M323

Created	@November 8, 2023 1:41 PM
Tags	

A1G

Lernnachweis: Eigenschaften von Funktionen Beschreiben und Unterschiede erläutern.

Im Rahmen dieses Lernziels habe ich mich mit den grundlegenden Eigenschaften von Funktionen in der funktionalen Programmierung auseinandergesetzt und dabei insbesondere den Unterschied zu anderen Programmieransätzen, wie beispielsweise der prozeduralen Programmierung, herausgearbeitet.

Funktionale Eigenschaften

Eine wichtige Eigenschaft funktionaler Programmierung ist die Verwendung von "pure functions". Eine "pure function" gibt für dieselben Eingabeparameter immer denselben Ausgabewert zurück und hat keine Seiteneffekte. Das bedeutet, sie beeinflusst den Zustand außerhalb der Funktion nicht.

Beispielcode

Hier ist ein Beispielcode, der eine "pure function" darstellt:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Die Funktion `factorial` berechnet die Fakultät einer Zahl und zeigt die Charakteristiken einer "pure function". Sie gibt für jede Eingabe denselben Ausgabewert zurück, ohne den Programmzustand zu ändern.

Unterschied zu Prozedur

Im Gegensatz dazu steht das Konzept der Prozeduren, bei dem der Fokus auf der Abfolge von Anweisungen liegt, die Schritt für Schritt ausgeführt werden. Im funktionalen Ansatz liegt der Schwerpunkt auf der Funktionsanwendung und Vermeidung von Zustandsänderungen.

Lernprozess und Reflexion

Anfangs fand ich es herausfordernd, mich von imperativen Ansätzen zu lösen und mich auf die rein funktionale Herangehensweise einzulassen. Die Auseinandersetzung mit "pure functions" half mir, den Nutzen von funktionaler Programmierung besser zu verstehen.

Die Anwendung des Konzepts in eigenen Projekten, wie zum Beispiel bei der Implementierung von rekursiven Funktionen, half mir, die Vorteile der funktionalen Programmierung zu schätzen. Die klare Trennung von Daten und Funktionen erleichtert das Verständnis und die Wartung des Codes.

Zukünftige Schritte

Um meine Kenntnisse weiter zu vertiefen, plane ich, mehr Projekte in funktionaler Programmierung zu realisieren. Besonders interessiert mich die Anwendung funktionaler Konzepte in der parallelen Programmierung und im Umgang mit großen Datenmengen.

Mit dem Verständnis der Funktionseigenschaften und des Unterschieds zu prozeduralen Ansätzen fühle ich mich bereit, die Prinzipien funktionaler Programmierung in zukünftigen Projekten effektiv einzusetzen.

A1F

Lernnachweis: Konzept von ‘Immutable Values’ erläutern und anwenden.

Während dieses Lernziels habe ich mich intensiv mit dem Konzept der *immutable values* in der funktionalen Programmierung auseinandergesetzt und dabei Beispiele angewendet. Dabei lag ein besonderer Fokus darauf, dieses Konzept im Vergleich zu anderen Programmiersprachen, insbesondere zu referenzierten Objekten, zu erklären.

Konzept der *immutable values*

Im funktionalen Ansatz sind Daten, die als immutable gelten, unveränderlich. Das bedeutet, dass ihre Werte nach der Erstellung nicht mehr geändert werden können. Im Gegensatz dazu können mutable Werte in anderen Ansätzen verändert werden, was zu Seiteneffekten führen kann.

Beispielcode

Hier ist ein Beispielcode, der das Konzept von *immutable values* verdeutlicht:

```
def immutable(string, n):
    # A1F
    lst = ' '.join(string).split()
    if len(n) > 1:
        new = ' '.join(n).split()
        return lst + new
    return lst + [n]
```

In diesem Beispiel wird die Funktion `immutable` verwendet, um eine Liste `lst` zu erstellen, die *immutable* ist. Die Liste wird durch Verkettung von Werten erzeugt, und es wird sichergestellt, dass die ursprünglichen Daten unverändert bleiben.

Unterschied zu referenzierten Objekten

Im Gegensatz zu referenzierten Objekten, bei denen Änderungen am Objekt selbst vorgenommen werden können, fördert das Konzept der *immutable values* in der funktionalen Programmierung eine sicherere und transparentere Handhabung von Daten. Dies minimiert Seiteneffekte und erleichtert das Verständnis des Programmflusses.

Lernprozess und Reflexion

Zu Beginn fand ich es herausfordernd, den Unterschied zwischen *immutable* und *mutable* Werten vollständig zu erfassen. Die Praxis, insbesondere durch die Anwendung in eigenen Codeprojekten, half mir jedoch, die Vorteile und Einschränkungen von *immutable values* besser zu verstehen.

Die Reflexion über meinen Code und die Diskussion mit meinen Mitschüler halfen mir, mein Verständnis zu vertiefen und mögliche Fallstricke zu identifizieren. Der klare Vorteil des Vermeidens von ungewollten Seiteneffekten wurde besonders deutlich.

Zukünftige Schritte

Um meine Kenntnisse weiter zu festigen, plane ich, mehr Projekte zu entwickeln, die auf dem Konzept der *immutable values* basieren. Dies wird mir helfen, die Vorteile dieses Ansatzes in verschiedenen Anwendungsszenarien zu erkennen und zu nutzen.

Mit einem fundierten Verständnis von *immutable values* fühle ich mich bereit, funktionalen Programmierstil konsequent in zukünftigen Projekten anzuwenden und somit robusten und wartbaren Code zu schaffen.

A1E

Problemen in verschiedenen Konzepten lösen und vergleichen

Während dieses Lernziels habe ich mich mit der Lösung von Problemen in den verschiedenen Programmierparadigmen (objektorientiert, prozedural und funktional) auseinandergesetzt. Ziel war es, die Herangehensweisen in den verschiedenen Konzepten zu verstehen und miteinander zu vergleichen.

Prozedurale Lösung

```
def quadrat_summe_prozedural(n):
    result = 0
    for i in range(n + 1):
        result += i ** 2
    return result
```

In der prozeduralen Lösung wird eine einfache Schleife verwendet, um die Quadratsumme bis zur Zahl `n` zu berechnen. Dies entspricht einem klassischen, schrittweisen Ansatz, bei dem der Fokus auf der Abfolge von Anweisungen liegt.

Objektorientierte Lösung

```
class QuadratSummeObjektorientiert:
    def __init__(self, n):
        self.n = n

    def berechne_quadrat_summe(self):
        result = 0
        for i in range(self.n + 1):
```

```
            result += i ** 2
        return result
```

Die objektorientierte Lösung verwendet eine Klasse, um den Zustand (`n`) zu kapseln und eine Methode `berechne_quadrat_summe` bereitzustellen, um die Quadratsumme zu berechnen. Dies spiegelt den Fokus der objektorientierten Programmierung auf Datenkapselung und Methoden wider.

Funktionale Lösung

```
def quadrat_summe_funktional(n):
    return sum(map(lambda x: x ** 2, range(n + 1)))
```

Die funktionale Lösung nutzt die `map`- und `sum`-Funktionen sowie eine Lambda-Funktion, um die Quadratsumme zu berechnen. Hier steht die Anwendung von Funktionen im Vordergrund, und der Zustand wird durch die Vermeidung von Seiteneffekten minimiert.

Vergleich der Herangehensweisen

In der prozeduralen und objektorientierten Lösung steht die Schritt-für-Schritt-Abarbeitung der Anweisungen im Mittelpunkt, während die funktionale Lösung auf Funktionsanwendung und Vermeidung von Zustandsänderungen abzielt.

Der prozedurale Ansatz kann einfach sein, eignet sich jedoch möglicherweise nicht für komplexe Strukturen. Die objektorientierte Lösung bietet eine klarere Strukturierung durch die Verwendung von Klassen und Methoden. Die funktionale Lösung ist besonders elegant, da sie sich auf die Anwendung von Funktionen ohne Seiteneffekte konzentriert.

Lernprozess und Reflexion

Der Vergleich dieser Lösungen half mir, die Vor- und Nachteile der verschiedenen Programmierparadigmen zu verstehen. Der prozedurale Ansatz ist direkt und leicht verständlich, während die objektorientierte Lösung für größere Projekte besser skalierbar sein kann. Die funktionale Lösung beeindruckt durch ihre Klarheit und Wartbarkeit.

Die Diskussion mit Kollegen half mir, die Vorzüge jeder Herangehensweise zu schätzen und zu verstehen, wann welche am besten geeignet ist.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, Probleme in verschiedenen ***Paradigmen*** zu lösen und die Effizienz und Lesbarkeit meines Codes zu verbessern. Dies wird mir helfen, flexibler in der Auswahl des passenden ***Paradigmas*** für zukünftige Projekte zu sein.

B1G

Algorithmus erklären

Im Rahmen dieses Lernziels habe ich einen funktionalen Algorithmus entwickelt, um die Quadratsumme bis zur Zahl `n` zu berechnen. Ziel war es, nicht nur den Code zu schreiben, sondern auch den Algorithmus dahinter zu verstehen und erklären zu können.

Funktionaler Algorithmus zur Quadratsumme

```
def quadrat_summe(n):
    # B1G
    return sum(map(lambda x: x ** 2, range(1, n + 1)))
```

Dieser Algorithmus verwendet funktionale Programmierkonzepte, um die Quadratsumme zu berechnen. Hier wird die `map`-Funktion verwendet, um für jede Zahl von 1 bis `n` das Quadrat zu berechnen, und die `sum`-Funktion addiert diese Quadrate.

Erklärung des Algorithmus

1. **Bereich definieren:** Der Algorithmus arbeitet im Bereich von 1 bis `n`.
2. **Quadrieren der Zahlen:** Mithilfe der `map` Funktion wird für jede Zahl im Bereich das Quadrat berechnet.
3. **Summieren der Quadrate:** Die `sum` Funktion addiert alle Quadrate zusammen, um die Quadratsumme zu erhalten.

Vorteile funktionaler Herangehensweise

- **Klarheit und Lesbarkeit:** Die Verwendung von `map` und `sum` sorgt für einen klaren und lesbaren Code.
- **Vermeidung von Seiteneffekten:** Die Funktion ist *pure* und hat keine Seiteneffekte, da sie keine externen Variablen ändert.

Lernprozess und Reflexion

Die Entwicklung dieses Algorithmus half mir, die Funktionsweise von `map` und `sum` besser zu verstehen und zu schätzen. Die klare Struktur des funktionalen Codes ermöglicht eine einfache Erklärung des Algorithmus und fördert das Verständnis für funktionale Programmierkonzepte.

Die Reflexion über die Vorteile der funktionalen Herangehensweise half mir, die Bedeutung von *pure functions* und die Vermeidung von Seiteneffekten zu erkennen. Dies wird meine Herangehensweise an zukünftige Problemlösungen beeinflussen.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, mehr komplexe Algorithmen funktional zu implementieren. Dies wird mir helfen, die Tiefe der funktionalen Programmierung zu erfassen und mich auf fortgeschrittenere Anwendungsfälle vorzubereiten.

B1F

Algorithmen in funktionale Teilstücke aufteilen

Im Zuge dieses Lernziels habe ich einen funktionalen Algorithmus erstellt, der die Grundrechenoperationen zwischen zwei Zahlen durchführt. Das Hauptaugenmerk lag darauf, den Algorithmus in funktionale Teilstücke aufzuteilen, um die Wartbarkeit und Verständlichkeit zu verbessern.

Funktional aufgeteilter Algorithmus für Grundrechenoperationen

```
def calculations(a, b):
    #B1F
    def add(a, b):
```

```
        return a + b

    def subtract(a, b):
        return a - b

    def multiply(a, b):
        return a * b

    def divide(a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero!")
        return a / b

    added = add(a, b)
    subtracted = subtract(a, b)
    multiplied = multiply(a, b)
    divided = divide(a, b)

    return f'Added: {added},<br> Subtracted: {subtracted},<br> Multiplied: {multiplied},<b
r> Divided: {divided}'
```

Erklärung des funktionalen Ansatzes

1. **Teilfunktionen für jede Operation:** Jede Grundrechenoperation (Addition, Subtraktion, Multiplikation, Division) wird in separate Funktionen aufgeteilt.
2. **Vermeidung von Seiteneffekten:** Jede Teilfunktion führt ihre Operation aus und gibt das Ergebnis zurück, ohne den Zustand außerhalb der Funktion zu ändern.
3. **Kombination der Ergebnisse:** Die Ergebnisse der Teilfunktionen werden in einem String zurückgegeben.

Vorteile der funktionalen Aufteilung

- **Klare Struktur:** Die Trennung der Operationen in einzelne Funktionen schafft eine klare Struktur und erleichtert die Wartbarkeit.
- **Wiederverwendbarkeit:** Jede Teilfunktion kann unabhängig von anderen wiederverwendet werden, was die Code-Wiederverwendbarkeit fördert.

Lernprozess und Reflexion

Die Aufteilung des Algorithmus in funktionale Teilstücke half mir, die Vorteile der funktionalen Programmierung besser zu verstehen. Die klare Struktur und die klare

Abgrenzung der Verantwortlichkeiten jeder Teilfunktion trugen zur Lesbarkeit des Codes bei.

Die Reflexion über den funktionalen Ansatz half mir, die Bedeutung von Funktionen als eigenständige Bausteine zu schätzen und wie sie zur Schaffung komplexerer Algorithmen kombiniert werden können.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu vertiefen, plane ich, komplexere Algorithmen funktional zu zerlegen und dabei sicherzustellen, dass die Teilstücke gut definiert und wieder verwendbar sind. Dies wird dazu beitragen, meine Fähigkeiten in der funktionalen Programmierung zu festigen und auf fortgeschrittene Anwendungsfälle vorzubereiten.

B1E

Funktionen in zusammenhängende Algorithmen implementieren

Im Rahmen dieses Lernziels habe ich einen funktionalen Algorithmus erstellt, der verschiedene Operationen auf einer Liste von Zahlen durchführt. Das Ziel war es, Funktionen zu implementieren und sie in einem zusammenhängenden Algorithmus zu kombinieren.

Funktionaler Algorithmus für Liste von Zahlen

```
def calculate_algoritmen(lst):
    #B1E
    def add_one(lst):
        return list(map(lambda x: x + 1, lst))

    def subtract_one(lst):
        return list(map(lambda x: x - 1, lst))

    def multiply_by_two(lst):
        return list(map(lambda x: x * 2, lst))

    def divide_by_two(lst):
        return list(map(lambda x: x / 2, lst))

    added = add_one(lst)
    subtracted = subtract_one(lst)
    multiplied = multiply_by_two(lst)
    divided = divide_by_two(lst)
```

```
    return f'Added: {added},<br> Subtracted: {subtracted},<br> Multiplied: {multiplied},<b
r> Divided: {divided}'
```

Erklärung des funktionalen Algorithmus

1. **Teilfunktionen für jede Operation:** Jede Operation (Addition, Subtraktion, Multiplikation, Division) wird in separate Funktionen aufgeteilt.
2. **Anwendung der Funktionen auf die Liste:** Jede Teilfunktion wird auf die Eingangsliste angewendet, um eine neue Liste zu erzeugen.
3. **Kombination der Ergebnisse:** Die Ergebnisse der Teilfunktionen werden in einem String zurückgegeben.

Vorteile der funktionalen Umsetzung

- **Modularität:** Jede Operation ist in einer eigenen Funktion gekapselt, was die Wiederverwendbarkeit und Wartbarkeit verbessert.
- **Vermeidung von Seiteneffekten:** Die Funktionen arbeiten mit Kopien der Daten und verändern nicht den Zustand außerhalb der Funktionen.

Lernprozess und Reflexion

Die Umsetzung dieses funktionalen Algorithmus half mir, die Idee der Funktionen als modulare Bausteine für zusammenhängende Algorithmen zu vertiefen. Die klare Trennung der Verantwortlichkeiten jeder Funktion erleichtert das Verständnis und die Wartung des Codes.

Die Reflexion über den funktionalen Ansatz betonte die Wichtigkeit von Modularität und Wiederverwendbarkeit, was zu einem effizienten und sauberen Code führt.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, komplexere Algorithmen zu erstellen, die eine Vielzahl von funktionalen Operationen kombinieren. Dies wird mir helfen, die funktionalen Programmierkonzepte in unterschiedlichen Kontexten anzuwenden und zu vertiefen.

B2G

Funktionen als Objekte behandeln und als Variablen speichern

Im Rahmen dieses Lernziels habe ich Funktionen als Objekte behandelt und sie in Variablen gespeichert und weitergegeben. Dies wurde durch eine Flask-Webanwendung umgesetzt, bei der die Funktionen in einer separaten Datei (`functional.py`) definiert sind.

Funktionen als Objekte in `functional.py`

```
def add(a, b):
    # B2G & B2F
    return a + b

def cal(function, a, b):
    # B2G & B2F
    return function(int(a), int(b))
```

Flask-Webanwendung in `main.py`

```
from flask import Flask
from functional import add, cal

app = Flask(__name__)

@app.route('/b2g/<n>')
def function_as_object(n):
    function = add
    return f'{cal(function, students[int(n.split(",")[0])].scores, students[int(n.split
(",")[1])].scores)}'
```

Erklärung des Codes

1. **Funktionen als Objekte:** Die Funktionen `add` und `cal` in `functional.py` werden als Objekte behandelt.
2. **Speichern in Variablen:** Die Funktion `add` wird in der Flask-Webanwendung in der Variable `function` gespeichert.
3. **Weitergabe der Funktion:** Die Funktion `cal` nimmt eine Funktion als Argument (`function`) und wendet sie auf die übergebenen Werte an.

Vorteile der Behandlung von Funktionen als Objekte

- **Dynamische Funktionalität:** Durch die Möglichkeit, Funktionen in Variablen zu speichern und weiterzugeben, können unterschiedliche Funktionen je nach Bedarf verwendet werden.
- **Wiederverwendbarkeit:** Die Funktionen können modular behandelt und an verschiedenen Stellen der Anwendung wiederverwendet werden.

Lernprozess und Reflexion

Die Umsetzung dieses Codes half mir, die Konzepte der Funktionen als Objekte in der Praxis zu verstehen. Die Verwendung von Funktionen als Argumente ermöglicht eine dynamische und flexible Gestaltung von Algorithmen.

Die Reflexion über den funktionalen Ansatz betonte die Wichtigkeit von Flexibilität und Wiederverwendbarkeit in der funktionalen Programmierung.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, mehr komplexe Algorithmen zu erstellen, bei denen Funktionen dynamisch als Objekte behandelt werden. Dies wird mir helfen, die Anwendung dieses Konzepts in verschiedenen Anwendungsfällen zu vertiefen.

B2F

Funktionen als Argumente für andere Funktionen verwenden → Hochwertige Funktionen erstellen

Hochwertige Funktion mit Funktionen als Argumente

Funktionen als Objekte in `functional.py`

```
def add(a, b):
    # B2G & B2F
    return a + b

def cal(function, a, b):
```

```
    # B2G & B2F
    return function(int(a), int(b))
```

Flask-Webanwendung in `main.py`

```
from flask import Flask
from functional import add, cal

app = Flask(__name__)

@app.route('/b2f/<n>')
def hochwertige_funktion(n):
    return f'{cal(add, students[int(n.split(",")[0])].scores,students[int(n.split(",")
[1])].scores)}'
```

Erklärung des Codes

1. `add` **Funktion:** Diese Funktion nimmt zwei Argumente entgegen und führt die Addition durch.
2. `cal` **Funktion:** Diese Funktion nimmt eine Funktion (`add` in diesem Fall) sowie zwei Zahlen als Argumente und führt die übergebene Funktion mit den Zahlen als Argumenten aus.
3. `hochwertige_funktion` **Route:** Diese Route nimmt Parameter `n` entgegen, extrahiert die Noten zweier Studenten und führt die hochwertige Funktion aus, indem sie `add` als Funktion und die extrahierten Noten als Argumente verwendet.

Vorteile von Funktionen als Argumente

- **Wiederverwendbarkeit:** Die Fähigkeit, Funktionen als Argumente zu verwenden, ermöglicht die Schaffung allgemeiner Funktionen, die in verschiedenen Kontexten eingesetzt werden können.
- **Flexibilität:** Durch die Verwendung von Funktionen als Argumente kann die Funktionalität einer Funktion dynamisch variiert werden.

Lernprozess und Reflexion

Die Umsetzung dieses Codes verdeutlicht die Flexibilität funktionaler Programmierung. Die Fähigkeit, Funktionen als Argumente zu übergeben, ermöglicht die Erstellung

höherwertiger Funktionen, was besonders in dynamischen Umgebungen und komplexen Anwendungen nützlich ist.

Die Reflexion betont die Bedeutung der klaren Strukturierung von Funktionen und ihrer Verwendung in höherwertigen Funktionen.

Zukünftige Schritte

Um die Fähigkeiten weiter zu vertiefen, ist es ratsam, verschiedene Funktionen als Argumente zu verwenden und die Auswirkungen auf die Flexibilität und Lesbarkeit des Codes zu beobachten. Dies wird helfen, ein tieferes Verständnis für die Anwendung funktionaler Konzepte zu entwickeln.

B2E

Funktionen als Objekte und Argumente verwenden.

Im Rahmen dieses Lernziels wurde eine Flask-Webanwendung erstellt, bei der Funktionen als Objekte und Argumente verwendet werden, um eine höherwertige Funktion zu erstellen (Anwendung von Closures).

Funktionen als Objekte und Closures in `functional.py`

```
def multiplier(factor):
    # B2E
    def multiply(number):
        return number * factor
    return multiply
```

Flask-Webanwendung in `main.py`

```
from flask import Flask
from functional import multiplier

app = Flask(__name__)

@app.route('/b2e/<n>')
def function_as_object(n):
    factor = int(n.split(",")[0])
    multiplied = int(students[int(n.split(",")[1])].scores)
```

```
    multiplier_by_n = multiplier(factor)
    return f'{multiplier_by_n(multiplied)}'
```

Erklärung des Codes

1. **Funktionen als Objekte:** Die Funktion `multiplier` in `functional.py` wird als Objekt verwendet und gibt eine andere Funktion (`multiply`) zurück.
2. **Anwendung von Closures:** Die Funktion `multiply` ist eine Closure, da sie auf die Variable `factor` zugreifen kann, auch wenn sie außerhalb ihres ursprünglichen Gültigkeitsbereichs aufgerufen wird.

Vorteile der Anwendung von Closures

- **Zugriff auf äußere Variablen:** Closures ermöglichen den Zugriff auf Variablen, die außerhalb ihres ursprünglichen Gültigkeitsbereichs definiert sind, was zu einer flexibleren Gestaltung von Funktionen führt.
- **Sauberer Code:** Closures helfen, den Code sauber zu halten, indem sie den Zugriff auf bestimmte Variablen auf den benötigten Bereich beschränken.

Lernprozess und Reflexion

Die Umsetzung dieses Codes half mir, das Konzept von Closures in der Praxis zu verstehen. Die Verwendung von Funktionen als Objekte, die auf äußere Variablen zugreifen können, ermöglicht eine elegante Lösung für komplexe Aufgaben.

Die Reflexion über den funktionalen Ansatz betonte die Bedeutung von Closures für die Schaffung flexibler und wieder verwendbarer Funktionen.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, mehr komplexe Algorithmen zu erstellen, bei denen Closures eine entscheidende Rolle spielen. Dies wird dazu beitragen, mein Verständnis von funktionalen Programmierkonzepten zu vertiefen und auf fortgeschrittene Anwendungsfälle vorzubereiten.

B3G

Einfache Lambda-Ausdrücke schreiben

Lambda-Ausdruck für die Konvertierung in Großbuchstaben

```
def to_upper(names):
    # B3G
    return list(map(lambda x: x.upper(), names))
```

Erklärung des Codes

1. **Lambda-Ausdruck:** Der Lambda-Ausdruck `lambda x: x.upper()` ist eine anonyme Funktion, die auf jedes Element `x` angewendet wird.
2. **Einzelne Operation:** Der Lambda-Ausdruck führt die Operation `x.upper()` durch, die jeden String in Großbuchstaben konvertiert.
3. **Anwendung auf Liste:** Der Lambda-Ausdruck wird mit der `map`Funktion auf die Liste `names` angewendet, um eine neue Liste mit den konvertierten Strings zu erstellen.

Vorteile der Verwendung von Lambda-Ausdrücken

- **Kompaktheit:** Lambda-Ausdrücke sind kompakte, inline definierte Funktionen, die sich gut für einfache Operationen eignen.
- **Einzeilige Funktionalität:** Lambda-Ausdrücke sind besonders nützlich, wenn die Funktion sehr kurz ist und nur eine einzelne Operation durchführt.

Lernprozess und Reflexion

Die Implementierung dieses Codes half mir, die Einfachheit und Nützlichkeit von Lambda-Ausdrücken für kurze, prägnante Funktionen zu verstehen. Die Verwendung von Lambda-Ausdrücken kann den Code lesbarer machen, insbesondere wenn die Funktion sehr einfach ist.

Die Reflexion über den funktionalen Ansatz betonte die Bedeutung von Klarheit und Kompaktheit bei der Verwendung von Lambda-Ausdrücken für spezifische Aufgaben.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, Lambda-Ausdrücke in verschiedenen Kontexten zu verwenden und zu verstehen, wie sie effektiv in

komplexeren Funktionen integriert werden können. Dies wird meine Fähigkeit zur Anwendung funktionaler Programmierkonzepte weiter stärken.

B3F

Lambda-Ausdrücke mit mehreren Argumenten

Lambda-Ausdruck für die Addition von Zahlen

```
add_numbers = lambda x, y: x + y
```

Erklärung des Codes

1. **Lambda-Ausdruck:** Der Lambda-Ausdruck `lambda x, y: x + y` ist eine anonyme Funktion, die zwei Argumente `x` und `y` annimmt.
2. **Mehrere Argumente:** Im Lambda-Ausdruck können mehrere Argumente verarbeitet werden, in diesem Fall `x` und `y`.
3. **Operation:** Der Lambda-Ausdruck führt die Addition der beiden Argumente `x` und `y` durch.

Vorteile der Verwendung von Lambda-Ausdrücken mit mehreren Argumenten

- **Kompaktheit:** Lambda-Ausdrücke sind besonders nützlich für kurze, prägnante Funktionen, auch wenn mehrere Argumente verarbeitet werden müssen.
- **Einzeilige Funktionalität:** Der Code bleibt trotz der Verarbeitung mehrerer Argumente kompakt und übersichtlich.

Lernprozess und Reflexion

Die Implementierung dieses Codes half mir, die Flexibilität von Lambda-Ausdrücken bei der Verarbeitung mehrerer Argumente zu verstehen. Lambda-Ausdrücke können in kurzen, spezifischen Funktionen verwendet werden, die mehrere Argumente erfordern.

Die Reflexion über den funktionalen Ansatz betonte die Einfachheit und Klarheit, die Lambda-Ausdrücke in Situationen bieten, in denen eine vollständig benannte Funktion

nicht notwendig ist.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, Lambda-Ausdrücke mit mehreren Argumenten in verschiedenen Kontexten zu verwenden und zu verstehen, wie sie effektiv in funktionalen Programmierparadigmen eingesetzt werden können. Dies wird meine Fähigkeiten bei der Anwendung funktionaler Konzepte weiter stärken.

B3E

Lambda-Ausdrücke verwenden um den Programmfluss zu steuern

Lambda-Ausdruck zur Sortierung von Listen

```
def sort_by_user(lst, criteria):
    # B3E
    return sorted(lst, key=lambda x: x[criteria])
```

Erklärung des Codes

1. **Lambda-Ausdruck:** Der Lambda-Ausdruck `lambda x: x[criteria]` wird als Schlüssel für die `sorted` Funktion verwendet.
2. **Benutzerdefinierte Sortierung:** Der Lambda-Ausdruck gibt an, nach welchem Kriterium die Sortierung erfolgen soll. Hier wird das Kriterium durch den Parameter `criteria` bestimmt.
3. **Sortierung:** Die `sorted` Funktion sortiert die Liste `lst` basierend auf dem durch den Lambda-Ausdruck definierten Kriterium.

Vorteile der Verwendung von Lambda-Ausdrücken zur Sortierung

- **Dynamische Sortierung:** Der Lambda-Ausdruck ermöglicht eine dynamische Definition des Sortierkriteriums, was die Flexibilität erhöht.
- **Einzeilige Funktionalität:** Die Verwendung von Lambda-Ausdrücken ermöglicht eine kurze und prägnante Sortierungsfunktion.

Lernprozess und Reflexion

Die Implementierung dieses Codes half mir, die Anwendung von Lambda-Ausdrücken zur Steuerung des Programmflusses zu verstehen, insbesondere in Bezug auf die Sortierung von Listen. Die Verwendung von Lambda-Ausdrücken ermöglicht eine elegante und dynamische Sortierung.

Die Reflexion über den funktionalen Ansatz betonte die Bedeutung von Lambda-Ausdrücken für die Flexibilität bei der Steuerung des Programmflusses in funktionalen Programmierparadigmen.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, Lambda-Ausdrücke in verschiedenen Kontexten zur Steuerung des Programmflusses zu verwenden. Dies wird mir helfen, die Vielseitigkeit dieser Konstrukte in komplexeren Funktionen zu verstehen und anzuwenden.

B4G

Funktionen Map, Filter und Reduce einzeln auf Listen anwenden

Anwendung von Map, Filter und Reduce auf eine Liste

```
from functools import reduce

def map_filter_reduce(lst):
    # B4G
    # Map: Quadrieren der Elemente
    mapped = map(lambda x: x ** 2, lst)

    # Filter: Auswahl der geraden Zahlen
    filtered = filter(lambda x: x % 2 == 0, mapped)

    # Reduce: Summierung der Zahlen
    reduced = reduce(lambda x, y: x + y, filtered)

    return reduced
```

Erklärung des Codes

1. **Map-Funktion:** Der Lambda-Ausdruck `lambda x: x ** 2` wird auf jedes Element der Liste `lst` angewendet, um die Quadrate der Elemente zu erstellen.
2. **Filter-Funktion:** Der Lambda-Ausdruck `lambda x: x % 2 == 0` wird auf die durch die Map-Funktion erstellten Quadrate angewendet, um nur die geraden Zahlen auszuwählen.
3. **Reduce-Funktion:** Der Lambda-Ausdruck `lambda x, y: x + y` wird auf die durch die Filter-Funktion ausgewählten Elemente angewendet, um ihre Summe zu berechnen.

Vorteile der Verwendung von Map, Filter und Reduce

- **Modularität:** Die Verwendung von Map, Filter und Reduce ermöglicht eine klare Aufteilung der Operationen auf Listen in einzelne Schritte.
- **Lesbarkeit:** Der Code wird durch die Verwendung von funktionalen Programmierkonstrukten lesbarer und eleganter.

Lernprozess und Reflexion

Die Implementierung dieses Codes half mir, die Anwendung von Map, Filter und Reduce in funktionalen Programmierparadigmen zu verstehen. Diese Funktionen bieten eine leistungsstarke und klare Möglichkeit, Operationen auf Listen durchzuführen.

Die Reflexion über den funktionalen Ansatz betonte die Bedeutung von Map, Filter und Reduce für eine deklarative und saubere Programmierung.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, Map, Filter und Reduce in verschiedenen Kontexten anzuwenden und zu verstehen, wie sie effektiv in komplexeren Funktionen integriert werden können. Dies wird meine Fähigkeit zur Anwendung funktionaler Konzepte weiter stärken.

B4F

Map, Filter, Reduce kombiniert verwenden.

Kombinierte Anwendung von Map, Filter und Reduce

```
from functools import reduce

def map_filter_reduce_combined(lst):
    # B4F
    result = reduce(lambda x, y: x + y, map(lambda x: x ** 2, filter(lambda x: x % 2 == 0,
lst)))
    return result
```

Erklärung des Codes

1. **Filter-Funktion:** Der Lambda-Ausdruck `lambda x: x % 2 == 0` wird auf jedes Element der Liste `lst` angewendet, um nur die geraden Zahlen auszuwählen.
2. **Map-Funktion:** Der Lambda-Ausdruck `lambda x: x ** 2` wird auf die durch die Filter-Funktion ausgewählten geraden Zahlen angewendet, um ihre Quadrate zu erstellen.
3. **Reduce-Funktion:** Der Lambda-Ausdruck `lambda x, y: x + y` wird auf die durch die Map-Funktion erstellten Quadrate angewendet, um ihre Summe zu berechnen.

Vorteile der kombinierten Anwendung

- **Effizienz:** Durch die Kombination von Map, Filter und Reduce in einer einzigen Anweisung wird der Bedarf an Zwischenvariablen und Schleifen minimiert, was zu einem effizienteren Code führt.
- **Klarheit:** Die kombinierte Anwendung zeigt deutlich die Abfolge der Transformationen auf die Daten.

Lernprozess und Reflexion

Die Implementierung dieses Codes half mir, die kombinierte Anwendung von Map, Filter und Reduce für komplexere Datenverarbeitungsaufgaben zu verstehen. Dies ermöglicht eine effiziente und klare Transformation von Daten in funktionalen Programmierparadigmen.

Die Reflexion über den funktionalen Ansatz betonte die Bedeutung der effizienten und klaren Verwendung von funktionalen Konstrukten, insbesondere bei komplexeren Transformationen.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, Map, Filter und Reduce in verschiedenen Kombinationen und für verschiedene Datenverarbeitungsaufgaben zu verwenden. Dies wird meine Fähigkeit zur Anwendung funktionaler Konzepte in unterschiedlichen Kontexten weiter stärken.

B4E

Map, Filter, Reduce verwenden, um komplexe Datenverarbeitungsaufgaben zu lösen

Datenverarbeitung durch Filter, Map und Reduce

```
from functools import reduce

def filter_through_students(gender, students):
    # B4E
    # Filter: Auswahl der Studenten des angegebenen Geschlechts
    target_students = list(filter(lambda student: student.gender == gender, students))

    # Map: Extrahieren der Noten der ausgewählten Studenten
    scores_of_target_students = list(map(lambda student: student.scores, target_students))
    print(scores_of_target_students)  # Ausgabe der extrahierten Noten

    # Reduce: Berechnung des Durchschnitts der Noten
    avg_scores_of_target_students = reduce(lambda x, y: x + y, scores_of_target_students)
/ len(
        scores_of_target_students) if len(scores_of_target_students) > 0 else 0

    return avg_scores_of_target_students
```

Erklärung des Codes

1. **Filter-Funktion:** Durch die Filter-Funktion werden nur die Studenten ausgewählt, die das angegebene Geschlecht haben. Dies entspricht einer Datenaggregation basierend auf einem bestimmten Kriterium.
2. **Map-Funktion:** Die Map-Funktion extrahiert die Noten der ausgewählten Studenten. Dies ist eine Transformation der Datenstruktur, da nur die Noten extrahiert werden.

3. **Reduce-Funktion:** Die Reduce-Funktion wird verwendet, um den Durchschnitt der extrahierten Noten zu berechnen. Dies ist eine weitere Form der Datenaggregation.

Lernprozess und Reflexion

Die Implementierung dieses Codes half mir, die Anwendung von Map, Filter und Reduce für komplexe Datenverarbeitungsaufgaben zu verstehen. In diesem Fall werden die Funktionen verwendet, um Daten zu aggregieren und zu transformieren, wodurch der Programmierer in der Lage ist, spezifische Informationen aus den Daten zu extrahieren.

Die Reflexion über den funktionalen Ansatz betonte die Vielseitigkeit von Map, Filter und Reduce bei der Lösung verschiedener Datenverarbeitungsaufgaben.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, ähnliche Datenverarbeitungsaufgaben zu lösen und verschiedene Kombinationen von Map, Filter und Reduce zu verwenden. Dies wird mir helfen, meine Fähigkeiten zur Anwendung funktionaler Konzepte auf unterschiedliche Anwendungsfälle zu erweitern.

C1G

Refactoring-Techniken auszählen, die einen Code lesbarer & verständlicher machen

1. Verwendung von List Comprehension

```
# Vorher
squares = []
for x in range(10):
    squares.append(x ** 2)

# Nachher
squares = [x ** 2 for x in range(10)]
```

List Comprehension bietet eine kompakte und lesbarere Möglichkeit, Listen zu erstellen.

2. Einsatz von Generators

```
# Vorher
def square_generator(n):
    for x in range(n):
        yield x ** 2

# Nachher
square_generator = (x ** 2 for x in range(n))
```

Generators sind effiziente Sequenzen, die Elemente bedarfsweise erzeugen und können den Speicherverbrauch optimieren.

3. Anwendung von Dictionary Comprehension

```
# Vorher
squares_dict = {}
for x in range(5):
    squares_dict[x] = x ** 2

# Nachher
squares_dict = {x: x ** 2 for x in range(5)}
```

Dictionary Comprehension ermöglicht eine elegante Erstellung von Dictionaries.

4. Verwendung von `yield` für Generatoren-Funktionen

```
# Vorher
def square_generator(n):
    squares = []
    for x in range(n):
        squares.append(x ** 2)
    return squares

# Nachher
def square_generator(n):
    for x in range(n):
        yield x ** 2
```

Die Verwendung von `yield` in einer Funktion macht sie zu einem Generator, der die Werte bedarfsweise erzeugt und verbessert die Leistung und Speichereffizienz.

5. Vermeidung von unnötigen Zwischenschritten

```
# Vorher
evens = []
for num in range(10):
    if num % 2 == 0:
        evens.append(num)

squared_evens = [x ** 2 for x in evens]

# Nachher
squared_evens = [x ** 2 for x in range(0, 10, 2)]
```

Die Vermeidung von unnötigen Zwischenschritten führt zu einem klareren und effizienteren Code.

6. Einsatz von Funktionen für wiederholten Code

```
# Vorher
result1 = x * 2
result2 = y * 2

# Nachher
def double_value(value):
    return value * 2

result1 = double_value(x)
result2 = double_value(y)
```

Die Verwendung von Funktionen verbessert die Lesbarkeit und ermöglicht die Wiederverwendung von Code.

Lernprozess und Reflexion

Refactoring-Techniken tragen dazu bei, den Code effizienter und lesbarer zu gestalten. Die Auswahl der richtigen Technik hängt von der spezifischen Situation ab, und die fortlaufende Reflexion über den Code verbessert im Laufe der Zeit das Verständnis für optimale Lösungen.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, diese Refactoring-Techniken in unterschiedlichen Kontexten anzuwenden und dadurch meine Fähigkeiten zur

Codeoptimierung zu vertiefen. Dies wird mir helfen, effizienten und klaren Code in verschiedenen Situationen zu schreiben.

C1F

Mit Refactoring-Techniken den Code lesbarer und verständlicher machen

1. Verwendung von List Comprehension

```
def list_comprehension(lst):
    # C1F
    names = [student.name for student in lst]
    print(names)
    return names
```

Die Verwendung von List Comprehension vereinfacht die Erstellung einer Liste aus den Namen der Studenten und macht den Code kompakter.

2. Einsatz von Generators

```
def generator(lst):
    # C1F
    names = list(student.name for student in lst)
    print(names)
    return names
```

Die Umwandlung des Generator-Ausdrucks in eine Liste macht den Code lesbarer und ermöglicht die Verwendung der Namen der Studenten in anderen Teilen des Codes.

3. Reflexion über den Code

Die Refactoring-Techniken in diesen Funktionen verbessern die Lesbarkeit und Klarheit des Codes erheblich. List Comprehension und Generators ermöglichen eine effiziente Erstellung von Listen und sind besonders nützlich, wenn die resultierende Liste weiterverwendet oder modifiziert wird.

Zukünftige Schritte

Um meine Fähigkeiten weiter zu entwickeln, plane ich, weitere Gelegenheiten zu identifizieren, bei denen List Comprehension, Generators und andere Refactoring-Techniken angewendet werden können. Die kontinuierliche Anwendung dieser Techniken wird meine Fähigkeit zur Verbesserung der Codequalität weiter stärken.

C1E

Sicherstellen das die Auswirkungen des Refactorings keine unerwünschte Nebeneffekte hat.

Betrachtung von Refactoring mit Bedacht

Refactoring ist ein mächtiges Werkzeug, um Code zu verbessern, aber es erfordert Umsicht, um unerwünschte Nebeneffekte zu vermeiden. Hier sind einige Grundsätze:

1. **Testabdeckung sicherstellen:** Vor dem Refactoring ist es wichtig, umfassende Tests zu haben, die sicherstellen, dass die bestehende Funktionalität intakt bleibt.
2. **Inkrementelle Änderungen:** Statt den gesamten Code auf einmal zu refaktorieren, sollten Änderungen schrittweise und inkrementell erfolgen, wobei zwischen den Änderungen getestet wird.
3. **Klare Dokumentation:** Änderungen sollten klar dokumentiert werden, um Entwicklern und Teammitgliedern zu helfen, die Veränderungen zu verstehen und mögliche Auswirkungen zu erkennen.
4. **Code Reviews:** Code Reviews sind entscheidend, um sicherzustellen, dass mehrere Augen den refaktorierten Code überprüfen und mögliche Probleme erkennen.

Reflexion auf mögliche Auswirkungen

Ohne einen konkreten Code als Beispiel ist es schwer, spezifische Auswirkungen zu diskutieren. Ein generischer Ansatz zur Reflexion könnte folgendermaßen aussehen:

1. **Vergleich der Funktionalität:** Stellen Sie sicher, dass die Funktionalität nach dem Refactoring identisch mit der vorherigen ist. Testen Sie alle relevanten Szenarien.
2. **Überprüfung der Leistung:** Refactoring kann Auswirkungen auf die Leistung haben. Überprüfen Sie, ob der refaktorierte Code effizienter oder langsamer ist als

zuvor.

3. **Beobachtung von Seiteneffekten:** Überwachen Sie den Code auf unerwünschte Seiteneffekte, insbesondere wenn das Refactoring Teile des Codes betrifft, die von anderen Teilen des Systems abhängen.
4. **Verhalten bei Randfällen:** Testen Sie den Code in Randfällen, um sicherzustellen, dass er unter allen Bedingungen korrekt funktioniert.

Zukünftige Schritte

Um die Fähigkeiten in diesem Bereich zu vertiefen, ist es wichtig, an realen Projekten zu arbeiten und Erfahrungen mit dem Evaluieren der Auswirkungen von Refactorings zu sammeln. Durch kontinuierliche Praxis wird die Fähigkeit verbessert, sicherzustellen, dass Refactorings keine unerwünschten Auswirkungen haben.

> Dieses Portfolio wurde mit Unterstützung von ChatGPT verfasst.