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

5. November 2025

Entwurf von Schleifen

Fallstudie: Rechnen mit Polynomen

Polynome

Definition

Ein *Polynom vom Grad n* ist eine Folge von Zahlen $[p_0, p_1, \ldots, p_n]$, den *Koeffizienten*. Dabei ist $n \geq 0$ und der *Leitkoeffizient* $p_n \neq 0$.

Andere Schreibweise: $\sum_{i=0}^{n} p_i x^i$

Beispiele

$$\begin{array}{lcl} [] & \approx & 0 \\ [1] & \approx & 1 \\ [3,2,1] & \approx & 3 + 2x + x^2 \end{array}$$

Anwendungen

Kryptographie, fehlerkorrigierende Codes.

Darstellung von Polynomen in Python

- Liste von Gleitkommazahlen

```
type polynom = list[float]
```

- Diese *Typdefinition* definiert einen neuen Typ mit Namen `polynom`.
- Der Typ `polynom` ist gleichwertig zum Typ `list[float]`.
- Konvention für ein Polynom `p` (Erinnerung):
  `len(p) == 0 or p[-1] != 0`

Rechenoperationen auf Polynomen

- (Skalar) Multiplikation mit einer Zahl c

$$c \cdot [p_0, p_1, \dots, p_n] = [c \cdot p_0, c \cdot p_1, \dots, c \cdot p_n]$$

- Auswertung an der Stelle x_0

$$[p_0, p_1, \dots, p_n](x_0) = \sum_{i=0}^{n} p_i \cdot x_0^i$$

- Ableitung

$$[p_0, p_1, \dots, p_n]' = [1 \cdot p_1, 2 \cdot p_2, \dots, n \cdot p_n]$$

- Integration

$$\int [p_0, p_1, \dots, p_n] = [0, p_0, p_1/2, p_2/3, \dots, p_n/(n+1)]$$

Skalarmultiplikation

Skalarmultiplikation

$$c \cdot [p_0, p_1, \ldots, p_n] = [c \cdot p_0, c \cdot p_1, \ldots, c \cdot p_n]$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `scalar_mult` nimmt als Eingabe

- `c : float`, den Faktor,
- `p : polynom`, ein Polynom.

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

Skalarmultiplikation

Schritt 2: Funktionsgerüst

```
def scalar_mult(
        c : float,
        p : polynom
        ) -> polynom:
    # fill in, initialization
    for i in range(len(p)):
        pass # fill in action for each element
    return ...
```

Skalarmultiplikation

Schritt 3: Beispiele

```
assert(scalar_mult(42, []) == [])
assert(scalar_mult(42, [1,2,3]) == [42,84,126])
assert(scalar_mult(-0.1, [1,2,4]) == [-0.1,-0.2,-0.4])
```

Skalarmultiplikation

Schritt 4: Funktionsdefinition

```
def scalar_mult(
          c : float,
          p : polynom
          ) -> polynom:
    result = []
    for i in range(len(p)):
          result = result + [c * p[i]]
    return result
```

Rumpf der Skalarmultiplikation

```
result = []        # initialization
for i in range(len(p)):
    result = result + [c * p[i]]      # update
return result
```

Muster: Akkumulator

Rumpf der Skalarmultiplikation

```
result = []          # initialization
for i in range(len(p)):
    result = result + [c * p[i]]      # update
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.
- Jede Schleifeniteration aktualisiert das Ergebnis in `result`,
  indem das Ergebnis um das aktuelle Element `p[i]` erweitert wird.

Begründung

- $p = [p_0, p_1, \ldots, p_n]$
- `r = []`
- $r = []$
- `for i in range(len(p):`
- vorher: $r = [c \cdot p_0, \ldots, c \cdot p_{i-1}]$
- `r = r + [c * p[i]]`
- nachher: $r = [c \cdot p_0, \ldots, c \cdot p_i]$
- nach dem $n+1$-ten Durchlauf (letzter Durchlauf der Schleife):
  $r = [c \cdot p_0, \ldots, c \cdot p_n]$

Ein fehlendes Beispiel

Ein fehlendes Beispiel

Skalarmultiplikation mit 0

Auswertung

Auswertung

$$[p_0, p_1, \ldots, p_n](x_0) = \sum_{i=0}^{n} p_i \cdot x_0^i$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_eval` nimmt als Eingabe

- `p : polynom`, ein Polynom,
- `x : float`, das Argument.

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

Schritt 2: Funktionsgerüst

```
def poly_eval(
        p : polynom,
        x : float
        ) -> float:
    # fill in
    for i in range(len(p)):
        pass # fill in action for each element
    return ...
```

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)
```

Auswertung

Schritt 4: Funktionsdefinition

```
def poly_eval(
        p : polynom,
        x : float
        ) -> float:
    result = 0
    for i in range(len(p)):
        result = result + p[i] * x ** i
    return result
```

Ableitung

Ableitung

$$[p_0, p_1, \ldots, p_n]' = [1 \cdot p_1, 2 \cdot p_2, \ldots, n \cdot p_n]$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `derivative` nimmt als Eingabe

- `p : polynom`, ein Polynom.

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

Ableitung

Schritt 2: Funktionsgerüst

```
def derivative(
        p : polynom
        ) -> polynom:
    # initialization
    for i in range(len(p)):
        pass # fill in action for each element
    return ...
```

Ableitung

Schritt 3: Beispiele

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

Ableitung

Schritt 4: Funktionsdefinition

```
def derivative(
        p : polynom
        ) -> polynom:
    result = []
    for i in range(len(p)):
        if i > 0:
            result = result + [i * p[i]]
    return result
```

Integration

Integration

$$\int [p_0, p_1, \ldots, p_n] = [0, p_0, p_1/2, p_2/3, \ldots, p_n/(n+1)]$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `integral` nimmt als Eingabe

- `p : polynom`, ein Polynom.

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

Weitere Schritte

selbst

Binäre Operationen

Operationen mit zwei Polynomen

- Addition (falls $n \leq m$)

$$[p_0, p_1, \dots, p_n] + [q_0, q_1, \dots, q_m]$$
$$= [p_0 + q_0, p_1 + q_1, \dots, p_n + q_n, q_{n+1}, \dots, q_m]$$

- Multiplikation von Polynomen

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

Addition

Addition

$$(p_0, p_1, \ldots, p_n) + (q_0, q_1, \ldots, q_m)$$
$$= (p_0 + q_0, p_1 + q_1, \ldots, p_n + q_n, q_{n+1}, \ldots, q_m)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_add` nimmt als Eingabe

- `p : polynom`, ein Polynom.
- `q : polynom`, ein Polynom.

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

Achtung

Die Grade der Polynome können unterschiedlich sein!

Schritt 2: Funktionsgerüst

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

Frage

Was ist das Argument . . . von `range`?

Addition

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])
```

Addition

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))
```

Addition

Schritt 4: Funktionsdefinition, erster Versuch

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

Addition

Problem

Eine Assertion schlägt fehl!

Addition

Problem

Eine Assertion schlägt fehl!

```
assert(poly_add([], []) == [])
assert(poly_add([42], []) == [42])
```

Addition

Problem

Eine Assertion schlägt fehl!

```
assert(poly_add([], []) == [])
assert(poly_add([42], []) == [42])
```

Analyse

Zweite Assertion schlägt fehl für `i=0`!

```
Traceback (most recent call last):
  File "<outputdir>/py_default_default.py", line 124, in <module>
    assert(poly_add([42], []) == [42])
           ^^^^^^^^^^^^^^^^^^^
  File "<outputdir>/py_default_default.py", line 108, in poly_add
    result = result + [p[i] + q[i]]
```

Addition — Wunschdenken

Neuer Entwurfsschritt: Wunschdenken

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

Addition — Wunschdenken

Neuer Entwurfsschritt: Wunschdenken

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

Schritt 1: Bezeichner und Datentypen

Die Funktion `safe¡ndex` nimmt als Eingabe

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

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

Sichere Indizierung | Addition

Schritt 2: Funktionsgerüst

```
def safe_index(
        p : list[float],
        i : int,  # assume >= 0
        d : float
        ) -> float:
    # fill in
    return 0
```

Sichere Indizierung | Addition

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
```

Sichere Indizierung | Addition

Schritt 4: Funktionsdefinition

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

oder (alternative Implementierung des Funktionsrumpfes)

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

Neuer Ausdruck

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 " "`

Addition

Schritt 4: Funktionsdefinition mit Hilfsfunktion

```
def poly_add(
        p : polynom,
        q : polynom
        ) -> polynom:
    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
```

Multiplikation

Multiplikation

$$[p_0, p_1, \ldots, p_n] \cdot [q_0, q_1, \ldots, q_m]$$

$$= [p_0 \cdot q_0, p_0 \cdot q_1 + p_1 \cdot q_0, \ldots, \sum_{i=0}^{k} p_i \cdot q_{k-i}, \ldots, p_n \cdot q_m]$$

Multiplikation

$$[p_0, p_1, \ldots, p_n] \cdot [q_0, q_1, \ldots, q_m]$$
$$= [p_0 \cdot q_0, p_0 \cdot q_1 + p_1 \cdot q_0, \ldots, \sum_{i=0}^{k} p_i \cdot q_{k-i}, \ldots, p_n \cdot q_m]$$

Woher kommt diese Definition?

$$\left(\sum_{i=0}^{n} p_i x^i\right) \cdot \left(\sum_{j=0}^{m} q_j x^j\right) = \sum_{i=0}^{n}\sum_{j=0}^{m} p_i x^i \cdot q_j x^j = \sum_{i=0}^{n}\sum_{j=0}^{m} p_i \cdot q_j \cdot x^{i+j}$$
$$= \sum_{k=0}^{n+m}\sum_{i+j=k} p_i \cdot q_j \cdot x^k = \sum_{k=0}^{n+m}\sum_{i=0}^{k} p_i \cdot q_{k-i} \cdot x^k$$

Multiplikation

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_mult` nimmt als Eingabe

- `p : polynom` ein Polynom
- `q : polynom` ein Polynom

und liefert als Ergebnis das Produkt der Eingaben.

Multiplikation

Schritt 2: Funktionsgerüst

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

Multiplikation

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]
```

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 `maxlen = len (p) + len (q) - 1`

Schritt 4: Funktionsdefinition

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

Das k-te Element

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

noch eine Schleife!

Multiplikation

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)
```

Multiplikation

Schritt 4: Funktionsdefinition, final

```
def poly_mult(
          p : polynom,
          q : polynom
          ) -> polynom:
    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
```

Verbesserte Typannotationen

Verbesserte Typannotationen

Am Beispiel der sicheren Indizierung

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

- Laut Typannotation muss das Argument `p` immer `list[float]`, das Argument `d : float` und demzufolge das Ergebnis `float` sein.

Verbesserte Typannotationen

Am Beispiel der sicheren Indizierung

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

- Laut Typannotation muss das Argument `p` immer `list[float]`, das Argument `d : float` und demzufolge das Ergebnis `float` sein.
- Am Code sehen wir aber, dass keine arithmetischen Operationen auf `d` oder die Elemente von `p` angewendet werden, sondern dass diese einfach durchgereicht werden!

Verbesserte Typannotationen

Am Beispiel der sicheren Indizierung

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

- Laut Typannotation muss das Argument `p` immer `list[float]`, das Argument `d : float` und demzufolge das Ergebnis `float` sein.
- Am Code sehen wir aber, dass keine arithmetischen Operationen auf `d` oder die Elemente von `p` angewendet werden, sondern dass diese einfach durchgereicht werden!
- Eine solche Funktion heißt *parametrisch polymorph*, weil statt `float` ein beliebiger Typ verwendet werden darf.

Verbesserte Typannotationen

Typvariable

- Schreibweise für einen genaueren generischen Typ:

```
def safe_index[T](
        p : list[T],
        i : int,   # assume >= 0
        d : T
        ) -> T:
```

Verbesserte Typannotationen

Typvariable

- Schreibweise für einen genaueren generischen Typ:

```
def safe_index[T](
        p : list[T],
        i : int,   # assume >= 0
        d : T
        ) -> T:
```

- `T` ist eine Typvariable, die für einen beliebigen Typ steht.

Verbesserte Typannotationen

Typvariable

- Schreibweise für einen genaueren generischen Typ:

```
def safe_index[T](
        p : list[T],
        i : int,   # assume >= 0
        d : T
        ) -> T:
```

- `T` ist eine Typvariable, die für einen beliebigen Typ steht.
- Sie wird durch `[T]` eingeführt und darf in den Typannotationen der Kopfzeile verwendet werden.

- Schreibweise für einen genaueren generischen Typ:

```
def safe_index[T](
        p : list[T],
        i : int,   # assume >= 0
        d : T
        ) -> T:
```

- T ist eine Typvariable, die für einen beliebigen Typ steht.
- Sie wird durch [T] eingeführt und darf in den Typannotationen der Kopfzeile verwendet werden.
- Bei Verwendung von safe_index setzt der Typchecker einen passenden Typ ein, der konsistent verwendet werden muss.

Generischer Typ

- Ein generischer Typ enthält eine oder mehrere Typvariablen (wie `list[T]`).
- Er steht als “Abkürzung” für alle Typen, die man durch Einsetzen von erlaubten konkreten Typen für die Typvariablen herstellen kann.
- Ohne weitere Beschränkung sind **alle** konkreten Typen erlaubt.

Lexikographische Ordnung

Die lexikographische Ordnung

Gegeben

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

$$\vec{a} = \text{"}a_1 a_2 \dots a_m\text{"} \qquad \vec{b} = \text{"}b_1 b_2 \dots b_n\text{"}$$

Die lexikographische Ordnung

Gegeben

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

$$\vec{a} = \text{"}a_1 a_2 \ldots a_m\text{"} \qquad \vec{b} = \text{"}b_1 b_2 \ldots b_n\text{"}$$

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

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

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

$$\vec{a} = \text{"}a_1 a_2 \ldots a_k a_{k+1} \ldots a_m\text{"} \qquad \vec{b} = \text{"}a_1 a_2 \ldots a_k b_{k+1} \ldots b_n\text{"}$$

  - $k = m$

$$\vec{a} = \text{"}a_1 a_2 \ldots a_m\text{"} \qquad \vec{b} = \text{"}a_1 a_2 \ldots a_m b_{m+1} \ldots b_n\text{"}$$

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

Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_leq` nimmt als Eingabe

- `a : list[int]` eine Sequenz von Zahlen
- `b : list[int]` eine Sequenz von Zahlen

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

Lexikographische Ordnung

Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_leq` nimmt als Eingabe

- `a : list[int]` eine Sequenz von Zahlen
- `b : list[int]` eine Sequenz von Zahlen

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

Schritt 2: Funktionsgerüst

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

Lexikographische Ordnung

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
```

Lexikographische Ordnung

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

- Range `minlen = min (len (a), len (b))`

Lexikographische Ordnung

Schritt 4: Funktionsdefinition

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

Typannotation für lexleq (1)

Problem

- Der Typ `list[int]` charakterisiert Listen von Zahlen.
- Aber der Code funktioniert viel allgemeiner, wenn nur die Elemente vergleichbar vom gleichen Typ sind!
  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.

Typannotation für lexleq (2)

Verbesserung

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

`B` ist eine Typvariable, aber jetzt ist bekannt, dass sie für einen der aufgelisteten Typen **`int`**, **`float`** oder **`str`** steht.
D.h.: `a` und `b` sind beides Listen, deren Elemente entweder `int` oder `float` oder `str` sind und daher vergleichbar!

Typannotation für lexleq (2)

Verbesserung

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

`B` ist eine Typvariable, aber jetzt ist bekannt, dass sie für einen der aufgelisteten Typen **`int`**, **`float`** oder **`str`** steht.
D.h.: `a` und `b` sind beides Listen, deren Elemente entweder `int` oder `float` oder `str` sind und daher vergleichbar!

Bewertung: Noch nicht optimal…

ok, aber was ist mit **`list[int]`**, **`list[list[int]]`** usw? Alle diese Typen sind auch vergleichbar…
Bessere Konzepte in Rust, Haskell, Scala, …

`while`-Schleifen

`while`-Schleifen

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

`while`-Schleifen

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.

Einlesen einer Liste

Beispiel: Einlesen einer Liste

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.

Beispiel: Einlesen einer Liste

Schritt 2: Funktionsgerüst

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

Beispiel: Einlesen einer Liste

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 <u>nicht von vorne herein klar</u>.
- Dafür ist eine while-Schleife erforderlich.
- Die while-Schleife führt ihren Rumpf solange aus, bis eine leere Eingabe erfolgt.

Einlesen einer Liste

Beispiele

Eingabe:

```
>>> input_list()

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

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

Einlesen einer Liste

Schritt 4: Funktionsdefinition

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

Das Newton-Verfahren

Das Newton-Verfahren

Suche Nullstellen von stetig differenzierbaren Funktionen

Verfahren

$f : \mathbb{R} \to \mathbb{R}$ 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

Das Newton-Verfahren

Präzisierung

… für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

Das Newton-Verfahren

Präzisierung

… für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

Was heißt hier “nah genug”?

- Eine überraschend schwierige Frage …

… 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 $\frac{|x - x'|}{\max(|x|, |x'|)} < \varepsilon$

… 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 $\frac{|x-x'|}{\max(|x|,|x'|)} < \varepsilon$
- $\varepsilon > 0$ ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.

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 $\frac{|x-x'|}{\max(|x|,|x'|)} < \varepsilon$
- $\varepsilon > 0$ ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.
- Wir wählen: $\varepsilon = 2^{-30} \approx 10^{-9}$

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 $\frac{|x-x'|}{\max(|x|,|x'|)} < \varepsilon$
- $\varepsilon > 0$ ist eine Konstante, die von der Repräsentation von `float`, dem Verfahren und der gewünschten Genauigkeit abhängt. Dazu kommen noch Sonderfälle.
- Wir wählen: $\varepsilon = 2^{-30} \approx 10^{-9}$
- Genug für eine Hilfsfunktion!

Hilfsfunktion

Die freundlichen Pythonistas waren schon für uns aktiv. Das Modul `math` enthält eine passende Hilfsfunktion:

> `math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`
>
> *Return `True` if the values a and b are close to each other and False otherwise.*
>
> *Whether two values are considered close is determined according to given absolute and relative tolerances. If no errors occur, the result will be:*
>
> `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol).`

Newton-Verfahren

Schritt 1: Bezeichner und Datentypen

Die Funktion `newton` nimmt als Eingabe

- `f : polynom` ein Polynom
- `x0 : float` einen Startwert

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

Newton-Verfahren

Schritt 2: Funktionsgerüst

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

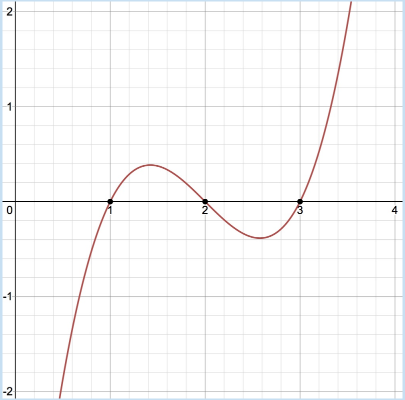
Newton-Verfahren

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.

Newton-Verfahren

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



Schritt 3: Beispiele

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

Newton-Verfahren

Schritt 4: Funktionsdefinition

```
def newton(
        f   : polynom,
        x0  : float
        ) -> float:
    deriv_f = derivative(f)
    xn = x0
    while not math.isclose( poly_eval (f, xn), 0):
        xn = xn - ( poly_eval (f, xn)
                  / poly_eval (deriv_f, xn))
    return xn
```

Das Collatz-Problem

Das Collatz-Problem

Verfahren (Collatz 1937)

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

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

Das Collatz-Problem

Verfahren (Collatz 1937)

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

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

Offene Frage

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

Das Collatz-Problem

Verfahren (Collatz 1937)

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

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

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]`

Collatz-Problem

```python
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
```

Collatz-Problem

Warum while?

- Es ist nicht bekannt, ob `collatz(n)` für jede Eingabe terminiert.
- Aber validiert für alle $n < 2^{71} \approx 2.36 \cdot 10^{21}$!
  Barina, David (2025). Improved verification limit for the convergence of the Collatz conjecture. The Journal of Supercomputing. 81 (7) 810. doi:10.1007/s11227-025-07337-0.

Abschließende Bemerkungen

Termination einer Schleife

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

Termination einer Schleife

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

Termination einer Schleife

- 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

Termination einer Schleife

- 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.

Termination einer Schleife

- 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.

Termination einer Schleife

- 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).

Termination einer Schleife

- 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.

Beispiel Zweierlogarithmus (Terminationsbedingung)

Zweierlogarithmus

$$\log_2 a = b$$
$$2^b = a$$

- für $a > 0$

Beispiel Zweierlogarithmus (Terminationsbedingung)

Zweierlogarithmus

$$\log_2 a = b$$
$$2^b = a$$

- für $a > 0$

für ganze Zahlen

$$\texttt{l2 (n)} = m$$
$$m = \lfloor \log_2 n \rfloor$$

- für $n > 0$

Implementierung Zweierlogarithmus

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

Terminationsbedingung

- Die `while`-Schleife terminiert, weil für alle `n>0` gilt, dass `n > n//2` und jede absteigende Folge von positiven ganzen Zahlen `n1 > n2 > ...` abbricht.
- Die Anzahl der Schleifendurchläufe ist durch $\log_2 n$ beschränkt.

Zusammenfassung

Zusammenfassung

- 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.
- 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.