

# Informatik I: Einführung in die Programmierung

## 7. Entwurf von Schleifen, While-Schleifen, Hilfsfunktionen und Akkumulatoren

Albert-Ludwigs-Universität Freiburg



UNI  
FREIBURG

Prof. Dr. Peter Thiemann

10. November 2021, 16. November 2021



# Entwurf von Schleifen

## Entwurf von Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
Addition  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



# Fallstudie: Rechnen mit Polynomen

Entwurf von  
Schleifen

**Fallstudie:**  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Definition

Ein *Polynom vom Grad  $n$*  ist eine Folge von Zahlen  $(a_0, a_1, \dots, a_n)$ , den *Koeffizienten*. Dabei ist  $n \geq 0$  und  $a_n \neq 0$ .

## Beispiele

- $()$
- $(1)$
- $(3, 2, 1)$

## Anwendungen

Kryptographie, fehlerkorrigierende Codes.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



- (Skalar) Multiplikation mit einer Zahl  $c$

$$c \cdot (a_0, a_1, \dots, a_n) = (c \cdot a_0, c \cdot a_1, \dots, c \cdot a_n)$$

- Auswertung an der Stelle  $x_0$

$$(a_0, a_1, \dots, a_n)[x_0] = \sum_{i=0}^n a_i \cdot x_0^i$$

- Ableitung

$$(a_0, a_1, \dots, a_n)' = (1 \cdot a_1, 2 \cdot a_2, \dots, n \cdot a_n)$$

- Integration

$$\int (a_0, a_1, \dots, a_n) = (0, a_0, a_1/2, a_2/3, \dots, a_n/(n+1))$$

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



# Skalarmultiplikation

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



$$c \cdot (a_0, a_1, \dots, a_n) = (c \cdot a_0, c \cdot a_1, \dots, c \cdot a_n)$$

## Schritt 1: Bezeichner und Datentypen

Die Funktion `skalar_mult` nimmt als Eingabe

- `c` : `complex`, den Faktor,
- `p` : `list[complex]`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def skalar_mult(  
    c : complex,  
    p : list[complex]  
    ) -> list[complex]:  
    # fill in, initialization  
    for a in p:  
        pass # fill in action for each element  
    return ...
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert(skalar_mult(42, []) == [])  
assert(skalar_mult(42, [1,2,3]) == [42,84,126])  
assert(skalar_mult(-0.1, [1,2,3]) == [-0.1,-0.2,-0.3])
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def skalar_mult(  
    c : complex,  
    p : list[complex]  
    ) -> list[complex]:  
    result = []  
    for a in p:  
        result = result + [c * a]  
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Rumpf der Skalarmultiplikation

```
result = []
for a in p:
    result = result + [c * a]
return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Rumpf der Skalarmultiplikation

```
result = []
for a in p:
    result = result + [c * a]
return result
```

## Variable `result` ist Akkumulator

- In `result` wird das Ergebnis aufgesammelt (akkumuliert)
- `result` wird vor der Schleife initialisiert auf das Ergebnis für die leere Liste
- Jeder Schleifendurchlauf erweitert das Ergebnis in `result` indem das Ergebnis mit dem aktuellen Element `a` erweitert wird

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

**Skalarmultiplikation**

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

**while-**  
Schleifen

Zusammen-  
fassung



# Auswertung

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

**Auswertung**

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



$$(a_0, a_1, \dots, a_n)[x_0] = \sum_{i=0}^n a_i \cdot x_0^i$$

## Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_eval` nimmt als Eingabe

- `p` : `list[complex]`, ein Polynom,
- `x` : `complex`, das Argument.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation

**Auswertung**

Ableitung  
Integration  
Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def poly_eval(  
    p : list[complex],  
    x : complex  
    ) -> complex:  
    # fill in  
    for a in p:  
        pass # fill in action for each element  
    return ...
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

**Auswertung**

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

**while-**  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert(poly_eval([], 2) == 0)
assert(poly_eval([1,2,3], 2) == 17)
assert(poly_eval([1,2,3], -0.1) == 0.83)
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

**Auswertung**

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def poly_eval(  
    p : list[complex],  
    x : complex  
    ) -> complex:  
    result = 0  
    i = 0  
    for a in p:  
        result = result + a * x ** i  
        i = i + 1  
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

**Auswertung**

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Alternative Funktionsdefinition

```
def poly_eval(  
    p : list[complex],  
    x : complex  
    ) -> complex:  
    result = 0  
    for i, a in enumerate(p): # <---  
        result = result + a * x ** i  
    return result
```

- `enumerate(seq)` liefert Paare aus (Laufindex, Element)
- Beispiel `list(enumerate([8, 8, 8])) == [(0, 8), (1, 8), (2, 8)]`

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

**Auswertung**

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

**while-**  
Schleifen

Zusammen-  
fassung



# Ableitung

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

**Ableitung**

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



$$(a_0, a_1, \dots, a_n)' = (1 \cdot a_1, 2 \cdot a_2, \dots, n \cdot a_n)$$

## Schritt 1: Bezeichner und Datentypen

Die Funktion `derivative` nimmt als Eingabe

- `p : list[complex]`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

**Ableitung**

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def derivative(  
    p : list[complex]  
    ) -> list[complex]:  
    # initialization  
    for a in p:  
        pass # fill in action for each element  
    return ...
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
**Ableitung**  
Integration  
Binäre Operationen  
Addition  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert(derivative([]) == [])  
assert(derivative([42]) == [])  
assert(derivative([1,2,3]) == [2,6])
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

**Ableitung**

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def derivative(  
    p : list[complex]  
    ) -> list[complex]:  
    result = []  
    for i, a in enumerate(p):  
        if i>0:  
            result = result + [i * a]  
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

**Ableitung**

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



# Integration

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

**Integration**

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



$$\int (a_0, a_1, \dots, a_n) = (0, a_0, a_1/2, a_2/3, \dots, a_n/(n+1))$$

## Schritt 1: Bezeichner und Datentypen

Die Funktion `integral` nimmt als Eingabe

- `p : list[complex]`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

## Weitere Schritte

selbst

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

**Integration**

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



# Binäre Operationen

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

**Binäre Operationen**

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



- Addition (falls  $n \leq m$ )

$$\begin{aligned}(a_0, a_1, \dots, a_n) + (b_0, b_1, \dots, b_m) \\ = (a_0 + b_0, a_1 + b_1, \dots, a_n + b_n, b_{n+1}, \dots, b_m)\end{aligned}$$

- Multiplikation von Polynomen

$$\begin{aligned}(a_0, a_1, \dots, a_n) \cdot (b_0, b_1, \dots, b_m) \\ = (a_0 \cdot b_0, a_0 \cdot b_1 + a_1 \cdot b_0, \dots, \sum_{i=0}^k a_i \cdot b_{k-i}, \dots, a_n \cdot b_m)\end{aligned}$$

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

**Binäre Operationen**

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

**while-**  
Schleifen

Zusammen-  
fassung



# Addition

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



$$(a_0, a_1, \dots, a_n) + (b_0, b_1, \dots, b_m) \\ = (a_0 + b_0, a_1 + b_1, \dots, a_n + b_n, b_{n+1}, \dots, b_m)$$

## Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_add` nimmt als Eingabe

- `p : list[complex]`, ein Polynom.
- `q : list[complex]`, ein Polynom.

Die Grade der Polynome ergeben sich aus der Länge der Sequenzen.

## Achtung

Die Grade der Polynome können unterschiedlich sein!

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def poly_add(
    p : list[complex],
    q : list[complex]
) -> list[complex]:
    # fill in
    for i in range(...): # <---
        pass # fill in action for each element
    return ...
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
**Addition**  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung

## Frage

Was ist das Argument ... von range?



## Schritt 3: Beispiele

```
assert(poly_add([], []) == [])  
assert(poly_add([42], []) == [42])  
assert(poly_add([], [11]) == [11])  
assert(poly_add([1,2,3], [4,3,2,5]) == [5,5,5,5])
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
**Addition**  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert(poly_add([], []) == [])  
assert(poly_add([42], []) == [42])  
assert(poly_add([], [11]) == [11])  
assert(poly_add([1,2,3], [4,3,2,5]) == [5,5,5,5])
```

## Antwort: Argument von range

```
maxlen = max(len(p), len(q))
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
**Addition**  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition, erster Versuch

```
def poly_add(
    p : list[complex],
    q : list[complex]
) -> list[complex]:
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        result = result + [p[i] + q[i]]
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Problem

Eine Assertion schlägt fehl!

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
**Addition**  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Problem

Eine Assertion schlägt fehl!

```
Traceback (most recent call last):
File ".../polynom.py", line 14, in <module>
    assert(poly_add([42], []) == [42])
File ".../polynom.py", line 10, in poly_add
    result = result + [p[i] + q[i]]
IndexError: list index out of range
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Problem

Eine Assertion schlägt fehl!

```
Traceback (most recent call last):
File ".../polynom.py", line 14, in <module>
    assert(poly_add([42], []) == [42])
File ".../polynom.py", line 10, in poly_add
    result = result + [p[i] + q[i]]
IndexError: list index out of range
```

## Analyse

Zweite Assertion schlägt fehl für  $i=0$ !

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Neuer Entwurfsschritt: Wunschdenken

**Abstrahiere** die gewünschte Funktionalität in einer **Hilfsfunktion**.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
**Addition**  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Neuer Entwurfsschritt: Wunschdenken

**Abstrahiere** die gewünschte Funktionalität in einer **Hilfsfunktion**.

## Schritt 1: Bezeichner und Datentypen

Die Funktion `safe_index` nimmt als Eingabe

- `p : list[complex]` eine Sequenz
- `i : int` einen Index (positiv)
- `d : complex` einen Ersatzwert für ein Element von `p`

und liefert das Element `p[i]` (falls definiert) oder den Ersatzwert.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def safe_index(  
    p : list[complex],  
    i : int, # assume  $\geq 0$   
    d : complex  
    ) -> complex:  
    # fill in  
    return 0
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
**Addition**  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert safe_index([1,2,3], 0, 0) == 1
assert safe_index([1,2,3], 2, 0) == 3
assert safe_index([1,2,3], 4, 0) == 0
assert safe_index([1,2,3], 4, 42) == 42
assert safe_index([], 0, 42) == 42
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def safe_index(  
    p : list[complex],  
    i : int, # assume  $\geq 0$   
    d : complex  
    ) -> complex:  
    return p[i] if i < len(p) else d
```

oder (alternative Implementierung des Funktionsrumpfes)

```
if i < len(p):  
    return p[i]  
else:  
    return d
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Bedingter Ausdruck (Conditional Expression)

```
expr_true if expr_cond else expr_false
```

- Werte zuerst *expr\_cond* aus
- Falls Ergebnis kein Nullwert, dann werte *expr\_true* als Ergebnis aus
- Sonst werte *expr\_false* als Ergebnis aus

## Beispiele

- `17 if True else 4 == 17`
- `"abc"[i] if i<3 else "␣"`

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

**while-**  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition mit Hilfsfunktion

```
def poly_add(
    p : list[complex],
    q : list[complex]
) -> list[complex]:
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        result = result + [
            safe_index(p,i,0) + safe_index (q,i,0)]
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

**Addition**

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



# Multiplikation

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



$$(p_0, p_1, \dots, p_n) \cdot (q_0, q_1, \dots, q_m) \\ = (p_0 \cdot q_0, p_0 \cdot q_1 + p_1 \cdot q_0, \dots, \sum_{i=0}^k p_i \cdot q_{k-i}, \dots, p_n \cdot q_m)$$

## Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_mult` nimmt als Eingabe

- `p : list[complex]` ein Polynom
- `q : list[complex]` ein Polynom

und liefert als Ergebnis das Produkt der Eingaben.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def poly_mult(  
    p : list[complex],  
    q : list[complex]  
    ) -> list[complex]:  
    # fill in  
    for k in range(...):  
        pass # fill in to compute k-th output element  
    return ...
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert poly_mult([], []) == []
assert poly_mult([42], []) == []
assert poly_mult([], [11]) == []
assert poly_mult([1,2,3], [1]) == [1,2,3]
assert poly_mult([1,2,3], [0,1]) == [0,1,2,3]
assert poly_mult([1,2,3], [1,1]) == [1,3,5,3]
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert poly_mult([], []) == []
assert poly_mult([42], []) == []
assert poly_mult([], [11]) == []
assert poly_mult([1,2,3], [1]) == [1,2,3]
assert poly_mult([1,2,3], [0,1]) == [0,1,2,3]
assert poly_mult([1,2,3], [1,1]) == [1,3,5,3]
```

## Beobachtungen

- Range  $\text{maxlen} = \text{len}(p) + \text{len}(q) - 1$

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def poly_mult(  
    p : list[complex],  
    q : list[complex]  
    ) -> list[complex]:  
    result = []  
    for k in range(len(p) + len(q) - 1):  
        rk = ... # k-th output element  
        result = result + [rk]  
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Das k-te Element

$$r_k = \sum_{i=0}^k p_i \cdot q_{k-i}$$

noch eine Schleife!

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Das k-te Element

$$r_k = \sum_{i=0}^k p_i \cdot q_{k-i}$$

noch eine Schleife!

## Berechnung

```
rk = 0
for i in range(k+1):
    rk = rk + (safe_index(p,i,0)
               * safe_index(q,k-i,0))
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition, final

```
def poly_mult(  
    p : list[complex],  
    q : list[complex]  
    ) -> list[complex]:  
    result = []  
    for k in range(len(p) + len(q) - 1):  
        rk = 0  
        for i in range(k+1):  
            rk = rk + (safe_index(p,i,0)  
                      * safe_index(q,k-i,0))  
        result = result + [rk]  
    return result
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

**Multiplikation**

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



# Extra: Lexikographische Ordnung

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

**Extra:**  
**Lexikographische  
Ordnung**

**while-**  
**Schleifen**

Zusammen-  
fassung

# Erinnerung: Lexikographische Ordnung



## Gegeben

Zwei Sequenzen der Längen  $m, n \geq 0$ :

$$\vec{a} = "a_1 a_2 \dots a_m"$$

$$\vec{b} = "b_1 b_2 \dots b_n"$$

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

**Extra:**  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung

# Erinnerung: Lexikographische Ordnung



## Gegeben

Zwei Sequenzen der Längen  $m, n \geq 0$ :

$$\vec{a} = "a_1 a_2 \dots a_m"$$

$$\vec{b} = "b_1 b_2 \dots b_n"$$

$\vec{a} \leq \vec{b}$  in der lexikographischen Ordnung, falls

Es gibt  $0 \leq k \leq \min(m, n)$ , so dass

- $a_1 = b_1, \dots, a_k = b_k$  und

$$\vec{a} = "a_1 a_2 \dots a_k a_{k+1} \dots a_m"$$

$$\vec{b} = "a_1 a_2 \dots a_k b_{k+1} \dots b_n"$$

- $k = m$

$$\vec{a} = "a_1 a_2 \dots a_m"$$

$$\vec{b} = "a_1 a_2 \dots a_m b_{m+1} \dots b_n"$$

- oder  $k < m$  und  $a_{k+1} < b_{k+1}$ .

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_leq` nimmt als Eingabe

- `a : list` eine Sequenz
- `b : list` eine Sequenz

und liefert als Ergebnis `True`, falls  $a \leq b$ , sonst `False`.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung

# Lexikographische Ordnung



## Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_leq` nimmt als Eingabe

- `a : list` eine Sequenz
- `b : list` eine Sequenz

und liefert als Ergebnis `True`, falls  $a \leq b$ , sonst `False`.

## Schritt 2: Funktionsgerüst

```
def lex_leq(a : list, b : list) -> bool:  
    # fill in  
    for k in range(...):  
        pass # fill in  
    return ...
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert lex_leq([], []) == True
assert lex_leq([42], []) == False
assert lex_leq([], [11]) == True
assert lex_leq([1,2,3], [1]) == False
assert lex_leq([1], [1,2,3]) == True
assert lex_leq([1,2,3], [0,1]) == False
assert lex_leq([1,2,3], [1,3]) == True
assert lex_leq([1,2,3], [1,2,3]) == True
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
assert lex_leq([], []) == True
assert lex_leq([42], []) == False
assert lex_leq([], [11]) == True
assert lex_leq([1,2,3], [1]) == False
assert lex_leq([1], [1,2,3]) == True
assert lex_leq([1,2,3], [0,1]) == False
assert lex_leq([1,2,3], [1,3]) == True
assert lex_leq([1,2,3], [1,2,3]) == True
```

## Beobachtungen

- $\text{Range minlen} = \min(\text{len}(a), \text{len}(b))$

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def lex_leq(  
    a : list,  
    b : list  
    ) -> bool:  
    minlen = min (len (a), len (b))  
    for k in range(minlen):  
        if a[k] < b[k]:  
            return True  
        if a[k] > b[k]:  
            return False  
    # a is prefix of b or vice versa  
    return len(a) <= len(b)
```

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
Addition  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



## Problem

- Der Typ `list` charakterisiert Listen mit beliebigen Elementen.
- Aber: Vergleich von beliebigen Listen ist nicht möglich!  
Beispiel: `lex_leq ("abc", [1,2,3])` liefert Fehler!
- Wir müssen sicherstellen:
  - 1 die Elemente haben den gleichen Typ und
  - 2 dieser Typ unterstützt Ordnungen.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

`while`-  
Schleifen

Zusammen-  
fassung



## Erster Versuch

```
A = TypeVar ("A")
```

definiert eine **Typvariable**. Damit kennzeichnet der Typ `list[A]` eine Liste, in der alle Elemente den gleichen Typ `A` haben, aber ...

- wir wissen nicht, was `A` ist und
- wir wissen nicht, ob `A` Ordnungen unterstützt.

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung

## Exkursion: Typannotation für lexleq (3)



### Erweiterte Lösung

```
B = TypeVar ("B", int, float, str)
```

... wieder eine Typvariable, aber jetzt ist bekannt, dass sie für einen der aufgelisteten Typen `int`, `float` oder `str` steht.

```
def lex_leq(a : list[B], b : list[B]) -> bool:
```

bedeutet: `a` und `b` sind beides Listen, deren Elemente entweder `int` oder `float` oder `str` sind und daher vergleichbar!

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen

Skalarmultiplikation

Auswertung

Ableitung

Integration

Binäre Operationen

Addition

Multiplikation

Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung

# Exkursion: Typannotation für lexleq (3)



## Erweiterte Lösung

```
B = TypeVar ("B", int, float, str)
```

... wieder eine Typvariable, aber jetzt ist bekannt, dass sie für einen der aufgelisteten Typen `int`, `float` oder `str` steht.

```
def lex_leq(a : list[B], b : list[B]) -> bool:
```

bedeutet: `a` und `b` sind beides Listen, deren Elemente entweder `int` oder `float` oder `str` sind und daher vergleichbar!

## Bewertung

ok, aber was ist mit `list[int]`, `list[list[int]]` usw? Alle diese Typen sind auch vergleichbar...

Entwurf von  
Schleifen

Fallstudie:  
Rechnen mit  
Polynomen  
Skalarmultiplikation  
Auswertung  
Ableitung  
Integration  
Binäre Operationen  
Addition  
Multiplikation  
Extra:  
Lexikographische  
Ordnung

while-  
Schleifen

Zusammen-  
fassung



# while-Schleifen

Entwurf von  
Schleifen

**while-**  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



Wiederholen eines Schleifenrumpfs, ohne dass vorher klar ist, wie oft.

## Beispiele

- Einlesen von mehreren Eingaben
- Das Newton-Verfahren zum Auffinden von Nullstellen
- Das Collatz-Problem

Entwurf von  
Schleifen

**while-**  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



Wiederholen eines Schleifenrumpfs, ohne dass vorher klar ist, wie oft.

## Beispiele

- Einlesen von mehreren Eingaben
- Das Newton-Verfahren zum Auffinden von Nullstellen
- Das Collatz-Problem

## Die `while`-Schleife

- Syntax:  
`while Bedingung:`  
    *Block*     # Schleifenrumpf
- Semantik: Die Anweisungen im *Block* werden wiederholt, solange die *Bedingung* keinen Nullwert (z.B. `True`) liefert.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



# Einlesen einer Liste

Entwurf von  
Schleifen

`while`-  
Schleifen

**Einlesen einer  
Liste**

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 1: Bezeichner und Datentypen

Die Funktion `input_list` nimmt keine Parameter, erwartet eine beliebig lange Folge von Eingaben, die mit einer leeren Zeile abgeschlossen ist, und liefert als Ergebnis die Liste dieser Eingaben als Strings.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung

# Beispiel: Einlesen einer Liste



## Schritt 2: Funktionsgerüst

```
def input_list() -> list[str]:  
    # fill in, initialization  
    while expr_cond:  
        pass # fill in  
    return ...
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def input_list() -> list[str]:  
    # fill in, initialization  
    while expr_cond:  
        pass # fill in  
    return ...
```

## Warum while?

- Die Anzahl der Eingaben ist nicht von vorne herein klar.
- Dafür ist eine while-Schleife erforderlich.
- Die while-Schleife führt ihren Rumpf wiederholt aus, solange nicht-leere Eingaben erfolgen.

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Beispiele

Eingabe:

```
>>> input_list()

[]
>>> input_list()
Bring
mal
das
WLAN-Kabel!

['Bring', 'mal', 'das', 'WLAN-Kabel!']
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def input_list() -> list[str]:  
    result = []  
    line = input()  
    while line:  
        result = result + [line]  
        line = input()  
    return result
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



# Das Newton-Verfahren

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

**Das  
Newton-Verfahren**

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Suche Nullstellen von stetig differenzierbaren Funktionen

### Verfahren

$f: \mathbb{R} \rightarrow \mathbb{R}$  sei stetig differenzierbar

- 1 Wähle  $x_0 \in \mathbb{R}$ ,  $n = 0$
- 2 Setze 
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
- 3 Berechne nacheinander  $x_1, x_2, \dots, x_k$  bis  $f(x_k)$  nah genug an 0.
- 4 Ergebnis ist  $x_k$

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung

# Das Newton-Verfahren

Präzisierung

... für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`



UNI  
FREIBURG

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung

# Das Newton-Verfahren

Präzisierung



UNI  
FREIBURG

... für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

Was heißt hier "nah genug"?

- Eine überraschend schwierige Frage ...

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung

# Das Newton-Verfahren

## Präzisierung



### ... für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

### Was heißt hier “nah genug”?

- Eine überraschend schwierige Frage ...
- Wir sagen:  $x$  ist nah genug an  $x'$ , falls  $\frac{|x-x'|}{|x|+|x'|} < \varepsilon$

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung

# Das Newton-Verfahren

## Präzisierung



### ... für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

### Was heißt hier “nah genug”?

- Eine überraschend schwierige Frage ...
- Wir sagen:  $x$  ist nah genug an  $x'$ , falls  $\frac{|x-x'|}{|x|+|x'|} < \varepsilon$
- $\varepsilon > 0$  ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



### ... für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

### Was heißt hier “nah genug”?

- Eine überraschend schwierige Frage ...
- Wir sagen:  $x$  ist nah genug an  $x'$ , falls  $\frac{|x-x'|}{|x|+|x'|} < \varepsilon$
- $\varepsilon > 0$  ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.
- Wir wählen:  $\varepsilon = 2^{-20} \approx 10^{-6}$

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung

# Das Newton-Verfahren

## Präzisierung



### ... für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

### Was heißt hier “nah genug”?

- Eine überraschend schwierige Frage ...
- Wir sagen:  $x$  ist nah genug an  $x'$ , falls  $\frac{|x-x'|}{|x|+|x'|} < \varepsilon$
- $\varepsilon > 0$  ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.
- Wir wählen:  $\varepsilon = 2^{-20} \approx 10^{-6}$
- Genug für eine Hilfsfunktion!

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



Die freundlichen Pythonistas waren schon für uns aktiv. `pytest` ist ein Modul zur Erstellung von Tests.<sup>1</sup> Darin ist eine passende Hilfsfunktion definiert:

```
from pytest import approx
```

Diese Funktion erzeugt eine approximative Zahl, bei der Operator `==` ähnlich wie “nah genug” implementiert ist.

Es reicht, wenn ein Argument approximativ ist.

---

<sup>1</sup>Falls nicht vorhanden: `pip3 install pytest`

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



Die freundlichen Pythonistas waren schon für uns aktiv. `pytest` ist ein Modul zur Erstellung von Tests.<sup>1</sup> Darin ist eine passende Hilfsfunktion definiert:

```
from pytest import approx
```

Diese Funktion erzeugt eine approximative Zahl, bei der Operator `==` ähnlich wie “nah genug” implementiert ist.

Es reicht, wenn ein Argument approximativ ist.

Alternative: verwende `math.isclose()` ...

---

<sup>1</sup>Falls nicht vorhanden: `pip3 install pytest`

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 1: Bezeichner und Datentypen

Die Funktion `newton` nimmt als Eingabe

- `f : list[float]` ein Polynom
- `x0 : float` einen Startwert

und verwendet das Newton-Verfahren zur Berechnung einer Zahl  $x$ , sodass  $f(x)$  “nah genug” an 0 ist.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 2: Funktionsgerüst

```
def newton(  
    f : list[float],  
    x0 : float  
    ) -> float:  
    # fill in  
    while expr_cond:  
        pass # fill in  
    return ...
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



## Warum while?

- Das Newton-Verfahren verwendet eine Folge  $x_n$ ,  
ohne dass von vorne herein klar ist, wieviele Elemente benötigt werden.
- Zur Verarbeitung dieser Folge ist eine while-Schleife erforderlich.
- Diese while-Schleife terminiert aufgrund der mathematischen / numerischen Eigenschaften des Newton-Verfahrens. Siehe Vorlesung Mathe I.

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



Beispielfunktion:  $f(x) = x^3 - 6x^2 + 11x - 6$



Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 3: Beispiele

```
p = [-6, 11, -6, 1]
assert newton (p, 0)    == approx (1)
assert newton (p, 1.1) == approx (1)
assert newton (p, 1.7) == approx (2)
assert newton (p, 2.5) == approx (1)
assert newton (p, 2.7) == approx (3)
assert newton (p, 10)  == approx (3)
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



## Schritt 4: Funktionsdefinition

```
def newton(  
    f : list[float],  
    x0 : float  
    ) -> float:  
    deriv_f = derivative(f)  
    xn = x0  
    while poly_eval (f, xn) != approx(0):  
        xn = xn - ( poly_eval (f, xn)  
                   / poly_eval (deriv_f, xn))  
    return xn
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



# Das Collatz-Problem

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
**Collatz-Problem**

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl  $n$  und definiere eine Folge  $n = a_0, a_1, a_2, \dots$ :

$$a_{i+1} = \begin{cases} a_i/2 & a_i \text{ gerade} \\ 3a_i + 1 & a_i \text{ ungerade} \end{cases}$$

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



## Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl  $n$  und definiere eine Folge  $n = a_0, a_1, a_2, \dots$ :

$$a_{i+1} = \begin{cases} a_i/2 & a_i \text{ gerade} \\ 3a_i + 1 & a_i \text{ ungerade} \end{cases}$$

## Offene Frage

Für welche Startwerte  $n$  gibt es ein  $i$  mit  $a_i = 1$ ?

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



## Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl  $n$  und definiere eine Folge  $n = a_0, a_1, a_2, \dots$ :

$$a_{i+1} = \begin{cases} a_i/2 & a_i \text{ gerade} \\ 3a_i + 1 & a_i \text{ ungerade} \end{cases}$$

## Offene Frage

Für welche Startwerte  $n$  gibt es ein  $i$  mit  $a_i = 1$ ?

## Beispiele (Folge der durchlaufenen Zahlen)

- [3, 10, 5, 16, 8, 4, 2, 1]
- [7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



```
def collatz (n : int) -> list[int]:  
    result = [n]  
    while n > 1:  
        if n % 2 == 0:  
            n = n // 2  
        else:  
            n = 3 * n + 1  
        result = result + [n]  
    return result
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



## Warum while?

- Es ist nicht bekannt, ob `collatz(n)` für jede Eingabe terminiert.
- Aber validiert für alle  $n < 20 \cdot 2^{58} \approx 5.7646 \cdot 10^{18}$  (Oliveira e Silva).

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem  
Abschließende  
Bemerkungen

Zusammen-  
fassung



# Abschließende Bemerkungen

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

**Abschließende  
Bemerkungen**

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

**Abschließende  
Bemerkungen**

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`  
Anzahl der Elemente in der Sequenz `seq`

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`  
Anzahl der Elemente in der Sequenz `seq`
  - `for i in range(...):`  
Größe des Range

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das

Newton-Verfahren

Das

Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`  
Anzahl der Elemente in der Sequenz `seq`
  - `for i in range(...):`  
Größe des Range
- Daher **terminiert** die Ausführung einer `for`-Schleife i.a.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das

Newton-Verfahren

Das

Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`  
Anzahl der Elemente in der Sequenz `seq`
  - `for i in range(...):`  
Größe des Range
- Daher **terminiert** die Ausführung einer `for`-Schleife i.a.
- Bei einer `while`-Schleife ist die Anzahl der Durchläufe **nicht a-priori klar**.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das

Newton-Verfahren

Das

Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`  
Anzahl der Elemente in der Sequenz `seq`
  - `for i in range(...):`  
Größe des Range
- Daher **terminiert** die Ausführung einer `for`-Schleife i.a.
- Bei einer `while`-Schleife ist die Anzahl der Durchläufe **nicht a-priori klar**.
- Daher ist stets eine Überlegung erforderlich, ob eine `while`-Schleife terminiert (**Terminationsbedingung**).

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das

Newton-Verfahren

Das

Collatz-Problem

Abschließende

Bemerkungen

Zusammen-  
fassung



- Die Anzahl der Durchläufe einer `for`-Schleife ist stets durch den Schleifenkopf vorgegeben:
  - `for element in seq:`  
Anzahl der Elemente in der Sequenz `seq`
  - `for i in range(...):`  
Größe des Range
- Daher **terminiert** die Ausführung einer `for`-Schleife i.a.
- Bei einer `while`-Schleife ist die Anzahl der Durchläufe **nicht a-priori klar**.
- Daher ist stets eine Überlegung erforderlich, ob eine `while`-Schleife terminiert (**Terminationsbedingung**).
- Die Terminationsbedingung **muss** im Programm z.B. als Kommentar dokumentiert werden.

Entwurf von  
Schleifen

`while`-  
Schleifen

Einlesen einer  
Liste

Das

Newton-Verfahren

Das

Collatz-Problem

Abschließende

Bemerkungen

Zusammen-  
fassung

# Beispiel Zweierlogarithmus (Terminationsbedingung)



## Zweierlogarithmus

$$\log_2 a = b$$

$$2^b = a$$

■ für  $a > 0$

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung

# Beispiel Zweierlogarithmus (Terminationsbedingung)



## Zweierlogarithmus

$$\log_2 a = b$$

$$2^b = a$$

■ für  $a > 0$

## für ganze Zahlen

$$\lfloor \log_2 n \rfloor = m$$

$$m = \lfloor \log_2 n \rfloor$$

■ für  $n > 0$

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung



```
def l2 (n : int) -> int:  
  m = -1  
  while n>0:  
    m = m + 1  
    n = n // 2  
  return m
```

Entwurf von  
Schleifen

while-  
Schleifen

Einlesen einer  
Liste

Das  
Newton-Verfahren

Das  
Collatz-Problem

Abschließende  
Bemerkungen

Zusammen-  
fassung

## Terminationsbedingung

- Die `while`-Schleife terminiert, weil für alle  $n > 0$  gilt, dass  $n > n // 2$  und jede Folge von positiven ganzen Zahlen  $n_1 > n_2 > \dots$  abbricht.
- Die Anzahl der Schleifendurchläufe ist durch  $\log_2 n$  beschränkt.



# Zusammenfassung

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.
- Der verwendete Range hängt von der Problemstellung ab.

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.
- Der verwendete Range hängt von der Problemstellung ab.
- **Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.**

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.
- Der verwendete Range hängt von der Problemstellung ab.
- **Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.**
- **while-Schleifen** werden verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.
- Der verwendete Range hängt von der Problemstellung ab.
- **Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.**
- **while-Schleifen** werden verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise
  - zur Verarbeitung von Eingaben

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.
- Der verwendete Range hängt von der Problemstellung ab.
- **Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.**
- **while-Schleifen** werden verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise
  - zur Verarbeitung von Eingaben
  - zur Berechnung von Approximationen

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung



- Funktionen über **Sequenzen** verwenden **for-in-Schleifen**.
- Ergebnisse werden meist in einer **Akkumulator** Variable berechnet.
- Funktionen über **mehreren Sequenzen** verwenden **for-range-Schleifen**.
- Der verwendete Range hängt von der Problemstellung ab.
- **Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.**
- **while-Schleifen** werden verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise
  - zur Verarbeitung von Eingaben
  - zur Berechnung von Approximationen
- Jede while-Schleife muss eine **dokumentierte Terminationsbedingung** haben.

Entwurf von  
Schleifen

while-  
Schleifen

Zusammen-  
fassung