LU01c - Funktionale Programmierung

Definition der Funktionalen Programmierung

Funktionale Programmierung ist ein Paradigma, das Funktionen als primäre Bausteine verwendet, Unveränderlichkeit von Daten fördert und einen deklarativen Stil betont. Es zielt darauf ab, Code klarer und vorhersagbarer zu gestalten.

Grundprinzipien der Funktionalen Programmierung

1. **Funktionen als Erste Klasse:** Sie können wie jede andere Variable behandelt werden.
2. **Unveränderlichkeit:** Einmal erstellte Daten werden nicht geändert.
3. **Deklarative Natur:** Der Code beschreibt, was zu tun ist, nicht wie.

Funktion oder Prozedur?

Bevor wir über funktionale Programmierung sprechen, muss geklärt sein, was hier überhaupt mit „Funktion„ gemeint ist. Denn das, was viele Sprachen „Funktion“ nennen, ist streng genommen oft eine **Prozedur**.

Der Unterschied im Überblick

	Funktion	**Prozedur**
Zweck	Berechnet einen Wert und gibt ihn zurück	Führt eine Aktion aus
Rückgabewert	immer vorhanden	keiner (in Python: `None`)
Wirkung nach aussen	idealerweise keine	genau darin liegt ihr Zweck: Ausgabe, Datei, Netzwerk, globaler Zustand
Aufruf steht als	Ausdruck: `y = f(x)`	Anweisung: `f(x)`
Ersetzbarkeit	Der Aufruf kann durch sein Ergebnis ersetzt werden (referenzielle Transparenz)	nicht ersetzbar – die Wirkung ginge verloren
Analogie	Ein Taschenrechner: Eingabe rein, Ergebnis raus	Ein Drucker: nimmt etwas entgegen und *tut* etwas damit

Beispiel: derselbe Zweck, zwei Formen

```python
# Prozedur: gibt nichts zurück, wirkt nach aussen (Konsolenausgabe)
def zeige_summe(zahlen):
    print(sum(zahlen))
```

```
# Funktion: liefert einen Wert, verändert nichts ausserhalb
def berechne_summe(zahlen):
    return sum(zahlen)

zahlen = [1, 2, 3, 4, 5]

r1 = zeige_summe(zahlen)        # Ausgabe: 15
r2 = berechne_summe(zahlen)     # keine Ausgabe

print(r1)                       # None   ← die Prozedur liefert nichts
zurück
print(r2)                       # 15     ← die Funktion liefert das Ergebnis
```

Unterrichtstipp

Lassen Sie die Klasse *vor* dem Ausführen raten, was `print(r1)` ausgibt. Die Antwort None überrascht fast immer – und macht den Unterschied auf einen Schlag greifbar: Eine Prozedur *tut* etwas, eine Funktion *liefert* etwas.

Warum das für die funktionale Programmierung zentral ist

Nur Funktionen im obigen Sinn lassen sich gefahrlos kombinieren, wiederverwenden und testen. Vergleichen Sie:

```
# Prozedur mit globalem Zustand
total = 0

def addiere_zum_total(x):
    global total
    total += x

addiere_zum_total(5)
addiere_zum_total(5)
print(total)                    # 10 – das Ergebnis hängt von der
Vorgeschichte ab


# Funktion ohne globalen Zustand
def addiere(a, b):
    return a + b

print(addiere(5, 5))            # 10
print(addiere(5, 5))            # 10 – immer dasselbe, egal wie oft
```

Der entscheidende Punkt: `addiere(5, 5)` kann man im Kopf jederzeit durch `10` ersetzen, ohne dass sich am Programm etwas ändert. Bei `addiere_zum_total(5)` geht das nicht – man muss den bisherigen Verlauf des Programms kennen, um zu wissen, was passiert. Genau diese Ersetzbarkeit heisst **referenzielle Transparenz** und ist der Grund, warum funktionaler Code leichter zu testen und

zu verstehen ist.

Verwandte Begriffe zur Abgrenzung

- **Methode**: Eine Funktion oder Prozedur, die zu einem Objekt gehört. In Python gut am Punkt erkennbar: `liste.append(4)`. Methoden verändern typischerweise den Zustand ihres Objekts – sind also meist Prozeduren.
- **Mathematische Funktion**: Bildet jede Eingabe eindeutig auf genau eine Ausgabe ab und tut sonst nichts. Genau dieses Ideal greift die funktionale Programmierung mit dem Begriff der **pure function** auf (siehe LU02).

Das Lehrbuch-Beispiel: `sort()` vs. `sorted()`

Python bietet dieselbe Aufgabe in beiden Formen an – ideal zum Durchspielen:

```
liste = [3, 1, 2]
print(liste.sort())     # None       ← Prozedur (Methode):
verändert die Liste
print(liste)            # [1, 2, 3]

liste2 = [3, 1, 2]
print(sorted(liste2))   # [1, 2, 3]  ← Funktion: liefert
eine neue Liste
print(liste2)           # [3, 1, 2]  ← Original bleibt
unverändert
```

`sorted()` ist die funktionale Variante: Sie gibt einen Wert zurück und lässt die Eingabe in Ruhe.

Achtung: In Python heisst beides `def`

Python kennt syntaktisch nur `def`. Ob etwas eine Funktion oder eine Prozedur ist, entscheidet also nicht das Schlüsselwort, sondern das **Verhalten**: Gibt es einen Wert zurück? Verändert es etwas ausserhalb?

Andere Sprachen trennen das explizit: Pascal unterscheidet `function` und `procedure`, Visual Basic `Function` und `Sub`.

Einführung in die Deklarative Programmierung

Deklarative Programmierung ist ein übergeordnetes Konzept, das funktional, logisch und einigen anderen Paradigmen gemeinsam ist. Es betont die Beschreibung der gewünschten Ergebnisse, ohne spezifische Anweisungen zu geben, wie diese Ergebnisse erreicht werden sollen.

Beispiel

Quadrieren einer Liste in Python auf funktionale Weise:

```
def quadrat(x):
    return x**2

zahlen = [1, 2, 3, 4, 5]
quadrate = map(quadrat, zahlen)
print(list(quadrate)) # Ausgabe: [1, 4, 9, 16, 25]
```

Warum das `list()` nötig ist

`map` berechnet die Werte nicht sofort, sondern liefert ein *Iterator-Objekt*, das die Quadrate erst bei Bedarf einzeln erzeugt. Man nennt das **Lazy Evaluation**. Ein `print(quadrate)` ohne `list()` gibt darum nur etwas wie `<map object at 0x…>` aus. Erst `list()` fordert alle Werte an und macht sie sichtbar.

Der Vorteil: Auch über Millionen von Elementen kann man so rechnen, ohne alle gleichzeitig im Speicher zu halten.

Fokus auf Funktionen, Unveränderlichkeit und Deklarativität

Funktionen

Funktionen sind das Herzstück der funktionalen Programmierung:

```
def anwenden(f, wert):
    return f(wert)

quadrat = lambda x: x**2
ergebnis = anwenden(quadrat, 5)
print(ergebnis) # Ausgabe: 25
```

Hier passiert etwas Bemerkenswertes: `quadrat` wird wie ein normaler Wert in einer Variablen abgelegt und als Argument weitergegeben. Das meint **Funktionen als Erste Klasse** (*first-class*). Eine Funktion wie anwenden, die selbst eine Funktion entgegennimmt oder zurückgibt, nennt man **höherwertige Funktion** (*higher-order function*).

Unveränderlichkeit

Die Unveränderlichkeit fördert sauberen, nachvollziehbaren Code:

```
original_liste = [1, 2, 3]
```

```
neue_liste = original_liste + [4] # Original bleibt unverändert
```

Deklarativität

Ein deklarativer Ansatz beschreibt, was getan werden soll:

```
# Deklarativ
summe = sum(zahlen)
```

Zusammenführung der Deklarativen und Funktionalen Programmierung

Deklarative Programmierung ist ein breiteres Konzept, das die funktionale Programmierung einschliesst. Während die funktionale Programmierung sich auf die Verwendung von Funktionen und Unveränderlichkeit konzentriert, bezieht sich die deklarative Programmierung auf die Abstraktion des „wie„ und konzentriert sich auf das „was“.

Mini-Übung

Funktion oder Prozedur? Ordnen Sie zu und begründen Sie:

1. `print(„Hallo“)`
2. `len([1, 2, 3])`
3. `liste.append(4)`
4. `round(3.14159, 2)`
5. `datei.write(„Text“)`
6. `max(noten)`

Umbauen: Schreiben Sie diese Prozedur in eine Funktion um:

```
def zeige_durchschnitt(zahlen):
    print(sum(zahlen) / len(zahlen))
```

Lösungshinweise

Funktionen: `len`, `round`, `max` – sie liefern einen Wert und verändern nichts. Prozeduren: `print`, `append`, `write` – sie geben `None` zurück und wirken nach aussen (Konsole, Liste, Datei).

Umbau: `return sum(zahlen) / len(zahlen)` statt `print(…)`. Die Ausgabe übernimmt dann die aufrufende Stelle – so bleibt die Berechnung testbar und wiederverwendbar.

Schlussfolgerung

Die funktionale Programmierung, mit Schwerpunkt auf der Verwendung von Funktionen, Unveränderlichkeit und einem deklarativen Ansatz, bietet einen klaren und wartbaren Weg, um Code zu schreiben. Sie ist Teil der grösseren deklarativen Paradigmenfamilie und findet in modernen Programmiersprachen wie Python Anwendung. Die Kombination von funktionalen und deklarativen Prinzipien führt zu Code, der einfacher zu lesen, zu verstehen und zu testen ist.

M323-LU01, M323-A1B

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m323/learningunits/lu01/funktional

Last update: **2026/08/18 10:13**