SAGE-Tutorium 01 im SoSe 2009

Lars Fischer*

29.04.2009

Inhaltsverzeichnis

1 Verwaltungstechnisches **2**

2 Teaser – Interaktive Worksheets **2**

3 Vom Problem zum Programm **5**

4 Variablen **5**
4.1 Definieren 5
4.1.1 Spielregeln für Variablennamen 6
4.2 Variablen haben einen Typen 6

5 Nützliche Hinweise **7**
5.1 Wie finde ich die Funktion zu ... ? 7
5.2 Listen von Zahlen 7
5.3 List-Comprehensions 8
5.4 Summe und Produkt von Listen 9

6 Aufgaben **9**

7 Fragen zur Vorlesung? **9**

8 Vorschau auf das nächste Mal **9**

9 Quellcode **10**

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

1 Verwaltungstechnisches

- Anderer Tag anderer Zeitpunkt?
- Zusammensetzung der Gruppe: aus welcher Vorlesung, welcher Studiengang (Bachelors?)
- Name, Studiengang, Vorlesung, EMail in Liste eintragen
- Falls Ihr fragen habt, zum Tutorium oder zur Vorlesung, so stellt sie. Meistens haben auch die anderen etwas davon.

2 Teaser – Interaktive Worksheets

SAGE erlaubt es Arbeitsblätter interaktiv zu gestalten. Damit kann ein Arbeitsblatt soweit vorbereitet werden, dass ein Benutzer (z.B. Schüler oder Eltern) experimentieren kann, ohne SAGE selber zu kennen. Das Herzstück von interaktiven Arbeitsblättern ist eine Funktion, der ein `@interact` vorangestellt wird. Dadurch werden für alle Argumente Funktion Eingabefelder bereitgestellt. Wir betrachten einige Beispiele:

```
@interact
def matrix_info(A= input_grid(3, 3, default=1, label="Matrix A:",
                to_value=matrix, width=4) ):
    d= A.det()

    print "Die Determinante ist %d."%d
    print "Die Zeilenstufenform ist:"
    print A.echelon_form()
    print "Die Eigenwerte sind %s."%A.eigenvalues()
    if d != 0:
        print "Die Gram-Schmidt-Matrix von A ist:"
        G,mu = A.gram_schmidt()
        print G
```

```
@interact
def plot_f_and_derivative( f=input_box( sin(x), label="Funktion f:") ):
     r=(0,7)
     g=f.derivative()
     Pf= plot(f , r)
     Pg= plot(g , r, rgbcolor="red")
     show(Pf+Pg)
```

```
@interact
def eratosthenes( #N=100,
                  N= input_box(default=100, label="Primzahlen bis zu:", type=int),
                  Schritt=(0..100) ):

    # Teiler von N sind <= sqrt(N)
    max_PZ = ceil( int( sqrt( N )) )

    Zahlen = range( 1, N + 1 ) # 1 bis N, inklusive N
    D={}

    for z in Zahlen:
        D[z] = True
    # jetzt hat D N-mal den Wert True, für alle k in 1,...,N: D[k] = True

    if Schritt > 0:
        D[1]=False # 1 ist kein Primzahl

    # nun fangen wir an, in jedem Schritt
    # die Vielfachen k*s mit k > 1 zu streichen.
    s=1
    k=2
    while s < Schritt+1: # für Schritt == 1 wird nichts gestrichen
        # suche die nächste, noch nicht gestrichene Zahl
        while not D[k]:
            k +=1
        PZ = k
        # erhöhe k für den nächsten Durchlauf:
        k+=1

        if PZ >= max_PZ:
            print "Die Primzahlen <=%d sind vollständig gesiebt!"%N
            Schritt = max_PZ
            break
        print "Schritt %d: Vielfache von %d wurden gestrichen."%(s,PZ)

        # wir haben eine PZ=k und D[k] ist nicht gestrichen
        # und PZ ist das s.-te Element, welches nicht gestrichen ist
        # nun werden die Vielfachen t*PZ mit t > 1 gestrichen:
        for t in range( 2*PZ, N+1, PZ ):  # Schrittweite PZ, PZ=1*PZ selber
                                          # bleibt erhalten
            D[t] = False

        # erhöhe s für den nächsten Schritt
```

```
        s+=1

    # Ausgabe
    print
    for z in Zahlen:
        if D[z]:
            print "% 4d"%(z),
        else:
            print "    ",
        if z%10==0:
            print
```

Eine Lösung der dritten Aufgaben von Blatt 1 ist mit viel weniger Zeilen möglich, oben lege ich nur Wert auf die Interaktivität. In obigem Beispiel kann man jeden Zwischenschritt verfolgen.

Ein viel kürzeres Sieb:

```
def primes_up_to(n):
    P= range(2,n+1)
    k= 0
    while P[k] < ceil(sqrt(n)): # P[k]^2 < n # falls ohne sqrt()
        p = P[k]
        for i in range(k+1,len(P)):
            if p.divides(P[i]):
                del P[i]
        k+=1
```

Um einen Eindruck zu bekommen, was mit den interaktiven Arbeitsblättern möglich ist, könnt Ihr die Seite `http://www.math.usm.edu/sage/` besuchen. Klickt links auf Calc I und dann auf die Vorschau-Bilder unter SAGElets. Eine wachsende Sammlung von interaktiven Arbeitsblättern findet sich auch unter `http://wiki.sagemath.org/interact`.

Ein Ziel dieser Veranstaltung ist es, Euch zu befähigen, solche Arbeitsblätter zu erstellen und Euren Unterricht damit zu bereichern.[1]

[1] Ich hoffe dieser Teaser ist nicht zu einem Taser geworden.

3 Vom Problem zum Programm

1. Zuerst sollte man sich über das Problem (die Aufgabenstellung) im Klaren sein:
   - Habe ich die Aufgabenstellung richtig verstanden?
   - Kann ich die Aufgabenstellung mit meinen eigenen Worten einem Mitglied meiner Gruppe erklären?
   - Haben alle in der Gruppe die Aufgabe so verstanden?
2. Gibt es Sonderfälle, welche Eingabeparameter sind zugelassen?
3. Kenne ich ein Beispiel/kann ich ein Beispiel finden, an dem ich mein späteres Programm testen kann? (**Dieser Schritt ist wichtig.**)
4. Welche Schritte sind notwendig, um das Beispiel zu lösen?
5. Diese Schritte aufschreiben (zuerst mit Bleistift auf Papier) als Pseudocode (siehe `http://de.wikipedia.org/wiki/Pseudocode`).
6. Das Beispiel an dem Pseudocode durchspielen.
7. Erst jetzt den Pseudocode in ein SAGE-Programm umformen.
8. Testen, ob das Programm bei dem Beispiel auch die richtige Lösung liefert.
9. Falls nein, dann Fehlersuche.

4 Variablen

Variablen sind Namen für Werte. Die Werte können Zahlen, Zeichenketten, Listen, etc. sein. Unter dem Namen können wir Zwischenergebnisse festhalten und später wiederverwenden. Stellt Euch vor wir müssten überall $3.14159265358979323846264338 3\ldots$ statt π in einem mathematischen Text schreiben.

4.1 Definieren

Vor der Benutzung muss eine Variable angelegt werden. Das geschieht durch die Zuweisung eines Wertes an einen (Variablen-)Namen. Danach existiert dieser Namen. Vorher existiert der Name nicht:

```
print IchExistiereNochNicht # das führt zu einem Fehler


IchExistiereNochNicht="Jetzt schon"
# Da die Variable eben angelegt wurde,
# klappt das jetzt:
print IchExistiereNochNicht
```

4.1.1 Spielregeln für Variablennamen

- Groß- und Kleinschreibung spielt eine Rolle: `a` und `A` sind zwei unterschiedliche Namen.
- Es dürfen Buchstaben, Ziffern und Unterstrich verwendet werden.
- Das erste Zeichen darf keine Ziffer sein.
- Umlaute sind in Variablennamen nicht erlaubt.

4.2 Variablen haben einen Typen

Ein Variable hat einen Typ. Der Typ legt die Bedeutung der Operationen fest. So bedeutet z.B. `"7"+"7"` etwas anderes als `7+7`. Einmal steht `+` zwischen Zeichenketten und einmal zwischen Zahlen. Variablen verschiedenen Typs sind meistens miteinander unverträglich. Z.B. bekommt man oft ein Liste mit Zahlen als Ergebnis zurück. Wir können dann mit den Elementen der Liste rechnen, nicht aber mit der Liste selber. (Insbesondere wenn die Liste nur ein Element enthält, führt das leicht zu Verwirrungen.)

```
print "Zeichenkette:", "7"+"7"
print "auch schön:", "6"*6
print "Zahlen:", 7+7
print "aber gemischt:", 7+"7" # Fehler!
```

```
L=prime_divisors(13)
print L
print type(L), type(L[0])
```

```
print L
print L*2 # Jetzt sind Fehler vorprogrammiert
print L[0] *2
```

(Addition und Multiplikation auf Listen bewirkt das gleiche, wie bei Zeichenketten.)

Siehe auch das zweite Beispiel zu den interaktiven Arbeitsblättern, in dessen letzter Zeile zwei Plots addiert wurden, um gemeinsam (via `show`) angezeigt zu werden.

Die Operationen, die eine Variable abhängig von ihrem Typ zulässt erfahren wir so:

```
dir(L)
L. # hier auf Tab drücken


S="Hallo"
dir(S)
S.# hier auf Tab drücken


D={}
dir(D)
D.# hier auf Tab drücken
```

5 Nützliche Hinweise

5.1 Wie finde ich die Funktion zu ... ?

Wenn wir eine Funktion suchen, deren Namen wir nicht kennen aber wir vermuten, dass der Funktionsname ein bestimmtes Stichwort enthält, so bietet sich die Funktion `seach_def` an. Das Ergebnis listet alle Dateien auf, in denen eine Funktion mit dem Stichwort definiert wird. Angenommen wir suchen eine Funktion zum erweiterten euklidischen Algorithmus, so erwarten wir, dass sie gcd im Namen enthält. `gcd` und Tab-Ergänzung führt hier nicht weiter, da nur Funktionen gefunden werden, die mit gcd beginnen.

```
print search_def("gcd", interact=False )
```

Hier führt uns gleich die dritte Zeile auf die Spur der Funktion `xgcd` und ein `xgcd?` bestätigt unseren Verdacht. (Ohne interact=False werden nur die Dateien angezeigt.)

5.2 Listen von Zahlen

SAGE unterstützt (im Gegensatz zu Python) eine einfachere Schreibweise für die Liste der Zahlen von 1 bis n (bzw. von a bis b) . Im Crashkurs hatte ich `range(1,n+1)` vorgestellt. Die andere Möglichkeit ist `[1..n]` (diesmal inklusive n). Daneben gibt es die Schreibweise `(1..n)`. Diese erzeugt ebenfalls die Zahlen von 1 bis n inklusive der beiden Intervallenden. Die letzte Schreibweise ist **nicht** das offene Intervall. Der Unterschied zwischen den beiden Schreibweisen ist nur technischer Natur. Die runden Klammern sind für große Zahlenlisten zu bevorzugen.

```
L1=[1..5]; print L1
L2=range(1,6); print L2
print L1 == L2
print [1..10] == range(1,10) # rechts fehlt ja die 10
print [10..15]
print [10,15]
print [10,..,15]  # das geht!
print [10,...,15] # das geht leider nicht: ... hat eine eigene
                  # Bedeutung in Python
```

5.3 List-Comprehensions

Wir werden es oft mit Mengen der Form $M_{n,k} := \left\{ l^k \mid l \in \{1, \ldots, n\} \right\}$ zu tun haben. Eine Möglichkeit ist es die Menge als leere Liste zu erzeugen und zu füllen.

```
# z.B.
n=10; k=3
L= []
for l in [1..n]: L.append(l^k)
print L
```

Anlegen und Füllen der Liste kann man aber vereinfachen. Dazu stellen SAGE/Python die List-Comprehensions zur Verfügung. In diesem Fall sieht das so aus:

```
n=10; k=3
M_nk = [ l^k for l in [1..n] ]
```

Viel näher kann man in einer Programmiersprache an die mathematische Mengen-Notation nicht kommen! Vor dem `for` steht die Beschreibung der Einträge der Liste, hinter dem `for` steht beschrieben, was `l` sein soll.

Das ganze kann man nun weiter treiben, da mehrere `for` und auch `if` zugelassen sind. (Aber ein `for` ist Pflicht.)

```
Punkte1 = [ (x,y) for x in [1..5] for y in [2..7] ]
#List-Comprehension = [ AUSDRUCK FOR dann 0 oder mehr weitere FOR oder IF ]
Punkte2 = [ (x,y) for x in [1..10] for y in [1..10] if is_odd(x) if y.is_even(y) ]
Punkte3 = [ (x,y) for x in [1..10] if is_odd(x) for y in [1..10]  if is_even(y) ]
Punkte4 = [ (x,y) for x in [1..10] for y in [1..10]  if is_odd(x) and is_even(y) ]
```

5.4 Summe und Produkt von Listen

Die Funktionen `sum` und `prod`, angewendet auf eine Liste, berechnen deren Summe bzw. deren Produkt.

```
print sum([1..100])
print prod([1..10])
print factorial(10) # 10!
```

6 Aufgaben

Aufgabe 1: In dem Wikipedia-Link `http://de.wikipedia.org/wiki/Pseudocode` findet Ihr am Ende den Pseudocode für eine Variante des Euklidischen Algorithmus. Erstellt gemäß des Pseudocodes eine SAGE-Funktion.

Diskutiert jeweils mit Eurem Nachbarn, welche Schritte (Pseudocode) und Funktionen zur Lösung der nächsten Aufgaben notwendig sind:

Aufgabe 2: Bestimme die Anzahl der Primzahlen zwischen 1000 und 10000.

Aufgabe 3: Bestimme die Summe der Primzahlen zwischen 1000 und 10000.

Aufgabe 4: Bestimme die Summe aller Zahlen, die durch 7 teilbar sind, im Bereich von 1 bis 5000. Benutzt einmal eine List-Comprehension und einmal eine for-Schleife.

7 Fragen zur Vorlesung?

8 Vorschau auf das nächste Mal

- Schleifen
- Funktionen und Sichtbarkeitsbereich von Variablen (Scope)
- Rekursion

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 Tutorium01.pdf unpack_files
  aus dem PDF herauslösen.

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