SAGE-Crashkurs SoSe 2009

Lars Fischer (lars . fischer (an der) uni - siegen . de)

Diese Zusammenstellung erläutert die wichtigsten Dinge, um SAGE in der Zahlentheorie-Vorlesung zum Lösen der ersten Übungsblätter zu benutzen.

Dieses Dokument gibt es in zwei Varianten einmal als statisches PDF-Dokument und einmal als interaktives SAGE-Worksheet. Wenn Ihr die Worksheet-Variante aus dem Public-Bereich des Sage-Server http://sage.mathematik.uni-siegen.de:8000/pub geöffnet habt, so könnt Ihr oben links auf *Log in to edit a copy* klicken. Dann habt Ihr eine eigene Kopie dieses Dokumentes und Ihr könnt die Erläuterungen direkt ausprobieren: ändert den Text in den Zellen und klickt auf den *evaluate* links unterhalb der ausgwählten Zelle.

1 Grundlagen

1.1 Anlegen eines Accounts auf dem SAGE-Server

Zuerst müsst Ihr eine Zugang zu dem Siegener-Sage-Server (http://sage.mathematik.uni-siegen.de:8000) anlegen. Das ist aber ganz einfach über den Link *Sign up for a new Sage Notebook account* möglich.

1.2 Einfache Rechnungen

Ihr könnt SAGE wie einen Taschenrechner verwenden. Es stehen die üblichen Operationen `+ - * /` sowie `^` zur Verfügung:

```
a=2*3
b=4+5
c=1+2*3
d=(1+2)*3
e=2^4
a,b,c,d,e
```

Wenn eine Zelle mehrere Zeilen enthält, so wird nur das Ergebnis der letzten Zeile angezeigt. Deswegen gibt es den `print` Befehl, der einfach sein Argument ausgibt:

```
print "Hallo Welt"
print 3+4
7-2
```

SAGE bzw. Python unterscheidet zwischen Zahlen und Zeichenketten. Bei Zeichenketten sind Anführungszeichen (" oder ') erforderlich. Wir brauchen nämlich einen Mechanismus, der die Variable `a` mit dem Wert 6 von dem Zeichen `"a"` unterscheidet.

```
print a
print "a"
```

Neben den üblichen Rechenoperationen gibt es in SAGE noch den *Rest bei Division*, der durch das Zeichen `%` berechnet wird, sowie die *Division mit Rest*, die durch das Zeichen `//` berechnet wird:

```
print "Division",7/5
print "Division mit Rest:", 7//5
r= 7% 5
print "Rest bei Division:", r
print (7//5) * 5 + r
```

2 Teilbarkeit und Primzahlen

Eine Zahl a ist durch eine Zahl b teilbar, wenn sie den Rest 0 bei Division durch b lässt (i.a.W. es gibt ein x mit $a = x \cdot b + 0$):

```
a=24
b=10
print a%b #Rest ungleich 0
b=12
print a%b #Rest 0
```

Das Zeichen # leitet einen Kommentar bis zum Zeilenende ein. Ein solcher Kommentar kann als Hinweis z.B. für den Korrekteur oder für einen selbst dienen.

In der Zahlentheorie spielen die Teiler und die Faktorisierung einer Zahl, sowie die Primzahlen eine besondere Rolle. Es gibt in SAGE Funktionen, die uns bei der Untersuchung dieser Objekte behilflich sind:

```
n=130
D=divisors(n) # die Liste der Teiler von n
print "Teiler:", D
for d in D:   # d nimmt nacheinander die Werte in D an
    print d,"\t", is_prime(d)
```

Die Funktion `divisors` berechnet eine Liste der Teiler der Zahl, die Funktion `is_prime` ist `True` (wahr), falls das Argument eine Primzahl ist, andernfalls ist der Wert `False` (falsch).

Die Zeile `for d in D:` oder eine entsprechende Konstruktion werden wir sehr häufig benötigen, um über die Elemente einer Liste zu laufen. Die Variable `d` nimmt nacheinander jeden Wert der Liste `D` an. Im Schleifenkörper, der durch die Einrückung kenntlich gemacht wird, können wir `d` benutzen. Oben haben wir z.B. geprüft, welche Teiler von `n` Primzahlen sind. Zu beachten sind der Doppelpunkt am Ende `for` Zeile, sowie die Einrückung des Schleifenkörpers.

Neben den Teilern werden wir auch häufig die Faktorisierung brauchen, dafür gibt es in SAGE die Funktion `factor`:

```
n=130
F= factor(n)
print "Faktorisierung von",n,":",F
for f in F:
    print f # Primfaktor UND Vielfachheit

F= factor(100)
print "Faktorisierung von 100:", F
for f in F:
    print f # Primfaktor UND Vielfachheit
```

2.1 Tab-Ergänzung und Hilfe

Eine sehr nützliche Funktion von SAGE ist die automatische Ergänzung von Befehlen. Wenn wir wissen wollen, welche Funktionen mit *prime* beginnen, so geben wir diese

Buchstaben in eine Zelle ein und drücken dann auf Tabulator, dadurch wird eine Liste mit möglichen Ergänzungen angezeigt:

```
prime # hinter dem e auf die Tabulator-Taste drücken
```

Eine weitere nützliche Einrichtung von SAGE ist die Funktionsdokumentation. Wenn wir z.B. erfahren wollen, was die Funktion `prime_range` bewirkt, so liefert `prime_range?` die Antwort. Insbesondere der EXAMPLES Abschnitt zeigt die typische Verwendung der Funktion. Die Variante mit zwei Fragezeichen liefert sogar die Implementierung der Funktion.

```
prime_range?
prime_range??
```

Oben rechts gelang Ihr über *Help* an die umfangreiche Online-Hilfe. Das *Reference Manual* ist im Inhaltsverzeichnis nach Kategorien geordnet und enthält einen alphabetischen Index.

2.2 Primzahlen und ggT

Mit `prime_range` sind wir in der Lage über die Primzahlen in einem Bereich zu iterieren (`for p in prime_range(100,1000): ...`). Desweiteren gibt die Funktion `primes_first_n` Zugriff auf die ersten n Primzahlen.

```
print "Primzahlen zwischen 20 und 50:"
for p in prime_range(20,50):
    print p,
print

print "Die ersten 25 Primzahlen:"
for p in primes_first_n(25):
    print p,
```

Die Funktion `prime_divisors` listet die Primteiler einer Zahl auf (`divisors` listet alle Teiler auf).

```
print "Alle Teiler:", divisors(130)
print "Nur Primteiler:", prime_divisors(130)
```

Wir hatten oben schon die Division mit Rest gesehen. In diesem Zusammenhang wird in der Vorlesung auch der größte gemeinsame Teiler und der Satz von Bézout behandelt. In SAGE berechnen wir den größten gemeinsamen Teiler mittels der Funktion `gcd`. Und die Aussage des Satzes von Bézout wird durch den erweiterten euklidischen Algorithmus bzw. die Funktion `xgcd` repräsentiert:

```
#ggT von 91 und 119
print "Der ggT ist", gcd(91,119)

# erweiterter eukl. Alg.: finde eine Darstellung x*91+y*119 = 7
lsg= xgcd(91, 119)
print "Die Lösung ist:", lsg
print lsg[0] == lsg[1] * 91 + lsg[2] * 119
```

2.3 Einschub über Listen

An dieser Stelle bietet sich ein kleiner Einschub über Listen an. Wir haben bereits einige Funktionen gesehen, die Listen zurückgeben (`divisors, prime_range, xgcd`). Eine Liste `L` ist eine Sammlung von Elementen einer bestimmten Länge. Die Länge einer Liste lässt sich mit `len` bestimmen. Auf die Elemente einer Liste `L` mit `n` Elementen greifen `L[0]` bis `L[n-1]` zu.

Alternativ bietet sich die Verwendung einer `for` Schleife an, die wir schon ein paar mal gesehen haben.

In SAGE ist auch eine Zeichenkette eine Liste von Buchstaben. Im Folgenden sehen wir einigen typische Operationen auf Listen:

```
S="Zahlentheorie"
print S, "hat die Länge", len(S)

print S[0], S[12], S[5]

# Bereichsauswahl:
print S[6:13]
# negative Indizes:
print S[-1],S[-8], S[-13]

for s in S:
   print s,
print

for i in range(len(S)):
```

```
   print i, S[i],
print

# wir drehen die Reihenfolge um
for i in range(len(S)):  # i läuft von 0 bis 12
    print S[-i-1],       # wir brauchen -1 bis -13
                         # oder i-13 oder 13-i
```

Hier liefert `range` die Liste der Zahlen von 0 bis (Argument -1).

2.4 Kongruenzrechnung

In der Vorlesung werden sehr bald die Restklassenringe $\mathbb{Z}/n\mathbb{Z}$ drankommen. Dabei werden Zahlen "bis auf den Rest bei Division durch n"betrachtet. Diese Menge hat die Struktur eines Ringes. D.h. man kann in diesen Restklassenringen Addieren und Multiplizieren.

Vorausgreifend sei hier ein Beispiel angegeben:

```
N=13
R=IntegerModRing(N)
print R

for r in R:
    print r,
print

print "12 + 1=",R(12)+ R(1)
print
print "r+R(13):"
for r in R:
    print r + R(13),
print

print "Ordnung von R(5)", R(5).order()
print "multiplikative Ordnung von R(7)", R(7).multiplicative_order()
print R(7)^-1
```

3 SAGE Programmierung

3.1 Dictionaries

Neben den Listen gibt es in SAGE einen weiteren nützlichen Datentyp. Das sind die Dictionaries (deutsch assoziatives Array). Im mathematischen Sinne sind sie eine Abbildung, die Schlüsseln aus einer Schlüsselmenge Werte zuordnet.

```
DLeer= {} # leeres dictionary
DMuster= { "Name": "Erika Mustermann",
           "Strasse": "Heidestraße 17",
           "Ort": "Köln"
           }

for k in DMuster:
     print "Unter dem Schlüssel '" + k + "' wurde '"+ DMuster[k] + "' gespeichert"

D={}
for i in srange(8,13):  # srange läuft über SAGE-Ganz-Zahlen, range
                        # nur über Python-Ganz-Zahlen,
                        # SAGE-Ganz-Zahlen haben mehr Funktionen
    D[i] = i.ndigits()
print D
```

3.2 Verzweigungen

Mit einer Verzweigung können wir unterschiedlich auf verschiedene Situationen reagieren. Die allgemeinste Form der Verzweigung ist:

```
if BEDINGUNG_1:
    ANWEISUNGEN_1
elif BEDINGUNG_2:
    ANWEISUNGEN_2
...
elif BEDINGUNG_n:
    ANWEISUNGEN_n
else:
    ANWEISUNGEN_ELSE
```

Die `elif` und `else` Teile können entfallen. Es gilt wieder, dass die Einrückung und die Doppelpunkte zu beachten sind. Wir betrachten einige Beispiele:

```
n=-3
if is_even(n):
    print "n ist gerade"


if is_even(n):
    print "n ist gerade"
else:
    print "n ist ungerade"


if n<0:
   print "n ist negativ"
elif n>0:
    print "n ist positiv"
else:
    print "n ist Null"
```

3.3 Schleifen

Neben der `for` Schleife, die wir bereits gesehen haben, gibt es noch die `while` Schleife. Sie führt ihren Schleifenkörper solange aus, wie die Bedingung erfüllt ist:

```
L= primes_first_n(100)

i=0
while L[i] < 100:
    print i,":",L[i]
    i+=1
```

Im obigen Beispiel geben wir von den ersten 100 Primzahlen nur diejenigen aus, die kleiner als 100 sind.

Im nächsten Beispiel wird die Summe der Primzahlen kleiner 1000 berechnet:

```
B=1000
p=2
s=0
while p < B:
   s+=p
```

```
   p=next_prime(p)
```

(Das Ergebnis können wir auch eleganter mit `sum(prime_range(1000))` erhalten.)

3.4 Funktionen

Das letzte Beispiel ist schon fast der Körper einer *Summe der Primzahlen kleiner als eine Zahl* Funktion. Eine Funktion ist ein Name für eine Anweisungsfolge, damit wir diese Anweisungen später erneut aufrufen können. Damit das Ganze flexibel bleibt, wird die Anweisungsfolge parametrisiert. In obigem Beispiel bietet sich B als Parameter an.

```
def SumOfPrimesLT(B):
    """Summe der Primzahlen kleiner als B.

    INPUT:
        B - positive ganze Zahl
    EXAMPLES:
        sage: SumOfPrimesLT(100)
        1060
        sage: SumOfPrimesLT(1000)
        76127
        """
    p=2
    s=0
    while p < B:       # solange p < B
       s+=p            # Aufsummieren
       p=next_prime(p) # nächste Primzahl

    return s           # Ergebnis zuückgeben
```

Hier wird eine Funktion mit Namen `SumOfPrimesLT` und dem Argument `B` definiert. Der Funktionskörper ist wieder eingerückt. Der Schleifenkörper innerhalb der Funktion ist eine weitere Stufe eingerückt. Die erste Zeichenkette ist der Text, den wir per `SumOfPrimesLT?` abrufen können.

Wenn wir die Funktion mit `SumOfPrimesLT(100)` aufrufen, so enthält im Funktionskörper das Argument `B` den Wert 100. Wenn das Ergebnis der Berechnung vorliegt, so wird es per `return` an den Aufrufer zurückgegeben. Die Funktion wird verlassen, sobald die Ausführung auf eine `return` Anweisung trifft. Gelangt die Ausführung ans Ende der Funktion (ohne vorher auf ein `return` zu treffen), so endet die Funktion dort.

Hier folgt ein Beispiel mit zwei Argumenten. Wir hatten oben gesehen, dass wir durch die Division mit Rest entschieden können, wann eine Zahl eine andere teilt. Damit können wir eine Funktion `is_divisible` schreiben:

```
def is_divisible(a,b):
    """Testet, ob a durch b teilbar ist.

    INPUT:
        a,b - ganze Zahlen
    EXAMPLES_
        sage: is_divisible(5,3)
        False
        sage: is_divisible(100,10)
        True
    """
    # lange Rede kurzer Sinn:
    return (a % b) == 0
```

(`is_divisible` ist hier nur als Beispiel gedacht, in SAGE funktioniert die viel elegantere Schreibweise `10.divides(100)`.)

```
def PolyZuNullstellenListe(L):
    p=1
    for x_i in L:
        p*=(x-x_i)
    for x_i in L:
        if p(x_i) != 0:
            print "Fehler bei %d"%x_i
    return p

    p=PolyZuNullstellenListe([ 1, 2 ,3, 0])
```

Anschließend können wir mit `p(3)` oder `p(4)` die Werte des Polynoms an verschiedenen Stellen berechnen.

4 Interaktive Arbeitsblätter

SAGE erlaubt es Arbeitsblätter interaktiv zu gestalten. Damit kann ein Arbeitsblatt soweit vorbereitet werden, dass ein Benutzer experimentieren kann, ohne SAGE zu kennen.

```
@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)
```

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

5 Letzte Worte

Abschließend möchte ich noch auf die SAGE-Homepage http://www.sagemath.org hinweisen. Dort findet Ihr ein Tutorial, sowie viele Beispiele im Wiki. Daneben gibt es sogar unter Stichwort Screencasts einige Videos rund um SAGE. Ihr findet ebenfalls einige Links zur Programmierung in Python.

Sehr nützlich sind auch die Sage Quick Reference Cards, die es unter http://wiki.sagemath.org/quickref gibt.

Ansonsten startet nächste Woche hier im CIP-Pool ein Tutorial rund um SAGE und Zahlentheorie im Allgemeinen.