a un carattere diverso da *a*, *b* o *c*.

Le parentesi ( ) consentono di creare gruppi di cattura, ovvero porzioni del pattern che possono essere estratte separatamente. Questo permette di raggruppare parti del pattern per applicare quantificatori o applicare modificatori.

Gli ancoraggi ^ e $ indicano rispettivamente l'inizio e la fine di una riga. Ad esempio, *^ciao* corrisponde a una riga che inizia con *"ciao"*, mentre *fine$* corrisponde a una riga che termina con *"fine"*.

Altre caratteristiche importanti includono l'uso del backslash \ per *l'escaping* di caratteri speciali, l'opzione di ricerca per l'insensibilità alle maiuscole/minuscole, la modalità multilinea e altre opzioni specifiche per modificare il comportamento delle espressioni regolari.

La comprensione di queste caratteristiche chiave è essenziale per costruire pattern di ricerca precisi e potenti utilizzando le espressioni regolari. Nelle prossime righe di questa sezione approfondiremo ciascuna di queste caratteristiche, fornendo esempi pratici per illustrarne l'utilizzo corretto e offrendo suggerimenti per sfruttarne appieno il potenziale nelle applicazioni di programmazione *Python*.

I metacaratteri

- Il metacarattere + indica una o più occorrenze del carattere o gruppo precedente. Ad esempio, il pattern ab+c corrisponde a *"abc"*, *"abbc"*, *"abbbc"* e così via.
- Il metacarattere **?** indica zero o una occorrenza del carattere o gruppo precedente. Ad esempio, il pattern colou?r corrisponde a *"color"* e *"colour"*.

- Le parentesi quadre **[ ]** definiscono una classe di caratteri, specificando un insieme di caratteri tra cui cercare una corrispondenza. Ad esempio, il pattern *[abc]* corrisponde a *"a"*, *"b"* o *"c"*.
- Il simbolo | rappresenta l'operatore di alternativa, corrispondendo a uno dei due pattern separati dall'operatore. Ad esempio, il pattern *red|blue* corrisponde a *"red"* o *"blue"*.
- Le parentesi **( )** permettono di creare gruppi di cattura, raggruppando parti del pattern. Questo permette di applicare quantificatori o applicare modificatori su un gruppo specifico. Ad esempio, il pattern *(ab)+* corrisponde a *"ab"*, *"abab"*, *"abababab"* e così via.
- Il carattere di backslash \ viene utilizzato per l'escaping dei metacaratteri speciali. Per esempio, \., * vengono utilizzati per cercare un punto o un asterisco letterali.
- Altri metacaratteri speciali, come **\d**, **\w**, **\s**, rappresentano classi di caratteri predefinite. Ad esempio, *\d* corrisponde a una cifra, *\w* corrisponde a una lettera o una cifra, e *\s* corrisponde a uno spazio bianco.

Comprendere il significato e l'utilizzo di questi metacaratteri speciali nelle espressioni regolari è fondamentale per costruire pattern di ricerca accurati e potenti.

I modelli di quantificatori

I modelli di quantificatori sono un'importante caratteristica delle espressioni regolari che permettono di specificare la quantità di occorrenze di un carattere o gruppo di caratteri. Questi quantificatori sono utilizzati per definire pattern di ricerca che possono essere più o meno restrittivi a seconda delle esigenze dell'applicazione.

I principali modelli di quantificatori includono:

- Asterisco *: Questo modello indica zero o più occorrenze del carattere o gruppo precedente. Ad esempio, il pattern *ab*c* corrisponde a *"ac"*, *"abc"*, *"abbc"*, *"abbbc"* e così via. Se il carattere o gruppo precedente non è presente, il pattern sarà ancora considerato valido.
- Più +: Questo modello indica una o più occorrenze del carattere o gruppo precedente. Ad esempio, il pattern *ab+c* corrisponde a *"abc"*,

"abbc", *"abbbc"* e così via. Il pattern non sarà valido se il carattere o gruppo precedente non è presente almeno una volta.

- Interrogativo **?**: Questo modello indica zero o una occorrenza del carattere o gruppo precedente. Ad esempio, il pattern *colou?r* corrisponde a *"color"* e *"colour"*. Il carattere o gruppo precedente può essere presente una volta o essere assente.
- Accolade **{n}**: Questo modello specifica un numero esatto di occorrenze del carattere o gruppo precedente. Ad esempio, il pattern *a{3}* corrisponde a *"aaa"*. Il pattern non sarà valido se il numero di occorrenze non corrisponde esattamente a *n*.
- Accolade **{n,}**: Questo modello indica almeno n occorrenze del carattere o gruppo precedente. Ad esempio, il pattern *a{2,}* corrisponde a *"aa"*, *"aaa"*, *"aaaa"* e così via. Il pattern sarà valido anche se ci sono più di *n* occorrenze.
- Accolade **{n,m}**: Questo modello specifica un intervallo di occorrenze del carattere o gruppo precedente, da *n* a *m*. Ad esempio, il pattern *a{2,4}* corrisponde a *"aa"*, *"aaa"* e *"aaaa"*. Il pattern non sarà valido se il numero di occorrenze è inferiore a *n* o superiore a *m*.

L'utilizzo dei modelli di quantificatori consente di definire pattern di ricerca più flessibili e precisi. È possibile combinare questi modelli con altri metacaratteri e classi di caratteri per ottenere risultati più specifici. Tuttavia, è importante prestare attenzione all'utilizzo corretto dei quantificatori per evitare risultati indesiderati o inefficienti.

Le classi di caratteri

Le classi di caratteri sono un'importante caratteristica delle espressioni regolari che permette di definire un insieme di caratteri tra cui cercare una corrispondenza. Queste classi consentono di specificare criteri più specifici per la ricerca di determinati caratteri all'interno di un testo.

Le principali caratteristiche delle classi di caratteri sono le seguenti:

- Parentesi quadre **[ ]**: le classi di caratteri sono delimitate dalle parentesi quadre. All'interno delle parentesi quadre, è possibile elencare una serie di caratteri o intervalli di caratteri separati da trattini. Ad esempio, *[abc]* corrisponde a uno qualsiasi dei caratteri *"a"*, *"b"* o *"c"*.

- Ranges di caratteri: i trattini (-) all'interno delle parentesi quadre sono utilizzati per definire intervalli di caratteri. Ad esempio, *[a-z]* corrisponde a qualsiasi carattere compreso tra *"a"* e *"z"* inclusi. Allo stesso modo, *[0-9]* corrisponde a qualsiasi cifra da *0* a *9*.
- Caratteri speciali nelle classi di caratteri: alcuni caratteri hanno un significato speciale all'interno delle classi di caratteri. Ad esempio, il metacarattere *"-"* ha un significato speciale quando viene utilizzato all'interno di una classe di caratteri per indicare un intervallo. Per includere il carattere *"-"* letterale nella classe, è necessario posizionarlo all'inizio o alla fine. Allo stesso modo, il carattere *"^"* ha un significato speciale quando viene utilizzato come primo carattere all'interno delle parentesi quadre, indicando la negazione della classe.
- Caratteri di escape: in alcune situazioni, può essere necessario includere caratteri speciali o metacaratteri letterali all'interno di una classe di caratteri. In tali casi, è possibile utilizzare il carattere di escape *""*, che permette di trattare il carattere successivo come un carattere letterale anziché come un metacarattere.

Le classi di caratteri consentono di cercare corrispondenze per una serie di caratteri specifici, fornendo una maggiore flessibilità e precisione nelle espressioni regolari. Possono essere combinati con altri metacaratteri e modelli di quantificatori per definire pattern di ricerca più sofisticati.

I gruppi di cattura

I gruppi di cattura sono utilizzati nelle espressioni regolari per raggruppare e catturare parti specifiche di un match. Utilizzando le parentesi tonde **( )**, è possibile delimitare il sotto-pattern che si desidera catturare e assegnare a un gruppo di cattura.

Ci sono due modi principali in cui i gruppi di cattura possono essere utilizzati:

- **Recupero dei gruppi di cattura**: dopo aver trovato una corrispondenza con un'espressione regolare, è possibile accedere ai gruppi di cattura all'interno della corrispondenza per ottenere le porzioni specifiche del testo corrispondente. Ad esempio, se si cerca un numero di telefono nel formato "XXX-XXX-XXXX", è possibile

utilizzare gruppi di cattura per estrarre le tre parti separate del numero di telefono (prefisso, parte centrale, suffisso).

- **Retrovisioni**: le retrovisioni consentono di fare riferimento a un match precedente all'interno della stessa espressione regolare. Utilizzando gruppi di cattura, è possibile riferirsi a un match precedentemente trovato e utilizzarlo nella stessa espressione regolare. Ad esempio, si potrebbe cercare parole duplicate utilizzando un gruppo di cattura e una retrovisione per confrontare il testo trovato con quello precedente.

Ecco alcuni punti chiave relativi ai gruppi di cattura:

- I gruppi di cattura sono delimitati dalle parentesi tonde ( ) all'interno di un'espressione regolare.
- Ogni gruppo di cattura viene assegnato a un numero univoco in base all'ordine di apertura delle parentesi. Il primo gruppo di cattura è il gruppo 1, il secondo gruppo è il gruppo 2 e così via.
- È possibile accedere ai gruppi di cattura all'interno di una corrispondenza utilizzando metodi specifici della libreria o del linguaggio di programmazione utilizzato.
- L'utilizzo dei gruppi di cattura consente di estrarre parti specifiche di un match per ulteriori elaborazioni o manipolazioni del testo.
- Le retrovisioni consentono di fare riferimento a un match precedentemente trovato all'interno della stessa espressione regolare, utilizzando il numero del gruppo di cattura corrispondente.

L'utilizzo dei gruppi di cattura nelle espressioni regolari offre una maggiore flessibilità e precisione nell'elaborazione dei testi corrispondenti. Consentono di estrarre parti specifiche di un match e di utilizzare informazioni precedenti all'interno della stessa espressione regolare.

Gli ancoraggi

Gli ancoraggi sono un'importante caratteristica delle espressioni regolari che consentono di specificare posizioni specifiche all'interno di un testo in cui cercare una corrispondenza. Gli ancoraggi sono utilizzati per delimitare l'inizio o la fine di una riga o una parola, fornendo criteri più precisi per la ricerca di corrispondenze.

Le principali caratteristiche degli ancoraggi sono le seguenti:

- Ancoraggio di inizio di riga ^: questo ancoraggio viene utilizzato per indicare l'inizio di una riga. Ad esempio, il pattern *^ciao* corrisponde a una riga che inizia con *"ciao"*. L'ancoraggio di inizio di riga viene spesso utilizzato con il flag multilinea (m), che viene utilizzato per abilitare l'ancoraggio di inizio linea (^) a corrispondere a ogni linea del testo di input invece che solo all'inizio dell'intera stringa. In altre parole, consente di considerare ogni linea del testo separatamente per le corrispondenze.
- Ancoraggio di fine di riga **$**: questo ancoraggio viene utilizzato per indicare la fine di una riga. Ad esempio, il pattern *fine$* corrisponde a una riga che termina con *"fine"*. Anche in questo caso, l'ancoraggio di fine di riga è spesso utilizzato con il flag multilinea.
- Ancoraggio di parola intera **\b**: questo ancoraggio viene utilizzato per indicare la fine o l'inizio di una parola. Ad esempio, il pattern *\bciao\b* corrisponde a *"ciao"* come parola indipendentemente dai caratteri circostanti. L'ancoraggio di parola intera può essere utilizzato per evitare corrispondenze parziali all'interno di parole più lunghe.
- Ancoraggio negativo **\B**: questo ancoraggio viene utilizzato per indicare una posizione che non è un confine di parola. Ad esempio, il pattern *\Bciao\B* corrisponde a *"ciao"* solo se non è presente all'inizio o alla fine di una parola. L'ancoraggio negativo può essere utilizzato per cercare corrispondenze all'interno di parole.

Gli ancoraggi consentono di stabilire criteri più precisi per la ricerca di corrispondenze nelle espressioni regolari. Possono essere utilizzati in combinazione con altri metacaratteri e modelli di quantificatori per ottenere risultati più specifici.

Lookahead positivo e negativo

I *lookahead* positivi e negativi sono costrutti avanzati utilizzati nelle espressioni regolari per effettuare delle previsioni sulla corrispondenza successiva senza consumare effettivamente i caratteri nella stringa di input.

- **Lookahead positivo (?=...)**

Il *lookahead* positivo si esprime con la sintassi (?=...). La sua funzione è verificare se un determinato pattern (rappresentato da ...) segue la posizione corrente nella stringa di input, senza includerlo nella corrispondenza finale.

- Il *lookahead* positivo afferma che il pattern specificato deve essere presente dopo la posizione corrente.
- Il *lookahead* positivo non consuma i caratteri corrispondenti. Dopo la valutazione del *lookahead* positivo, la ricerca riprende dalla stessa posizione iniziale.

Ad esempio, considera l'espressione regolare \w+(?=\d). Questo pattern cerca una sequenza di caratteri alfanumerici (\w+) che sia seguita da una cifra (\d). Il *lookahead* positivo (?=\d) verifica se la corrispondenza è seguita da una cifra, ma non include la cifra nella corrispondenza finale.

❖ **Lookahead negativo (?!...)**

Il *lookahead* negativo si esprime con la sintassi (?!...). La sua funzione è verificare se un determinato pattern (rappresentato da ...) non segue la posizione corrente nella stringa di input.

- Il *lookahead* negativo afferma che il pattern specificato non deve essere presente dopo la posizione corrente.
- Il *lookahead* negativo non consuma i caratteri corrispondenti. Dopo la valutazione del *lookahead* negativo, la ricerca riprende dalla stessa posizione iniziale.

Ad esempio, considera l'espressione regolare \w+(?!\d). Questo pattern cerca una sequenza di caratteri alfanumerici (\w+) che non sia seguita da una cifra (\d). Il *lookahead* negativo (?!\d) verifica che non ci sia una cifra dopo la corrispondenza, ma non include la cifra nella corrispondenza finale.

Utilizzo delle espressioni regolari in Python

Compilazione delle espressioni regolari

Il modulo *re* in *Python* fornisce la funzione *compile* che consente di creare un oggetto pattern. Questo oggetto pattern rappresenta un'espressione regolare compilata, che può essere utilizzata per effettuare corrispondenze e ricerche all'interno di stringhe.

La sintassi generale per utilizzare *re.compile* è la seguente:

```
pattern = re.compile(espressione_regolare, flag=0)
```

- *espressione_regolare* rappresenta l'espressione regolare che desideriamo compilare in un oggetto pattern.
- *flag* è un argomento opzionale che può essere utilizzato per specificare determinati comportamenti delle corrispondenze regolari. Ad esempio, il flag *re.IGNORECASE* consente di effettuare corrispondenze case-insensitive. Se non viene specificato alcun flag, viene utilizzato il valore predefinito 0.

Ecco un esempio pratico:

```
import re

pattern = re.compile(r'\d{2}-\d{2}-\d{4}')
```

In questo esempio, abbiamo utilizzato *re.compile* per creare un oggetto pattern che rappresenta un'espressione regolare per corrispondere a date nel formato "dd-mm-yyyy". L'espressione regolare *r'\d{2}-\d{2}-\d{4}'* corrisponde a due cifre seguite da un trattino, altre due cifre seguite da un trattino, e infine a quattro cifre.

Una volta creato l'oggetto pattern, possiamo utilizzarlo per effettuare corrispondenze e ricerche all'interno di stringhe utilizzando i metodi come *match*, *search* o *findall*. Ad esempio:

```
testo = "La data di oggi è 10-07-2023"

corrispondenza = pattern.search(testo)
if corrispondenza:
    print("Data trovata:", corrispondenza.group())
else:
    print("Nessuna data trovata.")

#output: Data trovata: 10-07-2023
```

In questo esempio, abbiamo utilizzato il metodo search sull'oggetto pattern per cercare una corrispondenza all'interno del testo. Se viene trovata una corrispondenza, viene stampata la data trovata.

Perché compilare un'espressione regolare?

Le espressioni regolari contengono pattern complessi che descrivono modelli di testo da cercare o manipolare. Quando si utilizza l'espressione regolare in *Python*, la funzione *re.compile* viene utilizzata per convertire l'espressione regolare in un oggetto pattern compilato. Ecco perché è consigliabile compilare un'espressione regolare:

- **Miglioramento delle prestazioni**: la compilazione dell'espressione regolare in un oggetto pattern porta a un miglioramento delle prestazioni durante le operazioni di corrispondenza e ricerca. L'oggetto pattern compilato viene ottimizzato internamente per rendere più efficienti le successive operazioni di matching su stringhe. Se si prevede di eseguire molte corrispondenze o ricerche con la stessa espressione regolare, la compilazione preliminare può offrire un notevole vantaggio in termini di velocità.
- **Riutilizzo dell'espressione regolare**: compilando l'espressione regolare in un oggetto pattern, è possibile riutilizzarla in diverse parti del codice. L'oggetto pattern è indipendente dal testo di input specifico su cui viene applicato, consentendo di utilizzare lo stesso pattern per eseguire corrispondenze su diverse stringhe senza dover ricompilare l'espressione regolare ogni volta. Ciò porta a un codice più pulito ed efficiente.
- **Leggibilità e manutenibilità**: separando la compilazione dell'espressione regolare in una fase di preparazione separata, il codice risulta più leggibile e più facile da mantenere. L'oggetto pattern contiene già l'espressione regolare compilata, rendendo il codice più chiaro e riducendo la duplicazione di codice.
- **Opzioni di flag**: durante la compilazione dell'espressione regolare, è possibile specificare opzioni di flag per modificare il comportamento delle corrispondenze. Queste opzioni possono includere la corrispondenza case-insensitive, il supporto per caratteri Unicode, la gestione di linee multiple, e altro ancora. La compilazione consente di

applicare facilmente queste opzioni a un oggetto pattern e di riutilizzarlo in modo coerente in tutto il codice.

Compilare un'espressione regolare in un oggetto pattern è quindi un'operazione raccomandata per migliorare le prestazioni, promuovere la riutilizzabilità del codice, aumentare la leggibilità e consentire l'applicazione coerente di opzioni di flag.

Metodi di ricerca e manipolazione delle espressioni regolari

Il modulo *re* in *Python* fornisce diversi metodi per eseguire ricerche e manipolazioni con le espressioni regolari. Di seguito sono descritti i principali metodi e viene fornito un esempio di output per ciascuno:

- Metodo ***search***

  - *search* cerca la prima corrispondenza dell'espressione regolare all'interno di una stringa e restituisce un oggetto corrispondenza.
  - Se viene trovata una corrispondenza, è possibile accedere ai dettagli utilizzando i metodi dell'oggetto corrispondenza come group().
  - Se nessuna corrispondenza viene trovata, restituisce None.

  Esempio:

```
import re

testo = "La mia email è test@example.com"
pattern = re.compile(r'\w+@\w+\.\w+')
corrispondenza = pattern.search(testo)
if corrispondenza:
    print("Corrispondenza trovata:", corrispondenza.group())
else:
    print("Nessuna corrispondenza trovata.")

#output: Corrispondenza trovata: test@example.com
```

❖ Metodo ***match***

- *match* cerca una corrispondenza all'inizio della stringa e restituisce un oggetto corrispondenza.
- Se la corrispondenza viene trovata all'inizio della stringa, è possibile accedere ai dettagli utilizzando i metodi dell'oggetto corrispondenza come group().
- Se la corrispondenza non viene trovata all'inizio della stringa, restituisce None.

Esempio:

```
import re

testo = "Ciao, come stai?"
pattern = re.compile(r'Ciao')
corrispondenza = pattern.match(testo)
if corrispondenza:
   print("Corrispondenza trovata:", corrispondenza.group())
else:
   print("Nessuna corrispondenza trovata.")

#output: Corrispondenza trovata: Ciao
```

❖ Metodo ***findall***

- *findall* trova tutte le corrispondenze dell'espressione regolare all'interno di una stringa e restituisce una lista contenente tutte le corrispondenze.
- Se nessuna corrispondenza viene trovata, restituisce una lista vuota.

Esempio:

```
import re

testo = "Questo è un test con numeri: 123 e 456"
pattern = re.compile(r'\d+')
```

```
corrispondenze = pattern.findall(testo)
if corrispondenze:
    print("Corrispondenze trovate:", corrispondenze)
else:
    print("Nessuna corrispondenza trovata.")

#output: Corrispondenze trovate: ['123', '456']
```

- Metodo ***finditer***

  - *finditer* trova tutte le corrispondenze dell'espressione regolare all'interno di una stringa e restituisce un iteratore che produce oggetti corrispondenza.
  - È possibile iterare sugli oggetti corrispondenza e accedere ai dettagli utilizzando i metodi come *group()*.

Esempio:

```
import re

testo = "Ciao mondo. Hello world."
pattern = re.compile(r'\b\w+\b')
corrispondenze = pattern.finditer(testo)
if corrispondenze:
    for corrispondenza in corrispondenze:
        print("Corrispondenza trovata:", corrispondenza.group())
else:
    print("Nessuna corrispondenza trovata.")
```

```
"""
Corrispondenza trovata: Ciao
Corrispondenza trovata: mondo
Corrispondenza trovata: Hello
Corrispondenza trovata: world
"""
```

❖ Metodo **sub**

- sub sostituisce tutte le corrispondenze dell'espressione regolare all'interno di una stringa con un testo specificato.
- Restituisce la stringa risultante dopo la sostituzione.

Esempio:

```
import re

testo = "Ciao mondo. Hello world."
pattern = re.compile(r'Hello')
nuovo_testo = pattern.sub("Hi", testo)
print("Nuovo testo:", nuovo_testo)

#output: Nuovo testo: Ciao mondo. Hi world.
```

Recupero dei gruppi di cattura

Supponiamo di voler cercare numeri di telefono nel formato "(XXX) XXX-XXXX" e estrarre separatamente l'indicativo internazionale e il numero locale.

```
import re

testo = "Il mio numero di telefono è (+123) 456-7890"
pattern = re.compile(r'\((\+\d+)\) (\d+-\d+)')
corrispondenza = pattern.search(testo)
if corrispondenza:
    indicativo = corrispondenza.group(1)
    numero_locale = corrispondenza.group(2)
    print("Indicativo internazionale:", indicativo)
    print("Numero locale:", numero_locale)
else:
    print("Nessuna corrispondenza trovata.")
```

```
"""
output:
Indicativo internazionale: +123
Numero locale: 456-7890
"""
```

In questo esempio, stiamo utilizzando l'espressione regolare r'\((\+\d+)\) (\d+-\d+)' per cercare numeri di telefono nel formato "(XXX) XXX-XXXX". L'uso dei gruppi di cattura ci consente di estrarre separatamente l'indicativo internazionale (gruppo 1) e il numero locale (gruppo 2).

Retrovisioni

Supponiamo di voler cercare parole duplicate all'interno di una stringa utilizzando retrovisioni.

```
import re

testo = "ciao ciao"
pattern = re.compile(r'(\b\w+\b) \1')
corrispondenza = pattern.search(testo)
if corrispondenza:
    parola_duplicata = corrispondenza.group()
    print("Parola duplicata trovata:", parola_duplicata)
else:
    print("Nessuna parola duplicata trovata.")

#output: Parola duplicata trovata: ciao ciao
```

In questo esempio, stiamo utilizzando l'espressione regolare *r'(\b\w+\b) \1'* per cercare parole duplicate. Utilizziamo un gruppo di cattura (gruppo 1) per catturare una parola completa *(\b\w+\b)*, seguita da uno spazio e una retrovisione (*\1*) che fa riferimento al contenuto del primo gruppo di cattura.

Esercizi

Esercizio 1: valida password

Dovrai sviluppare una funzione chiamata valida_password che accetti in input una stringa rappresentante una password e restituisca un valore booleano indicando se la password è valida o meno. La funzione dovrà utilizzare espressioni regolari per verificare la validità della password secondo i seguenti criteri:

Criteri per una password valida:

- La password deve avere una lunghezza compresa tra 8 e 16 caratteri.
- La password deve contenere almeno una lettera maiuscola.
- La password deve contenere almeno una lettera minuscola.
- La password deve contenere almeno una cifra.
- La password deve contenere almeno un carattere speciale tra @, #, $, %, ^, &, +, =.
- La password non deve contenere spazi bianchi.

```
import re

def valida_password(password):
    """
    Verifica se una password è valida secondo determinati criteri.

    Argomenti:
    - password: La password da verificare.

    Restituisce:
    - True se la password è valida, False altrimenti.
    """

    # Creazione del pattern per la password
    pattern = re.compile(r"""
        ^               # Ancoraggio all'inizio della stringa
        (?=.*[A-Z])     # Almeno una lettera maiuscola
        (?=.*[a-z])     # Almeno una lettera minuscola
```

```
    (?=.*\d)         # Almeno una cifra
    (?=.*[@#$%^&+=])  # Almeno un carattere speciale tra @#$%^&+=
    (?=\S+$)         # Nessuno spazio bianco consentito
    .{8,16}          # Lunghezza della password tra 8 e 16 caratteri
    $                # Ancoraggio alla fine della stringa
    """, re.VERBOSE)

  # Verifica della corrispondenza con il pattern
  corrispondenza = pattern.search(password)

  # Restituzione del risultato
  if corrispondenza:
      return True
  else:
      return False
```

```
# Esempi di utilizzo
password1 = "Password1@"  # Password valida
password2 = "abc123"      # Password non valida

print(valida_password(password1))  # Output: True
print(valida_password(password2))  # Output: False
```

In questo esempio, la funzione *valida_password* utilizza un oggetto pattern compilato con *re.compile* per verificare se una password soddisfa determinati criteri:

- La password deve iniziare all'inizio della stringa (^) e terminare alla fine della stringa ($) per garantire che non ci siano altri caratteri indesiderati.
- Vengono utilizzati gli ancoraggi (?=.*[A-Z]), (?=.*[a-z]), (?=.*\d), (?=.*[@#$%^&+=]) per garantire che la password contenga almeno una lettera maiuscola, una lettera minuscola, una cifra e un carattere speciale tra @#$%^&+=.

- Il modello di quantificatore {8,16} specifica che la lunghezza della password deve essere compresa tra 8 e 16 caratteri.
- Il flag *re.VERBOSE* consente di formattare l'espressione regolare in modo più leggibile utilizzando la modalità estesa.

Gli esempi di utilizzo mostrano l'applicazione della funzione *valida_password* su due password diverse, restituendo True per una password valida e False per una password non valida.

Esercizio 2: valida email

Sviluppa una funzione chiamata valida_indirizzo_email che prenda in input una stringa rappresentante un indirizzo email e restituisca un valore booleano indicante se l'indirizzo email è valido o meno. La funzione dovrà utilizzare espressioni regolari per verificare la validità dell'indirizzo email.

Requisiti dell'indirizzo email:

- L'indirizzo email deve essere composto da due parti separate dal simbolo '@'.
- La parte del nome utente (prima di '@') può contenere solo caratteri alfanumerici, punti e trattini.
- La parte del dominio (dopo '@') deve essere composta da una serie di blocchi di caratteri alfanumerici, trattini e punti separati da un singolo punto.
- Il dominio di primo livello (es. .com, .org) deve essere composto da almeno due lettere.

```
import re

def valida_indirizzo_email(email):
    """
    Verifica se un indirizzo email è valido secondo gli standard.

    Argomenti:
    - email: L'indirizzo email da verificare.

    Restituisce:
```

```
    - True se l'indirizzo email è valido, False altrimenti.
    """

    # Creazione del pattern per l'indirizzo email
    pattern = re.compile(r"""
        ^                   # Ancoraggio all'inizio della stringa
        [\w\.-]+            # Nome utente: caratteri alfanumerici, punto, trattino
        @                   # Simbolo '@'
        ([\w-]+\.)+         # Dominio: caratteri alfanumerici, trattino, punto (ripetuti almeno una volta)
        [a-zA-Z]{2,}        # Dominio di primo livello: almeno due lettere
        $                   # Ancoraggio alla fine della stringa
        """, re.VERBOSE)

    # Verifica della corrispondenza con il pattern
    corrispondenza = pattern.search(email)

    # Restituzione del risultato
    if corrispondenza:
        return True
    else:
        return False
```

```
# Esempi di utilizzo
email1 = "example@example.com"        # Email valida
email2 = "example@123.456.789"        # Email non valida

print(valida_indirizzo_email(email1))  # Output: True
print(valida_indirizzo_email(email2))  # Output: False
```

In questo esempio, la funzione *valida_indirizzo_email* utilizza un oggetto pattern compilato con *re.compile* per verificare se un indirizzo email soddisfa gli standard:

- L'indirizzo email deve iniziare all'inizio della stringa (^) e terminare alla fine della stringa ($) per garantire che non ci siano

altri caratteri indesiderati.

- Il nome utente può contenere caratteri alfanumerici, il punto e il trattino [\w\.-]+.
- Dopo il simbolo '@', il dominio deve contenere una serie di caratteri alfanumerici, il trattino e il punto ripetuti almeno una volta ([\w-]+\.)+.
- Il dominio di primo livello deve contenere almeno due lettere [a-zA-Z]{2,}.

Gli esempi di utilizzo mostrano l'applicazione della funzione *valida_indirizzo_email* su due indirizzi email diversi, restituendo *True* per un indirizzo valido e *False* per un indirizzo non valido.

Capitolo 8: Le eccezioni

Un'eccezione è un oggetto che rappresenta un errore o una condizione anomala che si verifica durante l'esecuzione di un programma. Quando si verifica un'anomalia, il programma viene interrotto e viene sollevata un'eccezione con un messaggio di errore che ne indica la causa.

Le eccezioni sono utilizzate per gestire gli errori in modo efficace. Quando si scrive del codice, ci si potrebbe imbattere in situazioni in cui si verifica un errore imprevisto, ad esempio, se si cerca di dividere un numero per zero, si cerca di accedere a un elemento di una lista che non esiste o si tenta di aprire un file che non esiste.

Blocco try-except

La gestione delle eccezioni in Python si basa sul blocco *try-except*. All'interno del blocco *try*, inseriremo il codice che potrebbe generare un'eccezione. Nel blocco *except*, definiremo come gestire l'eccezione specifica che ci aspettiamo.

Ad esempio, supponiamo di voler leggere il contenuto di un file specifico, ma non siamo sicuri che il file esista o che siamo autorizzati a leggerlo. Possiamo utilizzare il blocco *try-except* per gestire queste situazioni:

```
try:
    file = open("dati.txt", "r")
    contenuto = file.read()
    print(contenuto)
    file.close()
except FileNotFoundError:
    print("Il file non esiste.")
except PermissionError:
    print("Accesso negato al file.")
```

In questo esempio, il blocco *try* contiene il codice che potrebbe generare un'eccezione. Se il file non esiste, viene generata un'eccezione di tipo *FileNotFoundError*, che viene catturata dal blocco *except* corrispondente.

Allo stesso modo, se l'accesso al file è negato, viene generata un'eccezione di tipo *PermissionError*, che viene catturata dall'altro blocco *except*. All'interno di ogni blocco *except*, possiamo definire il codice da eseguire per gestire l'eccezione.

Blocco finally

Oltre al blocco *try-except*, possiamo anche utilizzare il blocco *finally* per definire del codice che deve essere eseguito indipendentemente dal fatto che un'eccezione sia stata generata o meno. Questo è utile per eseguire azioni di pulizia o rilascio di risorse, come la chiusura di un file, indipendentemente dall'esito delle operazioni precedenti.

```
try:
    file = open("dati.txt", "r")
    contenuto = file.read()
    print(contenuto)
except FileNotFoundError:
    print("Il file non esiste.")
except PermissionError:
    print("Accesso negato al file.")
finally:
    file.close()
```

In questo caso, il blocco *finally* viene eseguito sempre, indipendentemente dall'esito del blocco *try* o dall'eventuale eccezione generata. Pertanto, siamo sicuri che il file verrà chiuso correttamente.

Vedremo nel capitolo sulla gestione dei file, come questa problematica di chiusura forzata del file tramite chiamata delle funzione *close()* possa essere gestita in modo migliore.

L'istruzione raise

L'istruzione *raise* viene utilizzata per generare esplicitamente un'eccezione o un errore all'interno di un programma. Quando viene eseguita, l'istruzione *raise* solleva un'eccezione specificata o un'istanza di una classe eccezione,

interrompendo il flusso normale del programma e consentendo di gestire l'errore in un blocco *try-except* o in un'altra parte del codice.

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Divisione per zero non consentita")
    return a / b


try:
    result = divide(10, 0)
except ZeroDivisionError as error:
    print(error)
```

In questo esempio, la funzione *divide* effettua una divisione tra due numeri. Se il secondo numero *b* è uguale a zero, viene generata un'eccezione *ZeroDivisionError* utilizzando l'istruzione *raise* seguita dal tipo di eccezione e un messaggio di errore personalizzato. Successivamente, l'eccezione viene catturata nel blocco *try-except* e il messaggio di errore viene stampato.

Eccezioni predefinite

Ecco un elenco di alcune delle eccezioni predefinite in *Python*:

- **SyntaxError**: sollevata quando si verifica un errore di sintassi nel codice.
- **IndentationError**: sollevata quando si verifica un errore di indentazione nel codice.
- **TypeError**: sollevata quando si usa un operatore o una funzione con il tipo di dato sbagliato.
- **ValueError**: sollevata quando si passa un argomento ad una funzione che ha il tipo corretto ma un valore inaspettato.
- **IndexError**: sollevata quando si cerca di accedere ad un elemento di una sequenza (lista, tupla, stringa) che non esiste.
- **KeyError**: sollevata quando si cerca di accedere ad una chiave inesistente in un dizionario.
- **ZeroDivisionError**: sollevata quando si tenta di dividere per zero.

- **FileNotFoundError**: sollevata quando si tenta di aprire un file che non esiste.
- **IOError**: sollevata quando si verifica un errore di input/output.
- **ImportError**: sollevata quando si verifica un errore durante l'importazione di un modulo.
- **AttributeError**: sollevata quando si tenta di accedere ad un attributo di un oggetto che non esiste.
- **NameError**: sollevata quando si tenta di utilizzare una variabile non definita.

Questo è solo un elenco parziale delle eccezioni predefinite. Esistono anche molte altre eccezioni, e inoltre è possibile definire le proprie eccezioni personalizzate utilizzando la classe *Exception*.

Eccezioni personalizzate

Le eccezioni personalizzate consentono ai programmatori di creare un'eccezione specifica per un determinato scenario o errore che si verifichi all'interno del loro programma. Questo offre la possibilità di comunicare informazioni più dettagliate riguardo all'errore che si è verificato, semplificando così la diagnosi e la gestione del problema.

Definizione di una classe di eccezioni personalizzata

Per creare una nuova eccezione personalizzata, è necessario definire una nuova classe che erediti dalla classe base delle eccezioni di *Python*, *Exception*.

La nuova classe di eccezioni personalizzata può quindi essere personalizzata per includere attributi e metodi specifici.

```
class ValoreNonValidoError(Exception):
    """Eccezione personalizzata per un valore non valido."""

    def __init__(self, messaggio):
        self.messaggio = messaggio
        super().__init__(self.messaggio)

    def __str__(self):
```

```
        return f"ValoreNonValidoError: {self.messaggio}"
```

```
def controlla_valore(valore):
    if valore < 0:
        raise ValoreNonValidoError("Il valore non può essere negativo.")
```

```
# Esempio di utilizzo
try:
    valore_input = int(input("Inserisci un valore positivo: "))
    controlla_valore(valore_input)
    print("Valore valido!")
except ValoreNonValidoError as e:
    print(f"Errore: {e}")
```

In questo esempio, abbiamo definito una classe chiamata *ValoreNonValidoError* che eredita dalla classe base *Exception*. Questa classe rappresenta un'eccezione personalizzata per un valore non valido.

La classe *ValoreNonValidoError* ha un metodo *__init__* che prende un argomento *messaggio* e lo assegna all'attributo *messaggio*. Utilizziamo anche il metodo *__str__* per fornire una rappresentazione leggibile dell'eccezione quando viene stampata.

Successivamente, abbiamo una funzione chiamata *controlla_valore* che prende un valore come parametro. Se il valore è negativo, viene sollevata un'istanza della classe *ValoreNonValidoError* con un messaggio appropriato.

Nell'esempio di utilizzo, chiediamo all'utente di inserire un valore positivo. Quindi, chiamiamo la funzione *controlla_valore* passando il valore inserito. Se viene sollevata un'eccezione *ValoreNonValidoError*, viene catturata nell'istruzione *except* e viene stampato un messaggio di errore personalizzato.

Conclusione

L'utilizzo di eccezioni personalizzate offre diversi vantaggi nell'ambito della gestione degli errori in un programma:

- **Comunicazione dettagliata**: le eccezioni personalizzate consentono di fornire informazioni specifiche sull'errore che si è verificato, aiutando nella diagnosi e nella risoluzione del problema.
- **Struttura gerarchica**: le eccezioni personalizzate possono essere organizzate in una gerarchia, consentendo la gestione differenziata degli errori a vari livelli di astrazione nel codice.
- **Chiarezza del codice**: l'utilizzo di eccezioni personalizzate rende il codice più leggibile e comprensibile, in quanto evidenzia gli errori specifici che possono verificarsi e le relative azioni di gestione.
- **Riutilizzo del codice**: le eccezioni personalizzate possono essere riutilizzate in più parti del programma, consentendo una gestione coerente degli errori in tutto il codice.

Esercizi

Esercizio 1: conto bancario

In questo esercizio, creeremo una classe BankAccount che rappresenta un conto bancario. La classe avrà un saldo iniziale e metodi per depositare e prelevare denaro. Verranno gestite eccezioni come fondi insufficienti, importi di deposito o prelievo non validi e eccezioni personalizzate per limiti di saldo massimo.

```
Class SaldoInsufficienteError(Exception):
    “””Eccezione personalizzata per fondi insufficienti sul conto.”””

    Def __init__(self, saldo, 1mport):
        self.saldo = saldo
        self.importo = importo

    def __str__(self):
        return f”Saldo insufficiente! Saldo disponibile: {self.saldo}, Importo richiesto: {self.importo}”
```

```
class ImportoNonValidoError(Exception):
    “””Eccezione personalizzata per importo di deposito o prelievo non valido.”””
```

```
  Def __init__(self, importo):
     self.importo = importo

  def __str__(self):
     return f"Importo non valido! Importo richiesto: {self.importo}"
```

```
class BankAccount:
  """
  Classe che rappresenta un conto bancario.
  """

  def __init__(self, saldo_iniziale=0):
     self.saldo = saldo_iniziale

  def deposito(self, importo):
     """
     Deposita l'importo specificato sul conto.
     Solleva ImportoNonValidoError se l'importo non è positivo.
     """

     if importo <= 0:
        raise ImportoNonValidoError(importo)

     self.saldo += importo
     print(f"Depositati {importo}. Nuovo saldo: {self.saldo}")

  def prelievo(self, importo):
     """
     Preleva l'importo specificato dal conto.
     Solleva ImportoNonValidoError se l'importo non è positivo.
     Solleva SaldoInsufficienteError se il saldo è inferiore all'importo richiesto.
     """

     if importo <= 0:
```

```
            raise ImportoNonValidoError(importo)

        if importo > self.saldo:
            raise SaldoInsufficienteError(self.saldo, importo)

        self.saldo -= importo
        print(f"Prelievo di {importo}. Nuovo saldo: {self.saldo}")
```

```
# Test del programma
conto = BankAccount(100)

try:
    conto.deposito(-50)
except ImportoNonValidoError as e:
    print€

try:
    conto.prelievo(200)
except SaldoInsufficienteError as e:
    print€

try:
    conto.prelievo(50)
except SaldoInsufficienteError as e:
    print€
except ImportoNonValidoError as e:
    print(e)
```

- Vengono definiti due tipi di eccezioni personalizzate, *SaldoInsufficienteError* e *ImportoNonValidoError*, che ereditano dalla classe base *Exception*. Queste eccezioni personalizzate includono informazioni specifiche sull'errore e vengono utilizzate per gestire situazioni particolari.

- Viene definita la classe *BankAccount* che rappresenta un conto bancario. Il saldo iniziale viene impostato a 0 se non viene fornito un valore iniziale.
- Il metodo *deposito* consente di depositare un importo specificato sul conto. Se l'importo non è positivo, viene sollevata un'eccezione *ImportoNonValidoError*. In caso contrario, l'importo viene aggiunto al saldo e viene stampato il nuovo saldo.
- Il metodo *prelievo* consente di prelevare un importo specificato dal conto. Se l'importo non è positivo, viene sollevata un'eccezione *ImportoNonValidoError*. Se il saldo è inferiore all'importo richiesto, viene sollevata un'eccezione *SaldoInsufficienteError*. In caso contrario, l'importo viene sottratto dal saldo e viene stampato il nuovo saldo.
- Vengono creati due oggetti conto della classe *BankAccount*, uno con un saldo iniziale di 100.
- Viene utilizzato il blocco *try-except* per gestire le eccezioni sollevate durante l'esecuzione dei metodi deposito e prelievo.
- Nel primo blocco *try-except*, viene chiamato il metodo *deposito* con un importo negativo. Questo solleverà un'eccezione *ImportoNonValidoError* e il messaggio di errore appropriato verrà stampato.
- Nel secondo blocco *try-except*, viene chiamato il metodo *prelievo* con un importo maggiore del saldo disponibile. Questo solleverà un'eccezione *SaldoInsufficienteError* e il messaggio di errore appropriato verrà stampato.
- Nel terzo blocco *try-except*, viene chiamato il metodo *prelievo* con un importo valido. Questa operazione dovrebbe essere eseguita correttamente senza sollevare alcuna eccezione.
- Infine, viene gestita l'eccezione *ImportoNonValidoError* nel caso in cui venga sollevata accidentalmente durante il prelievo.

Esegui il codice per vedere l'output corretto e sperimentare con diversi scenari. Assicurati di leggere attentamente le descrizioni delle eccezioni personalizzate per capire quali informazioni vengono visualizzate negli errori.

Esercizio 2: conta parole

In questo esercizio, creeremo una funzione chiamata conta_parole che accetta il nome di un file come argomento e restituisce il numero di parole presenti nel file. Verranno gestite diverse eccezioni come file non trovato, errori di lettura del file e altre eccezioni impreviste.

```
def conta_parole(nome_file):
    """
    Conta il numero di parole presenti nel file specificato.
    Solleva FileNotFoundError se il file non viene trovato.
    Solleva IOError se si verifica un errore di lettura del file.
    """

    try:
        with open(nome_file, 'r') as file:
            contenuto = file.read()
            parole = contenuto.split()
            numero_parole = len(parole)
            return numero_parole

    except FileNotFoundError:
        raise FileNotFoundError("Errore: File non trovato!")
    except IOError:
        raise IOError("Errore: Si è verificato un errore di lettura del file!")
    except Exception as e:
        raise Exception("Errore imprevisto durante l'elaborazione del file:", str(e))
```

```
# Test del programma
try:
    nome_file = input("Inserisci il nome del file: ")
    numero_parole = conta_parole(nome_file)
    print("Il numero di parole nel file è:", numero_parole)

except FileNotFoundError as e:
    print(e)
```

```
except IOError as e:
    print(e)
except Exception as e:
    print(e)
```

- La funzione *conta_parole* viene definita con un parametro *nome_file*.
- All'interno della funzione, viene utilizzato il blocco *try-except* per gestire le eccezioni potenziali.
- Viene aperto il file specificato in modalità di lettura utilizzando il costrutto *with open* (vedi Capitolo 8: Gestione dei file) per garantire la chiusura automatica del file.
- Il contenuto del file viene letto utilizzando il metodo *read()* e assegnato alla variabile contenuto.
- Il contenuto viene suddiviso in parole utilizzando il metodo *split()* e il risultato viene assegnato alla lista parole.
- La lunghezza della lista parole viene calcolata utilizzando la funzione *len()* e il risultato viene assegnato alla variabile *numero_parole*.
- Se il file non viene trovato, viene sollevata un'eccezione *FileNotFoundError* con il messaggio di errore appropriato.
- Se si verifica un errore di lettura del file, viene sollevata un'eccezione *IOError* con il messaggio di errore appropriato.
- Se si verifica un'altra eccezione imprevista, viene sollevata un'eccezione generica *Exception* con il messaggio di errore appropriato.
- Nel blocco *try-except* esterno, viene chiesto all'utente di inserire il nome del file da elaborare.
- La funzione *conta_parole* viene chiamata con il nome del file e il numero di parole viene stampato.

Esegui il codice per vedere l'output corretto e sperimentare con diversi file. Assicurati di gestire correttamente le eccezioni per evitare errori imprevisti durante l'elaborazione del file.

Capitolo 9: Gestione dei file

La lettura e la scrittura di file sono operazioni comuni nella programmazione. *Python* offre una serie di strumenti e metodi per manipolare file in modo semplice ed efficiente. In questo capitolo, esploreremo come leggere e scrivere dati da/su file, aprendo così le porte alla gestione di dati persistenti.

Apertura di un file

Prima di poter leggere o scrivere dati su un file, è necessario aprirlo. *Python* fornisce la funzione built-in *open()* per aprire un file. Questa funzione richiede il percorso del file e la modalità di apertura desiderata (lettura, scrittura, append o altre opzioni).

Ad esempio, per aprire un file di testo chiamato "dati.txt" in modalità di lettura, possiamo fare quanto segue:

```
file = open("dati.txt", "r")
```

Una volta aperto, il file sarà associato a un oggetto *file*, che possiamo utilizzare per eseguire operazioni di lettura o scrittura.

Lettura dei dati da file

Dopo aver aperto un file in modalità di lettura, possiamo leggere i dati contenuti in esso. Esistono diversi metodi per leggere i dati, tra cui:

- **read()**: legge tutto il contenuto del file come una singola stringa.
- **readline()**: legge una singola riga dal file.
- **readlines()**: legge tutte le righe del file e le restituisce come una lista di stringhe.

Ad esempio, supponiamo che il file "dati.txt" contenga le seguenti righe di testo:

```
Prima riga
Seconda riga
Terza riga
```

Possiamo leggere il contenuto del file nel seguente modo:

```
file = open("dati.txt", "r")
contenuto = file.read()
print(contenuto)
file.close()
```

Scrittura dei dati su un file

Per scrivere dati su un file, è necessario aprire il file in modalità di scrittura o append (aggiunta). La modalità di scrittura "w" sovrascrive il contenuto esistente nel file, mentre la modalità di append "a" aggiunge i dati alla fine del file senza rimuovere il contenuto esistente.

Ad esempio, supponiamo di voler scrivere alcune righe di testo nel file "output.txt". Possiamo farlo nel seguente modo:

```
file = open("output.txt", "w")
file.write("Prima riga\n")
file.write("Seconda riga\n")
file.write("Terza riga\n")
file.close()
```

Chiusura di un file

Dopo aver finito di leggere o scrivere dati su un file, è importante chiuderlo utilizzando il metodo *close()* dell'oggetto *file*.

La chiusura del file rilascia le risorse di sistema associate ad esso e garantisce che tutti i dati siano correttamente scritti o letti.

Tuttavia, per semplificare questa operazione e assicurarsi che il file venga chiuso correttamente anche in caso di eccezioni, Python fornisce l'operatore {with()}. Quest'ultimo crea un contesto in cui il file viene aperto automaticamente all'interno del blocco di istruzioni ed è garantito che verrà chiuso correttamente una volta terminato il blocco.

Ecco un esempio di come utilizzare l'operatore {with} per leggere il contenuto di un file:

```
with open("dati.txt", "r") as file:
```

```
contenuto = file.read()
print(contenuto)
```

In questo esempio, il file viene aperto automaticamente all'interno del blocco *with* e assegnato all'oggetto *file*. Possiamo quindi leggerne il contenuto come visto in precedenza. Alla fine del blocco, il file viene automaticamente chiuso, indipendentemente dal fatto che il codice sia stato eseguito con successo o abbia generato un'eccezione.

L'uso dell'operatore *with* rende il codice più pulito e sicuro, eliminando la necessità di chiamare esplicitamente il metodo *close()*.

Lettura e scrittura di file CSV

I file CSV (Comma-Separated Values) sono un formato comune per archiviare e scambiare dati tabulari, dove i valori di ciascuna riga sono separati da virgole o altri delimitatori. In questo capitolo, esploreremo come leggere e scrivere file CSV utilizzando il modulo *csv*.

Lettura di un file CSV

La lettura di un file CSV in *Python* richiede l'importazione del modulo *csv* e l'utilizzo di alcune funzioni specifiche.

```
import csv

# Apertura del file CSV in modalità lettura
with open('dati.csv', 'r') as file:
    # Creazione dell'oggetto reader CSV
    reader = csv.reader(file)

    # Iterazione sulle righe del file CSV
    for riga in reader:
        # Processa la riga
        print(riga)
```

Scrittura su un file CSV

La scrittura su un file CSV è simile alla lettura. È sempre necessario importare il modulo *csv* e utilizzare le funzioni appropriate.

```
import csv

# Dati da scrivere nel file CSV
dati = [
    ['Nome', 'Cognome', 'Età'],
    ['Mario', 'Rossi', 25],
    ['Laura', 'Verdi', 30]
]

# Apertura del file CSV in modalità scrittura
with open('output.csv', 'w', newline='') as file:
    # Creazione dell'oggetto writer CSV
    writer = csv.writer(file)

    # Scrittura dei dati nel file CSV
    writer.writerows(dati)
```

Manipolazione dei dati CSV

Una volta letti o scritti i dati da un file CSV, è possibile manipolarli secondo le proprie necessità. Ad esempio, è possibile accedere ai valori specifici di una riga o effettuare calcoli sui dati.

```
import csv

# Lettura di un file CSV
with open('dati.csv', 'r') as file:
    reader = csv.reader(file)

    # Iterazione sulle righe del file CSV
    for riga in reader:
        # Accesso ai valori specifici di una riga
```

```
    nome = riga[0]      # accedo alla prima colonna (indice 0)
    cognome = riga[1]   # accedo alla seconda colonna (indice 1)
    eta = int(riga[2])  # accedo alla terza colonna (indice 2)

    # Esegui calcoli o operazioni sui dati
    # ...
```

Nell'esempio sopra, dopo aver letto un file, è possibile accedere ai valori specifici di ciascuna riga utilizzando l'indice corrispondente.

Considerazioni sulla manipolazione dei file CSV

Quando si lavora con file CSV, è importante tenere presente alcuni aspetti:

- Assicurarsi di gestire correttamente il delimitatore utilizzato nel file CSV. Per impostazione predefinita, viene utilizzata la virgola come delimitatore, ma è possibile specificare un delimitatore diverso utilizzando l'argomento {delimiter} delle funzioni {reader()} e {writer()}.

```
import csv

nome_file = "dati.csv"
delimitatore = ";"  # specifica il delimitatore utilizzato nel file

with open(nome_file, "r") as file_csv:
    lettore_csv = csv.reader(file_csv, delimiter=delimitatore)
    for riga in lettore_csv:
        print(riga)
```

- Prestare attenzione alla presenza di caratteri speciali come virgole o virgolette all'interno dei valori del file. Potrebbero causare problemi nella lettura o scrittura corretta dei dati. È possibile gestire tali caratteri utilizzando l'argomento *quoting* nelle funzioni *reader()* e *writer()*.

  Il *quoting* si riferisce all'uso dei caratteri di citazione (tipicamente virgolette o apici) per racchiudere i valori dei campi.

Quando si lavora con dati CSV, talvolta i valori dei campi possono contenere caratteri speciali, come virgole, punti e virgole o caratteri di nuova riga. Per evitare che questi caratteri speciali vengano interpretati come delimitatori di campo, è possibile racchiudere i valori tra i caratteri di citazione. Questo è particolarmente utile quando un valore di campo contiene il delimitatore stesso o spazi bianchi all'inizio o alla fine del valore.

Python offre diverse opzioni di citazione attraverso il modulo {csv}. Ecco i valori possibili per il parametro *quoting*:

- **csv.QUOTE_ALL**: citazione di tutti i campi, indipendentemente dalla presenza di spazi o delimitatori al loro interno. Ad esempio, "valore".
- **csv.QUOTE_MINIMAL**: citazione solo per i campi che contengono spazi o delimitatori. Ad esempio, "valore con spazio", ma non per "valore".
- **csv.QUOTE_NONNUMERIC**: citazione per tutti i campi tranne quelli numerici. Ad esempio, "valore" ma non per 25.
- **csv.QUOTE_NONE**: nessuna citazione, i campi vengono considerati come stringhe non citate.

```
import csv

nome_file = "dati.csv"
delimitatore = ","
quoting = csv.QUOTE_ALL  # o csv.QUOTE_MINIMAL,
                         # csv.QUOTE_NONNUMERIC, csv.QUOTE_NONE

with open(nome_file, "r") as file_csv:
    lettore_csv = csv.reader(file_csv, delimiter=delimitatore, \
                             quoting=quoting)
    for riga in lettore_csv:
        print(riga)
```

❖ Verificare la presenza di righe di intestazione nel file CSV. Se il file include una riga di intestazione con i nomi delle colonne, è possibile

saltarla durante la lettura utilizzando la funzione *next()* sull'oggetto reader.

In generale, la funzione *next()* viene utilizzata per ottenere la prossima riga.

```
import csv

nome_file = "dati.csv"

with open(nome_file, "r") as file_csv:
    lettore_csv = csv.reader(file_csv)

    # Lettura dell'header
    header = next(lettore_csv)
    print("Header:", header)

    # Lettura della prossima riga
    riga = next(lettore_csv)
    print("Prossima riga:", riga)
```

- Assicurarsi di gestire correttamente le eccezioni durante la lettura o la scrittura dei file CSV. Ad esempio, potrebbe essere necessario gestire eccezioni come *FileNotFoundError* quando si apre un file che non esiste o *PermissionError* quando non si dispone dei permessi di scrittura appropriati.

Gestione di file JSON

Il formato JSON (JavaScript Object Notation) è ampiamente utilizzato per archiviare e scambiare dati strutturati. Python fornisce un modulo integrato chiamato *json* che semplifica la lettura e la scrittura di file JSON. In questo capitolo, esploreremo come gestire tali file.

Lettura di un file JSON

La lettura di un file JSON in *Python* richiede l'importazione del modulo *json* e l'utilizzo di alcune funzioni specifiche.

```python
import json

# Apertura del file JSON in modalità lettura
with open('dati.json', 'r') as file:
    # Caricamento dei dati JSON
    dati = json.load(file)

    # Accesso ai dati (assicurati dell'esistenza delle chiavi)
    nome = dati['nome']
    età = dati['età']
    indirizzo = dati['indirizzo']
```

Nell'esempio sopra, il file JSON denominato "dati.json" viene aperto in modalità lettura. Successivamente, i dati in esso contenuti vengono caricati utilizzando la funzione *json.load(),* che converte il contenuto del file JSON in un oggetto *Python* (dizionario). Infine, è possibile accedere ai dati specifici utilizzando le chiavi corrispondenti, assicurandosi che la chiave sia presente all'interno del dizionario

Scrittura su un file JSON

La scrittura su file è simile alla lettura.

```python
import json

# Dati da scrivere nel file JSON
dati = {
    'nome': 'Mario Rossi',
    'età': 30,
    'indirizzo': 'Via Roma 123'
}

# Apertura del file JSON in modalità scrittura
with open('output.json', 'w') as file:
    # Scrittura dei dati JSON
    json.dump(dati, file)
```

Nell'esempio sopra, viene definito un dizionario dati contenente i dati da scrivere nel file JSON. Il file JSON denominato "output.json" viene aperto in modalità scrittura, dopodichè i dati vengono scritti nel file JSON utilizzando la funzione *json.dump()*.

Manipolazione dei dati JSON

Una volta letti o scritti i dati da un file JSON, è possibile manipolarli. Ad esempio, è possibile accedere ai valori specifici attraverso la chiave o aggiungere nuovi dati.

```
import json

# Lettura di un file JSON
with open('dati.json', 'r') as file:
    dati = json.load(file)

    # Accesso ai valori specifici
    nome = dati['nome']
    età = dati['età']

    # Modifica dei dati
    dati['indirizzo'] = 'Via Verdi 456'

# Scrittura su un file JSON
with open('dati_modificati.json', 'w') as file:
    json.dump(dati, file)
```

Nell'esempio sopra, dopo aver letto il file, è possibile accedere ai valori specifici utilizzando le chiavi corrispondenti. È anche possibile modificare i dati, aggiungere nuove chiavi o eliminare chiavi esistenti. Successivamente, i dati modificati possono essere scritti su un nuovo file utilizzando la funzione *json.dump()*.

Considerazioni sulla gestione dei file JSON

Durante la gestione dei file JSON in *Python*, è importante ricordarsi i seguenti aspetti:

- Assicurarsi che il file JSON sia ben formato. Un file JSON non valido potrebbe causare errori durante la lettura o la scrittura dei dati. È possibile verificare la validità di un file JSON utilizzando strumenti online o librerie come *jsonschema* in *Python*.

```
import json
from jsonschema import validate

# Definizione dello schema JSON
schema = {
    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "age": {"type": "integer", "minimum": 0},
        "email": {"type": "string", "format": "email"}
    },
    "required": ["name", "age"]
}

# Oggetto JSON da validare
json_data = '''
{
    "name": "Alice",
    "age": 30,
    "email": "alice@example.com"
}
'''

# Parsing dell'oggetto JSON
data = json.loads(json_data)

# Validazione dell'oggetto JSON rispetto allo schema
```

```
try:
    validate(data, schema)
    print("Validazione riuscita!")
except Exception as e:
    print("Validazione fallita:", e)
```

Assicurati di aver installato il pacchetto *jsonschema* utilizzando il comando *pip install jsonschema* prima di eseguire il codice.

- Prestare attenzione alla struttura dei dati JSON. Assicurarsi di utilizzare le chiavi corrette per accedere ai dati desiderati. È possibile eseguire controlli per verificare la presenza di determinate chiavi o per gestire eccezioni nel caso in cui una chiave non esista.
- Gestire correttamente le eccezioni durante la lettura o la scrittura dei file JSON. Ad esempio, potrebbe essere necessario gestire eccezioni come *FileNotFoundError* quando si apre un file JSON che non esiste o *PermissionError* quando non si dispone dei permessi di scrittura appropriati.
- Assicurarsi di comprendere la struttura dei dati JSON per poterli manipolare correttamente. Può essere utile utilizzare funzioni come *json.loads()* per caricare una stringa JSON in un oggetto *Python* o *json.dumps()* per convertire un oggetto *Python* in una stringa JSON.

Capitolo 10: Il debugging

Il debugging è un processo importante per risolvere errori all'interno del codice. In *Python*, ci sono diverse tecniche e strumenti che possono essere utilizzati per il debugging.

Una delle tecniche più semplici è la stampa di messaggi di debug all'interno del codice, ad esempio utilizzando la funzione *print()*. Questo consente di visualizzare i valori delle variabili e di seguire il flusso del programma. Tuttavia, questa tecnica può diventare inefficiente quando il codice diventa più complesso e si devono seguire molti passaggi.

Un'altra tecnica utile è l'utilizzo del debugger integrato di *Python*. Esso consente di fermare il programma in un determinato punto (detto *breakpoint*) e di eseguire il codice riga per riga, visualizzando il valore delle variabili in ogni passaggio. Per utilizzare il debugger, è necessario importare il modulo *pdb* e chiamare la funzione *pdb.set_trace()* in un punto del codice dove si desidera fermare l'esecuzione. In questo modo, il debugger si attiverà e si potrà iniziare ad eseguire il codice riga per riga.

Ecco un esempio di utilizzo:

```
import pdb

def somma(a, b):
    pdb.set_trace()
    return a + b

x = 2
y = 3
z = somma(x, y)
print(z)
```

In questo esempio, la funzione *somma()* contiene la chiamata alla funzione *pdb.set_trace()*. Quando il programma viene eseguito, si fermerà all'interno della funzione *somma(),* permettendo di controllare il valore delle variabili *a* e *b*.

Da qui, è possibile utilizzare alcuni comandi per controllare l'esecuzione del programma.

Alcuni dei comandi principali sono:

- **n (next)**: esegue la riga corrente e passa alla riga successiva
- **c (continue)**: riprende l'esecuzione del programma fino al prossimo breakpoint
- **s (step)**: esegue la riga corrente e si ferma all'interno di una funzione chiamata da quella riga
- **q (quit)**: interrompe l'esecuzione del programma

È inoltre possibile visualizzare il valore di una variabile utilizzando il comando *p nome_variabile*, dove *nome_variabile* è il nome della variabile di interesse.

Ad esempio, per visualizzare il valore di *a* nel nostro esempio, si può digitare:

```
(pdb) p a
2
```

Una volta terminato il debugging, è possibile uscire dal prompt del debugger digitando *q* (*quit*).

Il modulo *pdb* di *Python* è uno strumento molto potente che può essere utilizzato per identificare e risolvere i problemi nel codice. Tuttavia, può risultare complicato da utilizzare se non si ha esperienza con il debugging. Per questo motivo, ci sono anche altri strumenti più avanzati come *PyCharm* e *Visual Studio Code* che offrono un'interfaccia più user-friendly per il debugging.

Capitolo 11: Generazione delle documentazione del codice

La documentazione del codice svolge un ruolo cruciale nello sviluppo del software, poiché fornisce informazioni dettagliate sulle funzioni, i moduli e le classi implementate, facilitando la comprensione e l'utilizzo del codice da parte degli sviluppatori.

Esistono diversi strumenti e approcci per generarla in modo automatizzato e coerente. In questo capitolo, esploreremo le varie opzioni disponibili.

- **Docstrings**: sono stringhe di testo poste all'inizio di un modulo, di una funzione o di una classe, utilizzate per documentare il codice.

  Esse seguono una convenzione specifica, come ad esempio il formato *reStructuredText* o *Markdown* , e contengono informazioni sullo scopo, i parametri, il valore di ritorno e gli esempi di utilizzo del codice.

  Le *docstring* possono essere facilmente estratte utilizzando la funzione *help()* (vedi Aiuto in linea) di *Python* o possono essere incorporate in documentazione più ampia utilizzando strumenti di generazione della documentazione.

```
"""
Questo modulo fornisce funzioni utili per operazioni
matematiche di base.
"""

def somma(a, b):
    """
    Restituisce la somma di due numeri.

    Args:
        a (float o int): Il primo numero.
        b (float o int): Il secondo numero.

    Returns:
```

```
        float o int: La somma dei due numeri.
    """
    return a + b

def sottrazione(a, b):
    """
    Restituisce la differenza tra due numeri.

    Args:
        a (float o int): Il primo numero.
        b (float o int): Il secondo numero.

    Returns:
        float o int: La differenza tra i due numeri.
    """
    return a - b
```

- **Sphinx**: è uno strumento di generazione della documentazione ampiamente utilizzato nella comunità *Python*.

  Esso utilizza i file sorgenti del codice Python, comprese le *docstring*, per generare documentazione leggibile in diversi formati come HTML, PDF e EPUB.

  Sphinx supporta anche funzionalità avanzate come l'organizzazione gerarchica della documentazione, l'indicizzazione, i collegamenti incrociati e altro ancora.

  Per utilizzarlo è necessario definire un file di configurazione {conf.py} e creare una struttura di directory appropriata per i file sorgente e i file di output della documentazione.

  Si rimanda alla documentazione ufficiale per un maggiore approfondimento https://www.sphinx-doc.org/en/master/

- **Pydoc**: è uno strumento integrato in *Python* che genera la documentazione basata sulle *docstring* del codice.

Può essere utilizzato da linea di comando per generare automaticamente la documentazione per i moduli, le funzioni e le classi specificate.

Pydoc supporta diversi formati di output, come testo semplice, HTML o server web.

È possibile eseguire il comando *pydoc -w <nome_modulo>* per generare un file HTML con la documentazione del modulo specificato.

- **Altri strumenti**: esistono altri strumenti di generazione della documentazione in Python, come *Doxygen*, *MkDocs*, *Read the Docs* e molti altri.

  Questi strumenti offrono funzionalità aggiuntive, come l'integrazione con sistemi di versionamento del codice, la generazione automatica di diagrammi, l'hosting della documentazione online e altro ancora.

È possibile esplorare queste opzioni per trovare lo strumento che meglio si adatta alle esigenze specifiche del progetto e del team di sviluppo.

Best Practice

Oltre agli strumenti di generazione della documentazione, è importante seguire alcune best practice per garantire che la documentazione sia chiara, completa ed efficace:

- **Scrivere docstring informative e concise**: le *docstring* dovrebbero essere scritte in modo chiaro, fornendo una spiegazione completa delle funzioni, dei moduli o delle classi. Evitare l'uso di termini tecnici complessi quando non necessario e fornire esempi di utilizzo quando appropriato.
- **Seguire le convenzioni di stile**: seguire le linee guida di stile del codice *Python*, come quelle definite nel PEP 8, anche per la documentazione. Mantenere una formattazione uniforme e consistente per facilitare la lettura e la comprensione.
- **Aggiornare regolarmente la documentazione**: la documentazione deve essere tenuta aggiornata mano a mano che il codice evolve. Assicurarsi di rivedere e aggiornare le *docstring* ogni volta che si apportano modifiche significative al codice.

- **Utilizzare markup e formattazione adeguata**: utilizzare opportuni markup e formattazione nella documentazione per evidenziare parti importanti del testo, come titoli, elenchi, codice inline o blocchi di codice. Questo renderà la documentazione più leggibile e organizzata.
- **Aggiungere esempi significativi**: fornire esempi di utilizzo del codice nei commenti o nelle *docstring* può aiutare gli sviluppatori a capire come utilizzare correttamente le funzioni o le classi. Gli esempi dovrebbero essere significativi e coprire diversi casi d'uso.
- **Incorporare diagrammi o grafici**: a seconda della complessità del progetto, potrebbe essere utile includere diagrammi o grafici che illustrano il flusso del programma, l'architettura o le dipendenze. Questo può semplificare la comprensione del codice e dei concetti sottostanti.
- **Fornire una panoramica dell'API**: se il progetto ha un'API complessa con numerose funzioni o classi, fornire una panoramica dell'API nella documentazione può aiutare gli sviluppatori a navigare rapidamente attraverso le diverse parti del codice e trovare ciò di cui hanno bisogno.
- **Incorporare link e riferimenti**: quando possibile, fornire link o riferimenti a documentazione esterna, tutorial o risorse aggiuntive che possono essere utili per gli sviluppatori che consultano la documentazione.
- **Sottoporre la documentazione a revisione**: assicurarsi che la documentazione sia stata sottoposta a revisione da parte di altri membri del team per garantire l'accuratezza e la chiarezza delle informazioni fornite.

Seguendo queste best practice, è possibile creare una documentazione completa e di qualità che agevoli l'utilizzo e la manutenzione del codice. Una buona documentazione non solo aiuta gli sviluppatori a comprendere il funzionamento del codice, ma può anche favorire la collaborazione all'interno del team e semplificare il processo di onboarding per i nuovi membri del team.

Capitolo 11: I Design Pattern

La storia dei design pattern è legata allo sviluppo dell'ingegneria del software. Negli anni '70, mentre la disciplina dell'ingegneria del software stava ancora prendendo forma, i programmatori affrontavano sfide sempre più complesse nel progettare e sviluppare software. Durante questo periodo, emersero le prime idee riguardanti la riutilizzabilità del software e l'importanza di una buona progettazione.

Negli anni '80, i ricercatori e gli sviluppatori iniziarono a identificare e documentare soluzioni comuni a problemi ricorrenti nello sviluppo. Nel 1987, Christopher Alexander e Sara Ishikawa pubblicarono il libro "A Pattern Language", che introdusse il concetto di *pattern* come soluzioni generiche e riusabili per problemi di progettazione architettonica.

Successivamente, negli anni '90, il concetto di design pattern si diffuse nel campo dello sviluppo del software grazie al lavoro di Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, noti come *Gang of Four* (GoF). Nel 1994, pubblicarono il libro Design Patterns: Elements of Reusable Object-Oriented Software, che divenne una pietra miliare nell'ingegneria del software. Questo libro presentava 23 design pattern riconosciuti come soluzioni generiche per problemi di progettazione software.

Dopo la pubblicazione del libro di *GoF*, i design pattern divennero sempre più popolari e diffusi nella comunità degli sviluppatori. I programmatori iniziarono a riconoscere i vantaggi di utilizzare soluzioni consolidate per problemi comuni e a implementare i design pattern nei loro progetti. Vennero sviluppati anche altri libri e risorse che approfondivano i design pattern specifici per linguaggi di programmazione o contesti particolari.

Cosa sono i design pattern?

I design pattern, o "pattern di progettazione", sono soluzioni generali e riusabili per problemi comuni nell'ambito dello sviluppo del software. Sono considerati delle "migliori pratiche" che gli sviluppatori possono adottare per risolvere determinati problemi architetturali o di progettazione.

Un design pattern descrive un problema specifico che si verifica frequentemente nel contesto dello sviluppo del software e propone una soluzione generale che può essere adattata e applicata a diverse situazioni. Questi pattern sono stati identificati e documentati da esperti nel campo dell'ingegneria del software per fornire agli sviluppatori un modo comprovato e consolidato di affrontare problemi ricorrenti.

I design pattern non costituiscono codice specifico, ma piuttosto concetti o modelli che possono essere applicati in diversi linguaggi di programmazione; forniscono un modo per organizzare e strutturare il codice in modo efficiente, promuovendo la modularità, la flessibilità e la manutenibilità del software.

Ecco l'elenco dei 23 design pattern presentati nel libro Design Patterns: Elements of Reusable Object-Oriented Software:

- **Design pattern creazionali**: sono utilizzati per fornire un'astrazione per la creazione di oggetti complessi in modo flessibile e riutilizzabile. Consentono di nascondere i dettagli di implementazione specifici di una classe e fornire un'interfaccia comune per la creazione di oggetti. Essi aiutano a promuovere il basso accoppiamento, l'incapsulamento e la modularità del codice.

    - Singleton
    - Factory Method
    - Abstract Factory
    - Builder
    - Prototype

- **Design pattern strutturali**: sono utilizzati per definire le relazioni tra gli oggetti e fornire soluzioni per organizzare le classi in modo efficiente. Consentono di creare strutture complesse combinando oggetti in modo flessibile e riutilizzabile. Essi favoriscono la modularità, l'estensibilità e la manutenibilità del codice.

    - Adapter
    - Bridge
    - Composite
    - Decorator

- Facade
- Flyweight
- Proxy

❖ **Design pattern comportamentali**: sono utilizzati per gestire il comportamento e la comunicazione tra gli oggetti in un sistema. Forniscono soluzioni per l'organizzazione delle interazioni tra gli oggetti, la gestione degli algoritmi e la definizione di flussi di lavoro complessi.

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Ricorda che questi sono i design pattern proposti nel libro, ma nel corso del tempo ne sono stati identificati e documentati molti altri.

A tal proposito, ci sono anche design pattern architetturali, i quali riguardano l'organizzazione generale di un sistema software a un livello più ampio, mentre i design pattern riportati sopra si concentrano principalmente sulle soluzioni di progettazione a livello di classe e oggetto.

Alcuni esempi di design pattern architetturali includono:

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)
- Repository Pattern
- Dependency Injection (DI)

Inoltre, tieni ben presente che la scelta e l'uso dei design pattern dipendono dal contesto e dai requisiti specifici del progetto.

È importante sottolineare che, data la complessità e l'importanza dei design pattern, non entrerò nel dettaglio e nella spiegazione completa di ciascun pattern in questo capitolo. Se sei interessato a saperne di più su un particolare pattern o desideri una comprensione approfondita, ti consiglio di consultare il libro, che fornisce spiegazioni dettagliate, esempi di implementazione. Tuttavia, vedremo una possibile implementazione di quelli che sono a mio avviso alcuni dei più utili.

Perché è importante conoscere i design pattern?

Ecco alcune motivazioni convincenti per conoscere i design pattern:

- **Soluzioni comprovate**: i design pattern rappresentano soluzioni comprovate e consolidate per problemi comuni nell'ambito dello sviluppo del software. Essi sono il risultato di anni di esperienza e di studio da parte di esperti nel campo dell'ingegneria del software. Conoscere i design pattern ti consente di accedere a soluzioni già testate e ottimizzate, riducendo la possibilità di commettere errori e migliorando la qualità del tuo codice.
- **Miglioramento delle competenze di progettazione**: i design pattern promuovono una progettazione del software più strutturata, modulare e flessibile. Studiare i design pattern ti aiuta ad acquisire una maggiore comprensione delle best practice di progettazione e delle relazioni tra le componenti di un sistema. Questo ti renderà un programmatore più competente e ti permetterà di scrivere codice di qualità superiore.
- **Riusabilità del codice**: i design pattern favoriscono la creazione di codice riusabile. Poiché i pattern sono soluzioni generiche e modulari, possono essere applicati a diversi contesti e progetti. Questo significa che puoi utilizzare e adattare i design pattern esistenti per risolvere problemi simili in futuro, risparmiando tempo e sforzo nella progettazione e nello sviluppo.
- **Comunicazione efficace tra sviluppatori**: i design pattern forniscono un linguaggio comune tra gli sviluppatori. Quando tutti i membri di un team di sviluppo li conoscono, possono comunicare in modo più efficiente e chiaro riguardo alla progettazione del software. I pattern costituiscono uno "strumento di comunicazione"

che facilita lo scambio di idee e la comprensione reciproca, migliorando la collaborazione all'interno del team.

- **Miglioramento della manutenibilità del software**: i design pattern aiutano a rendere il software più mantenibile nel lungo termine. Poiché promuovono una progettazione modulare e flessibile, i pattern rendono più semplice apportare modifiche e aggiunte al codice senza causare effetti indesiderati su altre parti del sistema. Questo è particolarmente vantaggioso quando si lavora su progetti complessi e in continua evoluzione.
- **Preparazione per colloqui di lavoro**: molte aziende, durante i colloqui di lavoro per ruoli di sviluppo del software, pongono domande sulla conoscenza dei design pattern. Essere in grado di identificare e applicare i design pattern correttamente può fare la differenza nel processo di selezione e mostrare che sei un candidato ben preparato e competente.

Esempi di implementazione in Python

Singleton

Il *Singleton* ha lo scopo di garantire che una classe abbia una sola istanza e fornirne un punto di accesso globale. Le principali ragioni per utilizzare il pattern *Singleton* sono:

- **Controllo sull'istanza**: il Singleton permette di controllare l'istanza di una classe in modo che ne venga creata una sola durante l'intero ciclo di vita dell'applicazione. Questo può essere utile in situazioni in cui è necessario garantire l'accesso a un'istanza unica, ad esempio per gestire una risorsa condivisa o mantenere uno stato globale.
- **Punto di accesso globale**: il *Singleton* fornisce un punto di accesso globale all'istanza unica, consentendo a diversi componenti dell'applicazione di accedere e utilizzare l'istanza in modo coerente senza doverla passare esplicitamente tra di loro.
- **Condivisione di risorse**: il *Singleton* può essere utilizzato per condividere una risorsa costosa inizializzata una volta sola, come ad esempio un pool di connessioni al database o un gestore di file.

- **Controllo dei thread**: utilizzando il pattern *Singleton*, è possibile garantire che un'istanza sia accessibile e condivisa in modo sicuro da più *thread*. Questo aiuta a prevenire condizioni di *race condition* (vedi Introduzione alla concorrenza e al parallelismo) e assicura un accesso concorrente controllato alla risorsa.

È importante notare che il *Singleton* può essere considerato un pattern potente, ma può anche creare dipendenze nascoste e rendere il codice più difficile da testare. Pertanto, è consigliabile utilizzarlo con cautela e valutare attentamente la necessità di un'unica istanza e l'impatto che può avere sul design complessivo del sistema.

```
class Singleton:
    __instance = None

    @staticmethod
    def get_instance():
        if Singleton.__instance is None:
            Singleton()
        return Singleton.__instance

    def __init__(self):
        if Singleton.__instance is not None:
            raise Exception('Questo è un Singleton! Usa il metodo \
            get_instance() per ottenerne l\'istanza.')
        else:
            Singleton.__instance = self
```

```
# Utilizzo del Singleton
singleton_instance1 = Singleton.get_instance()
singleton_instance2 = Singleton.get_instance()

# Verifica se le due istanze sono identiche
print(singleton_instance1 is singleton_instance2)  # Output: True
```

In questo esempio, la classe *Singleton* implementa il pattern *Singleton*. L'attributo *__instance* tiene traccia dell'unica istanza della classe. Il metodo *get_instance()* viene utilizzato per ottenere l'istanza del Singleton: se non esiste, viene creata e assegnata a *__instance*, mentre se esiste già, viene restituita direttamente.

Nel codice di utilizzo, viene richiamato il metodo *get_instance()* due volte. Poiché il *Singleton* permette l'esistenza di una sola istanza, le due variabili *singleton_instance1* e *singleton_instance2* puntano alla stessa istanza. Pertanto, il confronto *singleton_instance1 is singleton_instance2* restituirà *True*.

Questa implementazione del *Singleton* garantisce che una sola istanza della classe venga creata durante l'esecuzione dell'applicazione.

Factory method

Il *Factory Pattern*, noto anche come *Factory Method*, è un design pattern creazionale che ha lo scopo di fornire un'interfaccia per la creazione di oggetti, ma lascia alle sottoclassi la responsabilità di decidere quale classe concreta istanziare. In sostanza, il *Factory Pattern* mira a separare la logica di creazione degli oggetti dal loro utilizzo.

Ecco alcune delle principali ragioni per cui si utilizza il *Factory Pattern*:

- **Incapsulamento della logica di creazione**: il *Factory Pattern* consente di incapsulare la logica di creazione degli oggetti in una classe separata chiamata *Factory*. Questo significa che il codice client non deve essere a conoscenza dei dettagli specifici di come vengono creati gli oggetti. La logica di creazione viene nascosta dietro l'interfaccia del *Factory*, favorendo l'incapsulamento e l'astrazione.
- **Astrazione della creazione degli oggetti**: il *Factory Pattern* permette di definire un'interfaccia comune per la creazione di oggetti, indipendentemente dalle loro implementazioni concrete. Ciò consente di scrivere codice client che dipende dall'interfaccia astratta del *Factory* invece che dalle classi concrete. In questo modo, è possibile sostituire facilmente le implementazioni concrete degli oggetti senza dover modificare il codice client.

- **Flessibilità e estensibilità**: utilizzando il *Factory Pattern*, è possibile estendere e aggiungere nuove varianti di oggetti senza modificare il codice client esistente. Basta creare una nuova sottoclasse del *Factory* e implementare il metodo di creazione per la nuova variante di oggetto desiderata. In questo modo, il codice client può utilizzare il *Factory* per creare oggetti diversi senza conoscere le dettagliate implementazioni specifiche.
- **Decoupling (disaccoppiamento)**: il *Factory Pattern* favorisce il disaccoppiamento tra il codice client e le classi concrete. Il client richiede la creazione di un oggetto al *Factory* utilizzando l'interfaccia astratta, senza dover conoscere la classe concreta sottostante. Questo rende il client indipendente dalle specifiche implementazioni degli oggetti e semplifica il riuso del codice.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Woof!"

class Cat(Animal):
    def sound(self):
        return "Meow!"

class AnimalFactory:
    def create_animal(self, animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
            raise ValueError("Invalid animal type.")
```

```
# Utilizzo del Factory per creare gli oggetti Animal

animal_factory = AnimalFactory()

dog = animal_factory.create_animal("dog")
print(dog.sound())  # Output: "Woof!"

cat = animal_factory.create_animal("cat")
print(cat.sound())  # Output: "Meow!"
```

In questo esempio, abbiamo un'interfaccia comune chiamata *Animal* che definisce il metodo astratto *sound()*. Successivamente, abbiamo due sottoclassi concrete *Dog* e *Cat* che implementano l'interfaccia *Animal* e forniscono la propria implementazione del metodo *sound()*.

La classe *AnimalFactory* è la fabbrica che si occupa dell'istanziazione degli oggetti *Animal*. Ha un metodo *create_animal(animal_type)* che accetta il tipo di animale desiderato come parametro. In base al tipo specificato, la fabbrica restituisce un'istanza della classe corrispondente (*Dog* o *Cat*).

Nell'esempio, viene creato un oggetto *AnimalFactory* e successivamente vengono utilizzati i metodi della fabbrica per creare oggetti *Dog* e *Cat*. Infine, viene chiamato il metodo *sound()* sugli oggetti creati per ottenere il suono corrispondente emesso da ciascun animale.

Adapter

Il pattern *Adapter*, anche conosciuto come *Wrapper*, è un design pattern strutturale che consente di convertire l'interfaccia di una classe in un'altra interfaccia che il client si aspetta. Questo consente a classi con interfacce diverse di lavorare insieme senza modificare il codice sorgente originale.

Il pattern *Adapter* è utile in diverse situazioni:

- **Adattamento di interfacce**: quando si ha una classe esistente con un'interfaccia incompatibile con quella richiesta dal client, *l'Adapter* consente di fornire un'interfaccia compatibile che può essere utilizzata dal client senza modificarne il codice.

- **Riuso di classi esistenti**: *l'Adapte*r consente di riutilizzare classi esistenti che hanno funzionalità desiderabili ma un'interfaccia incompatibile con il sistema in cui vengono utilizzate. Invece di riscrivere il codice o duplicare la logica, *l'Adapter* fornisce una conversione dell'interfaccia per consentire l'utilizzo della classe esistente.
- **Integrazione di sistemi esterni**: *l'Adapter* può essere utilizzato per integrare sistemi esterni o librerie con un sistema esistente. Se si desidera utilizzare un componente esterno che ha un'interfaccia diversa da quella richiesta dal sistema, *l'Adapter* fornisce un adattamento tra le due interfacce, consentendo una comunicazione senza problemi tra i componenti.

```
# Classe Target (interfaccia desiderata)
class PaymentProcessor:
    def process_payment(self, amount):
        pass

# Classe Adaptee (classe esistente da adattare)
class PayPalPaymentService:
    def pay_with_paypal(self, amount):
        print(f"Pagamento di {amount}€ effettuato con PayPal.")

# Adapter (adatta l'interfaccia dell'Adaptee al Target)
class PayPalAdapter(PaymentProcessor):
    def __init__(self, paypal_service):
        self.paypal_service = paypal_service

    def process_payment(self, amount):
        self.paypal_service.pay_with_paypal(amount)

# Utilizzo
paypal_service = PayPalPaymentService()
adapter = PayPalAdapter(paypal_service)
adapter.process_payment(50)
```

In questo esempio, immaginiamo di avere una classe *PaymentProcessor* che rappresenta un'interfaccia comune per l'elaborazione di pagamenti. Tuttavia, abbiamo anche una classe esistente *PayPalPaymentService* che gestisce i pagamenti tramite PayPal.

L'obiettivo è utilizzare la classe *PaymentProcessor* per elaborare pagamenti utilizzando il servizio PayPal, utilizzando la classe esistente *PayPalPaymentService*.

La classe *PayPalAdapter* si comporta da *Adapter*. Prende un oggetto *PayPalPaymentService* come argomento nel suo costruttore e implementa il metodo *process_payment()* dell'interfaccia *PaymentProcessor*. All'interno di questo metodo, viene chiamato il metodo *pay_with_paypal()* dell'oggetto *PayPalPaymentService*.

Nell'utilizzo, viene istanziato un oggetto *PayPalPaymentService* e viene creato un adattatore *PayPalAdapter* che prende l'oggetto *PayPalPaymentService* come argomento. Successivamente, viene chiamato il metodo *process_payment()* sull'adattatore, che a sua volta richiama il metodo *pay_with_paypal()* dell'oggetto *PayPalPaymentService*. In questo modo, possiamo utilizzare la classe *PaymentProcessor* per elaborare pagamenti utilizzando il servizio PayPal tramite l'adattatore.

Questo esempio illustra come l'*Adapter* Pattern possa essere utilizzato per integrare servizi esterni o librerie con un'interfaccia comune, consentendo a diverse parti del sistema di comunicare tra loro in modo uniforme, indipendentemente dalle loro specifiche implementazioni.

Observer

Il pattern *Observer* è un design pattern comportamentale che consente di stabilire una relazione di dipendenza uno-a-molti tra oggetti, in modo che quando l'oggetto principale (noto come soggetto osservato) cambia il suo stato, tutti gli oggetti dipendenti (noti come osservatori) vengono automaticamente notificati e aggiornati.

Il pattern *Observer* è utile in diverse situazioni:

- **Sincronizzazione di oggetti correlati**: quando due o più oggetti dipendono dallo stato di un altro oggetto e devono essere sincronizzati, il pattern *Observer* permette di stabilire questa dipendenza in modo strutturato. In questo modo, quando il soggetto osservato cambia il suo stato, tutti gli osservatori vengono notificati e possono aggiornarsi coerentemente.
- **Gestione eventi**: il pattern *Observer* è ampiamente utilizzato nella gestione di eventi e notifiche. Quando si desidera notificare una serie di oggetti quando un evento si verifica, il pattern Observer semplifica l'implementazione fornendo una struttura flessibile ed efficiente per la gestione delle notifiche.
- **Separazione di responsabilità**: utilizzando il pattern *Observer*, si promuove la separazione di responsabilità tra il soggetto osservato e gli osservatori. Il soggetto osservato non deve conoscere i dettagli specifici di ogni osservatore, ma solo l'interfaccia comune che gli osservatori devono implementare. Ciò consente una maggiore modularità e facilità di manutenzione nel sistema.
- **Scalabilità**: il pattern *Observer* favorisce la scalabilità del sistema in quanto consente di aggiungere o rimuovere osservatori in modo dinamico senza influire sul soggetto osservato o su altri osservatori. Questo rende il sistema più flessibile e adattabile a cambiamenti futuri.

```
# Classe Subject (soggetto osservato)
class WeatherStation:
    def __init__(self):
        self.observers = []
        self.temperature = None

    def add_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def set_temperature(self, temperature):
        self.temperature = temperature
```

```
        self.notify_observers()

    def notify_observers(self):
        for observer in self.observers:
            observer.update(self.temperature)

# Classe Observer (osservatore)
class TemperatureDisplay:
    def update(self, temperature):
        print(f"La temperatura è {temperature} gradi Celsius.")

# Utilizzo
weather_station = WeatherStation()
temperature_display = TemperatureDisplay()
weather_station.add_observer(temperature_display)

weather_station.set_temperature(25)
weather_station.set_temperature(30)

weather_station.remove_observer(temperature_display)

weather_station.set_temperature(27)
```

In questo esempio, abbiamo una classe *WeatherStation* che costituisce il soggetto osservato. Ha una lista di osservatori registrati *observers* e un attributo temperature che rappresenta la temperatura corrente.

La classe *WeatherStation* ha metodi per aggiungere, rimuovere e notificare gli osservatori. Il metodo *set_temperature* viene chiamato per impostare la temperatura e poi notifica gli osservatori chiamando il metodo update su ciascun osservatore.

La classe *TemperatureDisplay* costituisce l'osservatore. Implementa il metodo *update* che viene chiamato dal soggetto osservato quando viene notificato di un cambiamento nella temperatura. In questo caso, il metodo *update* semplicemente stampa la temperatura.

Nell'utilizzo, viene creato un oggetto *WeatherStation* e un oggetto *TemperatureDisplay*. L'osservatore (*TemperatureDisplay*) viene aggiunto al soggetto osservato (*WeatherStation*) utilizzando il metodo *add_observer*. Successivamente, vengono impostate due temperature diverse tramite il metodo *set_temperature*, il che porta alla notifica dell'osservatore e alla stampa della temperatura corrispondente. Infine, l'osservatore viene rimosso dal soggetto osservato utilizzando il metodo *remove_observer*. Quando viene impostata una nuova temperatura, l'osservatore non viene più notificato.

In questo esempio, il pattern Observer permette al soggetto osservato (*WeatherStation*) di mantenere una lista di osservatori e di notificarli automaticamente quando avviene un cambiamento di stato. Gli osservatori (*TemperatureDisplay*) possono essere aggiunti o rimossi dinamicamente dal soggetto osservato senza influire sul suo funzionamento. Questo promuove un basso accoppiamento tra soggetto e osservatori e consente una maggiore flessibilità nel sistema.

MVC (Model-View-Controller)

Il pattern *Model-View-Controller* (MVC) è un design pattern architetturale che separa la logica dell'applicazione in tre componenti principali: Modello (Model), Vista (View) e Controller.

Il pattern MVC è utile per diversi motivi:

- **Separazione delle responsabilità**: il pattern *MVC* promuove la separazione delle responsabilità all'interno dell'applicazione. Il Modello gestisce i dati e la logica di business, la Vista gestisce la presentazione dei dati all'utente e il Controller gestisce le interazioni e le azioni dell'utente. Questa separazione consente di mantenere il codice più organizzato, modularizzando le diverse funzionalità e facilitando la manutenibilità e la comprensione del sistema.
- **Riusabilità del codice**: la separazione dei componenti nel pattern *MVC* favorisce il riuso del codice. Ad esempio, è possibile utilizzare lo stesso Modello con diverse Viste o lo stesso Controller con diversi Modelli. Inoltre, grazie alla separazione dei compiti, è più facile

apportare modifiche o estendere il sistema senza dover riscrivere l'intera applicazione.

- **Testabilità**: grazie alla separazione del Modello, dalla Vista e dal Controller, è più semplice testare i singoli componenti in isolamento. Ad esempio, è possibile effettuare test unitari sul Modello senza dover interagire con l'interfaccia utente. Questo migliora l'affidabilità del codice e la qualità complessiva dell'applicazione.
- **Scalabilità**: il pattern *MVC* favorisce la scalabilità delle applicazioni. Ad esempio, è possibile aggiungere nuove Viste o Modelli senza dover apportare modifiche significative al codice esistente. Ciò consente di espandere e adattare l'applicazione per supportare nuove funzionalità o requisiti senza dover riscrivere tutto il sistema.

```
# Model (Modello dei dati)
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

# View (Vista)
class UserView:
    def display_user_details(self, user):
        print(f"Nome: {user.name}")
        print(f"Email: {user.email}")

# Controller
class UserController:
    def __init__(self, user, view):
        self.user = user
        self.view = view

    def update_user_details(self, name, email):
        self.user.name = name
        self.user.email = email

    def display_user_details(self):
```

```
        self.view.display_user_details(self.user)

# Utilizzo
user = User("Alice", "alice@example.com")
view = UserView()
controller = UserController(user, view)

controller.display_user_details()
controller.update_user_details("Alice Smith", "alice.smith@example.com")
controller.display_user_details()
```

In questo esempio, il pattern Model-View-Controller (MVC) viene utilizzato per gestire le informazioni di un utente.

La classe *User* rappresenta il modello dei dati, che contiene le informazioni dell'utente come nome e email.

La classe *UserView* rappresenta la vista, che si occupa di visualizzare le informazioni dell'utente. Il metodo *display_user_details*, consente di mostrato a schermo il nome e l'email dell'utente.

La classe *UserController* rappresenta il controller, che gestisce le interazioni tra il modello dei dati e la vista. Ha un'istanza di *User* e *UserView* come attributi. Il metodo *update_user_details* viene utilizzato per aggiornare le informazioni dell'utente, mentre il metodo *display_user_details* chiama il metodo corrispondente nella vista per visualizzare le informazioni aggiornate dell'utente.

Nell'utilizzo, viene creato un oggetto *User* con nome ed e-mail iniziali. Viene istanziata anche una vista *UserView* e un controller *UserController* che prende il modello User e la vista come argomenti nel suo costruttore.

Il controller viene utilizzato per visualizzare le informazioni dell'utente chiamando il metodo *display_user_details* che a sua volta richiama il metodo corrispondente nella vista. Successivamente, viene chiamato il metodo *update_user_details* per aggiornare le informazioni dell'utente e viene nuovamente chiamato il metodo per visualizzare le informazioni aggiornate.

In questo esempio, il modello (*User*), la vista (*UserView*) e il controller (*UserController*) lavorano insieme seguendo il pattern *Model-View-Controller*.

Capitolo 12: Concetti Avanzati

Le funzioni lambda

Le *funzioni lambda*, o *funzioni anonime*, sono una caratteristica del linguaggio che permette di definire funzioni senza un nome definito. Esse sono utili quando si desidera creare funzioni semplici senza dover dichiarare una funzione completa.

A differenza delle funzioni normali, le *funzioni lambda* non hanno un nome definito e possono essere create e utilizzate direttamente nel punto in cui sono necessarie.

Tuttavia, è importante usarle con parsimonia e tenere conto della loro comprensibilità per gli altri sviluppatori che leggeranno il tuo codice.

Sono un potente strumento da aggiungere alla tua cassetta degli attrezzi di programmazione *Python.*

Sintassi delle funzioni lambda

La sintassi generale per definire una funzione lambda è la seguente:

```
nome_funzione = lambda argomenti: espressione
```

Le funzioni lambda sono costituite da tre parti principali:

- La parola chiave *lambda* che indica che si sta definendo una funzioni lambda.
- Gli argomenti che rappresentano i parametri della funzione separati da virgole.
- L'espressione che rappresenta il corpo della funzione e restituisce il valore di ritorno.

Utilizzo avanzato delle funzioni lambda

Le funzioni lambda possono essere utilizzate in molti contesti e offrono una maggiore flessibilità rispetto alle funzioni tradizionali. Di seguito sono riportati alcuni esempi avanzati di utilizzo:

- sono spesso utilizzate come argomenti per le funzioni che richiedono una funzione come parametro. Questo rende il codice più conciso e leggibile.

```
numeri = [1, 2, 3, 4, 5]
risultati = list(map(lambda x: x * 2, numeri))
print(risultati)  # Output: [2, 4, 6, 8, 10]
```

Nell'esempio sopra, una funzioni lambda viene utilizzata come argomento per la funzione *map*. La funzioni lambda moltiplica ogni elemento della lista numeri per 2, producendo una nuova lista con i risultati corrispondenti.

- sono spesso utilizzate per definire funzioni di ordinamento personalizzate per gli oggetti. Possono essere passate come argomenti alla funzione *sorted()* o al metodo *sort()* di una lista.

```
studenti = [
    {"nome": "Alice", "eta": 20},
    {"nome": "Bob", "eta": 18},
    {"nome": "Charlie", "eta": 22}
]

studenti_ordinati = sorted(studenti, key=lambda x: x["eta"])
print(studenti_ordinati)
```

Nell'esempio sopra, una funzioni lambda viene utilizzata come argomento per la funzione *sorted()*. La funzioni lambda specifica che l'ordinamento deve essere basato sul valore dell'età di ogni studente. La lista studenti viene ordinata in base all'età e il risultato viene stampato.

- possono essere utilizzate per definire funzioni di filtraggio personalizzate utilizzate per filtrare gli elementi di una sequenza. Ad esempio, possono essere passate come argomenti alla funzione *filter()*.

```
numeri = [1, 2, 3, 4, 5]
numeri_pari = list(filter(lambda x: x % 2 == 0, numeri))
print(numeri_pari)  # Output: [2, 4]
```

Nell'esempio sopra, una funzioni lambda viene utilizzata come argomento per la funzione *filter()*. La funzioni lambda definisce la condizione che determina se un numero è pari e restituisce una nuova lista contenente solo i numeri pari dalla lista originale.

Esempio

Supponiamo di avere una lista di parole e vogliamo creare una nuova lista contenente solo le parole che iniziano con una lettera specifica. Possiamo utilizzare una funzioni lambda per filtrare le parole in base a questa condizione. Ecco una possibile implementazione:

```
parole = ["ciao", "casa", "barca", "giorno", "banana", "albero"]
lettera_iniziale = "b"

parole_filtrate = list(filter(lambda x: x.startswith(lettera_iniziale), \
                parole))
print(parole_filtrate)  # Output: ['barca', 'banana']
```

In questo esempio, utilizziamo una funzioni lambda come argomento per la funzione *filter()*. La funzioni lambda prende una parola (*x*) come argomento e verifica se inizia con la lettera specificata (*lettera_iniziale*) utilizzando il metodo *startswith()*. La funzione *filter()* restituisce una nuova lista contenente solo le parole che soddisfano la condizione.

Questo esempio illustra come le funzioni lambda possano essere utilizzate per creare funzioni di filtraggio personalizzate in modo conciso e leggibile.

I decoratori

I decoratori sono una caratteristica avanzata del linguaggio che consente di modificare o estendere il comportamento delle funzioni o delle classi senza modificarne direttamente il codice sorgente. I decoratori offrono una sintassi elegante per l'applicazione di modifiche a una funzione o a una classe, rendendo il codice più leggibile e mantenibile.

Che cosa sono i decoratori?

Un decoratore in *Python* è una funzione (o una classe) che prende come argomento un'altra funzione (o una classe) e restituisce una funzione (o una classe) modificata. In altre parole, i decoratori consentono di avvolgere una

funzione o una classe con un'altra funzione per modificarne il comportamento senza dover modificare direttamente il codice sorgente della funzione o della classe originale.

Sono spesso utilizzati per aggiungere funzionalità come il controllo degli accessi, il logging o la misurazione delle prestazioni alle funzioni esistenti.

Sintassi dei decoratori

La sintassi per l'applicazione di un decoratore a una funzione è la seguente:

```
@decoratore
def funzione():
    # Corpo della funzione
```

In questa sintassi, il decoratore viene applicato alla funzione utilizzando il simbolo "@" seguito dal nome del decoratore. Il decoratore stesso è una funzione che accetta la funzione originale come argomento e restituisce una funzione modificata.

Per comprendere meglio come funzionano, consideriamo un semplice esempio:

```
def mio_decoratore(funzione):
    def funzione_decorata():
        print("Prima dell'esecuzione della funzione")
        funzione()
        print("Dopo l'esecuzione della funzione")
    return funzione_decorata

@mio_decoratore
def mia_funzione():
    print("Esecuzione della mia_funzione")

# Chiamata alla funzione decorata
mia_funzione()
```

In questo esempio, definiamo una funzione chiamata decoratore che prende come argomento un'altra funzione (*funzione*). All'interno del decoratore,

definiamo una nuova funzione chiamata *funzione_decorata* che aggiunge del codice extra prima e dopo l'esecuzione della funzione originale.

Successivamente, utilizziamo il decoratore utilizzando il simbolo @ seguito dal nome del decoratore sopra la definizione della funzione *mia_funzione*. Ciò significa che la nostra funzione *mia_funzione* sarà decorata con la logica del decoratore *mio_decoratore*.

Quando chiamiamo *mia_funzione*, viene effettivamente chiamata la versione decorata della funzione. Questo significa che il codice nel decoratore viene eseguito prima dell'esecuzione della funzione originale; quindi, viene eseguita la funzione originale stessa e infine viene eseguito il codice dopo l'esecuzione della funzione.

L'output del programma sarà:

```
Prima dell'esecuzione della funzione
Esecuzione della mia_funzione
Dopo l'esecuzione della funzione
```

Come puoi vedere, il decoratore ha aggiunto del codice extra prima e dopo l'esecuzione della funzione originale *mia_funzione*.

Decoratori con argomenti

I decoratori con argomenti sono una variante avanzata dei decoratori standard. Consentono di personalizzare il comportamento del decoratore stesso fornendo argomenti durante l'applicazione dello stesso a una funzione o a una classe. L'aggiunta di argomenti ai decoratori offre maggiore flessibilità e possibilità di configurazione.

Sintassi dei decoratori con argomenti

La sintassi per l'applicazione di un decoratore con argomenti a una funzione è la seguente:

```
@decoratore(arg1, arg2, ...)
def funzione():
    # Corpo della funzione
```

In questa sintassi, il decoratore viene applicato alla funzione utilizzando il simbolo "@" seguito dal nome del decoratore e dagli argomenti del decoratore, separati da virgole.

Per implementare un decoratore con argomenti, è necessario definire tre livelli di funzioni annidate. Ecco come potrebbe apparire un esempio di decoratore con argomenti:

```
def decoratore_con_argomenti(arg1, arg2, ...):
    def mio_decoratore(funzione):
        def funzione_modificata(*args, **kwargs):
            # Logica personalizzata del decoratore
            # Utilizza gli argomenti del decoratore
            return funzione(*args, **kwargs)
        return funzione_modificata
    return mio_decoratore
```

Nell'esempio sopra, definiamo una funzione esterna chiamata *decoratore_con_argomenti*, che accetta gli argomenti arg1, arg2, ecc. Questa funzione restituisce il decoratore interno.

Il decoratore interno, chiamato *mio_decoratore*, prende come argomento la funzione originale e definisce una nuova funzione chiamata *funzione_modificata*. All'interno di *funzione_modificata*, puoi aggiungere la logica personalizzata del decoratore utilizzando gli argomenti {arg1}, {arg2}, ecc. Infine, viene restituita *funzione_modificata* come risultato del decoratore.

Di seguito, vediamo un esempio di decoratore che limita il numero di volte in cui una funzione può essere chiamata.

```
def logga(nome_file):
    def decoratore(funzione):
        def funzione_decorata(*args, **kwargs):
            with open(nome_file, 'a') as file:
                file.write(f'Esecuzione della funzione: \
                        {funzione.__name__}\n")
            risultato = funzione(*args, **kwargs)
            return risultato
```

```
        return funzione_decorata
    return decoratore

@logga("log.txt")
def somma(a, b):
    return a + b

@logga("log.txt")
def sottrai(a, b):
    return a - b

# Chiamata alle funzioni decoratate
risultato1 = somma(3, 4)
risultato2 = sottrai(10, 5)

print("Risultato somma:", risultato1)
print("Risultato sottrazione:", risultato2)
```

In questo esempio, abbiamo definito un decoratore logga che prende come argomento il nome di un file in cui vogliamo registrare le informazioni di log. Il decoratore registra l'esecuzione di ogni funzione decorata nel file di log.

All'interno della funzione decoratore *funzione_decorata*, apriamo il file di log in modalità *append* ('a') e scriviamo il messaggio che indica l'esecuzione della funzione. Successivamente, viene chiamata la funzione originale e viene restituito il risultato.

Quando il decoratore viene utilizzato con la sintassi *@logga("log.txt")* sopra la definizione delle funzioni somma e sottrai, viene chiamata la funzione logga con il nome del file di log come argomento.

Le funzioni *somma* e *sottrai* sono quindi decorate con il decoratore *logga*, che registra l'esecuzione delle due funzioni nel file di log specificato.

Alla fine del programma, stampiamo i risultati delle due funzioni decorate e possiamo controllare il file di log per verificare le registrazioni.

È importante notare che l'esempio sopra è solo un'illustrazione del concetto di decoratore con argomenti e dell'uso di un file di log. In una situazione reale, è necessario gestire eventuali errori e considerare altri aspetti come la formattazione del log e la gestione delle eccezioni.

Le metaclassi

Le *metaclassi* sono una caratteristica avanzata del linguaggio che consentono di definire il comportamento delle classi. Una metaclasse può essere vista come una "classe di classi" che definisce come una classe debba essere creata e comportarsi.

Dunque, mentre le classi sono utilizzate per creare oggetti, le metaclassi sono utilizzate per creare classi.

Ecco alcune ragioni per cui dovresti conoscerle:

- **Personalizzazione del comportamento di creazione delle classi**: le metaclassi consentono di controllare il modo in cui le classi vengono create. Puoi intercettare la creazione di una classe e modificarne dinamicamente i suoi attributi, i suoi metodi o il suo comportamento generale. Questo ti dà un livello di flessibilità molto elevato nella definizione delle tue classi.
- **Creazione di API e framework avanzati**: le metaclassi possono essere utilizzate per creare API e framework avanzati che semplificano l'uso delle classi per gli utenti. Ad esempio, puoi definire una metaclasse che aggiunge funzionalità automatiche alle classi create, come la tracciatura dei metodi chiamati o l'inserimento di log. Questo può rendere l'utilizzo delle classi più semplice e più potente per gli sviluppatori che utilizzano il tuo framework.
- **Validazione e controllo delle classi**: le metaclassi consentono di eseguire controlli e validazioni sulle classi create. Puoi controllare se una classe rispetta determinate regole o requisiti, ad esempio assicurandoti che una classe abbia determinati attributi o che implementi determinati metodi. In questo modo, le metaclassi possono aiutare a garantire l'integrità e la correttezza del codice.
- **Creazione di classi dinamiche**: le metaclassi consentono di creare classi dinamicamente durante l'esecuzione del programma. Questo può

essere utile in situazioni in cui hai bisogno di creare classi in base a condizioni o configurazioni specifiche.

Utilizzo delle metaclassi

Per utilizzare una metaclasse, è necessario definire una classe personalizzata che eredita dalla classe *type*. La classe personalizzata si comporterà da metaclasse per le nuove classi che verranno create. È possibile sovrascrivere i metodi della classe *type* per personalizzare il comportamento della metaclasse.

Una volta definita una metaclasse, è possibile utilizzarla specificando il nome della metaclasse come argomento *metaclass* nella definizione di una classe.

```
class MetaclasseAggiungiMetodo(type):
    def __init__(cls, nome, basi, attributi):
        super().__init__(nome, basi, attributi)
        cls.nuovo_metodo = lambda self: print("Questo è il \
        nuovo metodo aggiunto da Metaclasse")

class MiaClasse(metaclass=MetaclasseAggiungiMetodo):
    pass

mia_istanza = MiaClasse()
mia_istanza.nuovo_metodo()  # Output: Questo è il nuovo metodo
                            #         aggiunto da Metaclasse
```

In questo esempio, definiamo una metaclasse chiamata *MetaclasseAggiungiMetodo* che eredita dalla classe *type*. Sovrascriviamo il metodo *__init__*, che viene chiamato quando una nuova classe viene creata.

Nel metodo *__init__*, aggiungiamo un nuovo metodo chiamato *nuovo_metodo* a tutte le classi create utilizzando la metaclasse. Il metodo *nuovo_metodo* è una funzione lambda che semplicemente stampa un messaggio.

Successivamente, definiamo la classe *MiaClasse* utilizzando l'argomento *metaclass* e specificando *MetaclasseAggiungiMetodo* come metaclasse.

Infine, creiamo un'istanza di *MiaClasse* chiamata *mia_istanza* e chiamiamo il metodo *nuovo_metodo* su di essa. Vedremo che il metodo viene chiamato correttamente e stampa il messaggio *"Questo è il nuovo metodo aggiunto da Metaclasse"*.

Questo esempio semplificato illustra come una metaclasse possa aggiungere comportamenti o attributi automaticamente a tutte le sottoclassi. Si può immaginare di utilizzare le metaclassi per implementare funzionalità comuni o per estendere il comportamento delle classi in modo specifico.

Vediamo adesso un esempio di come le metaclassi possono essere utilizzate per controllare e validare le classi durante la creazione.

Supponiamo di voler definire una metaclasse chiamata *ControlloAttributi* che controlla se una classe ha tutti gli attributi richiesti. In questo caso, vogliamo che ogni classe abbia gli attributi {nome} e {cognome}. Se una classe non soddisfa questa condizione, verrà sollevata un'eccezione.

```
class ControlloAttributi(type):
    def __new__(cls, name, bases, attrs):
        if 'nome' not in attrs or 'cognome' not in attrs:
            raise AttributeError(f"La classe {name} deve \
            avere gli attributi 'nome' e 'cognome'.")
        return super().__new__(cls, name, bases, attrs)

class Persona(metaclass=ControlloAttributi):
    nome = 'Mario'
    cognome = 'Rossi'

class Studente(metaclass=ControlloAttributi):
    nome = 'Luca'

# La classe Persona ha tutti gli attributi richiesti,
# quindi viene creata correttamente.
# La classe Studente manca dell'attributo 'cognome',
# quindi solleva un'eccezione.
```

Nell'esempio sopra, abbiamo definito la classe *ControlloAttributi* come una sottoclasse di *type*, che è la metaclasse predefinita di *Python*. Sovrascriviamo il metodo *__new__*, che viene chiamato durante la creazione di una classe.

All'interno di *__new__*, controlliamo se la classe in creazione ha gli attributi {nome} e {cognome} nel dizionario degli attributi (*attrs*). Se uno di questi attributi manca, solleviamo un'eccezione *AttributeError*. In caso contrario, restituiamo la classe creata utilizzando *super().__new__*.

Quando creiamo la classe *Persona* e *Studente* specificando *metaclass=ControlloAttributi*, la metaclasse *ControlloAttributi* viene utilizzata per controllare se le classi soddisfano i requisiti degli attributi. La classe *Persona* ha tutti gli attributi richiesti e viene creata correttamente. D'altra parte, la classe *Studente* manca dell'attributo *cognome*, quindi viene sollevata un'eccezione *AttributeError*.

Vediamo un ultimo esempio di come le metaclassi possono essere utilizzate per creare classi dinamicamente.

Supponiamo di voler creare una metaclasse chiamata *GeneratoreClasse* che genera classi dinamicamente in base a un parametro specificato. In questo caso, vogliamo creare classi che rappresentino animali e il parametro specificato sarà il nome dell'animale.

```
class GeneratoreClasse(type):
    def __new__(cls, name, bases, attrs):
        attrs['__init__'] = cls._custom_init
        return super().__new__(cls, name, bases, attrs)

    @staticmethod
    def _custom_init(self, nome, attributi):
        self.nome = nome
        for attributo, valore in attributi.items():
            setattr(self, attributo, valore)

class Animale(metaclass=GeneratoreClasse):
    pass
```

```
# Creazione dinamica della classe 'Gatto'
Gatto = type('Gatto', (Animale,), {})

# Creazione dinamica della classe 'Cane'
Cane = type('Cane', (Animale,), {})

# Creazione di istanze degli animali
mio_gatto = Gatto('Micio', {'razza': 'Persiano'})
mio_cane = Cane('Fido', {'razza': 'Labrador'})

print(mio_gatto.nome)  # Output: Micio
print(mio_gatto.razza)  # Output: Persiano

print(mio_cane.nome)  # Output: Fido
print(mio_cane.razza)  # Output: Labrador
```

Nell'esempio sopra, la classe *GeneratoreClasse* è una metaclasse che deriva dalla classe type e ne sovrascrive il metodo *__new__()*. Questo metodo viene chiamato quando viene creata una nuova classe dinamicamente. In questo caso, viene aggiunto un metodo *__init__()* personalizzato alle classi generate.

Il metodo *__init__()* personalizzato, definito come un metodo statico nella metaclasse *GeneratoreClasse*, viene utilizzato per inizializzare gli attributi dell'istanza con i parametri *nome* e *attributi*.

La classe *Animale* viene definita con la metaclasse *GeneratoreClasse*, quindi avrà il metodo *__init__()* personalizzato.

Le classi *Gatto* e *Cane* vengono generate dinamicamente utilizzando la funzione *type*. Vengono passati il nome della classe come stringa, le basi (in questo caso, la classe *Animale*), e un dizionario vuoto per gli eventuali attributi aggiuntivi.

Vengono create le istanze degli animali *mio_gatto* e *mio_cane* utilizzando le classi *Gatto* e Cane, rispettivamente. Vengono passati i parametri corretti per il nome e gli attributi.

Infine, vengono stampati i valori degli attributi delle istanze degli animali per verificarne il corretto funzionamento.

Il metodo type

Dato che nell'esempio precedente abbiamo utilizzato il metodo {type()}, vediamo qual'è la sua funzione e la sua utilità.

Ecco alcuni dei suoi utilizzi principali:

- **Determinare il tipo di un oggetto**: puoi utilizzare *type* per ottenere il tipo di un oggetto esistente. Ad esempio, *type(42)* restituirà *<class 'int'>,* indicando che 42 è un oggetto di tipo *int*.
- **Creare nuove classi dinamicamente**: puoi utilizzare *type* per creare nuove classi dinamicamente. Passando i parametri corretti, puoi definire un nome di classe come stringa, le basi (classi da cui ereditare) e un dizionario che contiene gli attributi e i metodi della classe. Questo è ciò che abbiamo fatto nell'esempio della metaclasse per generare dinamicamente le classi degli animali.
- **Controllare il tipo di un oggetto**: puoi utilizzare *type* per controllare se un oggetto è di un determinato tipo. Ad esempio, *type(obj) == int* restituirà *True}* se *obj* è un oggetto di tipo *int*.
- **Introspezione**: *type* può essere utilizzato per esaminare le informazioni sulle classi e gli oggetti durante l'introspezione. Ad esempio, puoi ottenere la lista delle basi di una classe chiamando *type(classe).__bases__*.

```
class MioPadre:
    pass

class MiaClasse(MioPadre):
    pass

# Ottenere la lista delle basi di una classe
basi = MiaClasse.__bases__
print(basi)  # Output: (<class '__main__.MioPadre'>,)

# Verificare l'ereditarietà di una classe
```

```
is_subclass = issubclass(MiaClasse, MioPadre)
print(is_subclass)  # Output: True

# Verificare l'istanza di una classe
my_object = MiaClasse()
is_instance = isinstance(my_object, MioPadre)
print(is_instance)  # Output: True
```

Introduzione alla concorrenza e al parallelismo

Nella programmazione, i concetti di concorrenza e parallelismo riguardano l'esecuzione di più attività contemporaneamente, ma con differenze significative nel modo in cui vengono gestite.

Concetto di concorrenza

La concorrenza si riferisce all'esecuzione simultanea di più attività, ma non necessariamente in modo parallelo. In altre parole, si tratta di avere più flussi di esecuzione che vengono eseguiti in modo intercalato nel tempo. La concorrenza può essere ottenuta utilizzando i *thread*, che sono piccoli sottoprocessi all'interno di un processo più grande. I *thread* possono essere eseguiti in parallelo su più core di una CPU o possono condividere il tempo di elaborazione di un singolo core.

Durante l'esecuzione concorrente, le attività possono essere eseguite contemporaneamente o in modo alternato, a seconda di come vengono pianificate dal sistema operativo o dall'interprete del linguaggio di programmazione.

Concetto di parallelismo

Il parallelismo, d'altra parte, si riferisce all'esecuzione simultanea di più attività in modo vero e proprio, su più core di una CPU o su più macchine. Le attività parallele vengono eseguite contemporaneamente e possono ottenere una maggiore velocità di elaborazione rispetto all'esecuzione sequenziale.

L'utilizzo del parallelismo richiede che le attività abbiano una natura veramente indipendente tra loro, in modo che possano essere eseguite senza causare interferenze o dipendenze tra di loro. Per sfruttare appieno il

parallelismo, è necessario che l'hardware di elaborazione supporti più core o più macchine disponibili.

Vantaggi e svantaggi della concorrenza e del parallelismo

L'utilizzo della concorrenza e del parallelismo in un'applicazione può offrire numerosi vantaggi, ma è anche importante considerare gli eventuali svantaggi.

Vantaggi della concorrenza e del parallelismo

- **Miglioramento delle prestazioni**: uno dei principali vantaggi della concorrenza e del parallelismo è il miglioramento delle prestazioni dell'applicazione. Sfruttando più thread o più processi per eseguire attività in modo simultaneo, è possibile ridurre i tempi di elaborazione complessivi. Questo può portare a un aumento della velocità dell'applicazione e a una maggiore capacità di gestire un carico di lavoro più elevato.
- **Utilizzo efficiente delle risorse**: la concorrenza e il parallelismo consentono di utilizzare in modo efficiente le risorse disponibili, come i core di una CPU. Con l'esecuzione di attività in parallelo, è possibile sfruttare al massimo la potenza di calcolo disponibile. Ciò si traduce in una maggiore efficienza nell'utilizzo delle risorse hardware e nella riduzione dei tempi di attesa per l'elaborazione.
- **Maggiore capacità di risposta**: la concorrenza può migliorare la capacità di risposta di un'applicazione. Separando determinate attività in *thread* separati, è possibile evitare blocchi o rallentamenti che potrebbero influire sulla capacità di risposta dell'intero sistema. Ad esempio, in un'applicazione con un'interfaccia utente, l'utilizzo di *thread* separati per eseguire operazioni di lunga durata consente di mantenere la risposta immediata all'interazione dell'utente.

Svantaggi della concorrenza e del parallelismo

- **Complessità della gestione**: l'implementazione della concorrenza e del parallelismo può aumentare la complessità del codice. La gestione di *thread* o processi separati richiede attenzione per la sincronizzazione

dei dati condivisi, la gestione degli stati e la prevenzione di problemi come le *race condition* o le **deadlock**. È necessario gestire attentamente la concorrenza per evitare situazioni indesiderate e bug difficili da individuare.

- **Overhead di gestione**: l'uso della concorrenza e del parallelismo comporta un certo overhead di gestione. Ciò significa che l'aggiunta di *thread* o processi aggiuntivi comporta un aumento dei consumi di memoria e una maggiore complessità nell'organizzazione delle attività. Inoltre, la sincronizzazione dei *thread* o dei processi può richiedere tempo e risorse aggiuntive.
- **Possibilità di race condition e deadlock**: la concorrenza può portare a situazioni indesiderate come *race condition* o le *deadlock*. Una *race condition* si verifica quando due o più *thread* o processi accedono contemporaneamente a una risorsa condivisa, causando risultati imprevedibili o inconsistenze nei dati. Una *deadlock*, invece, si verifica quando due o più *thread* o processi si bloccano reciprocamente, aspettando l'uno sull'altro per rilasciare una risorsa necessaria per procedere. Queste situazioni possono causare comportamenti imprevedibili o blocchi completi dell'applicazione. Per mitigare questi problemi, è importante utilizzare meccanismi di sincronizzazione adeguati, come i lock, i semafori o i meccanismi di comunicazione tra processi.

Introduzione al modulo threading

Il modulo *threading* fornisce un'interfaccia ad alto livello per la gestione dei *thread*. Con esso è possibile creare e controllare i *thread* all'interno di un programma, consentendo l'esecuzione concorrente di più attività.

Creazione di un thread

Per utilizzare il modulo *threading*, è necessario importarlo nel proprio codice tramite la dichiarazione *import threading*. Successivamente, è possibile creare un nuovo *thread* definendo una classe che eredita dalla classe *Thread* fornita dal modulo.

```
import threading
```

```
class MyThread(threading.Thread):
    def run(self):
        # Logica del thread
        print("Sono un thread!")

# Creazione di un'istanza del thread
my_thread = MyThread()

# Avvio del thread
my_thread.start()
```

Nell'esempio precedente, abbiamo definito una classe *MyThread* che eredita dalla classe *Thread*. La classe *MyThread* implementa un metodo chiamato *run(),* che contiene la logica che verrà eseguita all'interno del thread. Nel nostro caso, il metodo *run()* semplicemente stampa un messaggio.

Dopo aver creato un'istanza del *thread*, possiamo avviarlo chiamando il metodo *start()*. Questo avvia l'esecuzione del *thread* e chiama automaticamente il metodo *run()* definito nella nostra classe.

La sincronizzazione dei thread

La sincronizzazione è un aspetto cruciale nella gestione dei *thread* concorrenti. Per evitare problemi come le *race condition* o garantire l'accesso esclusivo alle risorse condivise, *Python* fornisce meccanismi di sincronizzazione come i lock e i semafori.

- **Lock**: il lock, o "blocco", è un meccanismo di sincronizzazione che permette a un solo *thread* alla volta di accedere a una risorsa condivisa. Quando un *thread* acquisisce un lock, gli altri *thread* che cercano di acquisirlo devono attendere fino a quando il lock viene rilasciato. Questo assicura l'accesso esclusivo alla risorsa condivisa, evitando le *race condition*.

```
import threading

# Creazione di un lock
lock = threading.Lock()
```

```
# Variabile condivisa
shared_variable = 0

def increment():
    global shared_variable

    # Acquisizione del lock
    lock.acquire()

    try:
        # Modifica della variabile condivisa
        shared_variable += 1
    finally:
        # Rilascio del lock
        lock.release()

# Creazione dei thread
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

# Avvio dei thread
thread1.start()
thread2.start()

# Attesa che i thread terminino l'esecuzione
thread1.join()
thread2.join()

# Stampa del risultato
print("Valore finale:", shared_variable)
```

Nell'esempio precedente, abbiamo creato un oggetto *Lock* utilizzando *threading.Lock()*. All'interno della funzione *increment()*, acquisiamo il lock chiamando *lock.acquire()* prima di modificare la variabile condivisa. Dopo aver completato la modifica, rilasciamo il lock

chiamando *lock.release()*. Questo garantisce che solo un *thread* alla volta possa accedere e modificare la variabile condivisa.

- **Semafori**: un semaforo è un oggetto con un valore intero associato ad esso. Questo valore rappresenta il numero di "permessi" disponibili per accedere a una risorsa condivisa. I semafori possono essere utilizzati per regolare l'accesso concorrente a una risorsa limitando il numero di *thread* che possono accedervi contemporaneamente.

Un semaforo offre due operazioni principali:

- **acquire()**: decrementa il valore del semaforo e blocca il thread se il valore diventa negativo, indicando che tutti i permessi sono stati utilizzati.
- **release()**: incrementa il valore del semaforo e sblocca un *thread* in attesa, se presente.

Per utilizzare i semafori è necessario importare il modulo *threading* e creare un oggetto *Semaphore* specificando il numero di permessi disponibili.

```
import threading

# Creazione di un semaforo con 3 permessi
semaforo = threading.Semaphore(3)

def access_resource():
    # Acquisizione del semaforo
    semaforo.acquire()

    try:
        # Sezione critica: accesso alla risorsa condivisa
        print("Accesso alla risorsa condivisa...")
        # ...
    finally:
        # Rilascio del semaforo
        semaforo.release()
```

```
# Creazione dei thread
thread1 = threading.Thread(target=access_resource)
thread2 = threading.Thread(target=access_resource)

# Avvio dei thread
thread1.start()
thread2.start()

# Attesa che i thread terminino l'esecuzione
thread1.join()
thread2.join()
```

Nell'esempio precedente, abbiamo creato un oggetto *Semaphore* specificando il numero di permessi pari a 3. All'interno della funzione *access_resource()*, acquisiamo il semaforo chiamando *semaforo.acquire()*. Se il semaforo ha permessi disponibili, il *thread* può accedere alla risorsa condivisa all'interno della sezione critica. Al termine dell'accesso, chiamiamo *semaforo.release()* per rilasciare il semaforo e consentire ad altri *thread* di accedere.

I semafori offrono diversi vantaggi nella sincronizzazione dei *thread*:

- **Controllo dell'accesso concorrente**: possono limitare il numero di *thread* che possono accedere a una risorsa condivisa contemporaneamente, evitando situazioni di congestione.
- **Gestione delle risorse**: i semafori possono essere utilizzati per gestire l'accesso a risorse limitate, come ad esempio i *thread* che accedano a un database o a un pool di connessioni di rete.
- **Prevenzione delle race condition**: utilizzando i semafori, è possibile sincronizzare l'accesso a una risorsa condivisa, evitando problemi come le *race condition* e garantendo la coerenza dei dati.

Tuttavia, è importante utilizzare i semafori con attenzione per evitare potenziali problemi come *deadlock* o *starvation*. Un *deadlock*, come già detto in precedenza, si verifica quando due o più *thread* si bloccano a vicenda, mentre la *starvation* si verifica quando un thread viene

continuamente privato dell'accesso alla risorsa a causa del comportamento di scheduling.

Inoltre, i semafori richiedono una corretta gestione delle operazioni di acquisizione e rilascio. Un errore nella loro gestione potrebbe portare a una situazione in cui i *thread* rimangono bloccati o i permessi non vengono correttamente rilasciati.

Introduzione al modulo multiprocessing

Il modulo *multiprocessing* è una libreria integrata di *Python* che fornisce un'interfaccia per la creazione e la gestione di processi multipli. A differenza del modulo *threading*, che si occupa di eseguire i *thread* all'interno di un singolo processo, il modulo *multiprocessing* consente di sfruttare appieno i vantaggi dei processi multipli su sistemi multi-core o multi-processore.

Creazione di processi

Il modulo *multiprocessing* offre diverse classi per la creazione e la gestione dei processi. La classe principale è *Process*, che rappresenta un singolo processo. Per creare un nuovo processo, è necessario istanziare la classe *Process* e fornire come argomento una funzione da eseguire al suo interno.

```
import multiprocessing

def worker():
    print("Processo figlio")

if __name__ == "__main__":
    # Creazione di un nuovo processo
    process = multiprocessing.Process(target=worker)

    # Avvio del processo
    process.start()

    # Attesa che il processo termini l'esecuzione
```

```
process.join()

print("Processo padre")
```

Nell'esempio precedente, abbiamo definito una funzione *worker()* che viene eseguita nel processo figlio. Utilizzando *multiprocessing.Process*, abbiamo creato un nuovo processo specificando la funzione *worker()* come target. Il metodo *start()* avvia l'esecuzione del processo, mentre *join()* attende che il processo termini l'esecuzione. Infine, il codice nel blocco *if __name__ == "__main__":* viene eseguito solo nel processo padre.

Gestione dei processi

Il modulo *multiprocessing* fornisce metodi per la gestione dei processi, come l'attesa che un processo termini l'esecuzione o la verifica dello stato di un processo. Vediamo alcuni dei metodi più comuni.

- **join()**: il metodo *join()* viene chiamato su un oggetto *Process* per attendere che il processo termini l'esecuzione. Viene utilizzato per sincronizzare il processo principale (padre) con il processo figlio.

```
process.join()
```

- **is_alive()**: il metodo *is_alive()* viene chiamato su un oggetto *Process* per verificare se il processo è ancora in esecuzione. Restituisce *True* se il processo è ancora attivo, altrimenti restituisce *False*.

```
if process.is_alive():
    print("Il processo è ancora in esecuzione")
else:
    print("Il processo ha terminato l'esecuzione")
```

- **terminate()**: il metodo *terminate()* viene chiamato su un oggetto *Process* per terminare immediatamente l'esecuzione del processo.

```
process.terminate()
```

Comunicazione e condivisione di dati tra processi

Il modulo *multiprocessing* non solo consente la creazione e la gestione di processi multipli, ma fornisce anche meccanismi per la comunicazione e la condivisione di dati tra di essi. Poiché i processi creati utilizzano spazi di memoria separati, è necessario utilizzare strumenti specifici per scambiare informazioni tra i processi. In questa sezione, esploreremo i principali meccanismi offerti dal modulo *multiprocessing* per la comunicazione e la condivisione di dati.

Code

Una delle modalità di comunicazione più comuni tra processi è l'utilizzo di code sincronizzate. Il modulo *multiprocessing* fornisce la classe *Queue* per la creazione di code condivise tra processi.

La classe *Queue* implementa una coda FIFO (First-In, First-Out), in cui i processi possono inserire dati utilizzando il metodo *put()* e prelevare dati utilizzando il metodo *get()*. Le code offrono un'interfaccia sicura e thread-safe per lo scambio di dati tra processi.

```
def worker(queue):
    value = queue.get()
    print("Processo figlio:", value)

if __name__ == "__main__":
    # Creazione di una coda condivisa
    queue = multiprocessing.Queue()

    # Inserimento di un valore nella coda
    queue.put("Ciao dal processo padre!")

    # Creazione di un nuovo processo
    process = multiprocessing.Process(target=worker, args=(queue,))

    # Avvio dell'esecuzione del processo
    process.start()

    # Attesa che il processo termini l'esecuzione
```

```
process.join()
```

Nell'esempio precedente, abbiamo creato una coda utilizzando *multiprocessing.Queue()*. Nel processo padre, abbiamo inserito il valore "Ciao dal processo padre!" nella coda utilizzando il metodo *put()*. Nel processo figlio, abbiamo utilizzato *get()* per prelevare il valore dalla coda e stamparlo.

Variabili condivise

In alcuni casi, potrebbe essere necessario condividere dati tra processi in modo più diretto rispetto all'utilizzo di code. Il modulo *multiprocessing* fornisce le classi *Value* e *Array* per la creazione di variabili condivise tra processi.

- **Classe Value**: la classe *Value* consente di creare una variabile condivisa di un determinato tipo di dati, come interi, float o booleani. La sintassi per la creazione è la seguente:

```
import multiprocessing


# Creazione di una variabile condivisa di tipo intero
# inizializzata a 0
data = multiprocessing.Value("i", 0)
```

- **Classe Array**: la classe *Array* consente di creare un array condiviso di un determinato tipo di dati. È possibile specificare il tipo di dati e la dimensione dell'array durante la creazione. La sintassi è la seguente:

```
import multiprocessing


# Creazione di un array condiviso di tipo float
# inizializzato con [1.0, 2.0, 3.0, 4.0]
data = multiprocessing.Array("f", [1.0, 2.0, 3.0, 4.0])
```

Una volta create le variabili condivise, è possibile accedere e modificarle da diversi processi. È importante tenere presente che, a differenza delle code, non esiste un meccanismo di sincronizzazione implicito per le variabili

condivise. Pertanto, è necessario utilizzare i meccanismi di sincronizzazione appropriati, come i lock, per garantire un accesso sicuro e coerente ai dati condivisi.

```
import multiprocessing

def worker(data):
    data.value = 42

if __name__ == "__main__":
    # Creazione di una variabile condivisa di tipo intero
    data = multiprocessing.Value("i", 0)

    # Creazione di un nuovo processo
    process = multiprocessing.Process(target=worker, args=(data,))

    # Avvio dell'esecuzione del processo
    process.start()

    # Attesa che il processo termini l'esecuzione
    process.join()

    # Stampa del valore della variabile condivisa
    print("Valore della variabile condivisa:", data.value)
```

Pool di processi

Nel contesto della programmazione parallela, il concetto di "pool di processi" si riferisce a un gruppo di processi pronti ad eseguire dei compiti in parallelo. Il modulo *multiprocessing* fornisce la classe *Pool* per creare e gestire facilmente un pool di processi.

Creazione di un Pool di processi

La creazione di un *Pool* di processi è semplice: è possibile specificare il numero massimo di processi da utilizzare nel Pool o, se non specificato, verrà utilizzato il numero massimo di processori disponibili nel sistema.

```
import multiprocessing
```

```
# Creazione di un Pool di processi con quattro processi
pool = multiprocessing.Pool(processes=4)
```

Esecuzione di compiti con il Pool

Una volta creato il *Pool* di processi, è possibile eseguire compiti paralleli utilizzando i metodi forniti dalla classe *Pool*. Il metodo principale per l'esecuzione di un compito è *apply_async()*, che accetta una funzione e i relativi argomenti. Restituisce un oggetto *AsyncResult* che può essere utilizzato per ottenere il risultato del compito in modo asincrono.

```
import multiprocessing

def square(x):
    return x ** 2

if __name__ == "__main__":
    # Creazione di un Pool di processi con quattro processi
    pool = multiprocessing.Pool(processes=4)

    # Esecuzione di un compito parallelo utilizzando apply_async()
    result = pool.apply_async(square, (5,))

    # Ottiene il risultato del compito in modo asincrono
    print(result.get())
```

Nell'esempio precedente, abbiamo definito una funzione *square()* che calcola il quadrato di un numero. Successivamente, abbiamo creato un *Pool* di processi con quattro processi. Abbiamo eseguito il compito parallelo utilizzando *apply_async()* per calcolare il quadrato di 5. Infine, abbiamo ottenuto il risultato del compito utilizzando il metodo *get()* sull'oggetto *AsyncResult*.

Esecuzione di compiti multipli con il Pool

Il *Pool* di processi nel modulo *multiprocessing* offre un modo conveniente per eseguire compiti multipli in parallelo. Utilizzando il metodo *map()* del

Pool, è possibile applicare una funzione a una sequenza di input, suddividere automaticamente la sequenza in porzioni e distribuire i compiti tra i processi nel *Pool*.

```
import multiprocessing

def square(x):
    return x ** 2

if __name__ == "__main__":
    # Creazione di un Pool di processi con quattro processi
    pool = multiprocessing.Pool(processes=4)

    # Esecuzione di compiti multipli utilizzando map()
    numbers = [1, 2, 3, 4, 5]
    results = pool.map(square, numbers)

    # Stampa dei risultati
    print(results)
```

Nell'esempio precedente, abbiamo definito una funzione *square()* che calcola il quadrato di un numero. Successivamente, abbiamo creato un *Pool* di processi con quattro processi. Utilizzando il metodo *map(),* abbiamo applicato la funzione *square()* a una sequenza di numeri [1, 2, 3, 4, 5]. Il *Pool* di processi ha suddiviso automaticamente la sequenza in porzioni e ha distribuito i compiti ai processi disponibili nel *Pool*. Infine, abbiamo ottenuto i risultati dei compiti tramite la lista *results*.

L'utilizzo del metodo *map()* semplifica notevolmente l'esecuzione di compiti multipli, in quanto si occupa automaticamente della suddivisione dei compiti e della raccolta dei risultati. Tuttavia, è importante tenere presente che l'ordine dei risultati potrebbe non corrispondere all'ordine degli elementi nella sequenza di input, poiché i compiti vengono eseguiti in parallelo.

Il *Pool* di processi è particolarmente vantaggioso quando si hanno compiti indipendenti che possono essere eseguiti contemporaneamente. Sfruttando

il parallelo, è possibile ottenere un notevole miglioramento delle prestazioni e accelerare l'esecuzione di operazioni computazionalmente intensive.

Introduzione ai concetti di GIL (Global Interpreter Lock)

Il *GIL (Global Interpreter Lock)* è un meccanismo di sincronizzazione che viene utilizzato per garantire la coerenza dei dati all'interno dell'interprete *Python*. Il *GIL* è una peculiarità del *CPython*, l'implementazione di riferimento di *Python*, ed è importante comprenderne il funzionamento e l'impatto sulle prestazioni delle applicazioni.

Cos'è il GIL?

Il *GIL* è un meccanismo di lock che impedisce l'esecuzione simultanea di *thread Python* all'interno dello stesso processo. In altre parole, il *GIL* permette solo a un singolo *thread Python* di eseguire codice *Python* alla volta, anche su macchine multicore. Questo significa che, nonostante l'utilizzo di *thread*, l'esecuzione parallela di codice *Python* viene limitata da questo meccanismo di lock.

Motivazioni del GIL

Il *GIL* è stato introdotto nel *CPython* per semplificare la gestione della memoria e i dettagli dell'allocazione delle risorse in un ambiente multithreading. Grazie al *GIL*, *CPython* può utilizzare strutture dati interne in modo più efficiente, evitando la necessità di gestire esplicitamente la concorrenza delle operazioni di base.

Impatto sulle prestazioni

A causa del *GIL*, i *thread Python* non possono sfruttare appieno i vantaggi del parallelismo su sistemi multicore. Anche se si utilizzano *thread* multipli per eseguire operazioni in modo concorrente, il *GIL* li limita ad essere eseguiti in sequenza, anziché contemporaneamente. Di conseguenza, il *GIL* può essere un ostacolo per applicazioni che richiedono un alto grado di parallelismo computazionale.

Tuttavia, è importante sottolineare che il *GIL* può avere un impatto significativo solo su determinati tipi di applicazioni. Se il codice contiene operazioni bloccanti come l'I/O di rete, l'accesso a file o l'esecuzione di codice in linguaggi esterni, il *GIL* viene rilasciato durante queste

operazioni, consentendo ad altri *thread Python* di essere eseguiti. Pertanto, in scenari di utilizzo intensivo di I/O, il *GIL* potrebbe non costituire un limite significativo alle prestazioni.

Alcune alternative al GIL

Vediamo alcune alternative al *GIL*:

- **Utilizzo di processi multipli**: invece di utilizzare *thread*, si possono utilizzare processi multipli utilizzando il modulo *multiprocessing*. Poiché ogni processo avrà il proprio interprete *Python* indipendente, il *GIL* non rappresenterà un ostacolo per il parallelismo. Tuttavia, la comunicazione e la condivisione dei dati tra processi richiedono un'attenzione particolare a causa delle sfide legate alla sincronizzazione e alla serializzazione dei dati.
- **Utilizzo di estensioni scritte in C**: in alcune situazioni critiche in termini di prestazioni, si può considerare l'utilizzo di estensioni scritte in C o linguaggi simili. Queste estensioni possono essere eseguite senza il *GIL*, consentendo un vero parallelismo. Tuttavia, l'implementazione di estensioni richiede una conoscenza avanzata di programmazione e può complicare il processo di sviluppo.
- **Utilizzo di librerie specializzate**: esistono anche librerie specializzate, come *NumPy* e *pandas*, che offrono operazioni vettoriali e operazioni su array che sono eseguite in modo efficiente senza richiedere l'uso di *thread* o processi multipli. Queste librerie sfruttano al massimo le capacità di elaborazione dei processori sottostanti.

Utilizzo di librerie esterne come asyncio per la programmazione asincrona

La programmazione asincrona è una tecnica che consente di gestire in modo efficiente compiti non bloccanti e di migliorare le prestazioni delle applicazioni che richiedono operazioni di I/O intensivo. Una delle librerie più utilizzate per la programmazione asincrona in *Python* è *asyncio*.

Introduzione ad asyncio

asyncio è un modulo integrato nella libreria standard a partire dalla versione 3.4. Fornisce un framework per la scrittura di codice asincrono basato su *coroutines*, che sono funzioni speciali che possono essere sospese e riprese

in modo non bloccante. Le *coroutines* vengono eseguite all'interno di un *event loop*, che coordina l'esecuzione delle diverse attività asincrone.

Vediamo i concetti principali della programmazione asincrona.

- **Coroutines**: le *coroutines* sono funzioni asincrone che vengono definite utilizzando la sintassi *async def*. Consentono di sospendere l'esecuzione e aspettare che un'operazione asincrona si completi senza bloccare l'intero processo.
- **Event loop**: l'*event loop* è il motore principale di *asyncio* che coordina l'esecuzione delle *coroutines*. Gestisce l'ordine di esecuzione delle *coroutines*, permettendo loro di essere sospese e riprese in modo efficiente.
- **Task**: un *task* rappresenta un'attività asincrona all'interno dell'*event loop*. Può essere creato per eseguire una coroutine e monitorarne lo stato e i risultati.
- **Future**: un *Future* rappresenta il risultato di un'attività asincrona che potrebbe non essere ancora disponibile. Può essere usato per attendere l'ottenimento dei risultati di un'operazione asincrona.
- **waitable**: un oggetto che può essere usato con l'operatore *await* per sospendere l'esecuzione di una coroutine e attendere il completamento di un'operazione asincrona.

Ecco un esempio base che illustra come utilizzare *asyncio* per eseguire compiti asincroni:

```
import asyncio

async def task1():
    print("Inizio task 1")
    await asyncio.sleep(2)  # Simula un'operazione asincrona di 2 secondi
    print("Fine task 1")

async def task2():
    print("Inizio task 2")
    await asyncio.sleep(1)  # Simula un'operazione asincrona di 1 secondo
    print("Fine task 2")
```

```
async def main():
    print("Avvio dei task")
    # Esegue i task in modo asincrono
    await asyncio.gather(task1(), task2())
    print("Tutti i task sono stati completati")

asyncio.run(main())
```

In questo esempio, definiamo tre funzioni: *task1()*, *task2()* e *main()*. Le prime due funzioni rappresentano i compiti asincroni che vogliamo eseguire. Utilizziamo la funzione *asyncio.sleep()* per simulare delle operazioni asincrone che richiedono del tempo per essere completate.

La funzione *main()* è la nostra funzione principale che coordina l'esecuzione dei compiti asincroni. All'interno di questa funzione, utilizziamo *asyncio.gather()* per avviare i compiti *task1()* e *task2()* contemporaneamente. Questo ci consente di eseguirli in modo asincrono, senza dover aspettare il completamento di uno prima di iniziare l'altro.

Infine, utilizziamo *asyncio.run()* per eseguire la funzione *main()* all'interno di un *event loop* di *asyncio*. Questo avvia l'esecuzione dell'applicazione asincrona e gestisce l'ordine di esecuzione dei compiti.

Nel risultato, vedrai che i due task vengono avviati in modo asincrono, stampando i messaggi *"Inizio task X"*. Successivamente, la funzione *asyncio.sleep()* li sospende per un certo periodo di tempo simulato. Alla fine del periodo di sospensione, verranno stampati i messaggi *"Fine task X"*. Infine, la stampa *"Tutti i task sono stati completati"* indicherà che entrambi i compiti sono stati eseguiti.

Di seguito riportiamo alcuni dei vantaggi dell'utilizzo della programmazione asincrona:

- **Miglioramento delle prestazioni**: la programmazione asincrona permette di sfruttare in modo efficiente le risorse di sistema. Quando si eseguono operazioni di I/O, come la lettura o la scrittura su un file o una richiesta di rete, invece di bloccare l'intero processo in attesa della risposta, si sospende l'esecuzione della coroutine corrente consentendo ad altre coroutines di essere eseguite. Ciò consente di ottimizzare

l'utilizzo della CPU, ridurre i tempi di attesa e migliorare le prestazioni dell'applicazione.

- **Gestione semplificata dei compiti asincroni**: *asyncio* offre un'astrazione per la gestione delle attività asincrone. È possibile definire coroutines che rappresentano compiti specifici e utilizzare l'event loop per coordinare l'esecuzione di tali compiti. Inoltre, *asyncio* fornisce funzionalità come *asyncio.gather()* per eseguire più coroutines contemporaneamente e *asyncio.wait()* per attendere il completamento di un insieme di coroutines. Queste funzionalità semplificano la gestione dei compiti asincroni e consentono di organizzare il codice in modo più strutturato.
- **Scalabilità**: la programmazione asincrona è particolarmente utile per applicazioni che devono gestire un alto volume di richieste concorrenti. L'utilizzo di *coroutines* e dell'*event loop* permette di gestire facilmente numerose richieste simultanee senza creare un grande numero di *thread* o *processi*. Questo porta a un miglioramento delle prestazioni e a un'architettura più scalabile.
- **Facilità di sviluppo**: *asyncio* fornisce un modello di programmazione più semplice rispetto all'utilizzo diretto di *thread* o *processi*. Le coroutines e l'event loop rendono più facile scrivere codice asincrono leggibile e mantenibile. Inoltre, *asyncio* offre funzionalità come la gestione degli errori asincroni, la gestione della concorrenza e la sincronizzazione tra coroutines, semplificando la gestione delle sfide tipiche della programmazione asincrona.
- **Interoperabilità**: *asyncio* è progettato per essere compatibile con altre librerie e framework *Python*. Ad esempio, è possibile utilizzare *asyncio* insieme a librerie come *aiohttp* per lo sviluppo di applicazioni web asincrone.

Introduzione al modulo concurrent.futures

Il modulo *concurrent.futures* fornisce un'interfaccia ad alto livello per eseguire operazioni concorrenti in modo semplice e intuitivo. È stato introdotto nella libreria standard di *Python* a partire dalla versione 3.2.

Il modulo *concurrent.futures* fornisce un'interfaccia ad alto livello per eseguire compiti in modo concorrente utilizzando oggetti chiamati

"*executor*". Questo modulo semplifica la programmazione concorrente, consentendo di sfruttare il potenziale di esecuzione parallela senza dover gestire direttamente i *thread* o i *processi*.

Di seguito sono elencati i concetti chiave del modulo.

- **Executor**: l'executor è un'astrazione che rappresenta un pool di *thread* o *processi*, a seconda del tipo di executor utilizzato. Fornisce un'interfaccia semplice per sottoporre compiti da eseguire in parallelo.
- **Future**: un future rappresenta il risultato di un compito asincrono che potrebbe non essere ancora disponibile. È possibile utilizzare il future per ottenere il risultato di un compito quando è pronto.
- **Task**: un task è un'istanza di un future associato a un compito specifico. Può essere utilizzato per controllare lo stato e i risultati del compito asincrono.
- **ThreadPoolExecutor**: un tipo di executor che utilizza un pool di *thread* per eseguire i compiti in modo concorrente.
- **ProcessPoolExecutor**: un tipo di executor che utilizza un pool di *processi* per eseguire i compiti in modo concorrente.

Utilizzo del modulo concurrent.futures

- **Creazione di un executor**: è possibile creare un executor utilizzando le classi *ThreadPoolExecutor* o *ProcessPoolExecutor*. Si può specificare il numero di *thread* o *processi* desiderato come parametro di inizializzazione.
- **Sottomissione dei compiti**: utilizzando il metodo *submit()* dell'executor, si possono sottoporre i compiti da eseguire in modo concorrente. Il metodo restituisce un future associato al compito, che rappresenta il suo stato e il suo risultato futuro.
- **Ottenimento dei risultati**: per ottenere il risultato di un compito asincrono, si può utilizzare il metodo *result()* sul future corrispondente. Questo metodo blocca l'esecuzione finché il risultato non è disponibile. È anche possibile utilizzare il metodo *add_done_callback()* per specificare una funzione di callback che verrà eseguita una volta che il risultato è pronto.
- **Gestione di più compiti**: *concurrent.futures* fornisce anche il metodo *as_completed()*, che restituisce un iteratore che genera i

futuri man mano che vengono completati. Ciò consente di gestire dinamicamente i risultati dei compiti man mano che diventano disponibili.

Vediamo un esempio di utilizzo del modulo *concurrent.futures* per eseguire compiti in modo concorrente utilizzando un *ThreadPoolExecutor*:

```
import concurrent.futures

def task(name):
    print(f"Task {name} in esecuzione")
    result = name * 2
    return result

if __name__ == "__main__":
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = []
        for i in range(5):
            future = executor.submit(task, i)
            futures.append(future)

        # Ottenimento dei risultati
        for future in concurrent.futures.as_completed(futures):
            result = future.result()
            print(f"Risultato: {result}")
```

Nell'esempio precedente, abbiamo definito una funzione *task()* che prende un nome come argomento, esegue un'operazione e restituisce un risultato. All'interno del blocco *if __name__ == "__main__":* abbiamo creato un *ThreadPoolExecutor* utilizzando il costrutto *with* per garantire la chiusura pulita dell'executor.

Successivamente, abbiamo iterato su un range di valori e per ciascuno abbiamo sottomesso un compito all'executor utilizzando il metodo *submit()*. Questo metodo prende come argomenti la funzione da eseguire e gli eventuali argomenti da passare alla funzione. Restituisce un future associato al compito.

Abbiamo memorizzato tutti i futuri in una lista *futures*. Successivamente, abbiamo utilizzato *as_completed(futures)* per ottenere un iteratore che genera i futuri man mano che vengono completati. Per ciascun futuro, abbiamo utilizzato il metodo *result()* per ottenere il risultato del compito. Infine, abbiamo stampato il risultato ottenuto.

L'utilizzo di *concurrent.futures* semplifica notevolmente la gestione dei compiti concorrenti, consentendo di sottomettere i compiti all'executor e ottenere i risultati in modo semplice e efficiente.

Analisi delle prestazioni di applicazioni concorrenti o parallele

L'analisi delle prestazioni di applicazioni concorrenti o parallele è un aspetto cruciale per valutare l'efficacia e l'efficienza di tali sistemi. Attraverso l'analisi delle prestazioni, è possibile identificare i punti di forza e di debolezza di un'applicazione concorrente o parallela e ottimizzarla per ottenere le massime prestazioni. In questa sezione, esploreremo alcune tecniche e strumenti comuni per analizzare le prestazioni di tali applicazioni.

- **Misurazione dei tempi di esecuzione**: un modo fondamentale per valutare le prestazioni di un'applicazione concorrente o parallela è misurare i tempi di esecuzione dei compiti.
  Si può utilizzare il modulo *time* di *Python* per registrare i tempi di inizio e di fine dei compiti e calcolare la durata totale dell'esecuzione. È possibile confrontare i tempi di esecuzione tra diverse implementazioni o configurazioni.
- **Analisi dell'utilizzo delle risorse**: è importante monitorare l'utilizzo delle risorse come la CPU, la memoria e l'uso di rete per valutare l'efficienza dell'applicazione.
  Si possono utilizzare strumenti di monitoraggio delle risorse come *top* (per Linux), *Task Manager* (per Windows) o librerie specifiche come {*psutil*} in *Python* per ottenere misurazioni quantitative dell'utilizzo delle risorse durante l'esecuzione dell'applicazione. Analizzando i dati

sull'utilizzo delle risorse, si possono individuare potenziali colli di bottiglia o situazioni in cui l'applicazione potrebbe essere sovraccarica.

- **Scalabilità**: l'analisi della scalabilità è importante per determinare come le prestazioni dell'applicazione variano in base al numero di risorse (come *thread* o *processi*) allocate. Si possono effettuare test di carico aumentando gradualmente il numero di compiti o *thread* e misurando i tempi di esecuzione e l'utilizzo delle risorse.
  È importante valutare se l'applicazione riesce a ottenere un miglioramento delle prestazioni in modo proporzionale all'aumento delle risorse allocate o se si verificano limitazioni.
- **Strumenti di profilazione**: gli strumenti di profilazione consentono di analizzare il comportamento dell'applicazione, identificando le parti di codice che richiedono più tempo o risorse.
  Python fornisce vari strumenti di profilazione come il modulo *cProfile* o librerie di terze parti come *line_profiler* o *memory_profiler*.
  Utilizzando questi strumenti, è possibile ottenere una panoramica dettagliata delle prestazioni dell'applicazione, identificare le funzioni più costose in termini di tempo o risorse e ottimizzarle se necessario.
- **Test di stress**: i test di stress sono un'importante fase dell'analisi delle prestazioni di applicazioni concorrenti o parallele. Questi test consentono di valutare le prestazioni dell'applicazione in condizioni di carico estremo e di identificare eventuali limitazioni o degradi delle prestazioni. Durante i test di stress, si possono simulare situazioni in cui l'applicazione è sottoposta a un alto carico di lavoro, ad esempio aumentando il numero di compiti, richieste concorrenti o dati elaborati. Questi test permettono di valutare la stabilità dell'applicazione, la sua capacità di gestire il carico di lavoro e se si verificano eventuali problemi come rallentamenti, crash o errori.

Testing avanzato

Il testing è un'attività fondamentale nello sviluppo del software. Per garantire che il nostro codice funzioni correttamente e senza errori, è importante scrivere test accurati che verifichino il comportamento delle nostre funzioni e classi.

Python fornisce il modulo *unittest*, che offre un framework di testing integrato che semplifica la scrittura e l'esecuzione dei test unitari. In questo capitolo, esploreremo i concetti di base di *unittest* e impareremo come utilizzarlo per testare il nostro codice.

unittest è ispirato al framework di testing JUnit di Java e adotta l'approccio del testing basato su unità. Consiste in una serie di classi e metodi che ci permettono di definire i casi di test, eseguirli e verificare i risultati attesi.

Struttura di un caso di test

In *unittest*, un caso di test è definito come una classe che eredita dalla classe *unittest.TestCase*.

All'interno di questa classe, definiremo metodi per i singoli test.

```
import unittest

class MyTestCase(unittest.TestCase):
    def test_something(self):
        # Codice del test
        pass
```

All'interno del caso di test, i metodi di test devono iniziare con il prefisso "test". Questo permette al framework *unittest* di riconoscere i metodi di test e di eseguirli correttamente.

Asserzioni

Le asserzioni sono il cuore dei test unitari. Ci permettono di verificare che il comportamento del nostro codice sia conforme alle aspettative. {unittest} fornisce una serie di metodi di asserzione che possiamo utilizzare all'interno dei nostri casi di test.

- **assertEqual(a, b)**: verifica che a e b siano uguali.
- **assertTrue(x)**: verifica che x sia True.
- **assertFalse(x)**: verifica che x sia False.
- **assertRaises(exception, callable, *args, **kwargs)**: verifica che *callable* sollevi l'eccezione *exception* quando chiamato con gli argomenti specificati.

Esecuzione dei test

Per eseguire i test, possiamo utilizzare la funzione *unittest.main()*. Questa funzione scopre i casi di test all'interno di un modulo e li esegue. Possiamo eseguire i test direttamente da riga di comando o chiamando la funzione *unittest.main()* nel nostro script.

Ecco un esempio di esecuzione dei test direttamente da riga di comando:

```
python -m unittest test_module.py
```

Possiamo anche utilizzare l'opzione *-v* per ottenere un output più verboso, che mostrerà i dettagli dei test eseguiti.

```
python -m unittest -v test_module.py
```

Esempio

Ecco un esempio completo di come utilizzare il modulo *unittest* per scrivere e eseguire test unitari.

```
import unittest

# Funzione da testare
def add(a, b):
    return a + b

# Classe di test
class TestAddFunction(unittest.TestCase):

    def test_add_positive_numbers(self):
        result = add(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        result = add(-2, -3)
        self.assertEqual(result, -5)

    def test_add_zero(self):
```

```
        result = add(5, 0)
        self.assertEqual(result, 5)

# Esecuzione dei test
if __name__ == '__main__':
    unittest.main()
```

In questo esempio, abbiamo definito una semplice funzione *add()* che somma due numeri.

Successivamente, abbiamo creato una classe di test *TestAddFunction* che eredita da *unittest.TestCase*. All'interno di questa classe, abbiamo definito tre metodi di test: *test_add_positive_numbers()*, *test_add_negative_numbers()*, e *test_add_zero()*.

Ogni metodo di test inizia con il prefisso "test" e contiene una serie di asserzioni per verificare il comportamento della funzione *add()*.

Per eseguire i test, utilizziamo la funzione *unittest.main()* all'interno di un blocco *if __name__ == '__main__':*.

Questo assicura che i test vengano eseguiti solo se lo script viene eseguito direttamente e non se viene importato come modulo in un altro script.

Per eseguire i test, puoi salvare questo codice in un file chiamato ad esempio *test_add.py* e da riga di comando eseguire il seguente comando:

```
python test_add.py
```

Dovresti ottenere un output che indica il numero totale di test eseguiti e se sono passati o falliti.

Per salvare i risultati dei test in un file HTML, puoi utilizzare il modulo *unittest* in combinazione con la libreria *HtmlTestRunner*. Ecco un esempio di come farlo:

```
import unittest
import HtmlTestRunner

# Funzione da testare
```

```
def add(a, b):
    return a + b

# Classe di test
class TestAddFunction(unittest.TestCase):

    def test_add_positive_numbers(self):
        result = add(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        result = add(-2, -3)
        self.assertEqual(result, -5)

    def test_add_zero(self):
        result = add(5, 0)
        self.assertEqual(result, 5)

# Esecuzione dei test e generazione del report HTML
if __name__ == '__main__':
    unittest.main(testRunner=HtmlTestRunner.HTMLTestRunner(\
                output='test_results'))
```

In questo esempio, abbiamo importato il modulo *HtmlTestRunner.HTMLTestRunner* per generare il report HTML.

Nel blocco *if __name__ == '__main__':*, abbiamo chiamato *unittest.main()* specificando il parametro *testRunner=HtmlTestRunner.HTMLTestRunner(output='test_results')* per utilizzare *HtmlTestRunner* come runner dei test e specificare la cartella di output del report HTML.

Una volta eseguito lo script di test, verrà generato un report HTML nella cartella specificata (*test_results*). Il report includerà informazioni sui test eseguiti, i risultati, le asserzioni fallite e altri dettagli.

Assicurati di avere installato la libreria *HTMLTestRunner* nel tuo ambiente *Python*. Puoi installarla utilizzando il comando *pip install html-testRunner*.

Il valore di hash

Il valore di hash in *Python* è un numero intero univoco assegnato a un oggetto che viene calcolato utilizzando la funzione *hash()*.

Il valore di hash è determinato dalla sequenza di bit dell'oggetto e viene utilizzato principalmente per scopi di ottimizzazione e per consentire l'uso di oggetti immutabili come chiavi all'interno di dizionari o elementi all'interno di set.

La funzione *hash()* restituisce un intero di dimensioni fisse, che varia a seconda dell'architettura del sistema. Il valore di hash è garantito essere lo stesso per oggetti che sono uguali in termini di contenuto. Ad esempio, due stringhe con lo stesso contenuto avranno lo stesso valore di hash, mentre due stringhe diverse avranno valori di hash diversi.

Ecco un esempio di utilizzo della funzione *hash()*:

```
stringa1 = "Ciao"
stringa2 = "Ciao"
num = 42

print(hash(stringa1))  # Stampa il valore di hash della stringa1
print(hash(stringa2))  # Stampa il valore di hash della stringa2
                       # (sarà lo stesso di stringa1)
print(hash(num))    # Stampa il valore di hash del numero
```

In questo esempio, entrambe le stringhe *stringa1* e *stringa2* hanno lo stesso contenuto, quindi avranno lo stesso valore di hash. Il numero *num* avrà un valore di hash diverso poiché è un tipo di dato diverso dalle stringhe.

I valori di hash vengono spesso utilizzati come base per l'implementazione di dizionari e set. Ad esempio, i dizionari utilizzano i valori di hash delle loro chiavi per determinare la posizione di archiviazione e consentono un rapido accesso ai valori corrispondenti. Poiché i valori di hash sono unici per oggetti uguali, questa tecnica garantisce un accesso efficiente agli elementi all'interno delle strutture dati.

Un aspetto importante da notare è che gli oggetti mutabili come le liste non possono essere utilizzati come chiavi all'interno di dizionari poiché il loro

valore di hash può cambiare durante il loro ciclo di vita. Solo gli oggetti immutabili, come le tuple o le stringhe, possono essere utilizzati come chiavi in un dizionario.

Esercizi

Esercizio 1: conto bancario

Il compito consiste nell'implementare una classe *BankAccount* che rappresenta un conto bancario. La classe dovrebbe avere i seguenti metodi:

- __init__(self, initial_balance): un metodo inizializzatore che accetta un parametro initial_balance per impostare il saldo iniziale del conto.
- deposit(self, amount): un metodo che permette di depositare una determinata amount nel conto.
- withdraw(self, amount): un metodo che permette di prelevare una determinata amount dal conto.
- get_balance(self): un metodo che restituisce il saldo attuale del conto.

L'obiettivo è implementare correttamente i metodi in modo che le operazioni di deposito e prelievo vengano gestite in modo sicuro e corretto, garantendo la coerenza del saldo. Assicurarsi di utilizzare la mutua esclusione per evitare potenziali race condition quando più thread o processi accedono contemporaneamente al conto bancario.

Successivamente, nel metodo main, creare un'istanza di BankAccount con un saldo iniziale di 1000. Avviare due thread o processi che eseguano attività di deposito e prelievo sul conto bancario in modo concorrente. Assicurarsi di gestire correttamente la sincronizzazione e l'accesso al conto bancario utilizzando blocchi di mutua esclusione appropriati. Al termine dell'esecuzione dei thread o processi, stampare il saldo finale del conto.

Di seguito una possibile implementazione.

```
import multiprocessing
import time

class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance
        self.lock = multiprocessing.Lock()

    def deposit(self, amount):
        with self.lock:
```

```
            self.balance += amount

    def withdraw(self, amount):
        with self.lock:
            if self.balance >= amount:
                self.balance -= amount
            else:
                print("Saldo insufficiente.")

    def get_balance(self):
        return self.balance
```

```
def process_task(account, process_id):
    for _ in range(5):
        time.sleep(1)
        account.deposit(100)
        account.withdraw(50)
        balance = account.get_balance()
        print(f"Processo {process_id}: saldo attuale: {balance}")
```

```
def main():
    # Creazione di un'istanza di BankAccount
    account = BankAccount(initial_balance=1000)

    # Esecuzione di processi
    process1 = multiprocessing.Process(target=process_task, args=(account, 1))
    process2 = multiprocessing.Process(target=process_task, args=(account, 2))
    process1.start()
    process2.start()

    process1.join()
    process2.join()

    # Stampa il saldo finale
```

```
    final_balance = account.get_balance()
    print(f"Saldo finale: {final_balance}")

if __name__ == "__main__":
    main()
```

Esercizio 2: calcolatrice

Il compito consiste nell'implementare una calcolatrice utilizzando lambda functions, classi e oggetti. La classe *Calcolatrice* deve avere un metodo chiamato *calcola* che accetta tre parametri: un'operazione rappresentata da una lambda function, il primo operando e il secondo operando. Il metodo calcola deve restituire il risultato del calcolo eseguito utilizzando la lambda function.

Devi seguire le seguenti indicazioni per completare il task:

- Definisci una classe chiamata *Calcolatrice*.
- All'interno della classe, implementa un metodo chiamato calcola che accetta tre parametri: operazione, a e b.
- Il metodo *calcola* deve eseguire il calcolo utilizzando la lambda function operazione e restituire il risultato.
- All'esterno della classe, crea un'istanza della calcolatrice chiamata calcolatrice.
- Definisci due lambda functions: una per la somma e una per il prodotto.
- Utilizza il metodo *calcola* dell'oggetto calcolatrice per eseguire un calcolo di somma e stampa il risultato.
- Utilizza nuovamente il metodo *calcola* dell'oggetto calcolatrice per eseguire un calcolo di prodotto e stampa il risultato.

```
class Calcolatrice:
    """
    Classe che rappresenta una calcolatrice.

    Metodi:
```

```
    - calcola: esegue un calcolo specifico utilizzando una lambda function.
    """

    def calcola(self, operazione, a, b):
        """
        Esegue un calcolo specifico utilizzando una lambda function.

        Parametri:
        - operazione: una lambda function che rappresenta l'operazione da eseguire.
        - a: il primo operando.
        - b: il secondo operando.

        Ritorna:
        - Il risultato del calcolo.
        """
        return operazione(a, b)
```

```
def main():
    """
    Funzione principale che esegue un esempio di utilizzo di lambda function,
    classi e oggetti per eseguire calcoli con una calcolatrice.
    """
    calcolatrice = Calcolatrice()

    # Esempio di calcolo: somma di due numeri
    somma = lambda x, y: x + y
    risultato_somma = calcolatrice.calcola(somma, 5, 3)
    print("Risultato della somma:", risultato_somma)

    # Esempio di calcolo: prodotto di due numeri
    prodotto = lambda x, y: x * y
    risultato_prodotto = calcolatrice.calcola(prodotto, 4, 2)
    print("Risultato del prodotto:", risultato_prodotto)
```

```
if __name__ == "__main__":
    main()
```

Capitolo 13: Esercizi finali

Esercizi teorici

- **Quali delle seguenti affermazioni descrive correttamente un linguaggio interpretato?**

    1. Il codice sorgente viene convertito in linguaggio macchina prima dell'esecuzione.
    2. Il codice sorgente viene eseguito direttamente senza una fase di compilazione.
    3. Il codice sorgente viene compilato in un file binario eseguibile.
    4. Il codice sorgente viene interpretato da un interprete virtuale.

- **Cosa rappresenta la "shell" in Python?**

    1. Un tipo di dato usato per la manipolazione di stringhe.
    2. Un ambiente di sviluppo integrato (IDE) per Python.
    3. Una modalità interattiva in cui è possibile eseguire comandi Python.
    4. Un modulo Python per l'accesso a funzionalità di sistema operativo.

- **Qual è il modo corretto per scrivere un commento in Python?**

    1. // Questo è un commento
    2. # Questo è un commento
    3. /* Questo è un commento */
    4. <!-- Questo è un commento -->

- **Qual è il principale scopo dell'indentazione del codice?**

1. Migliorare la leggibilità del codice.
2. Risparmiare spazio di archiviazione.
3. Definire i blocchi di codice condizionale.
4. Abilitare la scrittura di commenti nel codice.

❖ **Qual è la differenza tra un ciclo "for" e un ciclo "while"?**

1. Il ciclo "for" viene eseguito un numero predefinito di volte, mentre il ciclo "while" viene eseguito finché una condizione è vera.
2. Il ciclo "for" viene eseguito finché una condizione è vera, mentre il ciclo "while" viene eseguito un numero predefinito di volte.
3. Non ci sono differenze tra il ciclo "for" e il ciclo "while" in Python.
4. Nessuna delle precedenti

❖ **Quale costrutto può essere utilizzato all'interno di un ciclo per passare alla successiva iterazione senza eseguire il resto del codice all'interno del ciclo?**

1. break
2. continue
3. pass
4. Nessuna delle precedenti

❖ **Quale delle seguenti opzioni è un esempio di ciclo "for" che itera su una lista di numeri interi?**

1. for i in range(10):
2. for i in [1, 2, 3, 4, 5]:
3. for i in "Hello":
4. Nessuna delle precedenti

❖ **Quale delle seguenti istruzioni viene utilizzata per interrompere un ciclo in anticipo?**

1. continue
2. return
3. break
4. Nessuna delle precedenti

❖ **Cosa succede se si esegue un ciclo infinito in Python?**

1. Il ciclo viene eseguito per sempre e il programma potrebbe bloccarsi.
2. Il ciclo viene eseguito solo una volta.
3. Il ciclo viene saltato completamente.
4. Nessuna delle precedenti

❖ **Quale delle seguenti opzioni viene utilizzata per eseguire un blocco di codice un numero specifico di volte?**

1. for loop
2. while loop
3. if statement
4. Nessuna delle precedent

❖ **Qual è il valore di "i" dopo l'esecuzione del seguente ciclo "for" in Python?**

```
1. numbers = [1, 2, 3, 4, 5]
2. for i in numbers:
3.     if i == 3:
4.         break
```

1. 3
2. 4
3. 5
4. 2

❖ **Qual è il valore di "i" dopo l'esecuzione del seguente ciclo "while"?**

```
1. i = 0
```

```
2. while i < 5:
3.     i += 1
4. print(i)
```

1. 0
2. 4
3. 5
4. 2

❖ **Qual è l'output del seguente codice?**

```
1. i = 0
2. while i < 5:
3.     if i == 3:
4.         i += 2
5.         continue
6.     print(i)
7.     i += 1
```

1. 0 1 2 3 4
2. 0 1 2 4
3. 0 1 2 5
4. 2

❖ **Quale delle seguenti affermazioni sull'operatore "range" in Python è corretta?**

1. L'operatore "range" restituisce un elenco di numeri interi in ordine decrescente.
2. L'operatore "range" può essere utilizzato solo con cicli "for" e non con cicli "while".
3. L'operatore "range" restituisce un oggetto che rappresenta una sequenza di numeri interi.
4. Nessuna delle precedenti

❖ **Qual è l'output del seguente codice?**

```
1. numbers = [1, 2, 3, 4, 5]
2. for i in range(len(numbers)):
```

```
3.    if numbers[i] % 2 == 0:
4.        continue
5.    print(numbers[i])
```

1. 1 2 3 4 5
2. 1 3 5
3. 2 4
4. 2

❖ **Quale delle seguenti opzioni rappresenta una caratteristica dei dizionari (dict)?**

1. I dizionari sono ordinati in base all'ordine di inserimento degli elementi.
2. I dizionari consentono l'inserimento di elementi duplicati.
3. I dizionari utilizzano indici numerici per accedere ai loro elementi.
4. I dizionari sono strutture dati immutabili.

❖ **Quale delle seguenti strutture dati in Python è più adatta per rappresentare un grafo?**

1. Dizionario (dict)
2. Lista (list)
3. Insieme (set)
4. Nessuno dei precedenti

❖ **Quale delle seguenti affermazioni è vera per un set (insieme)?**

1. Mantiene l'ordine di inserimento degli elementi.
2. Permette l'inserimento di duplicati.
3. Supporta l'indicizzazione dei suoi elementi.
4. Garantisce l'accesso ai suoi elementi in ordine alfabetico.

❖ **Qual è l'obiettivo principale della programmazione a oggetti?**

1. Creare programmi più efficienti

2. Rendere il codice più complesso
3. Organizzare il codice in modo modulare e riutilizzabile
4. Eliminare la necessità di scrivere commenti nel codice

❖ **Cosa è un oggetto in programmazione a oggetti?**

1. Un'istanza di una classe
2. Un'operazione matematica
3. Una variabile globale
4. Un file di codice sorgente

❖ **Cosa sono gli attributi di un oggetto?**

1. Metodi che operano sull'oggetto
2. Dati che descrivono lo stato dell'oggetto
3. Commenti nel codice sorgente
4. Algoritmi di ordinamento

❖ **Cos'è l'incapsulamento in programmazione a oggetti?**

1. Nascondere le informazioni sensibili all'interno di un oggetto
2. Creare oggetti complessi con molte proprietà
3. Scrivere commenti nel codice sorgente
4. Eseguire il codice in una sandbox virtuale

❖ **Qual è la differenza tra un metodo di classe e un metodo di istanza in Python?**

1. Un metodo di classe è chiamato su un'istanza della classe, mentre un metodo di istanza è chiamato sulla classe stessa
2. Un metodo di classe è chiamato sulla classe stessa, mentre un metodo di istanza è chiamato su un'istanza della classe.
3. Non ci sono differenze tra un metodo di classe e un metodo di istanza in Python.
4. Nessuna delle precedenti

❖ **Cosa si intende per ereditarietà multipla in Python?**

1. La capacità di una classe di ereditare da più di una classe madre.
2. La capacità di una classe di ereditare solo da una classe madre.
3. La capacità di una classe di ereditare solo da più di una classe madre.
4. Nessuna delle precedenti

❖ **Quale metodo speciale viene utilizzato per restituire una rappresentazione leggibile di un oggetto?**

1. __str__()
2. __init__()
3. __repr__()
4. Nessuna delle precedenti

❖ **Cosa succede se due classi diverse hanno un metodo con lo stesso nome?**

1. Viene generato un errore durante l'esecuzione del programma.
2. Il metodo della classe madre viene sovrascritto dal metodo della classe figlia.
3. Entrambi i metodi possono essere chiamati senza problemi.
4. Nessuna delle precedenti

❖ **Cosa restituisce il metodo super() all'interno di una classe figlia?**

1. L'istanza della classe madre.
2. L'istanza della classe figlia
3. Il metodo super() non restituisce niente.
4. Nessuna delle precedenti

❖ **Qual è il concetto principale di polimorfismo?**

1. La capacità di una classe di ereditare da più di una classe madre.
2. La capacità di una classe di modificare il comportamento di un metodo ereditato.

3. La capacità di una classe di avere più metodi con lo stesso nome.
4. Nessuna delle precedenti

❖ **Cosa si intende per composizione in programmazione a oggetti?**

1. La capacità di una classe di ereditare da una classe madre.
2. La capacità di una classe di avere un'istanza di un'altra classe come attributo.
3. La capacità di una classe di ereditare da più di una classe madre.
4. Nessuna delle precedenti

❖ **Quale delle seguenti opzioni permette di aprire un file in modalità di scrittura in Python?**

1. open("file.txt", "r")
2. open("file.txt", "w")
3. open("file.txt", "a")
4. open("file.txt", "x")

❖ **Come si chiude un file aperto in Python?**

1. close(file)
2. file.close()
3. end(file)
4. stop(file)

❖ **Qual è la differenza tra le modalità di apertura "r" e "rb" di un file in Python?**

1. "r" apre il file in modalità di lettura testuale, mentre "rb" apre il file in modalità di lettura binaria.
2. "r" apre il file in modalità di lettura binaria, mentre "rb" apre il file in modalità di lettura testuale.
3. Non ci sono differenze tra "r" e "rb".

4. "r" apre il file in modalità di lettura, mentre "rb" apre il file in modalità di scrittura.

❖ **Qual è il modo consigliato per leggere un file di grandi dimensioni in Python?**

1. Utilizzare il metodo *read()* per leggere l'intero file in una singola operazione.
2. Utilizzare il metodo *readline()* per leggere una riga alla volta fino alla fine del file.
3. Utilizzare il metodo *readlines()* per ottenere una lista di tutte le righe del file.
4. Utilizzare un ciclo for con l'oggetto file direttamente, che itera automaticamente sulle righe del file.

❖ **Cosa fa il metodo *seek()* quando viene utilizzato con un oggetto file in Python?**

1. Scrive i dati correnti nel file.
2. Crea un nuovo file nel percorso specificato.
3. Imposta la posizione del puntatore del file su una determinata posizione.
4. Elimina il contenuto del file.

❖ **Cosa fa il blocco *try-except* in Python?**

1. Crea una nuova eccezione personalizzata.
2. Esegue il codice all'interno del blocco *try* e gestisce eventuali eccezioni che si verificano.
3. Ignora completamente qualsiasi errore che si verifica.
4. Termina immediatamente l'esecuzione del programma.

❖ **Qual è la differenza tra il blocco *except* e il blocco *finally* in un blocco *try-except-finally*?**

1. Il blocco *except* viene eseguito solo se si verifica un'eccezione, mentre il blocco *finally* viene sempre eseguito.

2. Il blocco *finally* viene eseguito solo se si verifica un'eccezione, mentre il blocco *except* viene sempre eseguito.
3. Entrambi i blocchi vengono eseguiti solo se si verifica un'eccezione.
4. Entrambi i blocchi vengono sempre eseguiti indipendentemente dal verificarsi o meno di un'eccezione.

❖ **Come si può ottenere l'informazione sull'errore specifico che ha causato un'eccezione in Python?**

1. Utilizzando il blocco *except* senza specificare il tipo di eccezione.
2. Utilizzando il blocco *except* specificando il tipo di eccezione desiderato.
3. Utilizzando il metodo *get_error()* all'interno del blocco *except*.
4. Utilizzando il metodo *traceback()* all'interno del blocco *except*.

❖ **Qual è la funzione utilizzata per formattare una stringa in Python?**

1. format()
2. print()
3. input()
4. len()

❖ **Qual è il modo corretto di formattare una stringa in Python utilizzando f-string (formatted string literals)?**

1. "Hello {name}!"
2. "Hello %s!" % (name)
3. "Hello {}".format(name)
4. "Hello " + name + "!"

❖ **Qual è il risultato dell'espressione regolare r"\d{3}"?**

1. Trova una corrispondenza per qualsiasi carattere numerico.

2. Trova una corrispondenza per tre caratteri numerici consecutivi.
3. Trova una corrispondenza per un carattere numerico seguito da due caratteri alfanumerici.
4. Trova una corrispondenza per qualsiasi carattere alfanumerico.

❖ **Quale di queste espressioni regolari trova solo corrispondenze che iniziano e terminano con la lettera "a"?**

1. r"a.*a"
2. r"^a.*a$"
3. r"a[a-z]*a"
4. r"a[.*]a"

❖ **Quale di queste espressioni regolari trova solo corrispondenze che contengono una sequenza di almeno tre vocali consecutive?**

1. r"[aeiou]{3}"
2. r"(aeiou){3,}"
3. r"[aeiou]+{3}"
4. r"[aeiou]{3,}"

Esercizio finale: Database

Benvenuti nel capitolo dedicato all'esercizio di programmazione *Python* del nostro libro! In questo esercizio, utilizzeremo diverse librerie e concetti avanzati per creare un'applicazione interattiva e gestire dati con facilità.

La libreria *PyQt5* sarà il nostro punto di partenza. *PyQt5* è una libreria *Python* che ci permette di creare interfacce grafiche per le nostre applicazioni. Utilizzeremo le sue funzionalità per creare una *GUI* intuitiva e user-friendly.

La nostra applicazione sarà in grado di gestire dati utilizzando un database *SQLite*. La libreria *sqlite3* ci fornirà gli strumenti necessari per creare, interrogare e aggiornare il nostro database. Faremo uso di *SQL* (Structured Query Language) per eseguire operazioni di lettura e scrittura.

Ma non finisce qui! Durante l'esercizio, affronteremo anche la lettura di dati da file. Esploreremo come aprire, leggere e utilizzare le informazioni presenti in un file di testo o un documento strutturato in un formato specifico, come *JSON*.

Inoltre, metteremo in pratica i principi della programmazione a oggetti. Organizzeremo il nostro codice in classi e oggetti.

Questo ci permetterà di creare una struttura modulare, flessibile e facilmente mantenibile per la nostra applicazione.

Preparatevi ad affrontare un eccitante percorso di apprendimento!

Prima di iniziare, assicuratevi di avere installate le librerie necessarie sul vostro sistema e di avere una conoscenza di base della sintassi di *Python*. Siete pronti? Allora non perdiamo altro tempo e immergiamoci nell'avvincente mondo della programmazione *Python*.

Preparazione dell'ambiente di sviluppo

Prima di avventurarti nella scrittura del codice, è essenziale effettuare il setup adeguato dell'ambiente di sviluppo. Questo include l'installazione delle librerie necessarie e la configurazione dell'ambiente per garantire un flusso di lavoro fluido e privo di errori.

Installazione delle librerie

Per poter utilizzare le funzionalità di *PyQt5*, *sqlite3* e la lettura da file nel tuo progetto *Python*, è necessario installare le relative librerie e le loro dipendenze. Per semplificare questo processo, utilizzeremo un file *requirements.txt* che elenca tutte le librerie necessarie insieme alle versioni specifiche.

Assicurati di avere installato *pip* sul tuo sistema, che è lo strumento di gestione dei pacchetti predefinito per *Python*.

Dopodiché, segui questi passaggi per installare le librerie:

- Apri un editor di testo o un terminale.
- Crea un nuovo file vuoto chiamato {requirements.txt}.
- All'interno del file {requirements.txt}, aggiungi le seguenti righe:

```
PyQt5
```

Il modulo *sqlite3* è incluso nella libreria standard, quindi non è necessario installarlo separatamente. Puoi importarlo direttamente all'interno del tuo script o sessione interattiva.

- Salva il file *requirements.txt* nella directory in cui desideri eseguire l'installazione delle librerie.

Adesso puoi procedere con l'installazione delle librerie utilizzando uno dei seguenti metodi:

Metodo 1: Utilizzo di pip

- Apri un terminale.
- Naviga fino alla directory in cui si trova il file *requirements.txt*.
- Esegui il seguente comando per installare le librerie:

```
pip install -r requirements.txt
```

Questo comando leggerà il file *requirements.txt* e installerà automaticamente tutte le librerie elencate insieme alle loro dipendenze.

Metodo 2: Utilizzo di *pipenv*

Se preferisci utilizzare *pipenv* per creare un ambiente virtuale isolato per il tuo progetto, puoi seguire questi passaggi:

- Assicurati di avere *pipenv* installato sul tuo sistema. Se non l'hai ancora installato, puoi farlo eseguendo il comando ***pip install pipenv*** nel terminale.
- Apri un terminale e naviga fino alla directory in cui si trova il file *requirements.txt*.
- Esegui il seguente comando per creare un nuovo ambiente virtuale e installare le librerie:

```
pipenv install -r requirements.txt
```

Questo comando creerà un ambiente virtuale isolato, installerà le librerie elencate nel file *requirements.txt* e le registra nel file *Pipfile.lock*.

Con uno di questi metodi, dovresti essere in grado di installare correttamente le librerie necessarie, insieme alle loro dipendenze.

Creazione e visualizzazione di un database

Per garantire una struttura chiara e ben organizzata del nostro progetto, abbiamo suddiviso il codice in tre moduli distinti. Questo approccio modulare ci permette di separare le funzionalità specifiche in moduli dedicati, facilitando la manutenzione e il riutilizzo del codice.

Quella riportata di seguito è la struttura del progetto:

```
Database_dati_anagrafici
├── DatabaseBuilder.py
├── DatabaseViewer.py
├── main.py
├── dati_anagrafici.json
└── requirements.txt
```

Vediamo adesso cosa contiene ognuno dei file sopra elencati:

- **DatabaseBuilder.py**: in questo modulo, abbiamo raggruppato tutto il codice relativo alla creazione e alla gestione del database. Qui troviamo la classe *DatabaseBuilder*, che fornisce metodi per connettersi al database, creare tabelle, inserire dati e recuperare informazioni dal database stesso. Questo modulo svolge un ruolo fondamentale nella gestione del nostro database *SQLite3*.
- **DatabaseViewer.py**: in questo modulo concentriamo il codice per la visualizzazione del database utilizzando *PyQt5*. Qui troviamo la classe *DatabaseViewer*, che eredita da *QMainWindow* e fornisce un'interfaccia grafica che utilizza il componente *QTableView* per mostrare i dati del database in modo intuitivo e interattivo.
- **main.py**: modulo principale, abbiamo il codice per l'esecuzione dell'applicazione. Qui creiamo un'istanza della classe *DatabaseViewer* e avviamo l'applicazione utilizzando l'oggetto *QApplication* di *PyQt5*. Questo modulo costituisce il punto di ingresso dell'applicazione e coordina l'avvio e la gestione delle altre funzionalità.
- **dati_anagrafici.json**: contiene informazioni anagrafiche relative a una lista di individui. Ogni individuo è rappresentato da un oggetto *JSON* contenente diversi campi che descrivono i loro dati personali.

  I campi comuni presenti nell'oggetto JSON possono includere:

  - *nome*: il nome dell'individuo.
  - *cognome*: il cognome dell'individuo.
  - *data_nascita*: la data di nascita dell'individuo.
  - *luogo_nascita*: il luogo di nascita dell'individuo.
  - *indirizzo*: l'indirizzo di residenza dell'individuo.
  - *codice_fiscale*: il codice fiscale dell'individuo.
  - *sesso*: il genere dell'individuo.
  - *telefono*: il numero di telefono dell'individuo.
  - *email*: l'indirizzo email dell'individuo.
  - *professione*: la professione dell'individuo.

  Queste informazioni anagrafiche possono essere utilizzate per creare un registro delle persone, gestire un database di utenti o svolgere analisi demografiche, tra le molte altre applicazioni.

- **requirements.txt**: questo file elenca le dipendenze esterne richieste per eseguire correttamente l'applicazione. Queste dipendenze includono librerie, pacchetti o moduli specifici che devono essere installati nel sistema o nell'ambiente virtuale prima dell'esecuzione dell'applicazione.

L'organizzazione del codice in questi tre moduli ci consente di mantenere una struttura chiara e ben suddivisa, consentendo una migliore gestione delle responsabilità e una maggiore riusabilità del codice. Inoltre, ci offre la flessibilità di estendere o modificare specifiche parti senza dover impattare sull'intero progetto.

Ora che abbiamo una panoramica dell'organizzazione del codice, possiamo iniziare a esplorare ciascun modulo nel dettaglio per comprendere meglio come vengono implementate le diverse funzionalità dell'applicazione.

DatabaseBuilder.py

```
"""File: DatabaseBuilder.py"""

# Importa il modulo 'os' per lavorare con funzioni del sistema operativo
import os
# Importa il modulo 'sys' per interagire con l'interprete Python
import sqlite3
# Importa il modulo json per lavorare con dati in formato JSON
```

```
class DatabaseBuilder:
    def __init__(self, database_name=None):
        """Costruttore.

        Args:
            database_name (str): Il nome del database.

        """
        # Nome del database
        self.database_name = database_name
        # Lista per memorizzare le righe del database
```

```
    self.database_rows = []
```

```
def database_connect(self):
    """Questo metodo stabilisce la connessione al database SQLite."""
    # Connettiti al database
    self.connection = sqlite3.connect(self.database_name)

    # Ottieni un cursore per eseguire query
    self.cursor = self.connection.cursor()
```

```
def create_table(self, header=None):
    """
       Questo metodo crea una tabella nel database utilizzando
    un'intestazione specificata.
       L'intestazione contiene i nomi delle colonne e i relativi
    tipi di dati.
    Args:
        header (list): L'intestazione contenente i nomi delle colonne e i
                                relativi tipi di dati.
    """
    # Aggiungi l'estensione ".db" al nome del database
    #se non è presente
    if not self.has_file_extension(self.database_name):
        database = self.database_name + '.db'
    else:
        database = self.database_name

    # Verifica se il database non esiste già
    if not os.path.exists(database):
        try:
            # Crea una connessione al database
            self.connection = sqlite3.connect(database)
```

```
            # Ottieni un cursore per eseguire query
            self.cursor = self.connection.cursor()

            # Costruzione della query CREATE TABLE
            query = f"CREATE TABLE {self.database_name} ("
            for column in header:
                column_name, column_type = column
                query += f"{column_name} {column_type},"

            query = query.rstrip(",") + ")"

            # Esecuzione della query
            self.cursor.execute(query)

        except sqlite3.Error as e:
            print("Error connecting to the database:", e)
    else:
        print("Database already exists")
        # Connettiti al database esistente
        self.connection = sqlite3.connect(database)
        # Ottieni un cursore per eseguire query
        self.cursor = self.connection.cursor()

    # Conferma delle modifiche al database
    self.connection.commit()
```

```
def fetch_all(self):
    """Questo metodo esegue una query SQL per recuperare
    tutte le righe dal database."""
    # Costruzione della query
    query = "SELECT * FROM {database_name}".format(\
    database_name=self.database_name)

    # Esecuzione della query
```

```
    self.cursor.execute(query)

    # Memorizzazione delle righe recuperate dal database
    self.database_rows = self.cursor.fetchall()
```

```
def insert_elements(self, elements):
    """
    Questo metodo inserisce gli elementi forniti nel database.
       Se viene fornita una lista di elementi, verifica se ciascun
    elemento è già presente nel database.
       Se un elemento non è presente, viene inserito nel database
    utilizzando il metodo insert_element().

    Args:
        elements (list or tuple): Gli elementi da inserire nel database.
    """
    # Recupera tutte le righe dal database
    self.fetch_all()

    # Verifica se l'input 'elements' è di tipo list
    if isinstance(elements, list):
        for element in elements:
            # Verifica se l'elemento è già presente nel database
            if element not in self.database_rows:
                print(f'L'elemento {str(element)} \
                non è presente nel database: aggiungiamolo!")
                # Inserisce l'elemento nel database utilizzando
                # il metodo insert_element()
                self.insert_element(element)
            else:
                print(f'L'elemento {str(element)} è già presente \
                nel database")
    else:
```

```
        print('Hai fornito un input di tipo non supportato!')
```

```
def insert_element(self, row_values):
    """Questo metodo inserisce una singola riga di valori
    nel database.

    Args:
        row_values (tuple): I valori della riga da inserire nel database.
    """
    # Verifica se l'input 'elements' è di tipo tuple
    if isinstance(row_values, tuple):
        # Costruzione della query INSERT
        query = "INSERT INTO {} VALUES ({})".format(\
        self.database_name, ",".join(["?"] * len(row_values)))
        # Esecuzione della query INSERT con i valori della riga
        self.cursor.execute(query, row_values)
        # Conferma delle modifiche al database
        self.connection.commit()
    else:
        print('Hai fornito un elemento di tipo non supportato!')


def has_file_extension(self, filename):
    """
    Questo metodo verifica se una stringa ha un'estensione di file.
    Restituisce True se l'estensione è presente, False altrimenti.
    Args:
        filename (str): Il nome del file da controllare.

    Returns:
        bool: True se l'estensione è presente, False altrimenti.
    """
    # Controlla se la stringa termina con un'estensione di file
```

```
    # Ad esempio, ".db" indica che è presente un'estensione di file
    return '.' in filename and filename.rsplit('.', 1)[1].lower() \
    != 'db'
```

DatabaseViewer.py

```
"""File: DatabaseViewer.py"""

# Importa specifici widget da PyQt5 per la creazione
# dell'interfaccia grafica
from PyQt5.QtWidgets import QWidget, \
                        QApplication, \
                        QMainWindow, \
                        QTableView, \
                                QVBoxLayout
# Importa la classe QIcon da PyQt5 per gestire le icone
from PyQt5.QtGui import QIcon
# Importa il modulo QtSql da PyQt5 per lavorare con database
from PyQt5.QtSql import QSqlTableModel, QSqlDatabase
```

```
class DataBaseViewer(QMainWindow):
  def __init__(self, database_name=None):
    """Costruttore.

    Args:
      database_name (str): Il nome del database.
    """
    super().__init__()
    try:
      # Attributi
      self.database_name = database_name

      # Imposta l'icona della finestra
```

```
        self.setWindowIcon(QIcon("percorso_alla_tua_icona.ico"))

        # Imposta il titolo della finestra
        self.setWindowTitle("DataBase")

        # Imposta la geometria della finestra
        self.setGeometry(1000, 1000, 1000, 1000)

        # Crea un widget centrale
        self.central_widget = QWidget(self)

        # Crea un layout verticale per il widget centrale
        layout = QVBoxLayout(self.central_widget)

        # Crea il QTableView
        self.table_view = QTableView()

        # Aggiungi il QTableView al layout
        layout.addWidget(self.table_view)

        # Imposta il widget centrale come widget centrale
        # della finestra
        self.setCentralWidget(self.central_widget)
    except:
        print("Errore identificato durante l'inizializzazione!")
        return

def set_database_name(self, name):
    """
      Questo metodo imposta il nome del database utilizzato
    dall'applicazione.

    Args:
        name (str): Il nome del database da impostare.
    """
```

```
    self.database_name = name

def get_database_name(self):
    """
        Questo metodo restituisce il nome del database utilizzato
    dall'applicazione.

    Returns:
        str: Il nome del database.
    """
    return self.database_name

def move_to_middle_of_screen(self):
    """
        Questo metodo di permette di spostare la finestra principale
    nel centro del tuo desktop.
        Recupera le dimensioni (altezza e spessore) dello screen e
    della finestra, dopodichè
    calcola la posizione (x,y) in cui centrare la finestra.
    """
    # Ottiene il desktop dell'applicazione
    desktop = QApplication.desktop()

    # Ottiene la geometria dello schermo
    screen_rect = desktop.screenGeometry()

    # Ottiene la geometria della finestra
    window_rect = self.geometry()

    # Calcola la posizione (x,y) per il centro
    x = (screen_rect.width() - window_rect.width()) // 2
    y = (screen_rect.height() - window_rect.height()) // 2

    # Sposta la finestra nella posizione calcolata
```

```
    self.move(x, y)

def add_database(self):
    """
        Questo metodo aggiunge un database SQLite alla finestra
    principale.
        Viene utilizzato il driver QSQLITE per creare un'istanza
    di QSqlDatabase
        e viene impostato il nome del database. Successivamente,
    il database viene aperto.
    """
    # Crea un'istanza di QSqlDatabase con il driver QSQLITE
    self.db = QSqlDatabase.addDatabase("QSQLITE")

    # Imposta il nome del database
    self.db.setDatabaseName(self.database_name)

    # Apre il database
    self.db.open()

def create_model(self):
    """
        Questo metodo crea un modello QSqlTableModel e visualizza
    i dati all'interno di un QTableView.
        Il modello viene collegato al database self.db e la tabella
    viene impostata come il nome del database senza estensione.
        Successivamente, il modello viene selezionato per ottenere
    i dati dalla tabella.
        Infine, viene creato un'istanza di QTableView e il modello
    viene impostato come modello per la visualizzazione.
    """
    # Crea un'istanza di QSqlTableModel con il database self.db
    self.model = QSqlTableModel(None, self.db)
```

```
    # Imposta la tabella del modello come il nome del database
    # senza estensione
    self.model.setTable(self.database_name.replace('.db', ''))

    # Seleziona il modello per ottenere i dati dalla tabella
    self.model.select()

    # Crea un'istanza di QTableView
    self.table_view = QTableView()

    # Imposta il modello come modello per la visualizzazione
    # nel QTableView
    self.table_view.setModel(self.model)

def print_model_content(self):
    """
    Questo metodo stampa i contenuti del modello QSqlTableModel.
       Itera su ogni cella del modello e stampa il dato
    in essa contenuto.
    """
    # Ottiene il numero di righe e colonne del modello
    row_count = self.model.rowCount()
    column_count = self.model.columnCount()
    for row in range(row_count):
        for column in range(column_count):
            # Ottiene l'indice per una specifica cella nel modello
            index = self.model.index(row, column)
            # Ottiene il dato nella cella corrispondente all'indice
            data = self.model.data(index)
            # Stampa il dato
            print(data)

def add_table_to_window(self):
    """
```

```
        Questo metodo aggiunge la vista tabellare QTableView come
        widget centrale della finestra principale
    e ne imposta le proprietà.
    """
    # Imposta la vista tabellare come widget centrale
    # della finestra principale
    self.setCentralWidget(self.table_view)

    # Imposta le proprietà della vista tabellare
    self.set_table_properties()

def set_table_properties(self):
    """
        Questo metodo imposta le proprietà della vista
    tabellare QTableView.
        Vengono impostate le etichette degli headers,
    lo stile degli headers, l'estensione dell'ultimo header,
        il comportamento di modifica, il comportamento
    di selezione delle righe, la modalità di scorrimento verticale
    e le dimensioni della vista tabellare.
    """
    # Imposta le etichette degli header con uno stile specifico
    self.table_view.horizontalHeader().setStyleSheet(\
    "QHeaderView::section { background-color: lightblue; \
    font-weight: bold; }")
    self.table_view.verticalHeader().setStyleSheet(\
    "QHeaderView::section { background-color: lightblue; \
    font-weight: bold; }")

    # Imposta l'estensione dell'ultimo header
    self.table_view.horizontalHeader().setStretchLastSection(True)

    # Imposta il comportamento di modifica della vista
    # tabellare a NoEditTriggers (disabilita la modifica)
```

```
        self.table_view.setEditTriggers(QTableView.NoEditTriggers)

        # Imposta il comportamento di selezione delle righe della vista
        # tabellare a SelectRows (seleziona intere righe)
        self.table_view.setSelectionBehavior(QTableView.SelectRows)

        # Imposta la modalità di selezione della vista tabellare
        # a SingleSelection (seleziona una singola riga alla volta)
        self.table_view.setSelectionMode(QTableView.SingleSelection)

        # Imposta la modalità di scorrimento verticale della vista
        # tabellare a ScrollPerPixel (scorrimento verticale fluido)
        self.table_view.setVerticalScrollMode(QTableView.ScrollPerPixel)

        # Adatta le dimensioni delle colonne della vista tabellare
        # al contenuto
        self.table_view.resizeColumnsToContents()

        # Adatta le dimensioni delle righe della vista tabellare
        # al contenuto
        self.table_view.resizeRowsToContents()
```

main.py

```
"""File: main.py"""

# Importa i moduli
import sys
import json
from PyQt5.QtWidgets import QApplication
from DatabaseBuilder import DatabaseBuilder
from DatabaseViewer import DataBaseViewer

if __name__ == "__main__":
    app = QApplication([])
```

```
# Leggi dati anagrafici da file (.json)
filename = 'dati_anagrafici.json'
elements = []
with open(filename, "r") as f:
    # Carica il contenuto del file JSON
    dati_anagrafici = json.load(f)
    for element in dati_anagrafici:
        elements.append(tuple(element.values()))

database_name = 'dati_anagrafici'
database = database_name + '.db'

# Crea database
instance_db_builder = DatabaseBuilder(database_name)
# Creo l'header del database
# (partendo dalle chiavi del dizionario dei dati)
header = []
for key in dati_anagrafici[0].keys():
    header.append((key,'TEXT'))
instance_db_builder.create_table(header)
instance_db_builder.insert_elements(elements)

# Visualizza database
window = DataBaseViewer(database)
window.add_database()
window.create_model()
window.add_table_to_window()
window.move_to_middle_of_screen()
window.show()
sys.exit(app.exec_())
```

	nome	cognome	data_nascita	luogo_nascita	indirizzo	codice_fiscale	sesso	telefono	email	professione
1	Mario	Rossi	1985-07-15	Roma	Via Roma 123	RSSMRA85L15H...	M	3331234567	mario.rossi@exa...	Ingegnere
2	Laura	Bianchi	1990-03-21	Milano	Via Milano 456	BNCCHR90C21F...	F	3339876543	laura.bianchi@e...	Avvocato
3	Luigi	Verdi	1978-11-02	Napoli	Via Napoli 789	VRDLGI78S02I23...	M	3335554444	luigi.verdi@exa...	Medico
4	Giulia	Russo	1995-09-08	Palermo	Via Palermo 567	RSSGLI95P08G7...	F	3331112222	giulia.russo@ex...	Architetto

Figure 1 Output main.py

Risposte agli esercizi teorici

1. Risposta corretta: b. Il codice sorgente viene eseguito direttamente senza una fase di compilazione.
2. Risposta corretta: c. Una modalità interattiva in cui è possibile eseguire comandi Python.
3. Risposta corretta: b. # Questo è un commento
4. Risposta corretta: a. Migliorare la leggibilità del codice.
5. Risposta corretta: a. Il ciclo "for" viene eseguito un numero predefinito di volte, mentre il ciclo "while" viene eseguito finché una condizione è vera.
6. Risposta corretta: b. continue
7. Risposta corretta: b. for i in [1, 2, 3, 4, 5]:
8. Risposta corretta: c. break

9. Risposta corretta: a. Il ciclo viene eseguito per sempre e il programma potrebbe bloccarsi.
10. Risposta corretta: b. while loop
11. Risposta corretta: a. 3
12. Risposta corretta: c. 5
13. Risposta corretta: b. 0 1 2 4
14. Risposta corretta: c. L'operatore "range" restituisce un oggetto che rappresenta una sequenza di numeri interi.
15. Risposta corretta: b. 1 3 5
16. Risposta corretta: a. I dizionari sono ordinati in base all'ordine di inserimento degli elementi.
17. Risposta corretta: a. Dizionario (dict)
18. Risposta corretta: b. Permette l'inserimento di duplicati.
19. Risposta corretta: c. Organizzare il codice in modo modulare e riutilizzabile
20. Risposta corretta: a. Un'istanza di una classe
21. Risposta corretta: b. Dati che descrivono lo stato dell'oggetto
22. Risposta corretta: a. Nascondere le informazioni sensibili all'interno di un oggetto
23. Risposta corretta: b. Un metodo di classe è chiamato sulla classe stessa, mentre un metodo di istanza è chiamato su un'istanza della classe.
24. Risposta corretta: a. La capacità di una classe di ereditare da più di una classe madre.
25. Risposta corretta: a. __str__()
26. Risposta corretta: b. Il metodo della classe madre viene sovrascritto dal metodo della classe figlia.
27. Risposta corretta: a. L'istanza della classe madre.
28. Risposta corretta: b. La capacità di una classe di modificare il comportamento di un metodo ereditato.
29. Risposta corretta: b. La capacità di una classe di avere un'istanza di un'altra classe come attributo.
30. Risposta corretta: b. open("file.txt", "w")
31. Risposta corretta: b. file.close()
32. Risposta corretta: a. "r" apre il file in modalità di lettura testuale, mentre "rb" apre il file in modalità di lettura binaria.

33. Risposta corretta: d. Utilizzare un ciclo for con l'oggetto file direttamente, che itera automaticamente sulle righe del file.
34. Risposta corretta: c. Imposta la posizione del puntatore del file su una determinata posizione.
35. Risposta corretta: b. Esegue il codice all'interno del blocco try e gestisce eventuali eccezioni che si verificano.
36. Risposta corretta: a. Il blocco *except* viene eseguito solo se si verifica un'eccezione, mentre il blocco *finally* viene sempre eseguito.
37. Risposta corretta: b. Utilizzando il blocco *except* specificando il tipo di eccezione desiderato.
38. Risposta corretta: a. *format()*
39. Risposta corretta: a. "Hello {name}!"
40. Risposta corretta: b. Trova una corrispondenza per tre caratteri numerici consecutivi.
41. Risposta corretta: b. r"^a.*a$"
42. Risposta corretta: d. r"[aeiou]{3,}"

Se pensi che questo libro ti sia piaciuto e ti abbia aiutato ti chiedo solo dedicare pochi secondi a lasciare una breve recensione su Amazon. Questo è un sostegno fondamentale per noi autori.