SAGE-Tutorium 02 im SoSe 2009

Lars Fischer*

06.05.2009

Inhaltsverzeichnis

*WWW: `http://w3.countnumber.de/fischer`, EMail: vorname.nachname (bei der) uni-siegen.de

1 Worauf das Tutorium hinauflaufen soll

1.1 Bachelors

Für einen Schein: Erstellung und Präsentation eines kleinen Softwareprojektes.

1.2 Lehrer

Am Ende des Semesters: Erstellung von (unterrichtsrelevanten) Worksheets in Kleingruppen.

- Themenfindung: Was eignet sich überhaupt für den Computereinsatz (SAGE vs. Interaktive Geometrie Software).
- Präsentation in der letzten Sitzung.
- Diskussion: Bringt die Darstellung mit dem Computer etwas? Kann es besser vermittelt werden, als alleine durch Tafel und Kreide?

2 Wiederholung

- Beispiele für Interaktive Worksheets
- Variablen, der Typ bestimmt die Bedeutung der Operationen
- `1..n`
- List-Comprehensions

3 Schleifen

3.1 For-Schleifen

Die allgemeine Form der For-Schleife lautet umgangssprachlich:

```
Für jedes Element aus der Liste:
    tue irgendwas
```

Die For-Schleife wird also verwendet, um über Listen zu laufen. Wir werden sie ständig verwenden.

Hier iterieren wir über Listen.

```
for n in [1..100]:
    print "%3d"%n,
    # manueller Zeilenumbruch
    if n%10 == 0:
        print

for p in prime_range(1,1000):
    print p,
```

Hier ein Beispiel zu Dictionaries:

```
D= {}

for k in [1..100]:
    D[k] =  is_prime(k)
print D

# nur über die Schlüssel iterieren:
for k in D:
    print k, D[k]

# iterieren über Schlüssel und Werte:
for k,v in D.items():
    print k, v # v statt D[k]
```

Ein weiteres Element zur Ablaufsteuerung in Schleifen sind die Schlüsselworte `break` und `continue`. Dabei bricht `break` die Bearbeitung der Schleife ab. Z.B. wenn man gefunden hat, was man wollte. Das zweite Schlüsselwort `continue` setzt die Ausführung der Schleife mit dem nächsten Element der Liste fort.

```
for k in [1..100]:
    print k
    if k == 11:
        break
print "Am Ende der Schleife."
```

```
print "Am Anfang der Schleife."
for k in [1..100]:
    if not is_prime(k): # falls nicht prim
        continue
    # dies wird nur für k ausgeführt, die prim sind
    print k
print "Am Ende der Schleife."
```

Wie das `if`, hat auch die `for` Schleife ein `else`. Die Anweisungen aus dem `else`-Block werden ausgeführt, wenn die Schleife ordnungsgemäß beendet wird, aber nicht, wenn mittels `break` abgebrochen wurde.

```
print "Leere Liste:", range(-3)
for k in range(-3):
    #break # bei range(3) probieren
    print k
else:
    print "Am Ende der Liste."
```

Eine leere Liste (`range(-3)`) wird gar nicht durchlaufen. Aber wir gelangen in den `else` Teil.

Eine nicht-leere Liste (`range(3)`) wird durchlaufen und wir gelangen in den `else` Teil, außer das break wird aktiviert.

3.2 While-Schleifen

Bei einer `for`-Schleife ist die Anzahl der Durchläufe durch die Liste und deren Elemente vorgegeben. Die `while` Schleife führt ihren Schleifenkörper solange aus, wie eine Bedingung erfüllt ist. Wenn die Bedingung niemals versagt, dann hat man eine Endlos-Schleife erzeugt.

Die Bedingung wird vor jedem Schleifendurchlauf, also auch dem allerersten geprüft.

Die `while` Schleife ist uns zum ersten Mal beim ggT begegnet:

```
a=5*23*75
b=23*11*37
while b !=0: # Anzahl der Schritte unbestimmt
    r = a% b
    a=b
```

```
    b=r
print a


while False:
    print "hier kommt man niemals an"

#while True:
#    print "Endlos-Schleife: oben unter 'Action...' 'Interrupt' wählen"
```

Zur weiteren Steuerung können wir wieder auf `break` und `continue` zurückgreifen. `break` verlässt die Schleife sofort. `continue` überspringt den Rest des Schleifen-Blocks und geht zurück zum anfänglichen Test der Bedingung.

Es gibt ebenfalls die Möglichkeit einen `else`-Block anzugeben, der ausgeführt wird, sobald wir die Schleife normal verlassen. Normal bedeutet: nicht durch ein `break`.

```
print "Anfang der Schleife."
while False:
    print "Nie erreicht"
else:
    print "Else"


print "Anfang der Schleife."

n=randint(1,100)

while not is_prime(n):
    print n
    n+=1
else:
    print "Am Ende ist n:", n


n=randint(1,100)

while len(divisors(n)) < 7:
    n+=1
    if n % 7 ==0:
        continue
    print n
```

```
    if n %13 == 0:
        break
else:
    print n, divisors(n), len(divisors(n))
```

4 Funktionen

4.1 Allgemeines

Es kommt oftmals vor, dass eine Abfolge von Befehlen als Einheit wiederverwendet werden soll. Indem wir eine Befehlsfolge als Funktion definieren, können wir die Befehle über ihren Funktionsnamen aufrufen. Die notwendige Flexibilität wird über Funktionsargumente erreicht.

Eine Funktion, die einen Wert berechnet, gibt diesen Wert per `return` zurück. Wenn die Funktion keinen Wert berechnen soll, so kann das `return` weggelassen werden.

Eine Funktiondefinition wird mittels `def` eingeleitet. Es folgt der Name der Funktion und zwischen runden Klammern () folgt eine (möglicherweise leere) Liste von Argumenten. Hinter der schließenden Klammer steht ein Doppelpunkt, dann folgt eingerückt der Funktionskörper:

```
def FunktionOhneArgumente():
    print "Hallo."

FunktionOhneArgumente() # Man beachte die Klammern!
# Der Name allein hat einen Wert und führt
# nicht zu einer Fehlermeldung.
FunktionOhneArgumente
```

4.2 Argumente

```
def FunktionMitArgument(name):
    print "Hallo", name

FunktionMitArgument( "Welt" )
# das ist jetzt ein Fehler:
FunktionMitArgument(  )
```

```
# das nicht, da der Funktionsname ein gültiger Wert ist:
FunktionMitArgument


def FunktionMit2Argumenten(Anrede, Name):
    print "Hallo", Anrede, Name
FunktionMit2Argumenten( "Herr", "Müller Lüdenscheid")

# def FunktionMit3Argumenten ( A1, A2, A3):
#     ......
```

4.2.1 Standard Argumente

Manchmal ist ein Argument nur in Ausnahmefällen erforderlich. Damit wir es nicht bei jeder Verwendung der Funktion angeben müssen (Weglassen des Wertes führt ja zu einer Fehlermeldung), können wir einen Standardwert angeben.

```
def Erhoehe( wert, increment=1):
    return wert+increment

print Erhoehe(0)    # nur ein Wert notwendig, da increment einen Standardwert hat
print Erhoehe(10,2) # hier wird ein anderer Wert angegeben
```

Damit die Zuordnung beim Aufruf eindeutig ist, müssen Standard-Argumente die letzten Argumente sein. Im Fall von mehreren Standard-Argumenten, von denen wir nur einen Wert ändern wollen, können wir Folgendes beim Aufruf machen:

```
def BspF1(wert, startwert=1, increment=1):
    return startwert + wert+ increment

print BspF1( 10)
print BspF1( 10, 0) # startwert anpassen, wegen Reihenfolge bei Definition
print BspF1( 10, increment=2) # nur increment anpassen
# beide anpassen, Reihenfolge bei namentlicher Nennung egal
print BspF1( 10, increment=2, startwert=-10)
```

Die Funktion `ndigits` verwendet z.B. Standard-Argumente. Sie berechnet die Anzahl der Stellen zu einer bestimmten Basis. Der Vorgabewert ist Basis 10, man kann aber auch nach den Stellen zu anderen Basen fragen:

```
a=100
print a.ndigits()
print a.ndigits(2)
print a.ndigits(8)
print a.ndigits(16)
```

4.3 Scope

In diesem Abschnitt lernen wir, wie SAGE mit Namenskonflikten umgeht. Also was passiert, wenn wir einen Variablennamen mehrfach verwenden.

4.3.1 Namensräume

SAGE verwaltet eine Tabelle mit Variablennamen und deren Werten. Wenn wir eine neue Variable anlegen, so wird der Namensraum erweitert. Eine Zeile `n=1` veranlasst SAGE nach `n` in der Tabelle zu suchen. Ist die Suche erfolglos, so wird die Tabelle um `n` erweitert. Ist die Suche allerdings erfolgreich, so bekommt die Variable `n` den neuen Wert 1 zugewiesen (der alte Wert wird überschrieben).

Anfänglich existiert der globale Namensraum, in dem unsere Variablen landen und der Builtin-Namensraum in dem die Python/SAGE-eigenen Namen liegen. Globale Variablen haben Vorrang von Builtins (d.h. sie stehen weiter vorne in der Tabelle und werden früher gefunden).

```
print sys # das System-Modul
dir(sys)

sys="überschrieben"
print sys # jetzt nur noch ein String
```

Eine Funktion führt eine weitere Hierarchie ein. Sie bekommt ihren eigenen lokalen Namensraum. Die lokalen Namen verdecken während der Ausführung der Funktion Namen aus dem globalen Namensraum. Nach der Ausführung der Funktion wird der lokale Namensraum gelöscht und die globalen Variablen sind wieder sichtbar (sofern sie durch lokale Variablen verdeckt waren).

Es gilt die Vorrang-Regel LGB: lokal, global, builtin.

```
v="globales v"
print "vorher:",v

def f():
    # unser globales v wird nicht angetastet
    v="lokales v"
    print "in der Funktion:", v

f()

print "nachher",v
```

In der Funktion haben wir auch Zugriff auf die globalen Namen:

```
g="globales g"
v="globales v"
print "vorher:",v,g

def f():
    # unser globales v wird nicht angetastet
    v="lokales v"
    # aber auch g ist sichtbar
    print "in der Funktion:", v,g

f()

print "nachher",v,g
```

Die Argumente werden ebenfalls in dem lokalen Namensraum angelegt, Änderungen an den Argumenten wirken sich (meistens, siehe spätere Sitzung zu Python-Stolperfallen) nicht aus, da wir mit einer Kopie arbeiten (call by value).

```
a="global"
print "vorher", a

def g(a):
    print "in der Funktion:", a

# in der Funktion hat a den Wert 7
g(7)
#das ändert aber nichts an dem globalen a
```

```
print "nach der Funktion:",a
```

Das entspricht der Auffassung, dass Funktionen als Eingabe-Schnittstelle nur ihre Argumentliste haben sollen und die Ausgabe-Schnittstelle nur durch return bereitgestellt wird.

Die Funktion sollte keine globalen Variablen ändern können, das ist Standard in SAGE/Python. Im vorvorigen Beispiel hat die Zuweisung an `v` nur eine lokale Variable angelegt und nicht die globale verändert. Trotzdem war lesender Zugriff auf das globale `g` möglich.

Manchmal kann es in Ausnahmefällen erwünscht sein, aus einer Funktion heraus globale Variablen zu verändern. Das müssen wir in SAGE mit den Schlüsselwort `global` explizit anfordern:

```
g= "Anfangswert"
print "vor der Funktion:", g

# diese Funktion hat einen Seiteneffekt:
def Haesslich():
    global g # jetzt ist g auch innerhalb von f das globale g
    g=3
    print "in der Funktion:", g

Haesslich()

print "nach der Funktion:", g
# Jetzt sind Probleme vorprogrammiert, da
# die Funktion den Wert und auch den Typ geändert hat
```

Man sagt die Funktion hat einen Seiteneffekt. Solche Seiteneffekte sind in größeren Programmen eine Fehlerquelle, die nur schwer zu entdecken ist. Deswegen sollte man damit sehr vorsichtig umgehen und den Seiteneffekt deutlich kenntlich machen (dokumentieren).

Das Ganze kann manchmal zu verwirrenden Fehlermeldungen führen, die daher kommen, dass SAGE/Python eine globale Variable ohne weiteres lesen kann, bei Zuweisung (ohne vorher global zu benutzen) eine lokale Variable anlegen will und dann den lesenden Zugriff ebenfalls auf die lokale Variable ausführen will.

```
l="global"
def h1():
    print "vorher:",l # lesend auf das globale l
    l = "lokale"
    # ab hier denkt Python l soll lokal sein und deswegen
    # macht die erste Zeile Probleme
    print "nachher:",l
h1()
print "außerhalb:",l
```

Erfolgt die Zuweisung zuerst, so ist wieder klar, was gemeint ist und das Problem entsteht nicht:

```
l="global"
def h2():
    l = "lokale"
    print "nachher:",l
h2()
print "außerhalb:",l
```

5 Rekursion

In der Informatik hat man es oft mir rekursiven Datenstrukturen zu tun (verkettete Listen, Bäume). Auf diesen Datenstrukturen bieten sich rekursive Funktionen an.

Auch in der Mathematik wimmelt es nur so von Dingen, die rekursiv (also selbstbezüglich) definiert sind. Bekannte Beispiele sind Fakultät und Fibonacci-Zahlen:

5.1 Beispiele

```
def fak(n):
    # Rekursion endet bei 1
    if n == 1:
        return 1
    else:
        # Rekursion: hier ruft fak sich selbst auf:
        return n*fak(n-1)

[ fak(k) for k in [1..10] ]
```

Hier sehen wir bereits das Grundprinzip rekursiver Funktionen: Zur Berechnung eines Wertes ruft sie sich selbst mit einem anderen Wert auf. Zu dessen Berechnung ruft sie sich mit einem noch anderen Wert auf ....

Desweiteren sind einer oder mehrere Ausgangswerte bekannt mit denen die Rekursion terminiert.

Wir müssen besonders aufpassen, das wir immer die Abbruchs-Bedingung treffen: `fak(-1)` würde bis ans Ende aller Tage laufen, wenn nicht vorher der Speicher knapp würde.

Rekursion bietet oftmals elegante Lösungen, wenn das Problem (die Definition) rekursiv ist. Hier können wir genauso elegant eine nicht-rekursive Implementierung finden:

```
# nicht rekursiver Funktion, die das Produkt der Zahlen 1..n berechnet?
def fak2(n):
    return # hier könnte Deine Idee stehen
```

Schon bei der Berechnung des ggT haben wir gesehen, dass gilt $ggT(a,b) = ggT(a, ggT(b, a \mod b))$ und $ggT(a,0) = a$. Damit können wir den ggT rekursiv berechnen. Wir haben aber bereits gesehen, dass wir den ggT per Schleife berechnen können:

```
def ggT_rek(a,b):
    if b == 0:
        return a
    else:
        return ggT_rek(b, a%b )


def ggT(a,b):
    while b >0:
        r= a %b
        a=b
        b=r
    return a


# Vergleich per Zufallszahlen
L = [ ( randint(1,1000),randint(1,1000))  for k in [1..20] ]
print L
print [ ggT_rek(T[0], T[1]) ==  ggT(T[0], T[1]) for T in L ]
```

Solche einfachen Rekursionen lassen sich oft zu einer Schleife umformen. Lösungen, die ohne Rekursion auskommen sind häufig schneller aber weniger elegant.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Hier haben wir eine doppelte Rekursion. Das Laufzeitverhalten ist schlimm und wird immer schlimmer:

```
%time
#fib(12)
#fib(20)
fib(25)
#fib(30)
```

Hier ist Rekursion elegant aber jeder kann auf dem Papier, durch Anfertigen einer Wertetabelle fib(100) bestimmen. Daran scheitert obige elegante Implementierung. Daran wird auch der Fortschritt der Computer-Technik in den nächsten Jahren nichts ändern.

Warum ist das so? Was können wir besser machen?

```
%hide
# Fibonacci mit Wertetabelle:
def fib_dic(n, D={0:0, 1:1} ):
    if D.has_key(n):
        return D[n]
    else:
        T= fib_dic(n-1, D) + fib_dic(n-2)
        D[n] = T
        return T


%time
# Jetzt ist 100 kein Problem:
fib_dic(100)
```

5.2 Allgemeines

Malt euch mal den Aufruf-Baum zu `fib(6)` auf und vergleicht mit dem von `fak(6)`. Daran seht Ihr das mehrfache Rekursion keine gute Idee ist. Wenn Sie nicht zu vermeiden ist, dann überlegt vorher, wie groß die Argumente bzw. der Aufruf-Baum wird.

6 Fragen zur Vorlesung?

7 Aufgaben

Aufgabe 1: Bestimmt zu 10 verschiedenen Werten von n die Ordnung aller Elemente in $\mathbb{Z}/n\mathbb{Z}$.

Aufgabe 2: Bestimmt zu 10 verschiedenen Primzahlen p die Ordnung aller Elemente in $\mathbb{Z}/p\mathbb{Z}$. Fällt Euch etwas auf?

Aufgabe 3: Falls Ihr für die vorigen beiden Aufgaben keine Funktion erstellt habt, so schreibt nun eine Funktion def Orders(n): .... die die Ordnungen aller Elemente in $\mathbb{Z}/n\mathbb{Z}$ ausgibt.

Aufgabe 4: Die Schüler einer Klasse sollen sich zu Gruppen gleicher Größe ordnen. Sie versuchen zuerst, sich zu Dreiergruppen zusammen zu finden, doch es bleibt ein Schüler übrig. Bei Vierergruppen bleibt ebenfalls 1 Schüler übrig. Bei Fünfergruppen klappt es endlich. Wieviele Schüler sind mindestens in der Klasse?

Hinweis: Es gibt die Funktionen `crt` und `CRT_list`. Versucht zuerst die Lösung alleine mit `crt` zu berechnen.

8 Nächstes Mal

- Allgemeines zu Boolschen Ausdrücken (Bedingungen)
- Python-Stolperfallen und typische Fehlerquellen
- Generatoren und Listen, multi-ranges
- LaTeX 2_ε-Ausdrücke in Worksheets

9 Quellcode

Das gesamte Worksheet ist als Text-Datei in dem PDF eingebettet.

- Im Acrobat-Reader lässt es sich unter dem Büroklammer-Symbol in der linken Leiste herunterladen.
- Okular zeigt es im File-Menu als Embeded Files an.
- Unter Linux kann man die Text-Datei auch mit
  pdftk Tutorium02.pdf unpack_files
  aus dem PDF herauslösen.

Anschließend lässt sich die Text-Datei mit der Upload-Funktion des SAGE-Notebooks hochladen.