SAGE-Tutorium 03 im SoSe 2009

Lars Fischer*

13.05.2009

Inhaltsverzeichnis

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

1 Wiederholung

- Schleifen
- Funktionen und Namensräume
- Rekursion Teil 1

Letzte Sitzung kam die Frage auf, ob man alle benutzerdefinierten Variablen löschen kann. Das geht mit der `reset()` Funktion.

```
reset?
```

2 Rekursion Teil 2

Eine anschauliche Beschreibung für Rekursion lautet ungefähr so:

Rekursion: Um Rekursion zu verstehen, müsst Ihr den Begriff **Rekursion** nachschlagen.

2.1 Stolperfallen bei Rekursion

- vergessene oder nicht erreichte Abbruchbedingung
- Overhead durch Funktionsaufrufe, die bei einer iterativen Lösung entfallen
- exponentielle Laufzeit bei mehrfachen rekursiven Aufrufen

Merksatz: »Rekursiv geht schief!«

2.2 Fraktale Anwendungen

Eine Reihe von Fraktalen werden rekursiv berechnet, siehe `http://de.wikipedia.org/wiki/Fraktal`.

Hier zwei Beispiele:

```
def PythagorasBaum( N=5 ):
    global G, factor, R
    alpha = pi/6
    G= Graphics()

    # Faktor für die Skalierung:
    factor = RR( cos( alpha ) )
    # Drehmatix zu dem Winkel alpha:
    R=matrix(RR, [[ cos(alpha), -sin(alpha)], [sin(alpha), cos(alpha) ] ] )

    def zeichneSchritt( schritt, P1, P2):
        global G

        # Quadrat über P1, P2:
        deltaX= RR(P2[0] - P1[0])
        deltaY= RR(P2[1] - P1[1])
        P3 = [RR(P2[0] - deltaY), RR(P2[1] + deltaX) ]
        P4 = [RR(P1[0] - deltaY), RR(P1[1] + deltaX) ]

        if schritt >= N: # am Ende ein gefülltes Quadrat
            G += polygon( [ P1, P2, P3, P4 ], rgbcolor=(1,0,0) )
            return
        else: # sonst ein nur der Rahmen des Quadrates
            G += line( [ P1, P2, P3, P4,P1 ], rgbcolor=(0,1,0) )


        # Für das Dreieck:
        #   Die Strecke P4,P3 um alpha drehen und skalieren:
        #   dazu den Vektor v = P3-P4 mit R rotieren
        v= vector(RR, [P3[0]-P4[0], P3[1]-P4[1] ])
        #   Dann erhalten wir P5= P4 + v*R*factor.
        P5= vector(RR, P4) + R * v * factor

        # Falls die Dreiecke eine Farbe bekommen sollen
        #G += polygon( [P4,P3,P5], rgbcolor=(0,0,1) )

        # Weitere Quadrate über den Katheten:
        zeichneSchritt(schritt+1, P4, P5 )
        zeichneSchritt(schritt+1, P5, P3 )

    zeichneSchritt( 0, [0,0], [1,0] )
    G.show( axes= False , aspect_ratio=1)
    #print G
```

```
PythagorasBaum(1) # hier bis ca. 10 gehen



%time

def SierpinskiDreieck(N=5):
    global G
    G= Graphics()
    factor= RR(sqrt(3)/2)

    def zeichneDreieckeDerStufe(n, laenge, P ):
        global G
        lh= RR(laenge/2)
        if n >= N:
            G+= polygon([P, [P[0]+laenge, P[1]], [P[0]+lh, P[1]+factor*laenge] ],
                        rgbcolor= (0,0,0) )
            return
        else:
            zeichneDreieckeDerStufe(n+1, lh, P)
            zeichneDreieckeDerStufe(n+1, lh, [ P[0]+lh, P[1] ])
            zeichneDreieckeDerStufe(n+1, lh, [ RR(P[0]+lh/2), RR(P[1]+factor*lh) ])
            return

    zeichneDreieckeDerStufe( 1, 1, [0,0])
    G.show(axes=False)
    #print G

SierpinskiDreieck(1) # hier bis ca. 8 oder 9 gehen



@interact
def drawFractal( Fraktal= [ "Pythagoras Baum", "Sierpinski Dreieck" ],
                 Stufe= range(1,10) ):
    if Fraktal == "Sierpinski Dreieck":
        SierpinskiDreieck(Stufe )
    else:
        PythagorasBaum(Stufe )
```

Technische Hinweise:

- Die inneren Funktionen haben keinen inneren Namensraum innerhalb des äußeren. Sie haben keinen Zugriff auf die Variablen der äußeren Funktionen, deswegen jeweils das `global G`.
- Das `RR` bewirkt die Umwandlung der Zahlen in endliche Fließkommazahlen. Ohne `RR` würden es symbolische Variablen sein (z.B. $\cos(\pi/6) = \frac{\sqrt{3}}{2}$). Das Rechnen damit dauert wesentlich länger.

2.3 Eine erstaunliche Verbindung zwischen Sierpinski-Dreieck und Kongruenzrechnung

In diesem Abschnitt kriege ich tatsächlich die Kurve vom Sierpinski-Dreieck zur Kongruenzrechnung. Mit anderen Worten: in der Mathematik hängt alles irgendwie zusammen.

Auf dem letzten Aufgabenzettel kamen Binomialkoeffizienten vor. Diese berechnet man (wenn kein Taschenrechner oder Computer zur Verfügung steht) z.B. mit dem Pascalschen Dreieck.

```
def berechnePascalDreieck(Stufe):
    D={}
    maxlen = len(str( binomial(Stufe, int(Stufe/2) ) )) +2

    for n in [0..Stufe]:
        for l in [0..n]:
            if l == 0 or l==n:
                D[(n,l)] = 1
            else:
                # Addition der beiden Werte über der
                # aktuellen Position
                D[(n,l)] = D[(n-1,l-1)] + D[(n-1,l)]
    return D

B=berechnePascalDreieck(5)



def zeigePascalDreieck(Stufe):
    P= berechnePascalDreieck(Stufe)
    maxlen = len(str( P[(Stufe, int(Stufe/2) )] )) +3
    if is_odd(maxlen): maxlen+=1

    for n in [0..Stufe]:
        for l in [0..n]:
```

```
            if l==0: # vorne einige Leerzeichen einfügen
                # fragt nicht, wie diese Formel zustandekommt:
                spaces= int( (maxlen * (Stufe-n)/2 +1) )
                print " "*(spaces),
            print " %s"%str(P[(n,l)]).rjust(maxlen-1),
        print
zeigePascalDreieck(9)
```

Wir malen nun ein Pascalsches Dreieck und färben die Zellen ein: schwarz, falls der Eintrag ungerade und weiß falls der Eintrag gerade ist.

```
%time
BW= ( (1,1,1) , (0,0,0) )
TriCol = ( (1,0,0), (0,1,0), (0,0,1) )

def KongruenzDreieck(Stufe = 10, Farben= BW, Modul=2 ):
    N= len(Farben)

    PT= berechnePascalDreieck(Stufe)
    G= Graphics()

    boxSize= RR(2)
    y= RR(Stufe*boxSize)
    for n in [0..Stufe]:
        x= - RR(boxSize * n/2)
        for l in [0..n]:
            #print n,l
            col= Farben[ PT[(n,l)] % N ] # Rest als Index für die Farbe
            # oder falls ein anderer Modul vorgegeben wird:
            # weiß, falls es teilbar ist, sonst schwarz
            if Modul != 2:
                r = PT[(n,l)] % Modul
                if r==0:
                    col= Farben[0]
                else:
                    col= Farben[1]

            G+= polygon( [ (x,y), (x+boxSize, y), (x+boxSize, y-boxSize),
                           (x, y-boxSize) ], rgbcolor=col )
            x+= boxSize
        y-= boxSize
```

```
    G.show(axes=False, aspect_ratio=1)

KongruenzDreieck( 63, BW ) # 63 oder 127, das dauert aber
#KongruenzDreieck( 53, TriCol ) # 63 oder 127, das dauert aber
#KongruenzDreieck( 63, Modul=16) # für andere Modul sieht es ganz anders aus
```

(Zugegeben, der Zusammenhang mit der Kongruenzrechnung ist weit hergeholt.)

Wenn Ihr irgendwann mal Eure Schüler beschäftigen müsst: hier findet Ihr eine Vorlage zu Ausmalen `http://member.ycom.at/~isteiner/druckpascal.htm`.

Viele bunte Bilder zu verschiedenen Moduln findet ihr hier: `http://www.matheplanet.com/matheplanet/nuke/html/article.php?sid=785`.

3 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"
```

4 Boolesche Ausdrücke

Boolesche Ausdrücke sind Ausdrücke die wahr oder falsch, `True` oder `False` sind. Sie stehen in erster Linien bei der Eingangsbedingung einer `while` Schleife oder bei einer Verzweigung mit `if`.

Weil `True` und `False` erst spät in Python eingeführt wurden, gibt es historisch bedingt einige Konventionen, welche Werte anderen Typs sich wie `False` verhalten:

```
Tests = {"None": None, "leerer String":"", "Null":0, "Null als float": 0.0,
         "Leere Liste": [] }
for k in Tests:
    if Tests[k]:
        print k," ist auch True"
    else:
        print k," ist auch False"
```

Andere Werte dieser Typen sind `True`:

```
Tests = {"nicht-leerer String":" ", "Eins": 1, "Eins als float": 1.0,
         "nicht-leere Liste":[11] }
for k in Tests:
    if Tests[k]:
        print k," ist auch True"
    else:
        print k," ist auch False"
```

Für alle anderen Fällen müssen wir den Vergleichswert angeben:

```
a=randint(1,10)
if a == 5:
    print "zufällig 5 getroffen"
```

Hinweis: ich verwechsele gerne `==` mit `=`. Wenn ich mir angewöhnen könnte die Konstante immer auf die linke Seite zu schreiben, so würde mir SAGE/Python dabei helfen diesen Fehler zu finden:

```
a=5
if 5 == a:
    print "zufällig 5 getroffen"

#if 5 = a: # einer Konstante kann man Nichts zuweisen
#    print "zufällig 5 getroffen"
```

In einer Bedingung reicht es aus, einen booleschen Ausdruck (d.h. etwas das Wahr oder Falsch ist) anzugeben. Es ist nicht notwendig, diesen nochmals mit `==True` zu prüfen:

```
if is_prime(13) : print "13 ist prim"
if is_prime(17) == True: print "17 ist prim"
```

`is_prime( )` selbst liefert schon True oder False.

4.1 Negation

Es ist `not B` die Negation, bzw. das Gegenteil der Bedingung `B`.

```
for k in [1..10]:
    if not is_even(k):
        print k
```

```
for k in [1..10]:
    if is_even(k):
        pass # pass mach gar nichts außer der Syntax zu genügen
    else:
        print k
```

4.2 Verknüpfungen Boolescher Ausrücke

Über die Schlüsselwörter `and`, `or` und `not` lassen sich verschiedene Bedingungen miteinander verknüpfen.

```
for k in [1..10]:
    if is_even(k) or is_odd(k):
        print k
```

```
for k in [1..10]:
    if is_even(k) and is_odd(k):
        print k
```

Zu beachten ist, dass eine Kette von Oder-Bedingungen wahr ist, sobald von links ein erster wahrer Wert gefunden wird: die Auswertung wird mit True beendet. Dabei werden die rechts vom wahren Wert stehenden Ausdrücke nicht mehr ausgewertet.

Eine Kette von Und-Bedingungen kann nicht mehr wahr werden, sobald ein falscher Wert erscheint. D.h. findet Python bei der Auswertung von links einen falschen Wert, so ist die ganze Und-Kette False. Dabei werden weiter rechts stehende Werte nicht mehr ausgewertet.

5 Stolpersteine bei Python

5.1 Häufige Fehler

- inkonsistente Einrückung
- vergessener Doppelpunkt
- vergessene öffnende oder schließende Klammern `()[]{}`
- inkonsistente Klammerung

- vergessene Klammern beim Aufruf einer Funktion
- Groß- und Kleinschreibung falsch
- vergessene Anführungszeichen wo Zeichenketten in Spiel kommen
- vergessenes schließendes Anführungszeichen
- = statt == geschrieben
- vergessenes Komma in Listen
- eine Liste der Länge N hat gültige Indices von 0 bis N-1
- inkompatible Typen: Brüche, wo ganze Zahlen erforderlich sind
- ...

5.2 Richtige Fallen

Eine ausführliche Erläuterung von 10 Fallen, die sich auf den ersten Blick nicht erschließen, findet sich unter 10 Python pitfalls (`http://zephyrfalcon.org/labs/python_pitfalls.html`).

6 Fragen zur Vorlesung?

6.1 Restklassenringe

Wir haben eine Ordnung in der additiven und in der multiplikativen Gruppe.

```
N= 31 # N= 5*7
print "Phi(N):",  euler_phi(N), divisors(euler_phi(N))
R=IntegerModRing(N) # auch Integers

print "Ordnung des Ringes:", R.order()
for k in R:
    if not k.is_unit():
        print k, k.order(), "Keine Einheit"
    else:
        print k, k.order(), k.multiplicative_order()
```

Die Elemente eines IntegerModRinges sind Restklassen und entsprechend verhalten sich Addition und Multiplikation. Wir können ein einzelnes Element mit der `Mod`-Funktion anlegen und mit der `lift`-Funktion in die ganzen Zahlen zurückholen. Zum Vergleich verschiedene Rechenoperationen mit einem Restklassenelement und dessen Lift in die ganzen Zahlen:

```
r= Mod(5,13)
z = lift(r)
print r, z
print type(r), type(z)
print "Negatives:",      -r, -z
print "Addition:",       r+r+r, z+z+z
print "Multiplikation:", r*5, (z*5)
print "Inverses:",       r^-1, z^-1
```

6.2 Potenzieren

Es ist theoretisch egal, ob wir zuerst Potenzieren oder zuerst zu Restklassen übergehen, da das Ergebnis am Ende gleich ist:

```
%time
Mod(10^10000000,13)
```

```
%time
Mod(10,13)^10000000
```

Im ersten Fall wird zuerst eine große Zahl berechnet. Im zweiten Fall wird immer wieder zu Resten mod 13 übergegangen, dadurch bleiben die Zahlen immer zwischen 0 und 12. Deswegen können wir bei der zweiten Variante noch riesige Potenzen ausrechnen, die mit der ersten Variante nicht mehr zu berechnen wären. (Bei Anwendung des Satzes von Euler (Reduktion der Potenz mod $\varphi(n)$) geht es nochmal schneller.)

6.3 Ordnung und Primitivwurzeln

Bestehen da Fragen oder kann jeder etwas mit diesen Begriffen anfangen?

Wie prüft Ihr, ob eine Zahl eine Primitivwurzel ist? (Ohne Computer und Taschenrechner in einem Rechentest: Rechne geschickt!)

7 Aufgaben

Aufgabe 1: Überlegt Euch, wie die Laufzeit der `SierpinskiDreieck( N )` und `PythagorasBaum( N )` Funktionen von der Stufe `N` abhängt.[1]

Aufgabe 2: Überlegt Euch, wie man das Sierpinski-Dreieck ohne Rekursion malen kann.

Aufgabe 3: Implementiert eine Funktion zum Zeichnen der Koch-Kurve (siehe `http://de.wikipedia.org/wiki/Koch-Kurve`).

Aufgabe 4: Findet zu jeder Primzahl $p < 100$ ein Primitivwurzel ω mit $\langle\omega\rangle = \mathbb{Z}/p\mathbb{Z}$.

8 Nächstes Mal

- Generatoren und Listen, multi-ranges
- schnelle Potenzierung, schnelle Multiplikation
- quadratische Reste
- Besteht Interesse an weiteren Informationen zu LaTeX?
- OOP?

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 Tutorium03.pdf unpack_files
  aus dem PDF herauslösen.

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

[1] Spoiler: Aktiviert jeweils das `print G` am Ende der Funktion.