Modul M323 – Funktionales Programmieren

Bank Marketing Data – Kundenanalyse & Abschlussprognose

Projekt-Dokumentation

Autoren:	Peter Ngo, Alex Uscata
Klasse :	INA 23A
Dozent:	Dieter Kopp
Datum:	4. Januar 2026

Inhaltsverzeichnis

1 Einleitung

Im Rahmen des Moduls M323 – Funktionales Programmieren wird in diesem Projekt eine Datenanalyse einer Bank-Marketingkampagne umgesetzt. Ziel ist es, eine imperativ geschriebene Lösung mit einer funktional refaktorierten Version zu vergleichen und die Vor- sowie Nachteile funktionaler Sprachelemente praxisnah zu evaluieren.

Als Datengrundlage dient ein realitätsnaher Datensatz einer portugiesischen Bank, welcher Kundendaten sowie den Erfolg einer Marketingkampagne enthält. Beide Programmversionen erzeugen denselben Output, unterscheiden sich jedoch grundlegend in der Programmierweise.

2 Wahl der imperativen Programmiersprache

2.1 Begründung der Sprachwahl

Für dieses Projekt wurde die Programmiersprache **Python** gewählt. Python eignet sich besonders gut für Datenanalysen, da die Sprache:

- leicht lesbar und verständlich ist
- imperative Programmierung vollständig unterstützt
- funktionale Sprachelemente integriert anbietet
- häufig im beruflichen Umfeld eingesetzt wird

Da Python sowohl imperative als auch funktionale Konzepte vereint, ist die Sprache ideal geeignet, um beide Programmierparadigmen direkt miteinander zu vergleichen.

2.2 Unterstützte funktionale Elemente in Python

Python ist keine rein funktionale Sprache, bietet jedoch zahlreiche funktionale Sprachelemente aus der Standardbibliothek:

- `map()` – Transformation von Daten
- `filter()` – Selektion von Elementen
- `reduce()` (aus `functools`) – Aggregation
- Lambda-Funktionen
- List Comprehensions
- Unveränderliche Datentypen (z. B. Tupel)

Diese Elemente ermöglichen eine deklarative und kompakte Beschreibung von Datenverarbeitungslogik.

2.2.1 Kurzbeispiele der funktionalen Elemente

```
# map: Transformation
list(map(lambda x: x*x, [1,2,3]))       # -> [1,4,9]

# filter: Selektion
list(filter(lambda x: x > 2, [1,2,3])) # -> [3]

# reduce: Aggregation
from functools import reduce
reduce(lambda acc, x: acc + x, [1,2,3], 0)   # -> 6
```

3 Projektantrag

3.1 Ausgangslage

Während einer Marketingkampagne hat eine Bank verschiedene Kundendaten erfasst. Der Datensatz liegt in Form einer CSV-Datei vor und ist nicht direkt für Analysen aufbereitet.

Ziel ist es, relevante Kennzahlen zu berechnen, Kundengruppen zu vergleichen und statistische Zusammenhänge zu untersuchen.

3.2 Zielsetzung

Das Projekt verfolgt folgende Ziele:

- Analyse der Erfolgsquote der Marketingkampagne
- Vergleich verschiedener Kundengruppen
- Umsetzung einer imperativen Version (V1.0)
- Refactoring in eine funktionale Version (V2.0)
- Vergleich beider Ansätze hinsichtlich Lesbarkeit und Wartbarkeit

3.3 Projektumfang

Das Projekt umfasst:

- Einlesen und Verarbeiten eines CSV-Datensatzes
- Textbasierte Konsolenausgabe
- Keine externen Bibliotheken
- Umsetzung in Python; Ausführung und Dokumentation der Experimente in einem Jupyter Notebook (Tooling).

4 Programmausgabe

Beide Versionen des Programms erzeugen denselben Output. Die Ausgabe erfolgt textbasiert in der Konsole.

4.1 Beispielhafter Output

```
============================
Bankkampagne     Erfolgsquote

Anzahl Kunden: 4521
Abschl sse (yes): 512
Erfolgsquote: 11.3 %

============================
Vergleich nach Education

primary   |    Balance: 850  | Erfolgsquote: 8.5 %
secondary |    Balance: 1200 | Erfolgsquote: 10.1 %
tertiary  |    Balance: 1800 | Erfolgsquote: 14.3 %

ANOVA F-Wert: 5.21
```

Hinweis: Version 1.0 (imperativ) und Version 2.0 (funktional) liefern identische Resultate.

5 Refactoring: Vergleich Version 1.0 vs. Version 2.0

Beide Programmversionen implementieren dieselben Produktfunktionen und erzeugen identische Resultate, unterscheiden sich jedoch in der Umsetzung: Version 1.0 verwendet primär imperative Sprachmittel (Schleifen, Hilfsvariablen, schrittweiser Aufbau von Resultaten), während Version 2.0 die gleiche Logik mit funktionalen Sprachmitteln (z. B. `map`, `filter`, `reduce`, Lambdas) ausdrückt.

5.1 Konkrete Beispiele aus dem Projekt

Beispiel A: Filtern von Datensätzen In Version 1.0 wird die Filterlogik mit einer Schleife und `continue` umgesetzt. In Version 2.0 wird dieselbe Logik deklarativ als Prädikatfunktion formuliert und per `filter` angewendet.

```python
# V1.0 (imperativ) - skizziert
out = []
for row in data:
    if housing is not None and row["housing"] is not housing:
        continue
    if loan is not None and row["loan"] is not loan:
        continue
    out.append(row)

```

```
# V2.0 (funktional) - skizziert
def pred(row):
    ...
    return True/False

out = list(filter(pred, data))
```

Beispiel B: Aggregation (Tabellenbreiten via reduce) Für die tabellarische Ausgabe werden in Version 2.0 Spaltenbreiten über ein `reduce`-Faltungsmuster berechnet, während Version 1.0 die Breiten schrittweise in Schleifen aktualisiert.

```
# V2.0 (funktional) - skizziert
widths = reduce(update_widths, rows, widths_init)
```

Beispiel C: Transformation numerischer Werte (map) Transformationen wie `balance^2 + 1` werden in Version 2.0 direkt als Abbildung über die Liste beschrieben (`map` + Lambda), während Version 1.0 die Werte typischerweise in einer Schleife sammelt.

```
# V2.0 (funktional) - skizziert
sq1 = list(map(lambda b: b*b + 1.0, balances))
```

5.2 Zusammenfassung der Unterschiede

- **Lesbarkeit:** Version 2.0 ist kompakter und beschreibt häufig *was* berechnet wird, statt *wie*.
- **Wartbarkeit:** Wiederverwendbare Hilfsfunktionen (Prädikate, Mapping-Funktionen) reduzieren Duplikation.
- **Fehleranfälligkeit:** Weniger mutable Zwischenzustände und weniger Hilfsvariablen verringern typische Fehlerquellen.

6 Fazit

6.1 Nutzen funktionaler Elemente

Der Einsatz funktionaler Sprachelemente hat in diesem Projekt mehrere Vorteile gebracht. Insbesondere Filter- und Transformationsoperationen konnten kompakter und klarer formuliert werden.

6.2 Vereinfachung durch Refactoring

Im Vergleich zur imperativen Version reduzierte sich die Code-Länge in Version 2.0 deutlich. Verschachtelte Schleifen und Hilfsvariablen konnten durch deklarative Ausdrücke ersetzt werden. Dies führte zu besserer Lesbarkeit und weniger Fehleranfälligkeit.

6.3 Wiederverwendung funktionaler Konzepte

Die funktionalen Konzepte würden in zukünftigen Projekten erneut eingesetzt werden, insbesondere bei:

- Datenfiltern
- Aggregationen
- Transformationen von Listen und Datensätzen

6.4 Anwendungsfälle im beruflichen Umfeld

Im betrieblichen Kontext eignen sich funktionale Sprachelemente besonders für:

- Datenanalyse
- Reporting
- Verarbeitung von Log- oder Kundendaten
- Automatisierte Auswertungen

Zusammenfassend lässt sich sagen, dass funktionale Programmieransätze eine sinnvolle Ergänzung zur imperativen Programmierung darstellen und in vielen Anwendungsfällen zu einer klareren und effizienteren Lösung führen.

Hinweis: Der vollständige Quellcode befindet sich im Projekt-Repository: github.com/stoicfist/Modul-323-Projektarbeit. Er wurde in dieser Dokumentation bewusst nicht vollständig abgedruckt, um den Fokus auf Konzept und Vergleich zu legen.