Informatik I: Einführung in die Programmierung

7.2 While-Schleifen

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Peter Thiemann

20. November 2019

while-Schleifen

while-Schleifen

UNI FREIBURG

Manchmal muss etwas wiederholt werden, ohne dass vorher klar ist, wie oft.

Beispiele

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

while-Schleifen

UNI FREIBURG

Manchmal muss etwas wiederholt werden, ohne dass vorher klar ist, wie oft.

Beispiele

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

Die `while`-Schleife

- Syntax der `while`-Anweisung:
  `while` *Bedingung*: # Schleifenkopf
  Anweisungen # Schleifenrumpf
- Der Schleifenrumpf wird nur betreten, falls *Bedingung* keinen Nullwert liefert.
- Der Schleifenrumpf (*Anweisungen*) wird wiederholt, solange die *Bedingung* keinen Nullwert liefert. Z.B. solange sie `True` ist.

UNI FREIBURG

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

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def input_list() -> list: # of string
    # fill in, initialization
    while CONDITION:
        # fill in
    return
```

Beispiel: Einlesen einer Liste

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def input_list() -> list: # of string
    # fill in, initialization
    while CONDITION:
        # fill in
    return
```

Warum while?

- Die Anzahl der Eingaben ist nicht von vorne herein klar.
- Dafür ist eine while-Schleife erforderlich.
- Die while-Schleife läuft weiter, solange nicht-leere Eingaben erfolgen.
- Die while-Schleife terminiert (d.h., sie wird nur endlich oft durchlaufen), sobald eine leere Eingabe erfolgt.

Einlesen einer Liste

UNI FREIBURG

Beispiele

Eingabe:

```
>>> input_list()

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

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

Einlesen einer Liste

UNI FREIBURG

Schritt 4: Funktionsdefinition

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

UNI FREIBURG

Das Newton-Verfahren

Das Newton-Verfahren

UNI FREIBURG

Suche Nullstellen von stetig differenzierbaren Funktionen

Verfahren

$f\colon \mathbb{R} \to \mathbb{R}$ sei stetig differenzierbar

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

. . . für Polynomfunktionen

- Erfüllen die Voraussetzung
- Ableitung mit `derivative`

. . . 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 $\frac{|x-x'|}{|x|+|x'|} < \varepsilon$

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

. . . 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.
- Genug für eine Hilfsfunktion!

. . . 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.
- Genug für eine Hilfsfunktion!
- Oder wir verwenden `math.isclose`.

Schritt 1: Bezeichner und Datentypen

Die Funktion `newton` nimmt als Eingabe

- `f : list` 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

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def newton(
        f  : list, # of float
        x0 : float
        ) -> bool:
    # fill in
    while CONDITION:
        # fill in
    return
```

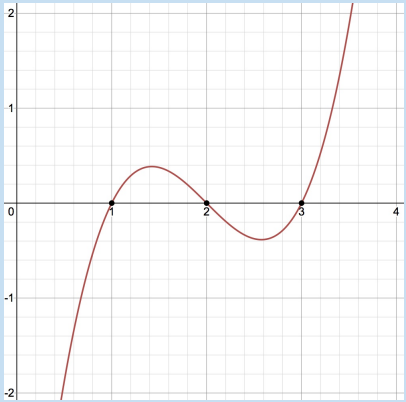
Newton-Verfahren

UNI FREIBURG

Warum while?

- Das Newton-Verfahren definiert eine Folge x_n, von der nicht von vorne herein klar ist, wieviele Elemente benötigt werden.
- Zur Berechnung dieser Folge ist eine while-Schleife erforderlich.
- Diese while-Schleife terminiert aufgrund der mathematischen / numerischen Eigenschaften des Newton-Verfahrens. Siehe Mathematik-Vorlesung.

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



Newton-Verfahren

UNI FREIBURG

Schritt 3: Beispiele

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

Schritt 4: Funktionsdefinition

```
def newton(
        f  : list, # of float
        x0 : float
        ) -> bool:
    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

UNI FREIBURG

Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl n.

- Falls n gerade, fahre fort mit $n/2$.
- Sonst fahre fort mit $3n+1$.
- Wiederhole bis $n = 1$.

Das Collatz-Problem

UNI FREIBURG

Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl n.

- Falls n gerade, fahre fort mit $n/2$.
- Sonst fahre fort mit $3n+1$.
- Wiederhole bis $n = 1$.

Offene Frage

Nach wievielen Wiederholungen wird $n = 1$ erreicht?

Das Collatz-Problem

UNI FREIBURG

Verfahren (Collatz 1937)

Starte mit einer positiven ganzen Zahl n.

- Falls n gerade, fahre fort mit $n/2$.
- Sonst fahre fort mit $3n+1$.
- Wiederhole bis $n = 1$.

Offene Frage

Nach wievielen Wiederholungen wird $n = 1$ erreicht?

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

UNI FREIBURG


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

Collatz-Problem

UNI FREIBURG

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).

Abschließende Bemerkungen

break, continue, else

Die Anweisungen `break`, `continue` und `else` wirken auf `while`-Schleifen genauso wie auf `for`-Schleifen:

- `break` beendet eine Schleife vorzeitig.
- `continue` beendet die aktuelle Schleifeniteration vorzeitig, d.h. springt zum Schleifenkopf um den nächsten Schleifentest durchzuführen.
- Schleifen können einen `else`-Zweig haben. Dieser wird nach Beendigung der Schleife ausgeführt, und zwar genau dann, wenn die Schleife *nicht* mit `break` verlassen wurde.

Termination der Schleife

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

Termination der Schleife

UNI FREIBURG

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

Termination der Schleife

UNI FREIBURG

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

Termination der Schleife

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

Termination der Schleife

UNI FREIBURG

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

Termination der Schleife

UNI FREIBURG

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

Termination der Schleife

UNI FREIBURG

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

Beispiel Terminationsbedingung: Zweierlogarithmus

UNI FREIBURG

Zweierlogarithmus

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

- für $a > 0$

Beispiel Terminationsbedingung: Zweierlogarithmus

UNI FREIBURG

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

UNI FREIBURG


```
def l2 (n : int) -> int:
    """ Zweierlogarithmus für n>0 """
    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 Folge `n1 > n2 > ...` von positiven ganzen Zahlen abbricht.
- Anzahl der Schleifendurchläufe ist durch $\log_2 n$ beschränkt.

Zusammenfassung

Zusammenfassung

- while-Schleifen dienen auch der Iteration (Wiederholung) von Berechnungen

Zusammenfassung

- while-Schleifen dienen auch der Iteration (Wiederholung) von Berechnungen
- while-Schleifen werden nur verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise

Zusammenfassung

- while-Schleifen dienen auch der Iteration (Wiederholung) von Berechnungen
- while-Schleifen werden nur verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise
  - zur Verarbeitung von Eingaben

Zusammenfassung

- while-Schleifen dienen auch der Iteration (Wiederholung) von Berechnungen
- while-Schleifen werden nur verwendet, wenn die Anzahl der Schleifendurchläufe nicht von vorne herein bestimmt werden kann oder soll, typischerweise
  - zur Verarbeitung von Eingaben
  - zur Berechnung von Approximationen

Zusammenfassung

- while-Schleifen dienen auch der Iteration (Wiederholung) von Berechnungen
- while-Schleifen werden nur 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.