LU05d - Decorators für Funktionen in Python

Ein `Decorator` in Python ist ein Entwurfsmuster, das es erlaubt, einer Funktion zusätzliche Funktionalität hinzuzufügen, ohne ihren Code zu ändern. Decorators sind sehr leistungsfähig und nützlich, da sie den Code sauber halten und die Prinzipien der Wiederverwendbarkeit und Kapselung fördern.

Grundkonzept

In Python sind Funktionen First-Class-Objekte, das bedeutet, sie können als Argumente an andere Funktionen übergeben, von anderen Funktionen zurückgegeben und in Variablen gespeichert werden. Ein `Decorator` ist eine Funktion, die eine andere Funktion nimmt und diese erweitert, ohne ihren Quellcode zu ändern.

Syntax

Die grundlegende Syntax für einen Decorator in Python ist die folgende:

```
def decorator_function(original_function):
    def wrapper_function():
        # zusätzliche Funktionalitäten hier
        original_function()
        # weitere zusätzliche Funktionalitäten
    return wrapper_function

@decorator_function
def display():
    print("Die Display-Funktion wurde aufgerufen")

display()
```

In diesem Beispiel ist `decorator_function` der Decorator, der die `display`-Funktion erweitert. Der „@“-Operator ist syntaktischer Zucker für `display = decorator_function(display)`.

Verwendung von Decorators

Decorators sind häufig in Web-Frameworks und bei der Entwicklung von APIs zu finden. Einige gängige Anwendungsfälle für Decorators sind:

- Logging
- Berechtigungsprüfung
- Caching
- Überwachung
- Änderung der Argumente oder des Rückgabewerts einer Funktion

Decorators für Funktionen mit Parametern

Ein Decorator kann auch Argumente akzeptieren und Rückgabewerte modifizieren. Dazu können die `*args` und `**kwargs` Syntax verwendet werden:

```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        print(f"Wrapper executed this before {original_function.__name__}")
        return original_function(*args, **kwargs)
    return wrapper_function
```

In diesem erweiterten Beispiel kann der `wrapper_function` jegliche Anzahl von Positional- und Keyword-Argumenten akzeptieren und an die `original_function` weitergeben.

Beispiel:

```
import time

# Decorator zum Messen der Ausführungszeit
def time_it(original_function):
    def wrapper_function(*args, **kwargs):
        start_time = time.time()
        result = original_function(*args, **kwargs)
        end_time = time.time()
        print(f"{original_function.__name__} ran in {end_time -
start_time:.10f} seconds")
        return result
    return wrapper_function

# Beispiel-Funktion, die den Decorator verwendet
@time_it
def calculate_factorial(num):
    factorial = 1
    for i in range(1, num + 1):
        factorial *= i
    return factorial
```

Decorators mit eigenen Parametern

Decorators können auch Parameter akzeptieren, um ihre Funktionalität weiter anzupassen. Diese Parameter können verwendet werden, um das Verhalten des Decorators basierend auf den übergebenen Werten zu steuern. In diesem Abschnitt werden wir uns ansehen, wie man funktionsbasierte Decorators mit Parametern erstellt. Ein Decorator mit Parametern ist im Grunde eine Funktion, die einen Decorator zurückgibt. Diese äußere Funktion akzeptiert die Parameter und gibt den eigentlichen Decorator (eine innere Funktion) zurück, der die zu dekorierende Funktion modifiziert.

Syntax

Die grundlegende Syntax für einen Decorator mit Parametern in Python ist die folgende:

```
def outer_function(parameter):
    def decorator_function(original_function):
        def wrapper_function(*args, **kwargs):
            print(f"Decorator Parameter: {parameter}")
            original_function(*args, **kwargs)
        return wrapper_function
    return decorator_function

@outer_function("Hello, World!")
def display():
    print("Die Display-Funktion wurde aufgerufen.")
```

In diesem Beispiel akzeptiert der äußere Decorator `outer_function` einen Parameter und gibt den eigentlichen Decorator `decorator_function` zurück. Der „@“-Operator ist hier syntaktischer Zucker für `display = outer_function(„Hello, World!“)(display)`.

Anwendungsbeispiele

Logging Decorator mit Level

Ein praktisches Beispiel könnte ein Logging-Decorator mit einem Log-Level als Parameter sein.

```
def logging_decorator(level):
    def decorator_function(original_function):
        def wrapper_function(*args, **kwargs):
            print(f"[{level}] Vor dem Aufruf von
{original_function.__name__}")
            result = original_function(*args, **kwargs)
            print(f"[{level}] Nach dem Aufruf von
{original_function.__name__}")
            return result
        return wrapper_function
    return decorator_function

@logging_decorator("INFO")
def add(a, b):
    return a + b
```

In diesem Beispiel wird der `logging_decorator` mit einem Log-Level als Parameter aufgerufen. Der Decorator gibt dann Log-Nachrichten mit dem entsprechenden Level aus.

M323-LU05