LU01.L09 - Vom Skript zur Funktion

1. Analyse der Ausgangslage

Alle drei `def`-Blöcke in `ausgangslage.py` 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

main.py

main.py

```
"""LU01.L09 - Musterloesung: Vom Skript zur Funktion."""


def artikel_hinzufuegen(lager, name, menge, preis):
    """
    Gibt ein NEUES Lager zurueck, das zusaetzlich den angegebenen
Artikel enthaelt.

    :param lager: Liste von Artikeln
    :param name: Name des neuen Artikels
    :param menge: Stueckzahl des neuen Artikels
    :param preis: Stueckpreis des neuen Artikels
    :return: neue Liste mit allen bisherigen Artikeln plus dem neuen
    """
    return lager + [{'name': name, 'menge': menge, 'preis': preis}]


def gesamtwert(lager):
    """
    Berechnet den Gesamtwert aller Artikel im Lager.

    :param lager: Liste von Artikeln
    :return: Gesamtwert als Zahl, bei leerem Lager 0
```

```
    """
    total = 0
    for artikel in lager:
        total += artikel['menge'] * artikel['preis']
    return total


def teuerster_artikel(lager):
    """
    Sucht den Artikel mit dem hoechsten Stueckpreis.

    :param lager: Liste von Artikeln
    :return: der teuerste Artikel, 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 zum Gesamtwert.

    :param lager: Liste von Artikeln
    :return: Ausgabetext als String
    """
    return f'Gesamtwert: {gesamtwert(lager):.2f} CHF'


def main():
    """Baut ein Lager auf und gibt die Auswertung aus."""
    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)

    name = teuerster_artikel(lager)['name']
    print(formatiere_gesamtwert(lager))
    print(f'Teuerster Artikel: {name}')


if __name__ == '__main__':
    main()
```

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. Die Tests, die den Kern treffen

Zwei der neun Tests prüfen genau das, worum es didaktisch geht:

main_test.py

main_test.py

```python
"""Tests zu LU01.A09 - Vom Skript zur Funktion."""

import pytest

import main

LAGER = [
    {'name': 'Maus', 'menge': 10, 'preis': 24.90},
    {'name': 'Tastatur', 'menge': 4, 'preis': 79.50},
    {'name': 'Monitor', 'menge': 2, 'preis': 249.00},
]


def test_gesamtwert():
    assert main.gesamtwert(LAGER) == pytest.approx(1065.00)


def test_gesamtwert_leer():
    assert main.gesamtwert([]) == 0


def test_gesamtwert_ist_wiederholbar():
    erster = main.gesamtwert(LAGER)
    zweiter = main.gesamtwert(LAGER)
    assert erster == pytest.approx(1065.00)
    assert zweiter == pytest.approx(1065.00)


def test_artikel_hinzufuegen_liefert_neue_liste():
    erweitert = main.artikel_hinzufuegen([], 'Dock', 1, 189.00)
```

```
    assert erweitert == [{'name': 'Dock', 'menge': 1, 'preis': 189.00}]


def test_artikel_hinzufuegen_veraendert_original_nicht():
    original = [{'name': 'Maus', 'menge': 10, 'preis': 24.90}]
    erweitert = main.artikel_hinzufuegen(original, 'Dock', 1, 189.00)
    assert len(original) == 1
    assert len(erweitert) == 2


def test_teuerster_artikel():
    assert main.teuerster_artikel(LAGER)['name'] == 'Monitor'


def test_teuerster_artikel_leer():
    assert main.teuerster_artikel(LAGER) is not None
    assert main.teuerster_artikel([]) is None


def test_formatiere_gesamtwert():
    assert main.formatiere_gesamtwert(LAGER) == 'Gesamtwert: 1065.00
CHF'


def test_keine_ausgabe_in_berechnung(capsys):
    ergebnisse = [
        main.gesamtwert(LAGER),
        main.teuerster_artikel(LAGER),
        main.formatiere_gesamtwert(LAGER),
        main.artikel_hinzufuegen(LAGER, 'Dock', 1, 189.00),
    ]
    ausgabe = capsys.readouterr().out
    assert None not in ergebnisse
    assert ausgabe == ''
```

- `test_artikel_hinzufuegen_veraendert_original_nicht` fällt durch, sobald jemand `append` auf der übergebenen Liste verwendet.
- `test_keine_ausgabe_in_berechnung` fällt durch, sobald eine berechnende Funktion druckt oder `None` liefert.

Dazu kommt das Linting: Ein `global` in `main.py` löst `W0603 global-statement` aus und kostet Punkte.

4. 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.
