



Metodi speciali in Python

Python

Definizione e caratteristiche

Un metodo unico è caratterizzato da due elementi chiave:

- **Decoratore @uniquemethod:** Questo decoratore identifica il metodo come unico e ne regola il comportamento all'interno della gerarchia delle classi.
- **Implementazione del metodo:** La definizione del metodo contiene la logica specifica da eseguire quando il metodo viene richiamato.



I metodi unici presentano le seguenti caratteristiche distintive:

- **Unicità:** Un metodo unico può essere definito solo una volta all'interno della gerarchia delle classi. Se una sottoclasse tenta di ridefinire lo stesso metodo unico, viene sollevata un'eccezione `TypeError`.
- **Ereditarietà:** I metodi unici non vengono ereditati automaticamente dalle classi sottostanti. Se una sottoclasse necessita di accedere al metodo unico definito nella classe padre, deve richiamarlo esplicitamente.
- **Risoluzione:** In caso di conflitti tra metodi unici con lo stesso nome in classi diverse ma correlate, la risoluzione avviene in base all'ordine di ricerca delle classi (Method Resolution Order - MRO).



Utilizzo dei metodi unici

I metodi unici trovano impiego in svariate situazioni:

- **Implementazione di metodi personalizzati per classi astratte:** I metodi unici permettono di definire metodi specifici per classi astratte, fornendo una guida alle classi sottostanti per l'implementazione di metodi coerenti.
- **Gestione di conflitti di nomi:** In scenari con classi correlate che definiscono metodi con lo stesso nome, i metodi unici garantiscono chiarezza e prevengono ambiguità.
- **Estensione di classi esterne:** Quando si lavora con classi di librerie esterne, i metodi unici offrono un modo sicuro per aggiungere funzionalità personalizzate senza modificare il codice originale.



Classi uniche e metodi unici

Le classi uniche, sebbene non siano un concetto formalmente definito in Python, si riferiscono a classi che sfruttano ampiamente i metodi unici per implementare comportamenti specifici.

Queste classi spesso fungono da base per altre classi, fornendo metodi unici come guida per l'implementazione coerente di metodi simili nelle classi derivate.

L'utilizzo di classi uniche e metodi unici favorisce un codice Python più modulare, flessibile e mantenibile, promuovendo principi di progettazione OOP solidi.



I Decoratori in Python: Spiegazioni e Esempi

I decorator in Python sono una funzionalità potente e versatile che consente di modificare e ampliare le funzionalità di funzioni e classi esistenti.

Sintassi

La sintassi per utilizzare i decorator è semplice:

- 1. @decoratore_funzione**
- 2. def funzione_da_decorare():**
- 3. # Corpo della funzione**



In questa sintassi:

- **@decoratore_funzione** indica che la funzione **funzione_da_decorare** verrà decorata dalla funzione **decoratore_funzione**.
- La funzione **decoratore_funzione** riceve la funzione da decorare come argomento.
- All'interno del decoratore, la funzione da decorare può essere modificata o estesa utilizzando diverse tecniche.



I Decoratori in Python: Spiegazioni e Esempi

```
1. import time
2.
3.
4. def registratore_tempo(funzione):
5.     def wrapper(*args, **kwargs):
6.         start_time = time.time()
7.         risultato = funzione(*args, **kwargs)
8.         end_time = time.time()
9.         tempo_esecuzione = end_time - start_time
10.        print(f"Tempo di esecuzione: {tempo_esecuzione:.2f} secondi")
11.        return risultato
12.
13.    return wrapper
14.
15.
16. @registratore_tempo
17. def funzione_esempio(n):
18.     for i in range(n):
19.         print(i)
20.
21. funzione_esempio(5)
```



In questo esempio:

- **Il decoratore `registratore_tempo` riceve la funzione `funzione_esempio` come argomento.**
- **All'interno del decoratore, viene definita una funzione interna `wrapper` che avvolge la funzione originale.**
- **La funzione `wrapper` registra il tempo di esecuzione della funzione originale e stampa il risultato.**
- **Il decoratore restituisce la funzione `wrapper` decorata.**
- **La funzione `funzione_esempio` viene decorata con `@registratore_tempo`, che farà sì che la funzione `wrapper` venga eseguita al posto suo.**
- **Quando `funzione_esempio` viene chiamata, il tempo di esecuzione viene registrato e stampato.**



Casi d'uso comuni dei decoratori

- **Aggiungere funzionalità a funzioni esistenti:** Come nell'esempio sopra, i decoratori possono essere utilizzati per aggiungere funzionalità come la registrazione del tempo, la gestione degli errori, la cache o la convalida degli input.
- **Creare comportamenti generici:** I decoratori possono essere utilizzati per creare comportamenti generici che possono essere applicati a diverse funzioni. Ad esempio, un decoratore può essere utilizzato per autenticare gli utenti prima di eseguire una funzione.
- **Semplificare il codice:** I decoratori possono semplificare il codice raggruppando la logica comune in funzioni separate che possono essere riutilizzate.



Casi d'uso comuni dei decoratori

```
1. import time
2.
3.
4. def registratore_tempo(funzione):
5.     def wrapper(*args, **kwargs):
6.         start_time = time.time()
7.         risultato = funzione(*args, **kwargs)
8.         end_time = time.time()
9.         tempo_esecuzione = end_time - start_time
10.        print(f"Tempo di esecuzione: {tempo_esecuzione:.2f} secondi")
11.        return risultato
12.
13.    return wrapper
14.
15.
16. @registratore_tempo
17. def funzione_esempio(n):
18.     for i in range(n):
19.         print(i)
20.
21. funzione_esempio(5)
```



Spiegazione:

- Questo codice definisce un decoratore `registratore_tempo` che registra il tempo impiegato per eseguire una funzione.
- Il decoratore riceve una funzione come argomento e restituisce una funzione interna wrapper che avvolge la funzione originale.
- La funzione wrapper registra il tempo di esecuzione della funzione originale e stampa il risultato.
- La funzione `funzione_esempio` viene decorata con `@registratore_tempo`, che farà sì che la funzione wrapper venga eseguita al posto suo.
- Quando `funzione_esempio` viene chiamata, il tempo di esecuzione viene registrato e stampato.



Casi d'uso comuni dei decoratori

```
1. def verifica_autorizzazione(livello_autorizzato):
2.     def decoratore_autorizzazione(funzione):
3.         def wrapper(*args, **kwargs):
4.             if livello_autorizzato <= livello_utente_corrente:
5.                 return funzione(*args, **kwargs)
6.             else:
7.                 raise Exception("Autorizzazione insufficiente")
8.
9.         return wrapper
10.
11.     return decoratore_autorizzazione
12.
13.
14. @verifica_autorizzazione(5)
15. def esegui_operazione_amministrativa():
16.     # Esegue un'operazione che richiede privilegi amministrativi
17.     pass
18.
19. livello_utente_corrente = 3
20.
21. try:
22.     esegui_operazione_amministrativa()
23. except Exception as e:
24.     print(f"Errore: {e}")
```



Spiegazione:

- Questo codice definisce un decoratore `verifica_autorizzazione` che verifica se l'utente corrente ha il livello di autorizzazione necessario per eseguire una funzione.
- Il decoratore riceve un `livello_autorizzato` come argomento e restituisce una funzione interna `decoratore_autorizzazione`.
- La funzione `decoratore_autorizzazione` riceve una funzione come argomento e restituisce una funzione wrapper.
- La funzione wrapper controlla se il `livello_utente_corrente` è maggiore o uguale al `livello_autorizzato`. Se lo è, esegue la funzione originale. Altrimenti, solleva un'eccezione.
- La funzione `esegui_operazione_amministrativa` viene decorata con `@verifica_autorizzazione(5)`, che richiede un livello di autorizzazione di 5 per essere eseguita.
- Il codice verifica se `livello_utente_corrente` è sufficiente per eseguire l'operazione. Se non lo è, viene sollevata un'eccezione.



Casi d'uso comuni dei decorator

```
1. def gestore_errori(funzione):
2.     def wrapper(*args, **kwargs):
3.         try:
4.             return funzione(*args, **kwargs)
5.         except Exception as e:
6.             print(f"Errore durante l'esecuzione della funzione: {e}")
7.             return None
8.
9.     return wrapper
10.
11.
12. @gestore_errori
13. def funzione_con_possibili_errori(divisore):
14.     return 10 / divisore
15.
16. risultato = funzione_con_possibili_errori(0)
17. print(f"Risultato: {risultato}")
18.
19. risultato = funzione_con_possibili_errori(5)
20. print(f"Risultato: {risultato}")
```



Spiegazione:

- **Questo codice definisce un decoratore `gestore_errori` che gestisce le eccezioni sollevate da una funzione.**
- **Il decoratore riceve una funzione come argomento e restituisce una funzione interna wrapper.**
- **La funzione wrapper tenta di eseguire la funzione originale. Se si verifica un'eccezione, stampa un messaggio di errore e restituisce `None`. Altrimenti, restituisce il risultato**



Buon Davante a tutti

