Modul M323 – Funktionales Programmieren

Bank Marketing Data

Kundenanalyse & Abschlussprognose

Projekt-Dokumentation

Autoren: Peter Ngo, Alex Uscata

Klasse: INA 23A

Dozent: Dieter Kopp

Datum: 10. Januar 2026

Inhaltsverzeichnis

1 Einleitung **3**

2 Wahl der imperativen Programmiersprache **3**

2.1 Begründung der Sprachwahl . . . 3

2.2 Unterstützte funktionale Elemente in Python . . . 3

2.2.1 Kurzbeispiele der funktionalen Elemente . . . 4

3 Datengrundlage **4**

3.1 Beschreibung des Datensatzes . . . 4

3.2 Hauptdimensionen . . . 4

3.3 Verwendete Variablen . . . 4

3.4 Herkunft des Datensatzes und Abgrenzung . . . 5

4 Projektantrag **5**

4.1 Ausgangslage . . . 5

4.2 Zielsetzung . . . 5

4.3 Projektumfang . . . 5

5 Programmausgabe **6**

5.1 Beispielhafter Output . . . 6

5.2 Interpretation der Ergebnisse . . . 6

5.3 Interpretation und statistische Einordnung . . . 6

6 Refactoring: Vergleich Version 1.0 vs. Version 2.0 **7**

6.1 Konkrete Beispiele aus dem Projekt . . . 7

6.1.1 Beispiel A: Filtern von Datensätzen . . . 7

Imperative Umsetzung (Version 1.0) . . . 7

Funktionale Umsetzung (Version 2.0) . . . 7

6.1.2 Beispiel B: Aggregation (Tabellenbreiten) . . . 8

Imperative Umsetzung (Version 1.0) . . . 8

Funktionale Umsetzung (Version 2.0) . . . 8

6.1.3 Beispiel C: Transformation numerischer Werte (map) . . . 8

Imperative Umsetzung (Version 1.0) . . . 8

Funktionale Umsetzung (Version 2.0) . . . 9

6.1.4 Zusammenfassung der Unterschiede . . . 9

7 Fazit **9**

7.1 Nutzen funktionaler Elemente . . . 9

7.2 Vereinfachung durch Refactoring . . . 9

7.3 Bewusste Designentscheidungen . . . 9

7.4 Wiederverwendung funktionaler Konzepte . . . 10

7.5 Grenzen funktionaler Elemente in Python . . . 10

7.6 Anwendungsfälle im beruflichen Umfeld . . . 10

8 Quellen **11**

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.

Die verwendeten Daten werden in Abschnitt 3 beschrieben. Der Einfluss funktionaler Sprachelemente wird anschließend im Refactoring-Vergleich in Abschnitt 6 analysiert.

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

```python
# 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 Datengrundlage

3.1 Beschreibung des Datensatzes

Als Datengrundlage dient ein Bank-Marketing-Datensatz einer portugiesischen Bank, der auf einer öffentlich dokumentierten Bank-Marketing-Studie basiert (vgl. [1]). Der Datensatz enthält Kundendaten aus einer Marketingkampagne sowie eine Zielvariable, welche angibt, ob ein Produktabschluss erfolgt ist (`complete = yes/no`).

Die Daten liegen in Form einer CSV-Datei vor und wurden im Projekt ausschließlich lesend verwendet.

3.2 Hauptdimensionen

Der Datensatz lässt sich in folgende Hauptdimensionen einteilen:

- **Demografie:** Alter (`age`), Bildungsstand (`education`), Familienstand (`marital`)
- **Finanzen:** Kontostand (`balance`)
- **Kontaktdaten:** Gesprächsdauer (`duration`)
- **Ergebnisvariable:** Produktabschluss (`complete`)

3.3 Verwendete Variablen

Im Projekt werden unter anderem folgende Variablen verwendet:

- `age` (numerisch)
- `job` (kategorisch)
- `marital` (kategorisch)
- `education` (kategorisch)
- `balance` (numerisch)
- `housing` (binär)
- `loan` (binär)

- `duration` (numerisch)
- `complete` (binär, Zielvariable)

3.4 Herkunft des Datensatzes und Abgrenzung

Der im Projekt verwendete Bank-Marketing-Datensatz stammt ursprünglich aus einer universitären Lehrveranstaltung zur multivariaten Datenanalyse an der Universität Basel (vgl. [2]). Der Datensatz wurde im Rahmen eines Assignments zur statistischen Auswertung von Bank-Marketingkampagnen eingesetzt und basiert auf realitätsnahen, vorverarbeiteten Kundendaten einer portugiesischen Bank.

Die Aufgabenstellung des ursprünglichen Assignments umfasste unter anderem Varianzanalysen, logaritmische Transformationen sowie Klassifikationsverfahren. In diesem Projekt wird der Datensatz jedoch ausschließlich als Datengrundlage verwendet und in einen neuen Kontext übertragen.

Die Zielsetzung dieses Projekts unterscheidet sich bewusst von der ursprünglichen universitären Aufgabenstellung: Der Fokus liegt nicht auf statistischer Modellbildung oder Inferenz, sondern auf dem Vergleich imperativer und funktionaler Programmierparadigmen im Sinne des Moduls M323 – Funktionales Programmieren.

4 Projektantrag

4.1 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.

4.2 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

4.3 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).

5 Programmausgabe

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

5.1 Beispielhafter Output

Das folgende Beispiel zeigt die Konsolenausgabe der Auswertungsfunktion *Group by Education* (Menüoption 5). Dabei werden alle Datensätze nach Bildungsniveau gruppiert und aggregierte Kennzahlen berechnet, darunter die Anzahl der Kunden, das durchschnittliche Alter, der durchschnittliche Kontostand sowie die Erfolgsquote der Marketingkampagne.

Die dargestellte Ausgabe wird sowohl von Version 1.0 (imperative Umsetzung) als auch von Version 2.0 (funktionale Umsetzung) identisch erzeugt.

```
                          GROUP BY EDUCATION
======================================================================
education | count | avg(age) | avg(balance) | success
----------+-------+----------+--------------+--------
primary   |  1002 |     49.0 |      1970.88 |   48.3%
secondary |  3815 |     40.7 |      1709.54 |   52.2%
tertiary  |  2657 |     39.7 |      2404.61 |   62.5%

ANOVA F-Wert: 41.93 (df=2,7471)
```

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

5.2 Interpretation der Ergebnisse

5.3 Interpretation und statistische Einordnung

Die Ergebnisse zeigen, dass der durchschnittliche Kontostand sowie die Erfolgsquote mit steigendem Bildungsniveau zunehmen. Kunden mit tertiärer Ausbildung weisen sowohl den höchsten durchschnittlichen Kontostand als auch die höchste Erfolgsquote auf.

Zur Bewertung der Unterschiede zwischen den Bildungsgruppen wird eine einfaktorielle Varianzanalyse (ANOVA) verwendet. Der ANOVA-F-Wert beschreibt das Verhältnis der Varianz zwischen den Gruppen zur Varianz innerhalb der Gruppen. Der hohe F-Wert deutet darauf hin, dass sich die Mittelwerte der Gruppen deutlich voneinander unterscheiden und die Varianz zwischen den Bildungsgruppen größer ist als die Varianz innerhalb der Gruppen.

Die angegebenen Freiheitsgrade (df) geben an, wie viele unabhängige Informationen in die Berechnung eingeflossen sind. Dabei beschreibt $df_{between}$ die Anzahl der verglichenen Gruppen minus eins, während df_{within} die Streuung der Werte innerhalb der Gruppen widerspiegelt [3].

6 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.

6.1 Konkrete Beispiele aus dem Projekt

6.1.1 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.

Imperative Umsetzung (Version 1.0)

```
def _apply_filters(data, housing, loan, balance_gt):
    out = []
    for row in data:
        if housing is not None and row.get("housing") is not housing:
            continue
        if loan is not None and row.get("loan") is not loan:
            continue
        bal = row.get("balance")
        if balance_gt is not None:
            if bal is None or bal <= balance_gt:
                continue
        out.append(row)
    return out
```

Funktionale Umsetzung (Version 2.0)

```
def _apply_filters(data, housing, loan, balance_gt):
    def pred(row):
        if housing is not None and row.get("housing") is not housing:
            return False
        if loan is not None and row.get("loan") is not loan:
            return False
        if balance_gt is not None:
            bal = row.get("balance")
            if bal is None or bal <= balance_gt:
                return False
```

```
        return True

    return list(filter(pred, data))
```

In der funktionalen Variante wird die Filterbedingung in einer separaten Prädikatfunktion gekapselt. Dies ermöglicht eine deklarative Beschreibung der Filterlogik und vermeidet explizite Kontrollstrukturen.

6.1.2 Beispiel B: Aggregation (Tabellenbreiten)

Für die tabellarische Ausgabe müssen die maximalen Spaltenbreiten über alle Zeilen hinweg ermittelt werden. In Version 1.0 erfolgt dies schrittweise mittels Schleifen und veränderlicher Zustände. Version 2.0 verwendet hingegen eine funktionale Faltung (`reduce`), welche die Aggregation deklarativ beschreibt.

Imperative Umsetzung (Version 1.0)

```
widths = [len(h) for h in headers]

for row in rows:
    for i, cell in enumerate(row):
        if len(cell) > widths[i]:
            widths[i] = len(cell)
```

Funktionale Umsetzung (Version 2.0)

```
widths = list(map(len, headers))

def upd(acc, row):
    return [max(acc[i], len(cell)) for i, cell in enumerate(row)]

widths = reduce(upd, rows, widths)
```

Die funktionale Variante beschreibt die Aggregation als reine Abbildung vom bisherigen Akkumulator und einer Zeile auf einen neuen Akkumulator. Dadurch entfällt explizite Zustandsänderung, was die Logik klarer und leichter testbar macht.

6.1.3 Beispiel C: Transformation numerischer Werte (map)

Numerische Transformationen wie `balance^2 + 1` werden in Version 1.0 schrittweise in einer Schleife berechnet und in einer Liste gesammelt. Version 2.0 beschreibt dieselbe Operation deklarativ als Abbildung über alle Werte mittels `map` und einer Lambda-Funktion.

Imperative Umsetzung (Version 1.0)

```
sq1 = []
for row in current:
```

```
    bal = row.get("balance")
    if isinstance(bal, (int, float)):
        b = float(bal)
        sq1.append(b * b + 1.0)
```

Funktionale Umsetzung (Version 2.0)

```
balances = [
    float(r["balance"])
    for r in current
    if isinstance(r.get("balance"), (int, float))
]

sq1 = list(map(lambda b: b * b + 1.0, balances))
```

Die funktionale Variante trennt klar zwischen der Auswahl gültiger Werte und der eigentlichen Transformation. Dadurch wird die Rechenlogik kompakter formuliert und leichter wiederverwendbar.

6.1.4 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.

7 Fazit

7.1 Nutzen funktionaler Elemente

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

7.2 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.

7.3 Bewusste Designentscheidungen

Im Rahmen der Transformation numerischer Variablen wurden bewusst unterschiedliche mathematische Abbildungen eingesetzt. Neben einer fachlich realistischen Transformation mittels Lo-

garithmus log(balance) wurde zusätzlich eine einfache quadratische Transformation $\text{balance}^2 + 1$ verwendet.

Der Logarithmus ist insbesondere im wirtschaftlichen Kontext sinnvoll, da er rechtsschiefe Verteilungen reduziert und Ausreißer abschwächt. Allerdings ist diese Transformation nur für positive Werte definiert und erfordert zusätzliche Prüfungen des Wertebereichs.

Die quadratische Transformation hingegen ist für alle numerischen Werte definiert und eignet sich besonders gut, um funktionale Abbildungen unabhängig von Sonderfällen zu demonstrieren. Dadurch konnte der Fokus im Projekt gezielt auf den Vergleich funktionaler Programmierkonzepte gelegt werden, ohne die Darstellung durch zusätzliche Randfallbehandlung zu verkomplizieren.

7.4 Wiederverwendung funktionaler Konzepte

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

- Datenfiltern
- Aggregationen
- Transformationen von Listen und Datensätzen

7.5 Grenzen funktionaler Elemente in Python

Obwohl Python zahlreiche funktionale Sprachelemente bereitstellt, handelt es sich nicht um eine rein funktionale Programmiersprache. Im Vergleich zu funktionalen Sprachen wie Haskell werden zentrale Konzepte wie Immutabilität oder Nebenwirkungsfreiheit nicht erzwungen, sondern lediglich konventionell eingehalten.

Variablen können weiterhin verändert werden, und Funktionen können unbeabsichtigt Seiteneffekte erzeugen. Die korrekte Anwendung funktionaler Konzepte hängt somit stark von der Disziplin der Entwickelnden ab und wird nicht durch den Sprachkern abgesichert.

Diese Einschränkungen wurden im Projekt bewusst in Kauf genommen, da der Fokus auf dem Vergleich imperativer und funktionaler Denkweisen innerhalb einer praxisnahen Sprache lag.

In einer rein funktionalen Sprache könnten diese Konzepte konsistenter und durch den Compiler abgesichert umgesetzt werden. Python eignet sich jedoch besonders gut, um funktionale Programmierkonzepte schrittweise in bestehende imperative Arbeitsweisen zu integrieren.

7.6 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 zur Entstehung der Arbeit

Zur Unterstützung bei der Dokumentation und beim Refactoring wurden KI-basierte Werkzeuge konsultiert. Die Projektarbeit wurde eigenständig umgesetzt; sämtliche Ergebnisse, Implementierungen und Auswertungen wurden von den Autoren geprüft, verstanden und verantwortet.

8 Quellen

Literatur

[1] S. Moro, R. Laureano, P. Cortez. *Using Data Mining for Bank Direct Marketing: An Application of the CRISP-DM Methodology.* Proceedings of the European Simulation and Modelling Conference (ESM 2011).

[2] Maringer, Dietmar und Kitzing, Emil. *Multivariate Datenanalyse – Assignment 1.* WWZ, Universität Basel, Herbstsemester 2024.

[3] Montgomery, Douglas C. *Design and Analysis of Experiments.* 9th Edition, Wiley, 2022.

[4] Projekt-Repository: https://github.com/stoicfist/Modul-323-Projektarbeit

Hinweis: Der vollständige Quellcode befindet sich im Projekt-Repository (vgl. [4]). Er wurde in dieser Dokumentation bewusst nicht vollständig abgedruckt, um den Fokus auf Konzept und Vergleich zu legen.