Rückblick: Funktional Programmieren

Verlangte Kompetenzen: Algorithmen und Teile von Applikationen deklarativ beschreiben und funktional implementieren.

1

Allgemeiner funktionaler `while`-Loop als Funktion:

Der Code unten verwendet funktionale Ansätze, um Schleifenverhalten zu realisieren, ohne dabei auf veränderbare Zustände zurückzugreifen, wie sie in imperativen Programmierparadigmen typisch sind.

Die zentrale Komponente der Implementierung ist die Funktion `function_while`, die zwei Funktionen als Parameter entgegennimmt: `cond` und `act`. Dabei legt `cond` die Abbruchbedingung fest, während `act` die in jeder Iteration auszuführende Operation definiert.

Die Funktion `function_while` nutzt Rekursion, um eine Schleifenfunktion zu erzeugen. Diese Schleifenfunktion führt die Aktion immer wieder auf den aktuellen Zustand aus, bis die festgelegte Abbruchbedingung erreicht ist.

Die erzeugte Schleifenfunktion bildet das Wiederholen von Aktionen auf eine einfache, funktionale Weise ab. Dabei kommt sie ganz ohne veränderbare Zustände aus und bleibt den Grundideen der funktionalen Programmierung treu.

```python
def function_while(cond, act):
    """
    Eine rekursive Schleifenfunktion, die rein funktional arbeitet.

    param cond: Eine Funktion, die den Zustand überprüft und True zurückgibt, wenn die Schleife fortgesetzt werden soll.
    param act:  Eine Funktion, die den Zustand verändert und einen neuen Zustand zurückgibt.
    return: Gibt eine Schleifenfunktion zurück, die den Zustand solange verändert,
            bis die Bedingung (cond) nicht mehr erfüllt ist.
    """
    def loop(state):
        if cond(state):  # Prüfe, ob die Bedingung erfüllt ist
            return loop(act(state))  # Rekursiver Aufruf mit verändertem Zustand
        else:
            return state  # Gibt den Endzustand zurück
    return loop


# Bedingung: Solange der aktuelle Wert kleiner oder gleich als 10 ist, läuft die Schleife weiter
def condition(state):
    # Der erste Wert im Tupel ist der aktuelle Zustand
    return state[0] < 10

# Aktion: Fügt das Quadrat des aktuellen Werts zum Tupel hinzu und gibt einen neuen Zustand zurück
def action(state):
    current, squares = state  # Tupel entpacken
    new_squares = squares + (current ** 2,)  # Neues Tupel erstellen (mit aktuellem Quadrat)
    return (current + 1, new_squares)  # Neuer Zustand als Tupel: aktueller Wert +1, neues Tupel

# Anfangszustand: Ein Tupel mit Startwert (0) und leerem Tupel für die Quadrate
initial_state = (0, ())

# Erstelle die Schleifenfunktion
func_result = function_while(condition, action)

# Starte die Schleife mit dem Anfangszustand
result = func_result(initial_state)

# Ausgabe der Ergebnisse
print(f'Quadrate: {result[1]}')  # Zeigt das Tupel der Quadrate
print(f'Anzahl Iterationen: {len(result[1])}')  # Zeigt die Anzahl der Iterationen
```

1. Keine Mutation: Der Zustand (`state`) wird nicht geändert. Stattdessen wird bei jeder Iteration ein neues `Tupel` erstellt.
2. Immutabilität: `Tupel` sind unveränderlich, wodurch unbeabsichtigte Änderungen ausgeschlossen sind.
3. Keine Seiteneffekte: Es gibt keine globalen Variablen oder Zustände, die von ausserhalb der Funktion beeinflusst werden.

2	Kennt den Unterschied zwischen imperativer und deklarativer/funktionaler Programmierung (deklaratives Programmierparadigma). Kennt Vorgehensweisen zur deklarativen Beschreibung von Problemen und Endzuständen. Weiterführende Dokumente: Funktionen als Grundlage der Programmierung Über die Funktionale Programmierung Der funktionale Stiel der Programmierung
3	Kennt Vor- und Nachteile funktionaler Programmierung. Weiterführende Dokumente: Beispiel: Ein (kaum) funktionstüchtiger Taschenrechner. Vorteil: Zustandslose Funktionen ohne Seiteneffekte liefern immer das gleiche Ergebnis (referenzielle Transparenz). Vorteil – Sie können (zumindest theoretisch!) beweisen, dass Ihr Code korrekt ist! Nachteil: (Übermässig) tiefe Rekursion Nachteil: Die funktionale Programmierung entspricht nicht (immer) der menschlichen Denkweisen.
4	Kennt Lambda Expression oder anonyme Funktionen in Python. • Kennt den Unterschied zwischen Statement und Expressions • Lambda Ausdrücke vertiefen • Durchflusskontrolle mit and und or • Lernen Sie inline 'if' Expressions!
5	Kennt Higher-Order Funktionen (Funktionen als Argumente und Rückgabewerte) in Python • Übergabe einer Funktion als Argument an eine andere Funktion • Verschachtelung einer Funktion in einer anderen Funktion • Rückgabe einer Funktion aus einer anderen Funktion • Das Operator-Modul – Operatoren als reguläre Funktionen • Dekoratoren (das Präfix @) • Dekoratoren mit Argumenten
6	Kennt funktionale Techniken und Entwurfsmuster in Python. • Currying – Ein Argument pro Funktion • Monaden – Variablen, die entscheiden, wie sie behandelt werden sollen • Memoisierung – Ergebnisse im Gedächtnis behalten • Closures – Operationen mit angehängten Daten
7	Fehler und Exception in Lambda Expressions in Python • Sie können keine Exceptions in Lamda-Expressions abfangen • Behandlung con Fehlern in Lambda-Expressions • Aufgabe: Ein voll funktionstüchtiger, interaktiver Taschenrechner • Functional Programming in Python