PYTHON HANDBOOK PER PRINCIPIANTI

LE BASI DEL LINGUAGGIO E UNA PANORAMICA GENERALE

Daniele Avolio

Indice

Capitolo 0

Chi sono?

Ciao! Sono *Daniele Avolio*, studente che durante la scrittura di questo documento sta per terminare la sua *Laurea Magistrale* Computer Science & Artificial Intelligence presso l'Università della Calabria! Sono un Laureato Triennale in Informatica e ho esperienza come sviluppatore web con la passione per il Machine Learning e in generale la Data Science. Potete trovare altre informazioni su di me sul mio LinkedIn o il mio sito web www.danieleavolio.it.

Perché questo documento?

Questo documento è stato scritto per aiutare chiunque voglia imparare a programmare in Python, partendo dalle basi e arrivando a concetti più avanzati. Python è un linguaggio di programmazione molto popolare e versatile, utilizzato in una vasta gamma di settori, tra cui sviluppo web, intelligenza artificiale, analisi dei dati, automazione e molto altro. Questo documento è stato scritto con l'obiettivo di fornire una guida completa e dettagliata alla programmazione in Python, con esempi pratici e spiegazioni chiare dei concetti fondamentali.

Cosa troverai in questo documento?

In questo documento troverai una serie di capitoli che coprono i concetti fondamentali della programmazione in Python, inclusi i tipi di dati, le strutture di controllo, le funzioni, le classi e molto altro. Ogni capitolo è progettato per essere letto in modo sequenziale, ma è possibile saltare tra i capitoli in base alle proprie esigenze e interessi. Ogni capitolo include spiegazioni dettagliate dei concetti, esempi pratici e suggerimenti per approfondire ulteriormente gli argomenti trattati.

Molte parti di questo documento sono state scritte utilizzando *LLM* (Large Language Model) per generare testo in modo automatico. Questo documento è stato scritto con l'obiettivo di fornire una guida completa e dettagliata alla programmazione in Python, con esempi pratici e spiegazioni chiare dei concetti fondamentali.

A chi è rivolto questo documento?

Questo documento è rivolto a chiunque voglia imparare a programmare in Python, indipendentemente dal livello di esperienza. Se sei un principiante assoluto, troverai una guida passo passo per imparare i concetti di base della programmazione in Python. Se sei già un programmatore esperto, troverai suggerimenti e approfondimenti su argomenti più avanzati e complessi. In ogni caso, questo documento è progettato per essere una risorsa utile e completa per chiunque voglia imparare a programmare in Python.

Capitolo 1: Introduzione alla Programmazione

1.1 Cos'è la programmazione?

La programmazione è l'arte di scrivere istruzioni per un computer in modo che possa eseguire determinate operazioni. Un programma è una sequenza di istruzioni scritte in un linguaggio di programmazione specifico, che viene tradotto in un linguaggio macchina comprensibile dal computer. La programmazione è una disciplina fondamentale nell'era digitale, poiché consente di

creare software, applicazioni e sistemi informatici che svolgono una vasta gamma di compiti e funzioni.

1.2 Perché imparare a programmare?

L'apprendimento della programmazione offre numerosi vantaggi e opportunità, tra cui:

- **Sviluppo di competenze tecniche**: La programmazione aiuta a sviluppare competenze tecniche e logiche che sono utili in molte aree della vita e del lavoro.
- **Creatività e innovazione**: La programmazione consente di creare nuovi prodotti, servizi e soluzioni innovative.
- **Opportunità di carriera**: Le competenze di programmazione sono molto richieste nel mercato del lavoro attuale e offrono numerose opportunità di carriera.
- **Problem solving**: La programmazione aiuta a sviluppare capacità di problem solving e pensiero critico.
- **Automazione e efficienza**: La programmazione consente di automatizzare compiti ripetitivi e migliorare l'efficienza dei processi.

1.3 Breve storia dei linguaggi di programmazione

I linguaggi di programmazione sono stati sviluppati per semplificare la scrittura di codice e rendere più accessibile la programmazione ai non esperti. Alcuni dei linguaggi di programmazione più noti e influenti includono:

- **Fortran**: sviluppato negli anni '50 per il calcolo scientifico.
- **Cobol**: sviluppato negli anni '60 per l'elaborazione dei dati aziendali.
- **C**: sviluppato negli anni '70 come linguaggio di programmazione di sistema.
- **Java**: sviluppato negli anni '90 come linguaggio di programmazione multipiattaforma.
- **Python**: sviluppato negli anni '90 come linguaggio di programmazione ad alto livello e facile da imparare.

1.4 Perché Python?

Python è uno dei linguaggi di programmazione più popolari e utilizzati al mondo, grazie alla sua semplicità, flessibilità e potenza. Python è noto per la sua sintassi chiara e leggibile, che lo rende ideale per principianti e esperti. Python è ampiamente utilizzato in una vasta gamma di settori, tra cui sviluppo web, intelligenza artificiale, analisi dei dati, automazione, sicurezza informatica e molto altro.

Python ha una vasta libreria standard e una comunità attiva di sviluppatori che contribuiscono a una vasta gamma di moduli e framework. Inoltre, Python è un linguaggio versatile che può essere utilizzato per sviluppare applicazioni desktop, web e mobile, nonché per l'elaborazione dei dati, l'automazione dei processi e molto altro.

Capitolo 2: Problemi, Algoritmi e Programmi

2.1 Il concetto di Algoritmo

Un algoritmo è una sequenza di istruzioni ben definite e ordinate che risolve un determinato problema o compito. Gli algoritmi sono alla base della programmazione e della risoluzione dei problemi informatici.

2.1.1 Definizione e caratteristiche di un algoritmo

Un algoritmo deve essere:

- **Definito**: deve essere preciso e ben definito, in modo che possa essere eseguito senza ambiguità.

- **Finito**: deve terminare dopo un numero finito di passaggi.
- **Efficiente**: deve risolvere il problema in modo efficiente, utilizzando un numero ragionevole di risorse.
- **Generale**: deve essere applicabile a una vasta gamma di casi e input.
- **Corretto**: deve produrre il risultato corretto per tutti gli input validi.
- **Deterministico**: deve produrre lo stesso risultato per lo stesso input.
- **Modulare**: deve essere suddiviso in parti più piccole e gestibili.
- **Riproducibile**: deve produrre lo stesso risultato per lo stesso input in ogni esecuzione.
- **Scalabile**: deve funzionare in modo efficiente anche per input di grandi dimensioni.

2.1.2 Esempi di algoritmi nella vita quotidiana

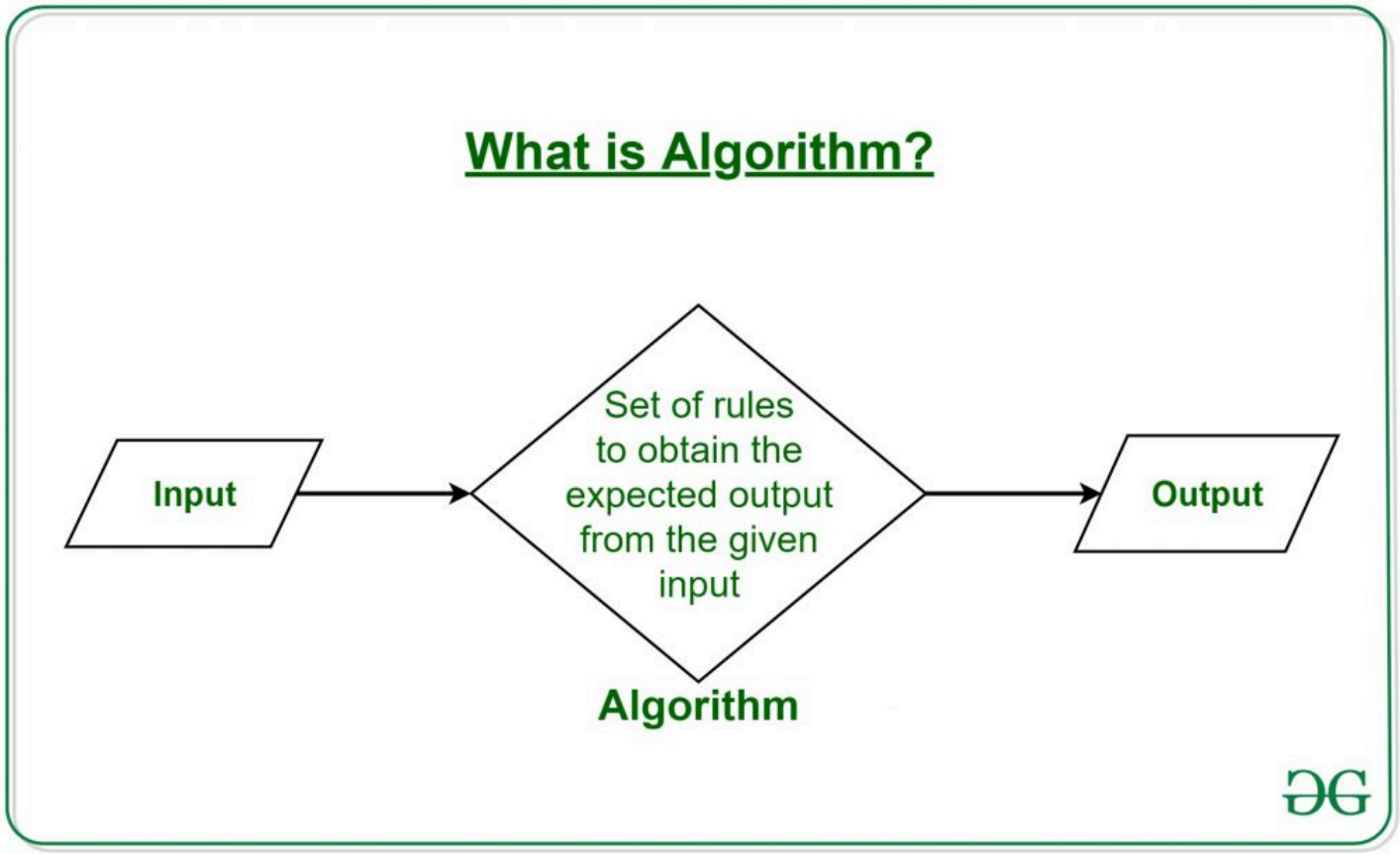
Banalmente, un algoritmo per fare un panino potrebbe essere:

```
1. Prendi due fette di pane.
2. Metti una fetta di formaggio su una fetta di pane.
3. Metti una fetta di prosciutto sulla fetta di formaggio.
4. Metti l'altra fetta di pane sopra il prosciutto.
5. Schiaccia il panino leggermente.
6. Mangia il panino.
```

Un algoritmo per fare la spesa potrebbe essere:

```
1. Prendi una lista della spesa.
2. Vai al supermercato più vicino.
3. Prendi un carrello.
4. Segui la lista della spesa e prendi gli articoli necessari.
5. Vai alla cassa e paga.
6. Torna a casa con la spesa.
```

Ecco, questi sono due esempi di algoritmi semplici che risolvono problemi comuni nella vita quotidiana. Quando si tratta di programmazione, gli algoritmi sono spesso più complessi e dettagliati, ma il concetto di base rimane lo stesso: una sequenza di istruzioni ben definite per risolvere un problema specifico.



2.2 Risoluzione algoritmica dei problemi

Quando si affronta un problema di programmazione, è importante seguire un processo strutturato per risolverlo in modo efficace. Questo processo può essere suddiviso in diverse fasi:

1. **Analisi del problema**: comprendere il problema, identificare i requisiti e definire i vincoli.
2. **Progettazione dell'algoritmo**: sviluppare un algoritmo che risolva il problema in modo efficiente e corretto.
3. **Implementazione dell'algoritmo**: tradurre l'algoritmo in un linguaggio di programmazione specifico.
4. **Test e debug**: verificare che il programma funzioni correttamente e correggere eventuali errori.

Chiaramente, per un principiante, la fase di progettazione dell'algoritmo è la più difficile, poiché richiede una buona comprensione del problema e delle tecniche di risoluzione. Tuttavia, con la pratica e l'esperienza, è possibile sviluppare competenze nel progettare algoritmi efficaci e ottimizzati.

2.3 Linguaggi di Programmazione

Ci sono davvero infiniti linguaggi di programmazione, ognuno con le proprie caratteristiche, vantaggi e svantaggi. Suddividiamo i linguaggi di programmazione in due categorie principali: linguaggi di alto livello e linguaggi di basso livello.

2.3.1 Linguaggi di alto e basso livello

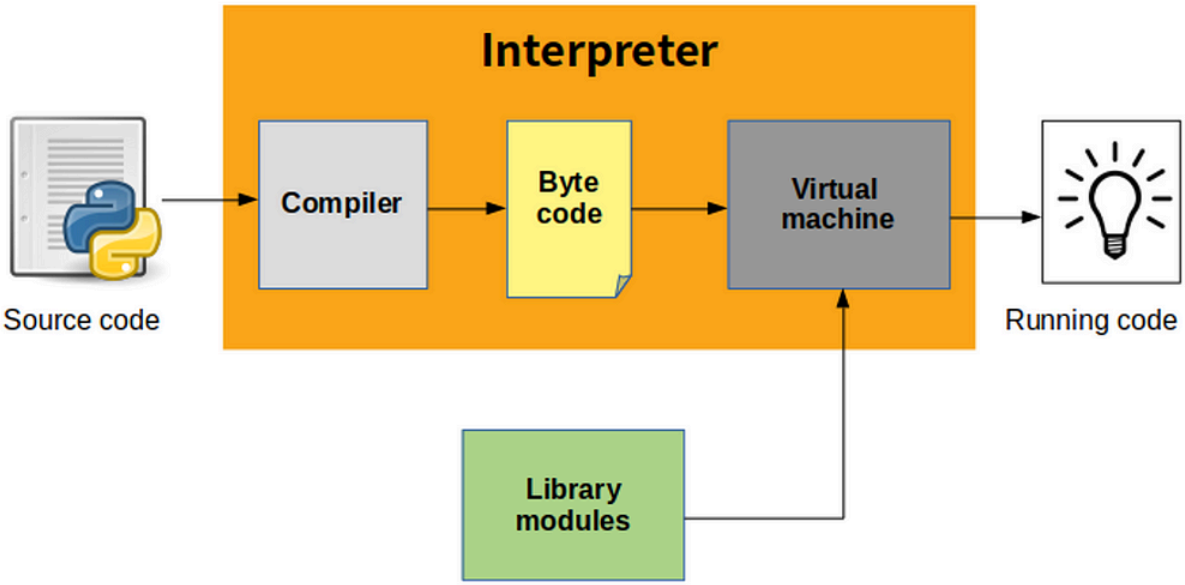
- **Linguaggi di alto livello**: sono linguaggi di programmazione che sono progettati per essere facili da leggere, scrivere e comprendere per gli esseri umani. Questi linguaggi sono più astratti e vicini al linguaggio naturale, il che li rende più accessibili ai principianti e ai non esperti. Esempi di linguaggi di alto livello includono Python, Java, C#, JavaScript, Ruby e molti altri.
- **Linguaggi di basso livello**: sono linguaggi di programmazione che sono più vicini al linguaggio macchina e richiedono una maggiore comprensione del funzionamento interno del computer. Questi linguaggi sono più difficili da leggere e scrivere, ma offrono un maggiore controllo sulle risorse del sistema. Esempi di linguaggi di basso livello includono Assembly, C e C++.

2.3.2 Compilatori e interpreti

- **Compilatori**: sono programmi che traducono il codice sorgente in un linguaggio di programmazione in codice macchina eseguibile. Il codice compilato può essere eseguito indipendentemente dall'ambiente di sviluppo in cui è stato creato. Esempi di linguaggi compilati includono C, C++, Java e Rust.
- **Interpreti**: sono programmi che eseguono il codice sorgente direttamente, senza la necessità di una fase di compilazione. Gli interpreti eseguono il codice riga per riga e forniscono feedback immediato sull'output. Esempi di linguaggi interpretati includono Python, JavaScript, Ruby e PHP.

2.3.3 Panoramica dei linguaggi più diffusi

- **Python**: è un linguaggio di programmazione ad alto livello, interpretato e multiparadigma, noto per la sua sintassi chiara e leggibile. Python è ampiamente utilizzato nello sviluppo web, nell'analisi dei dati, nell'intelligenza artificiale e in molte altre aree.
- **Java**: è un linguaggio di programmazione ad alto livello, compilato e multipiattaforma, noto per la sua portabilità e robustezza. Java è ampiamente utilizzato nello sviluppo di applicazioni enterprise, web e mobili.
- **JavaScript**: è un linguaggio di programmazione ad alto livello, interpretato e orientato agli oggetti, noto per la sua utilizzo nel web development. JavaScript è ampiamente utilizzato per lo sviluppo di siti web interattivi e dinamici.
- **C++**: è un linguaggio di programmazione ad alto livello, compilato e orientato agli oggetti, noto per la sua efficienza e flessibilità. C++ è ampiamente utilizzato nello sviluppo di sistemi operativi, videogiochi e applicazioni ad alte prestazioni.



2.4 Programmazione in Python - i primi passi

`Python` è un linguaggio di programmazione ad alto livello, interpretato e multiparadigma, noto per la sua sintassi chiara e leggibile. Python è ampiamente utilizzato nello sviluppo web, nell'analisi dei dati, nell'intelligenza artificiale e in molte altre aree.

Ora, perché è conveniente imparare Python? Beh, principalmente perché è molto facile da imparare e da usare, grazie alla sua sintassi chiara e leggibile. Inoltre, Python ha una vasta libreria standard e una comunità attiva di sviluppatori che contribuiscono a una vasta gamma di moduli e framework. Python è un linguaggio versatile che può essere utilizzato per sviluppare applicazioni desktop, web e mobile, nonché per l'elaborazione dei dati, l'automazione dei processi e molto altro. Per chi inizia a programmare, Python è forse la scelta più adatta perché permette di concentrarsi sui concetti di base della programmazione senza doversi preoccupare di dettagli complessi come una sintassi più ostica o una gestione della memoria più complessa come nel caso di `c` o `c++`.

2.4.1 Il concetto di programma

In **Python** un programma è una sequenza di istruzioni che vengono eseguite dall'interprete Python. Un programma Python può essere costituito da una o più istruzioni, che vengono eseguite in sequenza dall'alto verso il basso. Le istruzioni possono includere assegnazioni di variabili, chiamate a funzioni, cicli, condizioni e molto altro. Ecco un esempio di un semplice programma Python che stampa "Ciao, mondo!":

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

In questo esempio, `print("Ciao, mondo!")` è un'istruzione che stampa il messaggio "Ciao, mondo!" sulla console. Quando si esegue questo programma, l'interprete Python eseguirà l'istruzione e visualizzerà il messaggio di output. Ora, questo solitamente si esegue in un interprete Python

interattivo, ma è possibile scrivere il codice in un file di testo con estensione `.py` e poi eseguirlo da riga di comando con il comando `python nomefile.py`.

```
$ python hello.py
Ciao, mondo!
```

L'interprete controllerà il file `hello.py`, eseguirà il codice e visualizzerà l'output sulla console. Se ci dovessero essere più righe di codice, l'interprete eseguirà le istruzioni in sequenza dall'alto verso il basso.

2.4.2 Variabili e tipi di dati

Una nota dolente per chi arriva da altri linguaggi di programmazione è che in Python non è necessario dichiarare esplicitamente il tipo di una variabile. Python è un linguaggio a tipizzazione dinamica, il che significa che il tipo di una variabile è determinato automaticamente in base al valore assegnato. Ad esempio, in Python è possibile assegnare un valore intero a una variabile e successivamente assegnare una stringa allo stesso nome di variabile senza generare errori.

```
numero = 42
print(numero)  # Output: 42

numero = "quarantadue"
print(numero)  # Output: quarantadue
```

A lungo andare, questa flessibilità può essere molto comoda, ma può anche portare a errori difficili da individuare. È importante prestare attenzione al tipo di dati assegnato a una variabile e assicurarsi che sia coerente con il contesto in cui viene utilizzata. Nonoostante ciò, Python supporta i tipi di dati più comuni come interi, float, stringhe, booleani, liste, tuple, dizionari e molto altro. E' possibile indicare il tipo di una variabile, cosa che può essere utile per rendere il codice più chiaro e leggibile.

```
# Dichiarazione di variabili con tipi specifici
intero: int = 42

reale: float = 3.14

stringa: str = "Ciao, mondo!"

booleano: bool = True
```

2.4.3 Istruzioni e sintassi di base

A differenza di altri linguaggi di programmazione non è necessario terminare le istruzioni con un punto e virgola `;` in Python. Le istruzioni vengono separate automaticamente da una nuova riga o da un carattere di fine riga. Oltre a ciò, Python utilizza l'indentazione per definire i blocchi di codice, il

che significa che è necessario allineare correttamente le istruzioni all'interno di un blocco. Questo rende il codice più leggibile e strutturato, ma può anche causare errori se l'indentazione non è corretta. Non compaiono quindi le parentesi graffe `{}` come in altri linguaggi di programmazione.

```
if condizione:
    # Blocco di codice se la condizione è vera
    istruzione1
    istruzione2
else:
    # Blocco di codice se la condizione è falsa
    istruzione3
    istruzione4
```

In questo esempio, l'istruzione `if` controlla una condizione e, se è vera, esegue il blocco di codice all'interno del primo blocco indentato. Se la condizione è falsa, viene eseguito il blocco di codice all'interno del blocco `else` . L'indentazione è fondamentale per definire i blocchi di codice e deve essere coerente all'interno dello stesso blocco.

2.4.4 Input e output

In Python, è possibile interagire con l'utente attraverso l'input e l'output. L'istruzione `print()` viene utilizzata per visualizzare messaggi di output sulla console, mentre la funzione `input()` viene utilizzata per acquisire input dall'utente. Ecco un esempio di come utilizzare l'input e l'output in Python:

```
# Output di un messaggio
print("Inserisci il tuo nome:")

# Acquisizione di un input dall'utente
nome = input("")
print("Ciao, " + nome + "!")
```

In questo esempio, l'istruzione `print("Inserisci il tuo nome:")` visualizza un messaggio di output sulla console, mentre la funzione `input()` acquisisce un input dall'utente. Il valore inserito dall'utente viene memorizzato nella variabile `nome` e successivamente visualizzato con un messaggio di saluto.

Si può anche evitare di stampare un messaggio di output separato e chiedere direttamente all'utente di inserire il proprio nome:

```
nome = input("Inserisci il tuo nome: ")
print("Ciao, " + nome + "!")
```

Esercizio 2.4.1

Scrivi un programma Python che calcoli la somma dei primi `n` numeri interi positivi, dove `n` è un numero intero inserito dall'utente. Ad esempio, se l'utente inserisce `5`, il programma dovrebbe calcolare la somma `1 + 2 + 3 + 4 + 5` e visualizzare il risultato.

Soluzione

```
n = int(input("Inserisci un numero intero positivo: "))
somma = 0

for i in range(1, n + 1):
    somma += i

print("La somma dei primi", n, "numeri interi positivi è", somma)
```

Chiaramente non avendo ancora trattato i cicli, questa soluzione potrebbe non essere immediatamente comprensibile, ma è un buon esercizio per iniziare a pensare in termini di programmazione e algoritmi.

Capitolo 3: Programmazione in Python - Concetti Fondamentali

3.1 Strutture di Controllo

Le strutture di controllo sono fondamentali nella programmazione, poiché consentono di gestire il flusso di esecuzione del programma in base a determinate condizioni. In Python, le strutture di controllo principali includono istruzioni condizionali (`if`, `elif`, `else`), cicli (`for`, `while`), istruzioni `break` e `continue`.

E' importante comprendere come funzionano queste strutture di controllo e come utilizzarle in modo efficace per risolvere problemi e implementare algoritmi.

3.1.1 Istruzioni condizionali (if, elif, else)

Le istruzioni condizionali consentono di eseguire blocchi di codice in base a determinate condizioni. In Python, le istruzioni condizionali sono definite con le parole chiave `if`, `elif` (else if) e `else`. Ecco un esempio di come utilizzare le istruzioni condizionali in Python:

```
voto = 80

if voto >= 90:
    print("Voto: A")
elif voto >= 80:
    print("Voto: B")
elif voto >= 70:
    print("Voto: C")
```

```
else:
    print("Voto: D")
```

In questo esempio, l'istruzione `if` controlla se il voto è maggiore o uguale a 90 e, se è vero, stampa "Voto: A". Se la condizione non è vera, l'istruzione `elif` controlla se il voto è maggiore o uguale a 80 e, se è vero, stampa "Voto: B". Se nessuna delle condizioni precedenti è vera, l'istruzione `else` viene eseguita e stampa "Voto: D".

3.1.2 Operatori logici e di confronto

In Python, è possibile utilizzare operatori logici e di confronto per combinare condizioni e valutare espressioni booleane. Gli operatori logici più comuni includono `and`, `or` e `not`, mentre gli operatori di confronto includono `==` (uguale), `!=` (diverso), `<` (minore di), `>` (maggiore di), `<=` (minore o uguale a) e `>=` (maggiore o uguale a). Ecco un esempio di come utilizzare operatori logici e di confronto in Python:

```
x = 5
y = 10

if x > 0 and y > 0:
    print("Entrambi i numeri sono positivi")

if x > 0 or y > 0:
    print("Almeno uno dei numeri è positivo")

if not x < 0:
    print("Il numero x non è negativo")
```

In questo esempio, l'istruzione `if` utilizza gli operatori logici `and`, `or` e `not` per combinare condizioni e valutare espressioni booleane. Questi operatori consentono di creare condizioni complesse e gestire casi specifici in modo efficace.

3.1.3 Cicli (for, while)

I cicli sono utilizzati per eseguire ripetutamente un blocco di codice fino a quando una determinata condizione è soddisfatta. In Python, i cicli più comuni sono il ciclo `for` e il ciclo `while`. Ecco un esempio di come utilizzare i cicli in Python:

Ciclo `for`

Il ciclo `for` viene utilizzato per iterare su una sequenza di elementi, come una lista, una tupla o una stringa. Ecco un esempio di come utilizzare un ciclo `for` in Python:

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

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

In questo esempio, il ciclo `for` itera su ogni elemento della lista `frutta` e stampa il nome di ciascun frutto sulla console. La variabile `frutto` assume il valore di ciascun elemento della lista durante ogni iterazione del ciclo.

Ciclo `while`

Il ciclo `while` viene utilizzato per eseguire un blocco di codice finché una determinata condizione è vera. Ecco un esempio di come utilizzare un ciclo `while` in Python:

```
numero = 1

while numero <= 5:
    print(numero)
    numero += 1
```

In questo esempio, il ciclo `while` esegue il blocco di codice finché il valore della variabile `numero` è minore o uguale a 5. Durante ogni iterazione, il numero viene stampato sulla console e incrementato di 1. Il ciclo continua finché la condizione `numero <= 5` è vera.

3.1.4 Istruzioni break e continue

Queste due istruzioni sono utilizzate per controllare il flusso di esecuzione all'interno di un ciclo. L'istruzione `break` interrompe immediatamente l'esecuzione del ciclo e esce dal blocco, mentre l'istruzione `continue` salta il resto del blocco e passa alla successiva iterazione del ciclo. Spesso queste istruzioni permettono di ottenere un codice più performante e che rispetti le specifiche richieste.

```
for numero in range(1, 11):
    if numero == 5:
        break
    print(numero)
```

In questo esempio, il ciclo `for` itera su una sequenza di numeri da 1 a 10. Quando il numero è uguale a 5, l'istruzione `break` interrompe l'esecuzione del ciclo e esce dal blocco. Di conseguenza, solo i numeri da 1 a 4 vengono stampati sulla console.

```
for numero in range(1, 11):
    if numero % 2 == 0:
        continue
    print(numero)
```

In questo esempio, il ciclo `for` itera su una sequenza di numeri da 1 a 10. Quando il numero è pari (cioè il resto della divisione per 2 è 0), l'istruzione `continue` salta il resto del blocco e passa alla successiva iterazione del ciclo. Di conseguenza, solo i numeri dispari vengono stampati sulla console.

3.2 Funzioni e Metodi

Importante, le `funzioni` in Python sono blocchi di codice riutilizzabili che eseguono una determinata operazione o calcolo. Le funzioni consentono di suddividere il codice in parti più piccole e gestibili, rendendo il codice più modulare e leggibile. In Python, le funzioni sono definite con la parola chiave `def` seguita dal nome della funzione e da eventuali parametri.

3.2.1 Definizione e chiamata di funzioni

Le funzioni vengono definite con la parola chiave `def` seguita dal nome della funzione e da eventuali parametri tra parentesi. Ecco un esempio di come definire e chiamare una funzione in Python:

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

Per chiamare una funzione, è sufficiente utilizzare il nome della funzione seguito da eventuali argomenti tra parentesi. Ecco un esempio di come chiamare la funzione `saluta` definita in precedenza:

```
saluta("Alice")
saluta("Bob")
```

In questo esempio, la funzione `saluta` accetta un parametro `nome` e stampa un messaggio di saluto personalizzato sulla console. La funzione viene chiamata due volte con i nomi "Alice" e "Bob", che vengono passati come argomenti alla funzione.

3.2.2 Parametri e argomenti

Quando definiamo delle funzioni è possibile scegliere di passare dei parametri, ovvero dei valori che la funzione può utilizzare per eseguire un'operazione specifica. I parametri possono essere obbligatori o opzionali, e possono assumere valori predefiniti. Nell'esempio di prima abbiamo definito una funzione con il parametro *nome*. Ora definiamo 2 funzioni, una con un parametro obbligatorio ma con valore di base e una senza parametri.

```
def saluta(nome="Mondo"):
    print("Ciao, " + nome + "!")
```

```
def salutaNiente():
    print("Ciao, Mondo!")

saluta("Alice")
salutaNiente()
```

In questo esempio, la funzione `saluta` accetta un parametro `nome` con un valore predefinito "Mondo", che viene utilizzato se non viene specificato un valore. La funzione `salutaNiente` non accetta alcun parametro e stampa un messaggio di saluto predefinito. Quando chiamiamo la funzione `saluta` con il nome "Alice", il messaggio di saluto personalizzato viene visualizzato sulla console. Quando chiamiamo la funzione `salutaNiente` , viene visualizzato il messaggio di saluto predefinito "Ciao, Mondo!".

3.2.3 Valore di ritorno

Le funzioni in Python possono restituire un valore di ritorno, ovvero un valore calcolato o elaborato all'interno della funzione. Il valore di ritorno può essere utilizzato per passare informazioni o risultati da una funzione a un'altra parte del programma. Per restituire un valore di ritorno da una funzione, è possibile utilizzare l'istruzione `return` seguita dal valore da restituire. Creiamo una funzione per calcolare lo **stipendio** di un software engineer sottopagato.

```
def calcolaStipendio(ore_lavorate, tariffa_oraria):
    stipendio = ore_lavorate * tariffa_oraria
    return stipendio

ore = 40
tariffa = 20

stipendio = calcolaStipendio(ore, tariffa)
print("Lo stipendio è:", stipendio)
```

In questo esempio, la funzione `calcolaStipendio` accetta due parametri `ore_lavorate` e `tariffa_oraria` e calcola lo stipendio moltiplicando le ore lavorate per la tariffa oraria. Il valore dello stipendio calcolato viene restituito dalla funzione utilizzando l'istruzione `return` e memorizzato nella variabile `stipendio` . Infine, il valore dello stipendio viene stampato sulla console.

3.2.4 Scope delle variabili

L'argomento dello **scope** è uno dei più importanti in programmazione, e in Python non fa eccezione. Lo scope di una variabile definisce la parte del programma in cui la variabile è accessibile e può essere utilizzata. In Python, ci sono due tipi principali di scope: lo scope globale e lo scope locale. Le variabili definite all'interno di una funzione sono locali alla funzione stessa, mentre le variabili definite al di fuori di una funzione sono globali e possono essere utilizzate in tutto il programma. Definiamo due

variabili, un contatore globale e un contatore locale ad una funzione. Lo utilizzeremo per aumentare i soldi guadagnati dal nostro software engineer.

```
contatore_globale = 0

def aumentaStipendio(ammontare):
    global contatore_globale
    contatore = ammontare
    contatore_globale += contatore
    return contatore_globale

aumento = 100
nuovo_stipendio = aumentaStipendio(aumento)
print("Nuovo stipendio:", nuovo_stipendio)
```

Cosa stiamo facendo qui? Stiamo definendo una variabile globale (`contatore_globale`) e una funzione (`aumentaStipendio`) che accetta un parametro (`ammontare`) e aumenta il contatore globale di quell'ammontare. Utilizziamo la parola chiave `global` per indicare che stiamo facendo riferimento alla variabile globale all'interno della funzione. Quando chiamiamo la funzione `aumentaStipendio` con un aumento di 100, il contatore globale viene incrementato di 100 e il nuovo valore viene restituito e stampato sulla console.

La variabile **contatore** non è accessibile al di fuori della funzione `aumentaStipendio` , poiché è definita all'interno della funzione e quindi locale a essa. La variabile **contatore_globale**, invece, è accessibile in tutto il programma poiché è definita al di fuori della funzione e quindi globale.

3.2.5 Metodi degli oggetti built-in

Cosa sono i metodi degli oggetti built-in? Sono metodi predefiniti che possono essere utilizzati con oggetti di determinati tipi di dati in Python. Ad esempio, gli oggetti di tipo stringa hanno metodi come `upper()` , `lower()` , `capitalize()` , `split()` e molti altri che possono essere utilizzati per manipolare e trasformare le stringhe. Vediamo qualche esempi nel prossimo paragrafo.

3.3 Iterazione

L'iterazione è un concetto fondamentale nella programmazione, poiché consente di eseguire ripetutamente un blocco di codice fino a quando una determinata condizione è soddisfatta. In Python, l'iterazione può essere realizzata utilizzando cicli `for` e `while` , che consentono di eseguire un blocco di codice su una sequenza di elementi o finché una condizione è vera. Vediamo come funzionano questi cicli e come utilizzarli in Python.

3.3.1 Iterazione su sequenze (liste, tuple, stringhe)

Immaginiamo di avere una `lista` che contiene i nomi degli `studenti` e vogliamo stampare tutti quanti sulla console. Se facessimo cosi:

```
studenti = ["Alice", "Bob", "Charlie", "David", "Eve"]
print(studenti)
```

Otterremmo un output simile a questo:

```
["Alice", "Bob", "Charlie", "David", "Eve"]
```

Noi vogliamo stampare ogni studente su una riga separata. Per fare ciò, possiamo utilizzare un ciclo `for` per iterare su ogni elemento della lista e stamparlo sulla console. Vediamo come fare:

```
studenti = ["Alice", "Bob", "Lovaion", "AF64", "Miku"]

for studente in studenti:
    print(studente)
```

In questo esempio, il ciclo `for` itera su ogni elemento della lista `studenti` e stampa il nome di ciascuno studente sulla console. La variabile `studente` assume il valore di ciascun elemento della lista durante ogni iterazione del ciclo. L'output che otterremmo sarebbe:

```
Alice
Bob
Lovaion
AF64
Miku
```

3.3.2 Comprensione delle liste

Solitamente questo concetto viene espresso in inglese come `list comprehension`. Le comprensioni delle liste sono una caratteristica potente di Python che consente di creare liste in modo conciso e leggibile utilizzando una singola riga di codice. Le comprensioni delle liste sono spesso utilizzate per trasformare o filtrare una lista esistente in una nuova lista, ma in generale si possono eseguire molte operazioni sugli elementi senza dover scrivere un blocco di codice a parte. Attenzione poiché per scrivere una lista comprehension è necessario conoscere bene quello che si sta facendo.

Nel prossimo codice proviamo a creare una lista nuova partendo da quella degli studenti ma per ogni studente vogliamo cambiare la prima lettera di ogni studente con il numero di lettere del suo nome. Vediamo come fare:

```
studenti = ["Alice", "Bob", "Lovaion", "AF64", "Miku"]

studenti_modificati = [len(studente) + studente[1:] for studente in studenti]
```

Qui in particolare stiamo utilizzando `[1:]` per dire a Python di prendere tutti i caratteri della stringa tranne il primo. Quindi per ogni studente nella lista `studenti` stiamo creando una nuova stringa che ha come primo carattere il numero di lettere del nome dello studente e come resto della stringa il nome dello studente stesso. Il risultato sarà:

```
[5lice, 3ob, 6ovaion, 4F64, 4iku]
```

3.3.3 Funzione range()

La funzione `range()` è una funzione integrata di Python che genera una sequenza di numeri interi in un intervallo specificato. La funzione `range()` può essere utilizzata con uno, due o tre argomenti per definire l'inizio, la fine e l'incremento della sequenza. Vediamo qualche esempio utile.

Immaginiamo di avere bisogno di stampare i numeri da 1 a 10 sulla console. Possiamo farlo utilizzando la funzione `range()` in un ciclo `for` :

```
for numero in range(1, 11):
    print(numero)
```

In questo esempio, la funzione `range(1, 11)` genera una sequenza di numeri da 1 a 10 (escluso 11) e il ciclo `for` itera su ogni numero della sequenza e lo stampa sulla console. L'output sarà semplicemente i numeri da 1 a 10. Ora, immaginiamo di voler stampare i numeri pari da 0 a 10. Possiamo farlo utilizzando la funzione `range()` con un incremento di 2:

```
for numero in range(0, 11, 2):
    print(numero)
```

Il primo argomento di `range()` è l'inizio della sequenza, il secondo argomento è la fine della sequenza (escluso) e il terzo argomento è l'incremento della sequenza. In questo caso, la funzione `range(0, 11, 2)` genera una sequenza di numeri pari da 0 a 10 e il ciclo `for` itera su ogni numero della sequenza e lo stampa sulla console. L'output sarà i numeri pari da 0 a 10.

3.3.4 Iteratori e generatori

Altro concetto importante in Python sono gli iteratori e i generatori. Gli iteratori sono oggetti che consentono di iterare su una sequenza di elementi uno alla volta, mentre i generatori sono funzioni speciali che generano una sequenza di valori in modo efficiente. Gli iteratori e i generatori sono utilizzati per gestire grandi quantità di dati e per evitare di caricare tutti i dati in memoria contemporaneamente. Vediamo un esempio di come utilizzare un generatore per generare una sequenza di numeri pari:

```
def numeri_pari(n):
    for numero in range(0, n, 2):
        yield numero

for numero in numeri_pari(10):
    print(numero)
```

Cosa succede qui esattamente? Stiamo definendo una funzione `numeri_pari` che accetta un parametro `n` e utilizza un ciclo `for` per generare una sequenza di numeri pari da 0 a `n` con un incremento di 2. Tuttavia, invece di restituire tutti i numeri contemporaneamente, utilizziamo la parola chiave `yield` per restituire un numero alla volta. Quando chiamiamo la funzione `numeri_pari(10)` all'interno di un ciclo `for`, otteniamo una sequenza di numeri pari da 0 a 10 stampati sulla console. La keywordl `yield` è fondamentale per creare un generatore, poiché permette di restituire un valore alla volta senza dover memorizzare tutti i valori in memoria, e questo è particolarmente utile quando si lavora con grandi quantità di dati.

Tuttavia, a meno che non si stia lavorando con grandi quantità di dati, è più comune utilizzare una lista comprehension per generare una sequenza di numeri pari in modo più conciso e leggibile. I generatori sono utili quando si desidera evitare di caricare tutti i dati in memoria contemporaneamente o quando si desidera generare una sequenza di valori in modo efficiente.

3.4 Ricorsione

Il concetto di `ricorsione` è un altro concetto fondamentale in programmazione, che si riferisce alla capacità di una funzione di chiamare se stessa in modo ricorsivo. La ricorsione è spesso utilizzata per risolvere problemi che possono essere suddivisi in sottoproblemi più piccoli e simili al problema originale. In particolare modo viene utilizzata per risolvere problemi di tipo:

- Fibonacci
- Alberi binari
- Fattoriale

3.4.1 Concetto di ricorsione

La ricorsione in programmazione non è altro che la capacità di una funzione di chiamare se stessa. Questo concetto può sembrare strano all'inizio, ma è fondamentale per risolvere problemi complessi in modo elegante e conciso. La ricorsione è spesso utilizzata per risolvere problemi che possono essere suddivisi in sottoproblemi più piccoli e simili al problema originale. Facciamo un esempio calcolando il fattoriale di un numero. La formula per calcolare il fattoriale di un numero `n` è:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

Il codice per calcolare il fattoriale di un numero utilizzando la ricorsione potrebbe essere:

```
def fattoriale(n):
    if n == 0:
        return 1
    else:
        return n * fattoriale(n - 1)

numero = 5
risultato = fattoriale(numero)
print("Il fattoriale di", numero, "è", risultato)
```

Nel prossimo paragraph vedremo il concetto di casi base e casi ricorsivi, esempi di funzioni ricorsive e i vantaggi e svantaggi della ricorsione.

3.4.2 Casi base e casi ricorsivi

Nell'esempio di codice di prima il blocco `if n == 0:` è il caso base della ricorsione, ovvero il caso in cui la funzione restituisce un valore senza chiamare se stessa. Questo caso base è fondamentale per evitare che la funzione entri in un ciclo infinito e termini l'esecuzione correttamente. Il blocco `else:` è il caso ricorsivo della ricorsione, ovvero il caso in cui la funzione chiama se stessa con un argomento diverso. Questo caso ricorsivo consente alla funzione di risolvere il problema in modo iterativo, suddividendolo in sottoproblemi più piccoli fino a raggiungere il caso base.

Utilizzando `5` come esempio, il flow delle chiamate sarebbe:

```
fattoriale(5)
5 * fattoriale(4)
5 * 4 * fattoriale(3)
5 * 4 * 3 * fattoriale(2)
5 * 4 * 3 * 2 * fattoriale(1)
5 * 4 * 3 * 2 * 1 * fattoriale(0)
5 * 4 * 3 * 2 * 1 * 1
```

3.4.3 Esempi di funzioni ricorsive

Altri esempi di funzioni ricorsive includono il calcolo del numero di Fibonacci e la ricerca binaria. Il numero di Fibonacci è una sequenza di numeri in cui ciascun numero è la somma dei due numeri precedenti. La ricerca binaria è un algoritmo di ricerca che divide ripetutamente una lista in due metà e cerca un elemento in una delle due metà. Un esempio di fibonacci:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```
numero = 10
risultato = fibonacci(numero)
print("Il numero di Fibonacci di", numero, "è", risultato)
```

3.4.4 Vantaggi e svantaggi della ricorsione

La ricorsione ha vantaggi e svantaggi rispetto all'iterazione. I vantaggi della ricorsione includono la chiarezza del codice, la facilità di implementazione di algoritmi complessi e la gestione di problemi che possono essere suddivisi in sottoproblemi più piccoli. Tuttavia, la ricorsione può essere meno efficiente dell'iterazione in termini di utilizzo della memoria e tempo di esecuzione, poiché ogni chiamata ricorsiva aggiunge un nuovo frame di stack alla memoria e può portare a un sovraccarico dello stack in caso di ricorsione profonda. Inoltre, la ricorsione può essere più difficile da comprendere e debuggare rispetto all'iterazione, poiché il flusso di esecuzione può essere meno lineare e più complesso. Buona fortuna con la ricorsione!

Capitolo 4: Strutture Dati in Python

4.1 Liste

Le liste sono una delle strutture dati più comuni in Python e sono utilizzate per memorizzare una sequenza di elementi in un ordine specifico. Le liste possono contenere elementi di diversi tipi di dati, come interi, float, stringhe, booleani, liste annidate e molto altro. Le liste sono definite utilizzando parentesi quadre `[]` e gli elementi sono separati da virgole. Abbiamo visto vari esempi di liste in precedenza, ma vediamo come creare una lista vuota e come aggiungere elementi ad una lista esistente utilizzando il metodo `append()` e altri metodi comuni.

```
# Creazione di una lista vuota
linguaggi_di_programmazione = []

# Aggiunta di elementi alla lista
linguaggi_di_programmazione.append("Python")
linguaggi_di_programmazione.append("Java")
linguaggi_di_programmazione.append("JavaScript")
```

Ok, facciamo finta ora di aver fatto un'errore e vogliamo rimuovere un elemento dalla lista. Possiamo farlo utilizzando il metodo `remove()` o `pop()`. Il metodo `remove()` rimuove un elemento specifico dalla lista, mentre il metodo `pop()` rimuove l'elemento in una posizione specifica. Vediamo come fare:

```
# Aggiungiamo HTML per errore
```

```
linguaggi_di_programmazione.append("HTML")

# Rimuoviamo HTML dalla lista, siccome non è un linguaggio di programmazione
linguaggi_di_programmazione.remove("HTML")
```

Altrimenti, se volessimo rimuovere l'ultimo elemento della lista possiamo utilizzare il metodo `pop()` senza specificare alcun argomento:

```
# Rimuoviamo l'ultimo elemento dalla lista
linguaggi_di_programmazione.pop()
```

Altri metodi comuni per lavorare con le liste includono `insert()`, `extend()`, `index()`, `count()`, `sort()`, `reverse()` e molti altri. Le liste sono una delle strutture dati più flessibili e potenti in Python e sono ampiamente utilizzate per memorizzare e manipolare dati in modo efficiente. La cosa che conviene fare in questi casi è consultare la documentazione ufficiale di Python per avere una panoramica completa dei metodi disponibili per le liste.

4.2 Tuple

Le tuple sono simili alle liste, ma sono immutabili, ovvero non possono essere modificate dopo essere state create. Le tuple sono definite utilizzando parentesi tonde `()` e gli elementi sono separati da virgole. Le tuple sono spesso utilizzate per memorizzare insiemi di valori correlati che non devono essere modificati. Vediamo un esempio pratico di quando conviene utilizzare una tupla:

```
# Creazione di una tupla per una transazione finanziaria
transazione = ("2022-01-01", "Alice", 100.0, "Spesa")

movimenti = []

# Aggiunta della transazione alla lista dei movimenti
movimenti.append(transazione)
```

In questo modo abbiamo creato una tupla per rappresentare una transazione finanziaria, con la data, il nome del cliente, l'importo e la descrizione della transazione. Questa tupla è stata aggiunta a una lista di movimenti finanziari. Le tuple sono utili quando si desidera memorizzare un insieme di valori correlati che non devono essere modificati, ad esempio i dettagli di una transazione finanziaria o le coordinate di un punto nello spazio.

4.3 Dizionari

I `dizionari` sono forse la struttura dati più flessibile e potente in Python e sono utilizzati per memorizzare coppie chiave-valore in un ordine arbitrario. I dizionari sono definiti utilizzando parentesi graffe `{}` e le coppie `chiave-valore` sono separate da due punti `:`. I dizionari sono

spesso utilizzati per memorizzare dati strutturati in modo gerarchico e per accedere rapidamente ai valori utilizzando le chiavi. Negli altri linguaggi di programmazione sono spesso chiamati `mappe` o `hashmaps`.

Un esempio molto pratico di quando utilizzare un dizionario è la memorizzazione di un elenco di contatti telefonici. Vediamo come creare un dizionario di contatti telefonici:

```
# Creazione di un dizionario di contatti telefonici
contatti = {
    "Alice": "123-456-7890",
    "Bob": "234-567-8901",
    "Charlie": "345-678-9012"
}

# Aggiunta di un nuovo contatto al dizionario
contatti["David"] = "456-789-0123"
```

In questo esempio, abbiamo creato un dizionario di contatti telefonici con i nomi delle persone come chiavi e i numeri di telefono come valori. Ora, se volessimo rimuovere un contatto dal dizionario possiamo utilizzare il metodo `pop()` specificando la chiave del contatto da rimuovere:

```
# Rimozione di un contatto dal dizionario
contatti.pop("Charlie")
```

Invece, per accedere ai valori di un dizionario possiamo utilizzare la notazione delle parentesi quadre `[]` specificando la chiave del valore che vogliamo ottenere:

```
# Accesso ai valori del dizionario
print(contatti["Alice"])
```

In questo modo otterremmo il numero di telefono associato alla chiave "Alice".

Molti problemi di programmazione possono essere risolti in modo efficiente utilizzando i dizionari, poiché consentono di memorizzare e accedere rapidamente ai dati utilizzando le chiavi.

4.4 Insiemi

Gli `insiemi` sono una struttura dati in Python che memorizza una collezione di elementi unici in un ordine arbitrario. Gli insiemi sono definiti utilizzando parentesi graffe `{}` e gli elementi sono separati da virgole. Gli insiemi sono spesso utilizzati per rimuovere duplicati da una lista o per eseguire operazioni insiemistiche come unione, intersezione, differenza e differenza simmetrica. Spesso sono

utilizzati per eseguire operazioni di confronto tra due insiemi di dati. Vediamo un esempio pratico di quando utilizzare un insieme:

```
# Creazione di un insieme di colori
colori1 = {"rosso", "verde", "blu"}
colori2 = {"verde", "giallo", "viola"}

# Unione di due insiemi
colori_comuni = colori1.union(colori2)

# Intersezione di due insiemi
colori_diversi = colori1.difference(colori2)

# Differenza simmetrica di due insiemi
colori_unici = colori1.symmetric_difference(colori2)
```

In questo esempio, abbiamo creato due insiemi di colori e abbiamo eseguito diverse operazioni insiemistiche su di essi. L'unione di due insiemi restituisce un insieme contenente tutti gli elementi presenti in entrambi gli insiemi, l'intersezione di due insiemi restituisce un insieme contenente solo gli elementi comuni a entrambi gli insiemi e la differenza simmetrica di due insiemi restituisce un insieme contenente solo gli elementi che sono presenti in uno dei due insiemi ma non in entrambi.

Capitolo 5: Gestione degli Errori e Debugging

Nella programmazione è comune incontrare errori durante lo sviluppo di un programma. Gli errori possono essere di diversi tipi, come errori di sintassi, errori di runtime e errori logici. La gestione degli errori è un aspetto importante della programmazione e consente di identificare, diagnosticare e risolvere gli errori in modo efficace. In Python, è possibile gestire gli errori utilizzando le istruzioni `try`, `except` e `finally`.

5.1 Tipi di errori (sintassi, runtime, logici)

Gli errori di sintassi sono errori che si verificano quando il codice non rispetta le regole di sintassi del linguaggio di programmazione. Gli errori di runtime sono errori che si verificano durante l'esecuzione del programma, come divisioni per zero o accesso a indici fuori dai limiti di una lista. Gli errori logici sono errori che si verificano quando il programma produce un risultato diverso da quello atteso a causa di un errore nella logica del programma.

Vediamo qualche caso di errore comune in Python:

```
# Errore di sintassi
print("Ciao)

# Errore di runtime
```

```
numero = 0
risultato = 10 / numero

# Errore logico
def somma(a, b):
    return a - b

risultato = somma(5, 3)
```

In questo esempio, il primo errore è un errore di sintassi perché manca un apice alla fine della stringa. Il secondo errore è un errore di runtime perché stiamo cercando di dividere per zero, che è un'operazione non valida in matematica. Il terzo errore è un errore logico perché la funzione `somma` dovrebbe restituire la somma di due numeri, ma in realtà restituisce la differenza tra i due numeri.

5.2 Gestione delle eccezioni (try, except, finally)

Quando ci sono dei possibili errori nel codice, è possibile gestirli utilizzando le istruzioni `try`, `except` e `finally`. L'istruzione `try` consente di eseguire un blocco di codice che potrebbe generare un'eccezione, mentre l'istruzione `except` consente di gestire l'eccezione e di eseguire un blocco di codice alternativo in caso di errore. L'istruzione `finally` consente di eseguire un blocco di codice che deve essere eseguito in ogni caso, indipendentemente dal fatto che si sia verificata un'eccezione o meno. Vediamo un esempio di come utilizzare `try`, `except` e `finally` per gestire un errore di divisione per zero:

```
numero = 0

try:
    risultato = 10 / numero
except ZeroDivisionError:
    print("Errore: divisione per zero")
finally:
    print("Fine del programma")
```

In questo esempio, stiamo cercando di dividere per zero, che genererà un'eccezione di tipo `ZeroDivisionError`. Utilizzando l'istruzione `try`, possiamo eseguire il codice che potrebbe generare l'eccezione. Utilizzando l'istruzione `except`, possiamo gestire l'eccezione e stampare un messaggio di errore. Utilizzando l'istruzione `finally`, possiamo eseguire un blocco di codice che deve essere eseguito in ogni caso, indipendentemente dal fatto che si sia verificata un'eccezione o meno.

5.3 Tecniche di debugging

Il `debugging` è il processo di identificazione, diagnosi e risoluzione degli errori in un programma. Il debugging è un'attività fondamentale per gli sviluppatori di software e consente di garantire che il

programma funzioni correttamente e produca i risultati attesi. In Python, è possibile utilizzare diverse tecniche di debugging per identificare e risolvere gli errori nel codice. Alcune delle tecniche di debugging più comuni includono l'utilizzo di `print` per visualizzare i valori delle variabili, l'utilizzo di un debugger interattivo come `pdb`, l'utilizzo di strumenti di profiling per identificare i punti caldi del codice e l'utilizzo di test automatizzati per verificare il comportamento del programma.

```
def somma(a, b):
    print("Valore di a:", a)
    print("Valore di b:", b)
    return a + b

risultato = somma(5, 3)
```

In questo esempio, stiamo utilizzando l'istruzione `print` per visualizzare i valori delle variabili `a` e `b` all'interno della funzione `somma`. Questo ci consente di verificare che i valori delle variabili siano corretti e di identificare eventuali errori nel calcolo della somma. L'utilizzo di `print` è una tecnica di debugging semplice ma efficace che consente di visualizzare i valori delle variabili e di identificare rapidamente eventuali errori nel codice. Usando un IDE (Integrated Development Environment) come PyCharm o Visual Studio Code, è possibile utilizzare un debugger interattivo per eseguire il codice passo dopo passo, visualizzare i valori delle variabili e identificare i punti in cui si verificano gli errori. Questo è particolarmente utile per risolvere errori complessi e per comprendere il flusso di esecuzione del programma.

Capitolo 6: File e I/O

Per i non addetti ai lavori, l'I/O sta per Input/Output, ovvero l'interazione tra il programma e l'ambiente esterno.

Vediamo in questa sezione come leggere e scrivere file in Python, come gestire i percorsi dei file e come elaborare i dati da file.

6.1 Lettura e scrittura di file

Immaginiamo di avere un file contenenti i nomi degli studenti e i loro voti e vogliamo leggere i dati da file e calcolare la media dei voti. Quello che ci serve è:

1. Aprire il file in modalità lettura
2. Leggere i dati dal file
3. Chiudere il file

```
# Apertura del file in modalità lettura
with open("studenti.txt", "r") as file:
```

```
    # Lettura dei dati dal file
    studenti = file.readlines()

# Calcolo della media dei voti
voti = [int(studente.split()[1]) for studente in studenti]

media = sum(voti) / len(voti)

print("La media dei voti è:", media)
```

Spiegazione veloce di quello che sta succedendo. La parola chiave `with` viene utilizzata per aprire il file in modalità lettura e garantire che il file venga chiuso correttamente alla fine dell'istruzione `with`. Il metodo `readlines()` viene utilizzato per leggere i dati dal file e restituirli come una lista di righe. Utilizziamo una list comprehension per estrarre i voti da ciascuna riga e calcolare la media dei voti. Infine, stampiamo la media dei voti sulla console.

`open()` è una funzione integrata di Python che consente di aprire un file in diverse modalità, come lettura, scrittura, aggiunta, binaria, testuale e altro ancora. La lettura e la scrittura di file sono operazioni comuni in Python e sono utilizzate per elaborare dati da file, memorizzare dati su file e interagire con il sistema di file del sistema operativo. Il file viene trattato come una variabile e può essere aperto, letto, scritto e chiuso utilizzando i metodi integrati di Python.

Notare che il file `studenti.txt` deve essere presente nella stessa directory del file Python per poter essere letto correttamente. In caso contrario, verrà generato un errore di file non trovato. `split()` è un metodo delle stringhe di Python che consente di suddividere una stringa in una lista di sottostringhe utilizzando uno spazio come delimitatore. In questo caso, stiamo suddividendo ogni riga del file in due parti, il nome dello studente e il voto, estraendo il voto e convertendolo in un intero utilizzando `int()`. Siccome abbiamo detto che il file contiene il nome dello studente e il voto, possiamo usare `split()` per dividere la stringa in due parti e prendere la seconda parte che contiene il voto.

6.2 Gestione dei percorsi

La gestione dei percorsi dei file è un aspetto importante della programmazione e consente di lavorare con file e directory in modo efficace. In Python, è possibile utilizzare il modulo `os` per gestire i percorsi dei file e le directory, creare, spostare, rinominare e eliminare file e directory, e ottenere informazioni sui file e le directory. Vediamo un esempio di come utilizzare il modulo `os` per ottenere il percorso assoluto di un file e controllare se un file esiste:

```
import os

# Percorso attuale
percorso_attuale = os.getcwd()

print("Percorso attuale:", percorso_attuale)
```

```
# Percorso assoluto di un file
percorso_file = os.path.abspath("studenti.txt")

print("Percorso assoluto del file:", percorso_file)

# Verifica se il file esiste
if os.path.exists(percorso_file):
    print("Il file esiste")
else:
    print("Il file non esiste")
```

In questo esempio, stiamo utilizzando il modulo `os` per ottenere il percorso attuale del file Python utilizzando `os.getcwd()`, ottenere il percorso assoluto di un file utilizzando `os.path.abspath()`, e verificare se un file esiste utilizzando `os.path.exists()`. Il modulo `os` fornisce una vasta gamma di funzionalità per la gestione dei percorsi dei file e delle directory e consente di lavorare con file e directory in modo efficace.

6.3 Elaborazione di dati da file

L'elaborazione dei dati da file è un'attività comune in Python e consente di leggere, scrivere, manipolare e analizzare dati da file. I dati possono essere memorizzati in diversi formati, come testo, CSV, JSON, XML, database e altro ancora, e possono essere elaborati utilizzando le librerie integrate di Python o librerie di terze parti. Vediamo un esempio di come leggere e scrivere dati da un file CSV utilizzando il modulo `csv` di Python:

```
import csv

# Lettura dei dati da un file CSV
with open("studenti.csv", "r") as file:
    lettore = csv.reader(file)
    studenti = [riga for riga in lettore]

# Scrittura dei dati su un file CSV
with open("studenti_copia.csv", "w", newline="") as file:
    scrittore = csv.writer(file)
    scrittore.writerows(studenti)
```

In questo esempio, stiamo utilizzando il modulo `csv` di Python per leggere i dati da un file CSV utilizzando `csv.reader()` e scrivere i dati su un file CSV utilizzando `csv.writer()`. Il modulo `csv` fornisce una vasta gamma di funzionalità per lavorare con file CSV e consente di leggere e scrivere dati da file CSV in modo efficace. I dati da file possono essere elaborati utilizzando le librerie integrate di Python o librerie di terze parti, a seconda delle esigenze del progetto.

Capitolo 7: Moduli e Pacchetti

Python è famoso principalmente per la sua vasta libreria standard e per la sua comunità di sviluppatori che ha creato una vasta gamma di moduli e pacchetti per risolvere problemi comuni e complessi. I moduli sono file Python che contengono definizioni di funzioni, classi e variabili che possono essere utilizzate in altri programmi Python. I pacchetti sono raccolte di moduli che possono essere distribuiti e installati utilizzando il sistema di gestione dei pacchetti di Python, `pip`.

7.1 Importazione e utilizzo di moduli

Per esempio, la stra-grande maggioranza di librerie riguardanti l'intelligenza artificiale e il machine learning sono scritte per essere utilizzate in Python. Alcune come `numpy`, `pandas`, `matplotlib`, `scikit-learn`, `tensorflow`, `keras` e `pytorch` sono diventate fondamentali per chiunque voglia lavorare in questo campo. Ma non solo, Python è pieno di librerie per il web development, per la creazione di GUI, per il networking, per la sicurezza informatica, per la gestione dei database e molto altro ancora. Vediamo qualche esempio semplice di come importare e utilizzare un modulo in Python:

```
# Importazione di un modulo
import math

# Utilizzo di funzioni del modulo
print("Pi greco:", math.pi)

raggio = 5
area = math.pi * raggio ** 2

print("Area del cerchio:", area)
```

In questo esempio, stiamo importando il modulo `math` di Python utilizzando l'istruzione `import` e utilizzando le funzioni del modulo per calcolare il valore di Pi greco e l'area di un cerchio. Il modulo `math` fornisce una vasta gamma di funzioni matematiche e costanti matematiche che possono essere utilizzate per eseguire calcoli matematici complessi in Python. L'importazione di moduli è un aspetto importante della programmazione in Python e consente di utilizzare le funzionalità dei moduli in altri programmi Python.

7.2 Pacchetti Python più comuni

Quelli più utilizzati per lo scripting e per l'utilizzo più generale, senza includere argomenti più specifici come il machine learning, sono:

1. `numpy` : per la manipolazione di array e matrici multidimensionali
2. `pandas` : per la manipolazione e l'analisi dei dati
3. `matplotlib` : per la creazione di grafici e visualizzazioni

4. `requests` : per l'invio di richieste HTTP
5. `beautifulsoup4` : per l'estrazione di dati da pagine web
6. `scrapy` : per il web scraping
7. `flask` : per la creazione di applicazioni web
8. `django` : per la creazione di applicazioni web più complesse
9. `sqlalchemy` : per l'interazione con database SQL
10. `pytest` : per l'esecuzione di test automatizzati

Questi sono solo alcuni esempi di pacchetti Python comuni che possono essere utilizzati per risolvere problemi comuni e complessi in Python. La vasta libreria standard di Python e la sua comunità di sviluppatori hanno creato una vasta gamma di moduli e pacchetti che possono essere utilizzati per risolvere una vasta gamma di problemi in diversi settori e ambiti.

Capitolo 8: Programmazione Orientata agli Oggetti

Questo è probabilmente uno dei concetti più importanti e complessi della programmazione, ma è anche uno dei più potenti e flessibili. La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione che consente di organizzare il codice in classi e oggetti, che rappresentano entità del mondo reale e consentono di modellare il mondo reale in modo più efficace. In Python, tutto è un oggetto, ovvero un'istanza di una classe, e la programmazione orientata agli oggetti è ampiamente utilizzata per creare programmi complessi e modulari.

La programmazione orientata agli oggetti si basa su quattro concetti fondamentali: classi, oggetti, ereditarietà e polimorfismo. Le classi sono modelli per creare oggetti, gli oggetti sono istanze di classi, l'ereditarietà consente di creare nuove classi basate su classi esistenti e il polimorfismo consente di utilizzare oggetti di classi diverse in modo uniforme.

8.1 Concetti di base (classi, oggetti, metodi)

Classi

Una classe in programmazione è un modello per creare oggetti che rappresentano entità del mondo reale. Ad esempio, una classe `Auto` può essere utilizzata per creare oggetti che rappresentano automobili, con attributi come marca, modello, colore e metodi come accendere, spegnere, accelerare e frenare.

Quando si definisce una classe in Python, si utilizza la parola chiave `class` seguita dal nome della classe e due punti `:`. All'interno della classe, è possibile definire attributi e metodi che rappresentano le proprietà e i comportamenti degli oggetti della classe.

```
# Definizione di una classe
class Auto:
    # Attributi della classe
```

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

    # Metodi della classe
    def accendere(self):
        print("Accendere l'auto")

    def spegnere(self):
        print("Spegnere l'auto")

    def accelerare(self):
        print("Accelerare l'auto")

    def frenare(self):
        print("Frenare l'auto")
```

Il concetto di **costruttore** di una classe è fondamentale. Il costruttore è un metodo speciale chiamato `__init__` che viene eseguito quando si crea un nuovo oggetto della classe. Il costruttore può essere utilizzato per inizializzare gli attributi dell'oggetto con valori predefiniti. In questo caso, stiamo inizializzando gli attributi `marca`, `modello` e `colore` dell'oggetto `Auto` con i valori passati come argomenti al costruttore.

Il **self** è un riferimento all'oggetto stesso e viene utilizzato per accedere agli attributi e ai metodi dell'oggetto all'interno della classe. Quando si definiscono metodi di classe, il primo parametro deve essere `self` per accedere all'oggetto stesso.

I **metodi** della classe sono funzioni che rappresentano i comportamenti degli oggetti della classe. I metodi possono essere utilizzati per eseguire azioni sugli oggetti, come accendere, spegnere, accelerare e frenare un'auto. I metodi della classe possono accedere agli attributi dell'oggetto utilizzando il riferimento `self`.

Oggetti

Un oggetto in programmazione è un'istanza di una classe, ovvero un'entità del mondo reale che rappresenta un'istanza della classe. Ad esempio, un'auto è un oggetto che rappresenta un'istanza della classe `Auto`, con attributi come marca, modello, colore e metodi come accendere, spegnere, accelerare e frenare.

A differenza delle classi, che sono modelli per creare oggetti, gli oggetti sono istanze delle classi che rappresentano entità del mondo reale. Quando si crea un oggetto di una classe, si utilizza la parola chiave `class` seguita dal nome della classe e parentesi tonde `()`.

```
# Creazione di un oggetto della classe Auto
auto1 = Auto("Toyota", "Corolla", "Blu")
```

```
# Accesso agli attributi dell'oggetto
print("Marca:", auto1.marca)
print("Modello:", auto1.modello)
print("Colore:", auto1.colore)

# Utilizzo dei metodi dell'oggetto
auto1.accendere()
auto1.accelerare()
auto1.frenare()
auto1.spegnere()
```

In questo esempio, stiamo creando un oggetto `auto1` della classe `Auto` utilizzando il costruttore della classe con i valori `"Toyota"`, `"Corolla"` e `"Blu"` per gli attributi `marca`, `modello` e `colore`. Stiamo quindi accedendo agli attributi dell'oggetto utilizzando il riferimento `auto1.marca`, `auto1.modello` e `auto1.colore` e utilizzando i metodi dell'oggetto per accendere, accelerare, frenare e spegnere l'auto.

8.2 Ereditarietà

L'ereditarietà è un concetto fondamentale della programmazione orientata agli oggetti che consente di creare nuove classi basate su classi esistenti. La classe esistente è chiamata classe base o superclasse e la nuova classe è chiamata classe derivata o sottoclasse. La classe derivata eredita gli attributi e i metodi della classe base e può aggiungere nuovi attributi e metodi o modificare gli attributi e i metodi esistenti.

Pensiamo ad esempio di avere una classe `Veicolo` che rappresenta un veicolo generico con attributi come marca, modello e colore e metodi come accendere, spegnere, accelerare e frenare. Possiamo creare una classe `Auto` che eredita dalla classe `Veicolo` e aggiunge attributi e metodi specifici per un'auto, come numero di porte, numero di posti e tipo di carburante.

```
# Definizione di una classe base
class Veicolo:
    def __init__(self, marca, modello, colore):
        self.marca = marca
        self.modello = modello
        self.colore = colore

    def accendere(self):
        print("Accendere il veicolo")

    def spegnere(self):
        print("Spegnere il veicolo")

    def accelerare(self):
        print("Accelerare il veicolo")
```

```
    def frenare(self):
        print("Frenare il veicolo")

# Definizione di una classe derivata
class Auto(Veicolo):
    def __init__(self, marca, modello, colore, porte, posti, carburante):
        super().__init__(marca, modello, colore)
        self.porte = porte
        self.posti = posti
        self.carburante = carburante

    def aprire_porte(self):
        print("Aprire le porte dell'auto")

    def chiudere_porte(self):
        print("Chiudere le porte dell'auto")
```

In questo esempio, stiamo definendo una classe base `Veicolo` con attributi e metodi generici per un veicolo e una classe derivata `Auto` che eredita dalla classe base e aggiunge attributi e metodi specifici per un'auto. La classe derivata `Auto` utilizza il metodo `super()` per chiamare il costruttore della classe base e inizializzare gli attributi della classe base. La classe derivata `Auto` può quindi aggiungere nuovi attributi come `porte`, `posti` e `carburante` e nuovi metodi come `aprire_porte` e `chiudere_porte` specifici per un'auto.

8.3 Polimorfismo

Il polimorfismo è un concetto fondamentale della programmazione orientata agli oggetti che consente di utilizzare oggetti di classi diverse in modo uniforme. Il polimorfismo consente di trattare oggetti di classi diverse come se fossero oggetti della stessa classe e di eseguire operazioni sugli oggetti in modo uniforme.

Ad esempio, possiamo creare una funzione `guidare` che accetta un oggetto `Veicolo` come argomento e utilizza i metodi `accendere`, `accelerare`, `frenare` e `spegnere` dell'oggetto per guidare il veicolo. La funzione `guidare` può essere utilizzata con oggetti di classi diverse, come `Veicolo` e `Auto`, e può eseguire operazioni sugli oggetti in modo uniforme.

```
# Funzione polimorfica
def guidare(veicolo):
    veicolo.accendere()
    veicolo.accelerare()
    veicolo.frenare()
    veicolo.spegnere()

# Creazione di oggetti di classi diverse
veicolo = Veicolo("Toyota", "Corolla", "Blu")
```

```
auto = Auto("Toyota", "Corolla", "Blu", 4, 5, "Benzina")

# Utilizzo della funzione polimorfica
guidare(veicolo)
guidare(auto)
```

In questo esempio, stiamo definendo una funzione `guidare` che accetta un oggetto `Veicolo` come argomento e utilizza i metodi `accendere`, `accelerare`, `frenare` e `spegnere` dell'oggetto per guidare il veicolo. La funzione `guidare` può essere utilizzata con oggetti di classi diverse, come `Veicolo` e `Auto`, e può eseguire operazioni sugli oggetti in modo uniforme. Il polimorfismo consente di trattare oggetti di classi diverse come se fossero oggetti della stessa classe e di eseguire operazioni sugli oggetti in modo uniforme.

8.4 Incapsulamento

L'incapsulamento è un concetto fondamentale della programmazione orientata agli oggetti che consente di nascondere i dettagli di implementazione di un oggetto e di esporre solo le interfacce pubbliche dell'oggetto. L'incapsulamento consente di proteggere gli attributi e i metodi di un oggetto e di garantire che vengano utilizzati in modo corretto.

In Python, l'incapsulamento può essere realizzato utilizzando i metodi getter e setter per accedere e modificare gli attributi di un oggetto. I metodi getter vengono utilizzati per accedere agli attributi dell'oggetto e i metodi setter vengono utilizzati per modificare gli attributi dell'oggetto.

```
# Definizione di una classe con incapsulamento
class Persona:
    def __init__(self, nome, cognome):
        self.__nome = nome
        self.__cognome = cognome

    def get_nome(self):
        return self.__nome

    def set_nome(self, nome):
        self.__nome = nome

    def get_cognome(self):
        return self.__cognome

    def set_cognome(self, cognome):
        self.__cognome = cognome

# Creazione di un oggetto della classe con incapsulamento
persona = Persona("Mario", "Rossi")

# Accesso agli attributi dell'oggetto con incapsulamento
```

```
print("Nome:", persona.get_nome())
print("Cognome:", persona.get_cognome())

# Modifica degli attributi dell'oggetto con incapsulamento
persona.set_nome("Luigi")
persona.set_cognome("Verdi")

print("Nome:", persona.get_nome())
print("Cognome:", persona.get_cognome())
```

In questo esempio, stiamo definendo una classe `Persona` con attributi `nome` e `cognome` incapsulati utilizzando il doppio underscore `__` . Gli attributi incapsulati possono essere accessibili solo all'interno della classe e possono essere modificati solo utilizzando i metodi getter e setter della classe. I metodi getter vengono utilizzati per accedere agli attributi dell'oggetto e i metodi setter vengono utilizzati per modificare gli attributi dell'oggetto. L'incapsulamento consente di proteggere gli attributi e i metodi di un oggetto e di garantire che vengano utilizzati in modo corretto.

Continua

Chiaramente la programmazione orientata agli oggetti è un argomento molto vasto e complesso, e questo è solo un'introduzione. Per approfondire l'argomento, è possibile consultare la documentazione ufficiale di Python e altri tutorial e risorse online. Manca il concetto di classe e metodi **astratti** e **statici**, la **composizione** e l'**aggregazione**, il **design pattern** e molto altro ancora.

Capitolo 9: Progetti Pratici

9.1 Creazione di un'applicazione di gestione delle attività

Un'applicazione di gestione delle attività è un'applicazione che consente di creare, visualizzare, modificare ed eliminare attività da un elenco di attività. L'applicazione può essere utilizzata per tenere traccia delle attività quotidiane, settimanali o mensili e per organizzare il lavoro e la vita personale.

9.2 Creazione di una calcolatrice

Una calcolatrice è un'applicazione che consente di eseguire operazioni matematiche come addizione, sottrazione, moltiplicazione e divisione su numeri. L'applicazione può essere utilizzata per eseguire calcoli matematici semplici e complessi e per risolvere problemi matematici in modo rapido ed efficiente.

9.3 Creazione di un'applicazione di gestione delle spese

Un'applicazione di gestione delle spese è un'applicazione che consente di tenere traccia delle spese personali e di organizzare le finanze personali. L'applicazione può essere utilizzata per registrare le spese quotidiane, settimanali o mensili e per monitorare i costi e i budget.

9.4 Creazione di un'applicazione di gestione dei contatti

Un'applicazione di gestione dei contatti è un'applicazione che consente di memorizzare e organizzare i contatti telefonici e le informazioni di contatto delle persone. L'applicazione può essere utilizzata per creare un elenco di contatti, aggiungere nuovi contatti, modificare i contatti esistenti e cercare i contatti in base al nome o al numero di telefono.

Capitolo 10: Conclusioni

In questo tutorial abbiamo esplorato i concetti fondamentali della programmazione in Python, come variabili, tipi di dati, operatori, istruzioni di controllo, funzioni, liste, tuple, dizionari, insiemi, gestione degli errori, debugging, file e I/O, moduli e pacchetti, programmazione orientata agli oggetti e progetti pratici. Python è un linguaggio di programmazione versatile e potente che può essere utilizzato per creare una vasta gamma di applicazioni, da script semplici a programmi complessi.

La programmazione in Python è un'abilità fondamentale per gli sviluppatori di software e consente di creare programmi efficienti, modulari e scalabili. Python è ampiamente utilizzato in diversi settori e ambiti, come lo sviluppo web, il machine learning, l'analisi dei dati, l'automazione, la sicurezza informatica e altro ancora. Conoscere i concetti fondamentali della programmazione in Python è un passo importante per diventare un programmatore esperto e per creare programmi di alta qualità.

Spero che questo tutorial ti abbia fornito una panoramica completa della programmazione in Python e ti abbia aiutato a comprendere i concetti fondamentali della programmazione in Python. Se hai domande, suggerimenti o feedback, non esitare a contattarmi. Grazie per aver letto questo tutorial e buona programmazione in Python!

Scritto da Daniele Avolio