LU05c - Closures in Python

Ein `Closure` ist eine innere Funktion, die sich an den Zustand ihrer äußeren umgebenden Funktion erinnert, selbst wenn die äußere Funktion ihren Ausführungskontext bereits verlassen hat. Closures sind ein Konzept, das in vielen Programmiersprachen, einschließlich Python, vorkommt.

Grundlagen

- **Closure**: Eine innere Funktion, die Zugang zu den Variablen und dem Zustand ihrer äußeren Funktion hat, auch nachdem diese beendet wurde.
- **Äußere Funktion**: Die umschließende Funktion, die die innere Funktion definiert.
- **Innere Funktion**: Die Funktion, die als Closure dient, weil sie den Zustand der äußeren Funktion speichert.

Beispiel

Ein einfaches Beispiel für ein Closure:

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_five = outer_function(5)
print(add_five(3))  # Output: 8
```

In diesem Beispiel ist `inner_function` ein Closure, das den Wert von x speichert, selbst nachdem `outer_function` beendet wurde.

Animation: was im Speicher passiert

Die Animation zeigt genau dieses Beispiel Schritt für Schritt: wie der Rahmen von `outer_function` vom Aufruf-Stack verschwindet, während die Zelle mit `x = 5` bestehen bleibt. Der zweite Tab zeigt, dass zwei Aufrufe zwei **unabhängige** Zellen erzeugen.

Wie Closures arbeiten

- Innere Funktionen haben Zugriff auf die Variablen und den Zustand ihrer äußeren Funktion.
- Die innere Funktion ``erinnert`` sich an diesen Zustand, selbst wenn die äußere Funktion bereits beendet wurde.
- Das ermöglicht es, Zustand und Daten zwischen mehreren Aufrufen der inneren Funktion zu

speichern.

Closures in der Praxis

`add_five` zeigt den Mechanismus, aber niemand schreibt so etwas in echtem Code – dafür gibt es die Addition. Die folgenden vier Muster sind die Fälle, in denen Closures tatsächlich eingesetzt werden.

Die Faustregel: Ein Closure lohnt sich, wenn man eine Funktion braucht, die **eine bestimmte Einstellung schon kennt** – und man diese Einstellung nicht bei jedem Aufruf erneut mitgeben will.

1. Funktionsfabrik: konfigurierte Funktionen erzeugen

Statt bei jedem Aufruf den Steuersatz mitzugeben, erzeugt man einmal eine Funktion, die ihn bereits kennt:

```
def preis_mit_steuer(satz):
    def berechne(netto):
        return round(netto * (1 + satz / 100), 2)
    return berechne

schweiz = preis_mit_steuer(8.1)
deutschland = preis_mit_steuer(19)

print(schweiz(100))       # 108.1
print(deutschland(100))   # 119.0
```

`schweiz` und `deutschland` sind zwei unabhängige Funktionen mit je eigener gespeicherter Einstellung – genau das zeigt Tab 2 der Animation. Typische Vertreter: Rabattrechner, Einheitenumrechner, Validatoren mit unterschiedlichen Grenzwerten.

2. Kriterien für ''sorted()'' und ''filter()'' erzeugen

Funktionen wie `sorted()` erwarten eine Funktion mit genau einem Parameter. Wenn diese Funktion aber zusätzlich wissen muss, **wonach** sortiert werden soll, liefert ein Closure die Lösung:

```
def nach_feld(feld):
    def schluessel(eintrag):
        return eintrag[feld]
    return schluessel

produkte = [
    {"name": "Maus", "preis": 25},
    {"name": "Tastatur", "preis": 80},
```

```
    {"name": "Kabel", "preis": 12},
]

print(sorted(produkte, key=nach_feld("preis")))
print(sorted(produkte, key=nach_feld("name")))
```

So wird aus einer Sortier-Tabellenspalte im UI direkt eine Sortierfunktion – ohne für jede Spalte eine eigene Funktion zu schreiben.

3. Zustand kapseln statt globaler Variablen

Ein Closure kann sich zwischen den Aufrufen etwas merken. Mit `nonlocal` darf die innere Funktion die gespeicherte Variable auch verändern:

```
def zaehler_erstellen():
    anzahl = 0
    def hochzaehlen():
        nonlocal anzahl
        anzahl += 1
        return anzahl
    return hochzaehlen

klicks = zaehler_erstellen()
fehler = zaehler_erstellen()

print(klicks())   # 1
print(klicks())   # 2
print(fehler())   # 1  -> eigener, unabhängiger Zustand
```

Der entscheidende Punkt: `anzahl` ist von außen **nicht erreichbar**. Niemand kann den Zähler versehentlich auf 500 setzen. Das ist echte Datenkapselung – ohne globale Variable und ohne eigene Klasse. Genau darum geht es in der Aufgabe LU05.A09 - Refactoring: Vermeiden globaler Variablen durch Closures.

4. Callbacks in Oberflächen und Event-Systemen

Ein Button erwartet eine Funktion ohne Argumente. Trotzdem muss beim Klick klar sein, *welcher* Button gedrückt wurde – das Closure transportiert diese Information:

```
def klick_handler(button_name):
    def bei_klick():
        print(f"Button '{button_name}' wurde geklickt")
    return bei_klick

# button.on_click(klick_handler("Speichern"))
klick_handler("Speichern")()   # Button 'Speichern' wurde geklickt
```

Dasselbe Muster steckt hinter Event-Listenern in JavaScript, Callbacks in GUI-Frameworks und Retry-

Logik, die eine bereits konfigurierte Operation erneut ausführt.

5. Decorators

Decorators sind die verbreitetste Anwendung von Closures überhaupt: Die äußere Funktion nimmt eine Funktion entgegen, die innere „erinnert„ sich an sie und erweitert ihr Verhalten – etwa um Logging, Zeitmessung oder eine Berechtigungsprüfung. Details dazu auf LU05d - Decorators für Funktionen in Python.

Stolperfalle: späte Auswertung in Schleifen

Ein Closure speichert die **Variable**, nicht deren Wert zum Zeitpunkt der Definition. In einer Schleife führt das zu einem klassischen Fehler:

```
funktionen = [lambda: i for i in range(3)]
print([f() for f in funktionen])   # [2, 2, 2] – nicht [0, 1, 2]!
```

Alle drei Funktionen teilen sich dieselbe Variable `i`, die am Ende 2 ist. Die Lösung ist ein Default-Argument, das den Wert sofort kopiert:

```
funktionen = [lambda i=i: i for i in range(3)]
print([f() for f in funktionen])   # [0, 1, 2]
```

M323-LU05, M323-C2I