LU03f - Callback Funktionen

Callback Funktionen sind eine klassische Anwendung von First-Class- und Higher-Order-Functions.

Callbacks in der GUI Programmierung

Die Entwicklung von grafischen Benutzeroberflächen (GUIs) unterscheidet sich in vielerlei Hinsicht von der Entwicklung von Konsolen- oder Terminalanwendungen. Eine der Hauptunterschiede ist, dass GUI-Anwendungen ereignisgesteuert sind. Das bedeutet, dass statt in einer sequenziellen Reihenfolge von oben nach unten durch den Code zu gehen, die Ausführung des Codes oft von Benutzeraktionen wie Mausklicks, Tastatureingaben oder anderen Ereignissen ausgelöst wird. Hier kommen `Callbacks` ins Spiel.

Was sind Callbacks?

Ein `Callback` ist eine Funktion, die an eine andere Funktion als Argument übergeben wird und zu einem späteren Zeitpunkt in Reaktion auf ein Ereignis ausgeführt wird. In der GUI-Programmierung werden Callbacks häufig verwendet, um auf Benutzerereignisse zu reagieren.

Warum werden Callbacks in der GUI-Programmierung verwendet?

1. **Ereignisgesteuerte Natur von GUIs**: Da GUI-Anwendungen darauf warten, dass Benutzerereignisse auftreten, können wir nicht vorhersehen, wann diese Ereignisse eintreten werden. Callbacks bieten eine Möglichkeit, spezifische Codeblöcke in Reaktion auf bestimmte Ereignisse auszuführen.
2. **Modularität und Wiederverwendbarkeit**: Durch die Verwendung von Callbacks können Sie spezifische Aktionen isolieren, was den Code sauberer und wiederverwendbarer macht.
3. **Flexibilität**: Callbacks ermöglichen es, unterschiedliche Aktionen für dasselbe Ereignis zu definieren, je nach Kontext oder Zustand der Anwendung.

Beispiel

In vielen GUI-Frameworks, einschließlich `tkinter` in Python, wird ein Button-Widget mit einem Callback verknüpft, um zu definieren, was passieren soll, wenn der Benutzer auf den Button klickt.

```
import tkinter as tk

def on_button_click():
    print('Der Button wurde geklickt!')

root = tk.Tk()
button = tk.Button(root, text='Klicken Sie mich', command=on_button_click)
button.pack()
```

```
root.mainloop()
```

In diesem Beispiel ist `on_button_click` der Callback, der aufgerufen wird, wenn der Benutzer auf den Button klickt.

Fazit

Callbacks sind ein essenzielles Konzept in der GUI-Programmierung und bieten eine effektive Möglichkeit, auf Benutzerereignisse zu reagieren und einen dynamischen, interaktiven Workflow für Anwendungen zu erstellen.

Callbacks in der Asynchronen Programmierung

Asynchrone Programmierung ist ein Ansatz, bei dem Operationen ausgeführt werden können, ohne den Ablauf des gesamten Programms zu blockieren. Ein häufiges Szenario für asynchrone Aufrufe sind Netzwerkanfragen, z. B. API-Aufrufe, bei denen nicht vorhersehbar ist, wie lange sie dauern werden.

Warum Callbacks in asynchronen Aufrufen verwenden?

- **Nicht-blockierende Natur:** Bei einem synchronen Aufruf würde Ihr Code anhalten und warten, bis die Anfrage abgeschlossen ist. Bei einem asynchronen Aufruf kann der Code weiter ausgeführt werden, ohne auf die Antwort zu warten. Ein Callback ermöglicht es, spezifischen Code auszuführen, sobald die Antwort eintrifft.
- **Strukturierte Code-Organisation:** Durch die Verwendung von Callbacks können Sie spezifische Aktionen oder Folgeverarbeitungen klar definieren, die nach Abschluss einer Operation ausgeführt werden sollen.
- **Fehlerbehandlung:** Callbacks können so gestaltet werden, dass sie sowohl mit normalen Daten als auch mit Fehlern umgehen können. Dies ist besonders nützlich bei Netzwerkanfragen, bei denen viele Dinge schief gehen können.

Warum nicht einfach den Code direkt nach dem API-Aufruf ausführen?

- **Unvorhersehbare Antwortzeiten:** Bei synchronem Code, der auf eine Netzwerkanfrage folgt, haben Sie keine Kontrolle darüber, wann die Antwort eintrifft. Ihr gesamtes Programm würde warten, was zu einer schlechten Benutzererfahrung führen kann.
- **Ressourceneffizienz:** Asynchrone Aufrufe ermöglichen es, Ressourcen effizienter zu nutzen. Während Sie auf eine Antwort warten, kann die CPU andere Aufgaben erledigen.
- **Klarheit und Wartbarkeit:** Durch das Trennen der Logik in verschiedene Funktionen (z. B. eine für den API-Aufruf und eine andere für die Verarbeitung der Antwort) wird der Code sauberer und leichter zu warten.

Asynchron Programmieren in Python

In modernen Anwendungen, insbesondere bei I/O-intensiven Aufgaben wie Netzwerkanfragen, Datenbankzugriffen oder Dateioperationen, ist es oft erforderlich, Operationen asynchron auszuführen, um die Gesamtleistung der Anwendung zu verbessern. Python bietet mit `async` und `await` ein leistungsfähiges Werkzeug zur asynchronen Programmierung.

Was sind async und await?

- **`async`**: Das Schlüsselwort `async` definiert eine Funktion als asynchron. Eine solche Funktion gibt ein „Coroutine“-Objekt zurück, das später mit `await` aufgerufen werden kann.
- **`await`**: Das Schlüsselwort `await` wird verwendet, um das Ergebnis einer asynchronen Operation abzurufen. Es kann nur innerhalb einer `async`-Funktion verwendet werden.

Beispiel: Parallele Ausführung von Aufgaben

Eine der Hauptvorteile von asynchroner Programmierung ist die Fähigkeit, mehrere Aufgaben parallel auszuführen, insbesondere wenn es um Aufgaben geht, die auf externe Ressourcen warten, wie z.B. Netzwerkanfragen.

```
import asyncio

async def task_1():
    print('Task 1 gestartet')
    await asyncio.sleep(2)
    print('Task 1 abgeschlossen')

async def task_2():
    print('Task 2 gestartet')
    await asyncio.sleep(3)
    print('Task 2 abgeschlossen')

async def main():
    await asyncio.gather(task_1(), task_2())

if __name__ == '__main__':
    asyncio.run(main())
```

Beispiel: Asynchrone Dateioperation

In diesem Beispiel zeigen wir, wie Dateioperationen asynchron ablaufen können. Das Beispiel liest den Inhalt von zwei Dateien asynchron.

```
import aiofiles
import asyncio

async def read_file(file_name):
    async with aiofiles.open(file_name, 'r') as file:
        contents = await file.read()
        print(f'Inhalt von {file_name}: {contents}')

async def main():
    await asyncio.gather(
        read_file('file1.txt'),
        read_file('file2.txt')
```

```
    )

if __name__ == '__main__':
    asyncio.run(main())
```

Beispiel: API-Call mit Zeitüberschreitung

Ein häufiges Problem bei Netzwerkanfragen ist die Notwendigkeit, eine maximale Wartezeit festzulegen. Dies kann durch die Verwendung von Timeout-Parametern und asynchroner Programmierung gelöst werden.

```
import httpx
import asyncio

async def fetch_data():
    async with httpx.AsyncClient() as client:
        try:
            response = await
client.get('https://run.mocky.io/v3/685db531-06e7-4d66-bbf6-99de9f2feab3?moc
ky-delay=3000ms', timeout=5)
            print(response.text)
        except httpx.RequestError as exc:
            print(f'An error occurred while requesting data: {exc}')

async def main():
    await fetch_data()

if __name__ == '__main__':
    asyncio.run(main())
```

Beispiel: Daten von mehreren APIs gleichzeitig abrufen

Hier zeigen wir, wie mehrere API-Aufrufe parallel ablaufen können, um die Effizienz zu verbessern.

```
import httpx
import asyncio


async def fetch_data(url):
    async with httpx.AsyncClient() as client:
        response = await client.get(url)
        return response.text


async def main():
    urls = [
'https://run.mocky.io/v3/685db531-06e7-4d66-bbf6-99de9f2feab3?mocky-delay=30
00ms',
```

```
'https://run.mocky.io/v3/685db531-06e7-4d66-bbf6-99de9f2feab3?mocky-delay=2000ms'
    ]
    results = await asyncio.gather(*(fetch_data(url) for url in urls))
    for result in results:
        print(result)


if __name__ == '__main__':
    asyncio.run(main())
```

Diese Beispiele zeigen verschiedene nützliche Anwendungsfälle für die asynchrone Programmierung in Python. Besonders bei I/O-intensiven Aufgaben hilft sie, die Blockierung des Programms zu vermeiden und die Gesamtleistung zu verbessern.