LU01.L09 - Vom Skript zur Funktion

1. Analyse der Ausgangslage

Alle drei `def`-Blöcke sind **Prozeduren**:

Block	Warum Prozedur?
`artikel_hinzufuegen`	Kein `return`. Wirkt nach aussen: verändert die globale Liste `lager` und die globale Zahl `gesamtwert`.
`zeige_gesamtwert`	Kein `return`. Wirkt nach aussen: schreibt auf die Konsole. Liest zudem globalen Zustand statt Parameter.
`zeige_teuersten`	Kein `return`. Berechnet zwar etwas, gibt es aber nicht heraus, sondern druckt es direkt.

Alle drei geben None zurück und lassen sich deshalb nicht als Ausdruck verwenden.

2. Musterlösung

```
def artikel_hinzufuegen(lager, name, menge, preis):
    """Gibt ein NEUES Lager mit dem zusätzlichen Artikel zurück."""
    return lager + [{"name": name, "menge": menge, "preis": preis}]


def gesamtwert(lager):
    """Berechnet den Gesamtwert aller Artikel im Lager."""
    total = 0
    for artikel in lager:
        total += artikel["menge"] * artikel["preis"]
    return total


def teuerster_artikel(lager):
    """Gibt den teuersten Artikel zurück, bei leerem Lager None."""
    if not lager:
        return None
    teuerster = lager[0]
    for artikel in lager[1:]:
        if artikel["preis"] > teuerster["preis"]:
            teuerster = artikel
    return teuerster


def formatiere_gesamtwert(lager):
    """Erzeugt den Ausgabetext - gibt ihn zurück, statt ihn zu drucken."""
    return f"Gesamtwert: {gesamtwert(lager):.2f} CHF"
```

```
if __name__ == '__main__':
    lager = []
    lager = artikel_hinzufuegen(lager, 'Maus', 10, 24.90)
    lager = artikel_hinzufuegen(lager, 'Tastatur', 4, 79.50)
    lager = artikel_hinzufuegen(lager, 'Monitor', 2, 249.00)

    print(formatiere_gesamtwert(lager))             # Gesamtwert: 1065.00
CHF
    print(f"Teuerster Artikel: {teuerster_artikel(lager)['name']}")  #
Monitor

    # Nachweis 1: derselbe Aufruf liefert immer dasselbe Ergebnis
    print(gesamtwert(lager) == gesamtwert(lager))    # True

    # Nachweis 2: das Original bleibt unverändert
    erweitert = artikel_hinzufuegen(lager, 'Dock', 1, 189.00)
    print(len(lager), len(erweitert))                # 3 4

    # Randfall: leeres Lager
    print(teuerster_artikel([]))                     # None
```

Zum + in `artikel_hinzufuegen`

`lager + [neuer_artikel]` erzeugt eine **neue** Liste, `lager.append(…)` verändert die bestehende. Genau darin liegt der Unterschied zwischen der funktionalen und der prozeduralen Variante.

Der Preis: Die aufrufende Stelle muss das Ergebnis auffangen (`lager = artikel_hinzufuegen(…)`). Wer das vergisst, verliert den neuen Artikel – dafür kann er aber nie versehentlich fremde Daten beschädigen.

3. Antworten auf die Kontrollfragen

Frage 1: Ersetzbarkeit

`gesamtwert(lager)` hängt **nur** von seinem Parameter ab und verändert nichts. Der Aufruf kann darum jederzeit im Kopf durch seine Zahl ersetzt werden – das Programm verhält sich identisch. Diese Eigenschaft heisst **referenzielle Transparenz**.

Beim alten `artikel_hinzufuegen('Maus', 10, 24.90)` geht das nicht: Der Aufruf hat gar keinen Rückgabewert, den man einsetzen könnte, und seine Wirkung hängt davon ab, was vorher schon im globalen `lager` stand. Man muss die ganze Vorgeschichte des Programms kennen, um zu wissen, was passiert.

Frage 2: Rückgabewert einer Prozedur

```
ergebnis = zeige_gesamtwert()
print(ergebnis)     # None
```

Eine Prozedur ohne `return` liefert in Python implizit `None`. Die Zahl war zwar auf der Konsole zu sehen, ist im Programm aber nirgends mehr greifbar – sie lässt sich nicht weiterverwenden, nicht weiterrechnen und nicht testen.

Frage 3: Testbarkeit

Automatisch testbar sind `gesamtwert`, `teuerster_artikel`, `artikel_hinzufuegen` und `formatiere_gesamtwert`: Man ruft sie mit einer bekannten Eingabe auf und vergleicht den Rückgabewert mit dem erwarteten Wert.

```
def test_gesamtwert():
    lager = [{"name": "A", "menge": 2, "preis": 10.0}]
    assert gesamtwert(lager) == 20.0

def test_teuerster_bei_leerem_lager():
    assert teuerster_artikel([]) is None
```

Die alten Prozeduren lassen sich so nicht prüfen: Sie geben nichts zurück. Man müsste die Konsolenausgabe abfangen und zusätzlich den globalen Zustand vor jedem Test zurücksetzen – aufwendig und fehleranfällig. Das ist der praktische Grund, warum die funktionale Programmierung Berechnung und Ausgabe trennt.
