Advanced Python Features

Pasquale Mirtuono

June 6, 2024

Decorators - Introduzione

- I decorators permettono di modificare o migliorare funzioni e metodi senza cambiare il loro codice. Essi sono un concetto fondamentale in Python e vengono utilizzati per estendere o alterare il comportamento di funzioni e metodi in modo trasparente ed elegante.
- Questo è possibile grazie all'uso di funzioni o classi che prendono una funzione come input e ne restituiscono una versione modificata o arricchita.

Decorators - Utilità

- Utili per aggiungere funzionalità come logging, controllo di accesso e misurazione delle prestazioni. Per esempio, è possibile creare un decorator che registra l ora di inizio e fine di una funzione, permettendo di misurare il tempo di esecuzione della stessa.
- Un altro esempio è un decorator che controlla se un utente ha i permessi necessari per eseguire una determinata funzione, migliorando la sicurezza del codice. Inoltre, i decorators possono essere utilizzati per la gestione delle risorse, come la chiusura automatica di file o la gestione delle transazioni in database.

Decorators - Miglioramenti al Codice

- I decorators migliorano la leggibilità e la manutenibilità del codice, poiché separano chiaramente la logica aggiuntiva dalla logica principale della funzione.
- Questo approccio facilita la scrittura di codice modulare e riutilizzabile, riducendo la duplicazione e promuovendo l'organizzazione del codice.

Decorators - Applicazioni Avanzate

- Possono essere applicati sia a funzioni che a metodi di classe. In quest'ultimo caso, i decorators possono essere utilizzati per modificare il comportamento di metodi di istanza, di classe o statici, offrendo una grande flessibilità nella gestione del comportamento delle classi.

Decorators - Combinazioni

- È possibile combinare più decorators per arricchire ulteriormente una funzione o un metodo. Questa caratteristica consente di costruire una pipeline di trasformazioni, dove ogni decorator aggiunge un livello di funzionalità, mantenendo il codice principale pulito e focalizzato sul suo scopo originale.

Esempio di Decorator - Codice

```
def my_decorator(func):
def wrapper():
print("Something is happening before the function is called.")
func()
print("Something is happening after the function is called.")
return wrapper

@my_decorator
def say_hello():
print("Hello!")

say_hello()
```

Esempio di Decorator - Output

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

List Comprehensions

- Le list comprehensions forniscono un modo conciso per creare liste in Python.
- Riducono la necessità di cicli espliciti, rendendo il codice più leggibile e compatto.
- Sintassi: `[expression for item in iterable if condition]`
- Possono essere utilizzate per trasformare, filtrare e combinare liste in una singola riga di codice.

Dictionary Comprehensions

- Le dictionary comprehensions forniscono un modo conciso per creare dizionari da collezioni esistenti.
- Sintassi: `{key: value for item in iterable if condition}`
- Possono essere utilizzate per trasformare o filtrare i valori di un dizionario.
- Consentono di eseguire operazioni su ogni coppia chiave-valore e creare un nuovo dizionario in una singola riga di codice.

Set Comprehensions

- Le set comprehensions forniscono un modo conciso per creare insiemi senza duplicati.
- Sintassi: `{expression for item in iterable if condition}`
- Possono essere utilizzate per filtrare o trasformare i valori di un insieme.
- Consentono di eseguire operazioni su ogni elemento e creare un nuovo insieme in una singola riga di codice.

Lambda Functions - Introduzione

- Le funzioni lambda sono piccole funzioni anonime definite con la parola chiave `lambda`.
- Sintassi: `lambda arguments: expression`
- Utili per operazioni brevi o per passare come argomenti a funzioni di ordine superiore.

Utilizzo delle Lambda Functions

Le lambda functions possono essere utilizzate in diverse situazioni. Ecco alcuni esempi:

```
# Esempio 1: Somma di due numeri
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8

# Esempio 2: Filtraggio di una lista
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6, 8, 10]
```

Utilizzo delle Lambda Functions (Continuazione)

- Le lambda functions possono essere utilizzate per operazioni brevi e semplici senza la necessità di definire una funzione separata.
- Sono spesso utilizzate come argomenti per funzioni di ordine superiore come `map`, `filter`, e `reduce`.
- La loro sintassi concisa le rende ideali per situazioni in cui è necessario definire una funzione rapidamente.

Higher-Order Functions - Introduzione

- Le funzioni di ordine superiore possono prendere altre funzioni come argomenti o restituirle come risultati.
- Sono utili per creare codice flessibile e riutilizzabile, consentendo di separare la logica di alto livello dalla logica di basso livello.
- Esempi comuni includono `map`, `filter`, e `reduce`.

Funzione map

- La funzione `map` applica una data funzione a tutti gli elementi di una sequenza e restituisce un nuovo iterator che produce i risultati.
- Sintassi: `map(function, iterable)`
- Esempio:
  - `numbers = [1, 2, 3, 4, 5]`
  - `squared = list(map(lambda x:  x**2, numbers))`
  - `print(squared)`

Funzione filter

- La funzione `filter` crea un iteratore che produce solo gli elementi di un iterable per i quali una funzione specificata restituisce True.
- Sintassi: `filter(function, iterable)`
- Esempio:
  - `numbers = [1, 2, 3, 4, 5]`
  - `even = list(filter(lambda x:  x % 2 == 0, numbers))`
  - `print(even)`

Funzione reduce

- La funzione `reduce` applica una funzione a coppie di elementi di un iterable fino a quando l'iterable è ridotto a un singolo risultato.
- Sintassi: `functools.reduce(function, iterable[, initializer])`
- Esempio:
  - `import functools`
  - `numbers = [1, 2, 3, 4, 5]`
  - `total = functools.reduce(lambda x, y:  x + y, numbers)`
  - `print(total)`

Exception Handling - Introduzione

- La gestione delle eccezioni permette di gestire gli errori in modo elegante usando i blocchi `try`, `except`, `finally`, e `else`.
- È essenziale per scrivere programmi robusti e tolleranti ai guasti.

Context Managers - Introduzione

- I gestori di contesto, implementati usando l istruzione `with`, sono utilizzati per gestire risorse come file e connessioni di rete.
- Assicurano che le risorse siano correttamente acquisite e rilasciate, prevenendo perdite di risorse.

Context Managers - Utilizzo

- In Python, e possibile implementare un gestore di contesto definendo una classe con metodi `__enter__()` e `__exit__()`.
- Il metodo `__enter__()` viene chiamato quando si entra nel blocco `with`, mentre il metodo `__exit__()` viene chiamato alla sua uscita.

Type Hints

- Gli hint di tipo sono annotazioni che specificano i tipi di dati attesi per variabili e valori di ritorno delle funzioni.
- Migliorano la leggibilità del codice, aiutano nel debugging e facilitano il supporto degli strumenti.

Generators and Iterators

- I generatori sono funzioni che producono valori uno alla volta e mantengono il loro stato tra le chiamate.
- Utili per creare iteratori, oggetti che possono essere iterati (ad esempio, in un ciclo `for`) senza caricare l'intera sequenza in memoria.

Metaprogramming

- La metaprogrammazione implica la scrittura di codice che manipola altro codice a runtime.
- Tecniche includono l'uso di metaclassi, `getattr`, `setattr`, e `hasattr`.
- Utile per la generazione dinamica di codice e per l'implementazione di pattern avanzati come proxy e singleton.

Asynchronous Programming

- La programmazione asincrona permette l'esecuzione concorrente del codice, rendendo i programmi più efficienti, specialmente nei compiti I/O-bound.
- La libreria `asyncio` fornisce strumenti per scrivere codice asincrono usando `async def`, `await`, e `async with`.

Multiple Dispatch

- Il multiple dispatch permette a una funzione di comportarsi diversamente in base ai tipi dei suoi argomenti.
- Utile per creare funzioni polimorfiche che possono gestire vari tipi di input.

Dataclasses

- Le dataclass, introdotte in Python 3.7, semplificano la creazione di classi utilizzate principalmente per memorizzare dati.
- Generano automaticamente metodi speciali come `__init__`, `__repr__`, e `__eq__`, riducendo il boilerplate code e migliorando la leggibilità.