Lernnachweis zu Kompetenz A1G

**Kompetenz**:

Ich kann die Eigenschaften von Funktionen beschreiben (z.Bsp. pure function) und den Unterschied zu anderen Programmier-Strukturen erläutern (z.Bsp. zu Prozedur).

**Lernnachweis: Eigenschaften von Funktionen in der funktionalen Programmierung**

Im Rahmen dieses Lernfeldes habe ich die grundlegenden Eigenschaften von Funktionen in der funktionalen Programmierung erforscht. Eine besondere Art von Funktionen, die in der funktionalen Programmierung hervorgehoben wird, ist die sogenannte "pure function". Eine pure Funktion ist eine Funktion, die bei gleichen Eingabeparametern immer denselben Ausgabewert liefert und keine Seiteneffekte hat. Ein Beispiel hierfür ist die add-Funktion in meiner Flask-Anwendung:

Code Beispiel mit add Funktion:   
**from** **flask** **import** Flask, jsonify

app = Flask(\_\_name\_\_)

# Beispiel einer pure function

**def** **add**(x, y):

**return** x + y

# Route für die Anwendung der pure function

**@app.route**('/')

**def** **home**():

# Anwendung der pure function

result = add(**3**, **5**)

**return** jsonify({'result': result})

**if** \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Reflexion**:   
Die Integration von Codebeispielen in die Reflexion, insbesondere aus der erstellten Flask-Anwendung, hat mir geholfen, abstrakte Konzepte in konkreten Kontexten anzuwenden. Dies wurde durch das Beispiel der add-Funktion deutlich.

**Fazit**:

Die Erkundung funktionaler Programmierung und reiner Funktionen in Python vermittelte wertvolle Einblicke.

**Info**: Vollständiger Code im Flask Projekt (Github Repo).

Lernnachweis zu Kompetenz A1F:

**Kompetenz**:

Ich kann das Konzept von \*immutable values\* erläutern und dazu Beispiele anwenden. Somit kann ich dieses Konzept funktionaler Programmierung im Unterschied zu anderen Programmiersprachen erklären (z.Bsp. im Vergleich zu referenzierten Objekten).

**Lernnachweis zu A1F: Konzept von immutable values**

Durch die Flask-Applikation erlernte ich den Unterschied zwischen mutablen und unveränderlichen Werten in Python. Die Anwendung in Echtprojekten vertiefte mein Verständnis, und ich erkenne die Schlüsselrolle von immutable values in der funktionalen Programmierung.

Code Beispiel zu immutable und mutable:

**from** **flask** **import** Flask

app = Flask(\_\_name\_\_)

# Beispiel für eine Liste (mutable)

mutable\_list = [**1**, **2**, **3**]

# Beispiel für ein Tupel (immutable)

immutable\_tuple = (**1**, **2**, **3**)

**@app.route**('/')

**def** **index**():

**return** f"Mutable List: {mutable\_list}<br>Immutable Tuple: {immutable\_tuple}"

**if** \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Reflexion**:   
Anfänglich herausfordernd, wurde mir die Schlüsselrolle von immutable values durch praxisnahe Anwendungen in der Flask-Applikation klar. Peer-Reviews stärkten mein Verständnis. Ich plane, meine Kenntnisse in komplexeren Projekten zu vertiefen, und fühle mich nun sicherer im Umgang mit diesem Konzept.

**Fazit**:   
Erfolgreiches Verständnis von immutable values in Python durch praxisnahe Anwendung. Fundamentale Rolle für Robustheit in der funktionalen Programmierung erreicht.

**Info**: Vollständiger Code im Flask Projekt (Github Repo).

Lernnachweis zu Kompetenz A1E:

**Kompetenz**:

A1E: Ich kann aufzeigen wie Probleme in den verschiedenen Konzepten (OO, prozedural und funktional) gelöst werden und diese miteinander vergleichen.

**Lernnachweis zu A1E**:

Während des Studiums des Moduls 323 habe ich mich intensiv mit der Problemlösung in den verschiedenen Programmierparadigmen auseinandergesetzt, insbesondere im objektorientierten (OO), prozeduralen und funktionalen Ansatz.

Code Beispiel zu Kompetenz A1E:

**from** **flask** **import** Flask

app = Flask(\_\_name\_\_)

# Objektorientiertes Paradigma

**class** **User**:

**def** **\_\_init\_\_**(self, username, email):

self.username = username

self.email = email

**def** **display\_info**(self):

**return** f"Username: {self.username}, Email: {self.email}"

user1 = User("JohnDoe", "john@example.com")

# Prozedurales Paradigma

**def** **calculate\_sum**(numbers):

result = **0**

**for** num **in** numbers:

result += num

**return** result

numbers = [**1**, **2**, **3**, **4**, **5**]

total = calculate\_sum(numbers)

# Funktionales Paradigma

**def** **square\_numbers**(numbers):

**return** list(map(**lambda** x: x\*\***2**, numbers))

squared = square\_numbers(numbers)

**@app.route**('/')

**def** **index**():

**return** (

f"Objektorientiertes Paradigma: {user1.display\_info()}<br>"

f"Prozedurales Paradigma: Die Summe der Zahlen ist {total}<br>"

f"Funktionales Paradigma: Quadrierte Zahlen: {squared}"

)

**if** \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Lernprozess und Reflexion**:

Die Vergleichsanalyse dieser Paradigmen ermöglichte mir, ihre Stärken und Schwächen zu verstehen. Im prozeduralen Ansatz liegt der Fokus auf Verfahren und Schritten, während das objektorientierte Paradigma die Struktur betont. Das funktionale Paradigma zeichnet sich durch klare Funktionen und Unveränderlichkeit aus.

**Fazit**:

Die Anwendung in meiner Flask-Applikation half mir, die Vor- und Nachteile dieser Paradigmen in der Praxis zu erkennen. Zum Beispiel konnte ich mithilfe von Klassen und Objekten im OO-Paradigma die Anwendung besser strukturieren und warten.

**Info**: Vollständiger Code im Flask Projekt (Github Repo).