LU01.L13 - Dasselbe Problem in drei Paradigmen

Diese Aufgabe gehört inhaltlich zu LU01, wird aber erst **nach LU04** bearbeitet – die funktionale Fassung setzt `map`, `filter` und `reduce` voraus.

1. prozedural.py

```
noten = [('Alina', 5.5), ('Ben', 3.5), ('Chiara', 4.0), ('Dario', 5.0),
('Elif', 3.9)]

if __name__ == '__main__':
    summe = 0
    ungenuegend = []
    beste = noten[0][1]

    for name, note in noten:
        summe += note
        if note < 4.0:
            ungenuegend.append(name)
        if note > beste:
            beste = note

    durchschnitt = summe / len(noten)

    print(f'Durchschnitt: {durchschnitt:.2f}')
    print(f'Ungenuegend: {ungenuegend}')
    print(f'Beste Note: {beste}')
```

Drei Variablen werden parallel fortgeschrieben. Wer eine vierte Kennzahl braucht, ergänzt eine vierte Variable und eine weitere Zeile im Schleifenkörper – die Schleife wird mit jeder Kennzahl unübersichtlicher.

2. objektorientiert.py

```
class Klasse:
    def __init__(self, noten=None):
        self._noten = list(noten) if noten else []

    def hinzufuegen(self, name, note):
        self._noten.append((name, note))     # veraendert den eigenen
Zustand

    def durchschnitt(self):
```

```
        return sum(note for _, note in self._noten) / len(self._noten)

    def ungenuegende(self):
        return [name for name, note in self._noten if note < 4.0]

    def beste_note(self):
        return max(note for _, note in self._noten)


if __name__ == '__main__':
    klasse = Klasse([('Alina', 5.5), ('Ben', 3.5), ('Chiara', 4.0),
                     ('Dario', 5.0), ('Elif', 3.9)])

    print(f'Durchschnitt: {klasse.durchschnitt():.2f}')
    print(f'Ungenuegend: {klasse.ungenuegende()}')
    print(f'Beste Note: {klasse.beste_note()}')
```

3. funktional.py

```
from functools import reduce


def noten_werte(noten):
    """Loest die Noten aus den Paaren heraus."""
    return map(lambda paar: paar[1], noten)


def durchschnitt(noten):
    return reduce(lambda a, b: a + b, noten_werte(noten)) / len(noten)


def ungenuegende(noten):
    return list(map(lambda paar: paar[0],
                    filter(lambda paar: paar[1] < 4.0, noten)))


def beste_note(noten):
    return reduce(lambda a, b: a if a > b else b, noten_werte(noten))


def note_hinzufuegen(noten, name, note):
    return noten + [(name, note)]           # neue Liste statt Veraenderung


if __name__ == '__main__':
    noten = [('Alina', 5.5), ('Ben', 3.5), ('Chiara', 4.0),
             ('Dario', 5.0), ('Elif', 3.9)]
```

```
    print(f'Durchschnitt: {durchschnitt(noten):.2f}')
    print(f'Ungenuegend: {ungenuegende(noten)}')
    print(f'Beste Note: {beste_note(noten)}')
```

Kürzer geht es mit den eingebauten Funktionen

`sum` und `max` sind selbst schon Reduktionen – `reduce` von Hand zu schreiben, macht hier nur sichtbar, was in ihnen steckt:

```
def durchschnitt(noten):
    return sum(note for _, note in noten) / len(noten)

def beste_note(noten):
    return max(note for _, note in noten)
```

Beide Fassungen sind funktional. Verlangt ist in der Aufgabe die explizite mit `reduce`, weil man daran sieht, dass *jede* Aggregation dasselbe Muster hat: Startwert, Verknüpfung, Ergebnis.

Vergleichstabelle

Kriterium	Prozedural	Objektorientiert	Funktional
Wo liegt der Zustand?	In lokalen Variablen des Skripts, die während der Schleife wachsen.	Im Objekt, in `self._noten` – er überlebt jeden Methodenaufruf.	Nirgends dauerhaft. Die Daten werden als Argument hineingegeben und als Ergebnis herausgereicht.
Zweimal dieselbe Berechnung?	Der Schleifencode müsste komplett wiederholt werden, inklusive Zurücksetzen der Variablen.	Kann zwei verschiedene Werte liefern, sobald zwischendurch `hinzufuegen` aufgerufen wurde.	Liefert garantiert immer dasselbe – das Ergebnis hängt nur vom Argument ab.
Wie testet man einzeln?	Gar nicht sauber: Die Berechnung steckt im Hauptprogramm und ist nicht aufrufbar.	Man muss zuerst ein Objekt in einen bestimmten Zustand bringen, dann die Methode aufrufen.	`assert durchschnitt(testdaten) == erwartet` – eine Zeile, kein Aufbau nötig.
Was muss man wissen, um eine Zeile zu verstehen?	Den bisherigen Verlauf der Schleife: Welche Werte haben `summe`, `beste` gerade?	Den aktuellen Zustand des Objekts – der von aussen nicht sichtbar ist.	Nur die Signatur der Funktion und ihre Argumente.
Gleichzeitiger Zugriff?	Problematisch: geteilte Variablen können sich gegenseitig überschreiben.	Problematisch: zwei Programmteile mit derselben Objektreferenz sehen Änderungen des jeweils anderen.	Unkritisch: Es gibt keinen geteilten veränderlichen Zustand.
Anzahl Codezeilen	mittel (~18)	am meisten (~20, plus Klassengerüst)	am wenigsten im Berechnungsteil, dafür in unabhängigen Einheiten

Zur Zeilenzahl

Die Zeilenzahl ist das schwächste der sechs Kriterien. Bewerten Sie in Ihrer Abgabe vor allem die ersten fünf Zeilen der Tabelle – Kürze und Qualität sind nicht dasselbe.

Antworten auf die Reflexionsfragen

1. Eine vierte Kennzahl ergänzen

Am einfachsten funktional: Man schreibt eine weitere unabhängige Funktion `median(noten)`. Nichts Bestehendes wird angefasst, nichts kann kaputtgehen.

Am schwierigsten prozedural: Die neue Kennzahl braucht eine weitere Variable vor der Schleife und eine weitere Zeile darin. Die Schleife übernimmt damit noch eine Aufgabe mehr – ein Verstoss gegen das Prinzip, dass ein Codeblock für genau eine Sache zuständig sein soll.

Die OO-Fassung liegt dazwischen: Eine neue Methode ist schnell ergänzt, sie hängt aber am Zustand des Objekts.

2. Warum derselbe Aufruf zwei Ergebnisse liefert

```
klasse = Klasse(noten)
print(round(klasse.durchschnitt(), 3))   # 4.38
klasse.hinzufuegen('Fabio', 2.0)
print(round(klasse.durchschnitt(), 3))   # 3.983
```

`durchschnitt()` hat keine Argumente. Sein Ergebnis hängt ausschliesslich vom verborgenen Zustand `self._noten` ab – und der hat sich zwischen den beiden Aufrufen geändert. Der Aufruf ist damit **nicht referenziell transparent**: Man kann ihn nicht durch seinen Wert ersetzen, ohne die Vorgeschichte zu kennen.

In der funktionalen Fassung ist das ausgeschlossen, weil die Daten sichtbar im Argument stehen:

```
print(durchschnitt(noten))                          # 4.38
erweitert = note_hinzufuegen(noten, 'Fabio', 2.0)
print(durchschnitt(noten))                          # 4.38 - unveraendert
print(durchschnitt(erweitert))                      # 3.983 - andere Daten,
anderer Name
```

Unterschiedliche Ergebnisse haben hier unterschiedliche Namen. Genau das ist der Gewinn.

3. Die OO-Fassung ist im Kern imperativ

Erkennbar an drei Punkten:

- `hinzufuegen` gibt nichts zurück (None) und *tut* stattdessen etwas – es ist eine **Prozedur** im Sinn von LU01c.
- `self._noten.append(…)` verändert eine bestehende Datenstruktur, statt eine neue zu erzeugen.
- Die Reihenfolge der Aufrufe entscheidet über das Ergebnis – das Kennzeichen zustandsbehafteter, imperativer Programmierung.

Objektorientierung ist also **kein Gegensatz** zur imperativen Programmierung, sondern eine Art, imperativen Code zu organisieren: Sie bündelt Zustand und die darauf arbeitenden Prozeduren zu Einheiten. Der Gegensatz verläuft zwischen *Zustand verändern* und *Werte erzeugen*, nicht zwischen *OO* und *funktional*.

4. Leere Notenliste

```
TypeError: reduce() of empty iterable with no initial value
```

`reduce` ohne Startwert nimmt das erste Element als Anfang – bei einer leeren Folge gibt es keines. Zwei saubere Behandlungen:

```
# Variante A: Startwert angeben
def summe(noten):
    return reduce(lambda a, b: a + b, noten_werte(noten), 0)

# Variante B: den Randfall zur Bedingung der Funktion machen
def durchschnitt(noten):
    if not noten:
        raise ValueError('Durchschnitt einer leeren Notenliste ist
undefiniert')
    return reduce(lambda a, b: a + b, noten_werte(noten)) / len(noten)
```

Bei der Summe ist `0` ein sinnvoller Startwert. Beim Durchschnitt nicht: Er wäre eine Division durch null. Und bei `beste_note` gibt es überhaupt keinen neutralen Startwert – hier ist Variante B die einzig ehrliche Antwort. Das ist ein guter Moment, um zu sehen, dass nicht jede Aggregation einen sinnvollen Anfangswert hat.

5. Wann welches Paradigma?

Mögliche Antworten – die Beispiele der Lernenden dürfen abweichen, müssen aber begründet sein:

Paradigma	Passende Situation	Begründung
Prozedural	Ein kurzes Wartungsskript, das einmal pro Monat Logdateien aufräumt.	Der Aufwand für Struktur lohnt sich nicht; das Skript hat einen Ablauf und ein Ende.
Objektorientiert	Eine GUI-Anwendung oder ein Spiel, in dem Objekte über die Zeit ihren Zustand ändern (Fenster, Spielfigur, Warenkorb).	Zustand ist hier der Zweck der Sache, nicht der Feind. Ein Objekt bündelt ihn an einer klar zuständigen Stelle.

Paradigma	Passende Situation	Begründung
Funktional	Datenaufbereitung und Auswertung: Importe filtern, transformieren, aggregieren – und alles, was parallel oder verteilt läuft.	Ohne geteilten veränderlichen Zustand sind die Schritte einzeln testbar, beliebig kombinierbar und parallelisierbar.

Fazit: In der Praxis werden die drei Stile gemischt. Eine typische Anwendung hat objektorientierte Struktur an der Oberfläche, einen funktionalen Kern für die Berechnungen und prozedurale Ränder dort, wo mit Dateien, Netzwerk und Benutzern interagiert wird.
