Informatik I

18. Schleifen und Iteration

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

25. Januar 2011

Informatik I

25. Januar 2011 — 18. Schleifen und Iteration

Schleifen und Iteration

- Wir haben sowohl in Scheme als auch in Python schon einige rekursive Prozeduren bzw. Methoden/Funktionen gesehen.
- In Scheme haben wir insbesondere endrekursive Prozeduren betrachtet; diese haben konstanten Platzverbrauch ...

Schleifen und Iteration II

```
=> (it-factorial-1 15             1)
=> (it-factorial-1 14            15)
=> (it-factorial-1 13           210)
=> (it-factorial-1 12          2730)
=> (it-factorial-1 11         32760)
=> (it-factorial-1 10        360360)
=> (it-factorial-1  9       3603600)
=> (it-factorial-1  8      32432400)
=> (it-factorial-1  7     259459200)
=> (it-factorial-1  6    1816214400)
=> (it-factorial-1  5   10897286400)
=> (it-factorial-1  4   54486432000)
=> (it-factorial-1  3  217945728000)
=> (it-factorial-1  2  653837184000)
=> (it-factorial-1  1 1307674368000)
=> (it-factorial-1  0 1307674368000)
=> 1307674368000
```

Schleifen und Iteration III

- Wir haben endrekursive Funktionen auch als iterativ bezeichnet.
- Doch wie damals angekündigt, werden wir den Begriff “iterativ” nun etwas anders definieren/charakterisieren.

`while`-Schleifen

- Eine Schleife ist in der imperativen Programmierung ein Block, also eine Folge von Anweisungen, der mehrmals ausgeführt wird.
- Wie oft, hängt von Bedingungen ab.
- Es gibt die allgemeinste Form, die `while`-Schleife, und eine speziellere Form, die `for`-Schleife.
- Insbesondere in Python kommt die `for`-Schleife in der Praxis häufiger vor, doch die `while`-Schleife ist grundlegender und soll deshalb hier zuerst betrachtet werden.
- Unser Beispiel in diesem Kapitel ist die Fakultätsfunktion.

18.1 Für eine feste Zahl

Ein Beispiel

mystery.pyfactorial.py

```
i = 10
result = 1
while i > 0:
    result *= i
    i -= 1
print(result)
```

Was berechnet dieses Programm?
Die Fakultät von 10.

Programmablauf

factorial.py

```
i = 10
result = 1
while i > 0:
    result *= i
    i -= 1
print(result)
```

```
i = 10
result = 3628800
```

Ein allgemeines Programm

- Das kleine Beispielprogramm diente dazu, Schleifen vorzustellen.
- Allerdings erfüllt es nicht die Anforderung der Generalität: es sollte die Fakultät einer beliebigen Zahl berechnen.
- Das können wir auf zwei Arten erreichen (die erste Art wird nochmal zwei Untervarianten haben) ...

18.2 Benutzereingaben

- Eine weitere Schleife
- Schleifen abbrechen
- Shellaufrufe

Erste Art: Benutzereingaben

- Bei imperativen Programmen spielt Zeit eine große Rolle: Programmzeilen sind Befehle, die nacheinander ausgeführt werden. Diese Sichtweise unterscheidet sich von der funktionalen: dort gibt es eine Eingabe und die gesamte Ausführung eines Programms besteht aus der Auswertung einer Prozedur für diese Eingabe.
- Deshalb ist es ganz natürlich, dass das Programm im Laufe der Zeit den Benutzer immer wieder mal nach einer Eingabe fragt und eine Variable auf den Wert dieser Eingabe setzt.
- Wir haben allerdings auch in der imperativen/objektorientierten Programmierung Funktionen/Methoden, d.h. wir nehmen häufig auch die funktionale Sichtweise ein.

Fakultätsprogramm mit Benutzereingabe

factorial.py

```
n = input("Gib eine Zahl ein: ")
i = int(n)
result = 1
while i > 0:
    result *= i
    i -= 1
print("Fakultaet von", n, "ist", result)
```

Die Funktion `input` druckt ihr Argument aus und wartet dann eine Benutzereingabe ab. Sobald diese erfolgt ist (beendet durch die Eingabetaste), wird die Benutzereingabe als String zurückgegeben, also hier an `n` zugewiesen.

Verwendung des Programms

Python-Interpreter

```
>>> from factorial import *
Gib eine Zahl ein:  4
Fakultaet von 4 ist 24
>>> from factorial import *
>>>
```

- Das Programm kann im Python-Interpreter gestartet werden, allerdings ist das nicht sehr praktisch: nach einmaligem Ablauf müssten wir das Programm erneut laden, aber das tut der Interpreter normalerweise nicht; wir müssten also erst den Interpreter beenden und ihn dann nochmal neu starten.
- Hier wiederum zwei Lösungen ...

Eine weitere Schleife

factorial.py

```
n = 1
while n >= 0:
    n = input("Gib eine Zahl ein: ")
    n = int(n)
    i = n
    result = 1
    while i > 0:
        result *= i
        i -= 1
    if n >= 0:
        print("Fakultaet von", n, "ist", result)
```

Warum ist die Umwandlung von `n` in ein `int` nötig? Wegen des Vergleichs `n >= 0`

Verwendung des Programms mit zusätzlicher Schleife

Python-Interpreter

```
>>> from factorial import *
Gib eine Zahl ein:  2
Fakultaet von 2 ist 2
Gib eine Zahl ein:  7
Fakultaet von 7 ist 5040
Gib eine Zahl ein:  10
Fakultaet von 10 ist 3628800
Gib eine Zahl ein:  -1
>>>
```

Unnötige/schädliche Berechnungen

- Wir haben das Programm so konzipiert, dass die Eingabe einer negativen Zahl das Signal zum Abbrechen der Schleife ist.
- Allerdings wird in diesem Fall einiger Code noch ausgeführt, der unnötig ist und im Allgemeinen sogar schädlich sein könnte . . .

Unnötige Ausführungen in factorial.py

factorial.py

```
n = 1
while n >= 0:
    n = input("Gib eine Zahl ein: ")
    n = int(n)
    i = n
    result = 1
    while i > 0:
        result *= i
        i -= 1
    if n >= 0:
        print("Fakultaet von", n, "ist", result)
```

- Die roten Zeilen werden unnötigerweise noch ausgeführt.
- Zum Glück richten sie in diesem Fall keinen Schaden an, aber auch nur wegen der letzten if-Abfrage.

Unnötige/schädliche Berechnungen vermeiden

- Im Allgemeinen könnte durch die Ausführung von solchen Zeilen ein Schaden entstehen, etwa die Ausgabe

  `Fakultaet von -5 ist 1`

  oder, viel schlimmer, Nichtterminierung.
- Es gibt zwei Konstrukte, um dies zu vermeiden: `break` und `continue`.
- Wir wollen unser Programm so konzipieren: Eingabe -1 soll zum Abbruch führen, jede andere negative Zahl soll als Versehen interpretiert werden und es soll zur Eingabe einer neuen Zahl aufgefordert werden.

Fakultät mit break und continue

factorial.py

```
n = 1
while n >= 0:
    n = input("Gib eine Zahl ein: ")
    n = int(n)
    if n == -1:
        break
    if n < -1:
        n = 0
        continue
    print("Jetzt gehts los.")
    i = n
    result = 1
    while i > 0:
        result *= i
        i -= 1
    print("Fakultaet von", n, "ist", result)
print("Ende")
```

Verwendung des Programms

Python-Interpreter

```
>>> from factorial_break import *
Gib eine Zahl ein:  6
Jetzt gehts los.
Fakultaet von 6 ist 720
Gib eine Zahl ein:  7
Jetzt gehts los.
Fakultaet von 7 ist 5040
Gib eine Zahl ein:  -2
Gib eine Zahl ein:  4
Jetzt gehts los.
Fakultaet von 4 ist 24
Gib eine Zahl ein:  -1
Ende
>>>
```

Vereinfachtes Programm

factorial.py

```
while True:
    n = input("Gib eine Zahl ein: ")
    n = int(n)
    if n == -1:
        break
    if n < -1:
        continue
    print("Jetzt gehts los.")
    i = n
    result = 1
    while i > 0:
        result *= i
        i -= 1
    print("Fakultaet von", n, "ist", result)
print("Ende")
```

Programme in der Shell starten

Anstatt eine zweite Schleife zu schreiben, könnten wir auch das Programm statt im Interpreter in der Shell aufrufen (wie wir schon gesehen haben). Hier nochmal das alte Programm:

factorial.py

```
n = input("Gib eine Zahl ein: ")
i = int(n)
result = 1
while i > 0:
    result *= i
    i -= 1
print("Fakultaet von", n, "ist", result)
```

In der Shell

Shell

```
# python3 factorial_shell.py
Gib eine Zahl ein:  5
Fakultaet von 5 ist 120
# python3 factorial_shell.py
Gib eine Zahl ein:  10
Fakultaet von 10 ist 3628800
```

Wenn wir Programme für das richtige Leben schreiben, werden diese häufig aus einer Shell aufgerufen.

18.3 Eine Funktion factorial

Zweite Art: Eine Funktion `factorial`

Anstatt eine Benutzereingabe zu verwenden, können wir auch eine Funktion schreiben, die sich bequem im Interpreter aufrufen lässt:

`factorial.py`

```
def factorial(n):
    i = n
    result = 1
    while i > 0:
        result *= i
        i -= 1
    return result
```

Verwendung der Funktion factorial

Python-Interpreter

```
>>> from factorial import *
>>> factorial(10)
3628800
>>> factorial(4)
24
```

18.4 Iteration von unten

Iteration von unten

In den bisherigen Varianten des Fakultätsprogramms wurde die Variable `i` von `n` bis 1 heruntergezählt.
Man kann sie auch von 1 bis `n` heraufzählen:
factorial.py

```
def factorial(n):
    i = 1
    result = 1
    while i <= n:
        result *= i
        i += 1
    return result
```

18.5 Endrekursion

Iteration vs. Rekursion

- Programme (Funktionen . . .), die Schleifen statt Rekursion benutzen, bezeichnet mal als iterativ.
- Rekursion: um eine Funktion für ein Argument x zu berechnen, tu so, als wäre das Resultat für "den Vorgänger" von x schon berechnet und berechne aus diesem Resultat das Endresultat ("vom Großen ins Kleine").
- Iteration: Berechne ausdrücklich erst "kleine" Ergebnisse und dann immer "größere".

Endrekursion in Python

- In der imperativen Programmierung sind sowohl iterative als auch rekursive Berechnungen üblich. Iteration ist häufig effizienter, aber Rekursion ist häufig wesentlich natürlicher und übersichtlicher.
- Das größte Problem bei rekursiven Funktionen ist der hohe Speicherbedarf, weil i.A. jeder Aufruf wieder neue Variablen anlegt. Dies ist nicht der Fall für endrekursive Funktionen.

Rekursive Fakultätsfunktion in Python

Hier ist die Definition einer rekursiven Fakultätsfunktion:
factorial.py

```
def factorial(n):
    if n <= 1:
        return 1
    else:
        intermediate = factorial(n-1)
        result = n * intermediate
        return result
```

Die Definition ist ein wenig umständlich, damit man deutlicher erkennt, dass die Funktion nicht endrekursiv ist: nach dem rekursiven Aufruf folgen noch zwei weitere Zeilen.

Endrekursive Fakultätsfunktion in Python

Hier ist die Definition einer endrekursiven Fakultätsfunktion:

factorial.py

```
def factorial(n):
    return factorial_aux(n, 1)

def factorial_aux(n, acc):
    if n == 0:
        return acc
    else:
        n1 = n - 1
        acc1 = acc * n
        return factorial_aux(n1, acc1)
```

Die Definition ist ebenso umständlich, aber man sieht deutlich, dass die Funktion endrekursiv ist: nach dem rekursiven Aufruf folgt nichts mehr!

Endrekursion in der imperativen Programmierung

- In der imperativen Programmierung gibt es eine besonders klare Definition von Endrekursion: Eine rekursive Funktion ist endrekursiv, wenn das letzte, was die Funktion tut, ist, sich selbst rekursiv aufzurufen.
- In der imperativen Programmierung würde man wohl nicht sagen (wie wir es für Scheme taten): ein endrekursives Programm ist iterativ, sondern vielleicht eher: ein endrekursives Programm ahmt eine iterative Berechnung nach.

18.6 for-Schleifen

for-Schleifen

- Neben `while`-Schleifen gibt es in imperativen Programmiersprachen normalerweise auch noch die so genannten `for`-Schleifen.
- Diese gestatten es, eine Variable von 0 bis zu einem bestimmten Wert heraufzuzählen.

Fakultätsprogramm mit for-Schleife

factorial.py

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

- range(n) ist ein so genanntes Bereichsobjekt; es stellt die Sequenz der n Zahlen von 0 bis n-1 dar.
- Die Schleife, genauer ihr einzeiliger Rumpf, wird n mal durchlaufen, wobei i nacheinander die Werte 0, 1, . . . , n-1 hat.

Syntaktischer Zucker

`for`-Schleifen sind syntaktischer Zucker. Die `for`-Schleife

`for-Schleife`

```
for i in range(n):
    Rumpf
```

kann ersetzt werden durch:

`while-Schleife`

```
i = 0
while i < n:
    Rumpf
    i += 1
```

Bemerkungen zu `for`-Schleifen

- Insbesondere in Python kommt die `for`-Schleife in der Praxis häufiger vor als die `while`-Schleife.
- Insbesondere in Python sind `for`-Schleifen viel allgemeiner als hier präsentiert:
  - Man braucht nicht bei 0 zu beginnen.
  - Man kann die Schrittweite bestimmen, insbesondere auch herunterzählen.
  - Man kann nicht nur durch einen Zahlenbereich hindurchzählen, sondern auch z.B. durch eine verlinkte Liste (sofern wir die dortige Implementierung noch erweitern), d.h., eine Variable nacheinander den Wert der Listenelemente annehmen lassen.

  Diese Verallgemeinerungen werden wir noch sehen.

18.7 Zusammenfassung

Keine Klassen

- Ob es wohl jemandem auffiel? In diesem Kapitel haben wir nicht über Klassen oder Objekte gesprochen.
- Schleifen bzw. Iteration sind typische Merkmale für imperative Programmierung.
- Zwar verwenden auch objektorientierte Programme Schleifen, aber das hat nichts mit der Objektorientierung zu tun. Daher in diesem Kapitel: keine Objekte!
- Wir sehen wieder einmal: “Objektorientiert” und “imperativ” ist kein strikter Gegensatz!
- Demnächst wollen wir aber auch für unsere objektorientierten Programme (verlinkte Listen) manche Methoden iterativ machen.

Zusammenfassung

- Schleifen sind ein wichtiges Konstrukt der imperativen Programmierung.
- Eine `while`-Schleife ist ein Block, der so oft ausgeführt wird, bis eine gegebene Bedingung verletzt ist.
- Imperative Programe müssen nicht unbedingt Funktionen enthalten. Man kann ein Programm auch so betrachten, dass die Zeilen darin nacheinander ausgeführt werden und dass man sich für die Bildschirmeingabe und -ausgabe interessiert.

Zusammenfassung II

- Als Beispiel haben wir die Fakultät betrachtet. Um diese zu berechnen, muss in einer Schleife ein Index herauf- oder heruntergezählt werden.
- Endrekursion heißt: als letztes ruft eine Funktion sich selbst auf.
- `for`-Schleifen sind syntaktischer Zucker für das Heraufzählen einer Variable.