Closures in Python

Einführung

Eine Closure ist eine Funktion, die auf Variablen in einem umgebenden Scope zugreifen kann, in dem sie definiert wurde, auch nachdem der umgebende Scope nicht mehr existiert. Closures sind nützlich, wenn eine Funktion von Variablen abhängt, die sich außerhalb ihres direkten Bereichs befinden.

Grundlagen der Closures

Eine Closure entsteht, wenn eine Funktion eine andere Funktion zurückgibt und die zurückgegebene Funktion auf Variablen aus dem umgebenden Scope zugreift.

Beispiel

```
def äußere_funktion(x):
    def innere_funktion(y):
        return x + y
    return innere_funktion

closure = äußere_funktion(5)
print(closure(3))  # Ausgabe: 8
```

In diesem Beispiel bildet `innere_funktion` eine Closure, die auf die Variable `x` aus dem umgebenden Scope zugreift, selbst nachdem `äußere_funktion` bereits abgeschlossen ist.

Anwendungsfälle von Closures

Closures können in verschiedenen Situationen nützlich sein, wie zum Beispiel:

- **Zustandsspeicherung**: Eine Funktion kann Zustände zwischen Aufrufen speichern.
- **Callback-Funktionen**: Eine Callback-Funktion kann eine Closure verwenden, um auf lokale Variablen zuzugreifen.
- **Memoisierung**: Eine Funktion kann Ergebnisse zwischenspeichern und wiederverwenden.

Beispiel: Zustandsspeicherung

```
def zähler(start):
    zählerstand = start
    def inkrement():
        nonlocal zählerstand
        zählerstand += 1
        return zählerstand
    return inkrement

counter = zähler(0)
```

```
print(counter())  # Ausgabe: 1
print(counter())  # Ausgabe: 2
print(counter())  # Ausgabe: 3
```

In diesem Beispiel speichert die Closure `inkrement` den Zustand des Zählers zwischen den Aufrufen.

Closures sind ein leistungsfähiges Werkzeug in Python, das es ermöglicht, Funktionen mit Zugriff auf Variablen in ihrem umgebenden Scope zu erstellen. Sie sind besonders nützlich in Situationen, in denen Zustände zwischen Funktionen gespeichert werden müssen oder wenn Callbacks benötigt werden, die auf lokale Variablen zugreifen müssen.

Dekorateure (Decorators) in Python

Einführung

Dekorateure sind ein mächtiges und expressives Werkzeug in Python, das es erlaubt, das Verhalten einer Funktion oder Methode zu modifizieren, ohne ihren Code zu ändern. Sie sind ideal für das Hinzufügen von Funktionalitäten "on-the-fly" und für die Anwendung von Aspekten wie Logging, Zugriffskontrolle, Messung und Caching auf eine saubere und wiederverwendbare Weise.

Grundlagen der Dekorateure

Ein Dekorateur ist im Wesentlichen eine Funktion, die eine andere Funktion als Argument nimmt und eine Funktion zurückgibt. Der Dekorateur hat die Möglichkeit, Aktionen vor und nach dem Aufruf der ursprünglichen Funktion durchzuführen.

Syntax

```
def mein_dekorator(funktion):
    def wrapper():
        # Aktionen vor dem Aufruf
        funktion()
        # Aktionen nach dem Aufruf
    return wrapper
```

Anwendung eines Dekorateurs

```
@mein_dekorator
def gruß():
    print("Hallo Welt!")

gruß()
```

Beispiele

Logging-Dekorateur

Ein einfacher Dekorateur, der das Logging von Funktionsaufrufen ermöglicht.

```
def logging_dekorator(funktion):
    def wrapper(*args, **kwargs):
        print(f"Aufruf von {funktion.__name__} mit {args} und {kwargs}")
        ergebnis = funktion(*args, **kwargs)
        print(f"{funktion.__name__} zurückgegeben {ergebnis}")
        return ergebnis
    return wrapper

@logging_dekorator
def addiere(a, b):
    return a + b

addiere(5, 3)
```

Leistungsüberwachungs-Dekorateur

Misst die Ausführungszeit einer Funktion.

```
import time

def zeitmessung_dekorator(funktion):
    def wrapper(*args, **kwargs):
        startzeit = time.time()
        ergebnis = funktion(*args, **kwargs)
        endzeit = time.time()
        print(f"{funktion.__name__} Ausführungszeit: {endzeit - startzeit}
Sekunden.")
        return ergebnis
    return wrapper

@zeitmessung_dekorator
def langlaufende_funktion():
    time.sleep(2)

langlaufende_funktion()
```

Übungen

1. **Schreiben Sie einen Dekorateur**: Implementieren Sie einen Dekorateur `repeat(n)`, der eine Funktion n-mal wiederholt.
2. **Dekorateur mit Argumenten**: Erstellen Sie einen Dekorateur `authenticate`, der eine Funktion nur dann ausführt, wenn ein gegebenes Passwort korrekt ist.
3. **Verschachtelte Dekorateure**: Wenden Sie mehrere Dekorateure auf eine Funktion an und beobachten Sie, wie sie zusammenarbeiten.