Sage 3: Programmieren, Funktionen, Listen

28.11.2018

Inhaltsverzeichnis

1. Programmieren in Sage
2. Rekursive Funktionen mit Gedächtnis
3. Manipulation von Listen
4. Transformationen von Listen: map und filter
5. Funktionen: Lambda
6. Listenverarbeitung: reduce
7. List Comprehension
8. Tupel
9. Listenverarbeitung: zip
10. Der Datentyp Dictionary
11. Zeitmessung

Programmieren in Sage

Grundsätzlich basiert Sage auf Python, und verwendet daher dieselbe Syntax für Schleifen, if-Abfragen, Funktionen und so weiter.

Vorsicht: Sage verwendet Python 2, nicht Python 3, es gibt daher einige Unterschiede (z.B. print-Statement).

Zusätzlich zu den vielen Bibliotheken für Mathematik bringt Sage auch einige Syntax-Erweiterungen und -Modifikationen mit (z.B. mathematische Operatoren, vereinfachte Range-Listen).

Die folgenden Inhalte sind größtenteils eine Wiederholung aus Programmieren 0:

Kontrollstrukturen

If-Statement

	If	If-Else	If-Elif-Else
In C:	<pre>if (condition) { statements }</pre>	<pre>if (condition) { statements } else</pre>	—

```
{
  statements
}
```

**In
Sage/Python:**

```
if condition:
  statements
```

```
if condition:
  statements
else:
  statements
```

```
if condition:
  statements
elif condition:
  statements
...
elif condition:
  statements
else:
  statements
```

Beispiele

In C:

```
float x = 355.0/113.0;
if (x >= 3 && x <= 5)
{
  printf("%d is between 3 and 5\n", x);
}
```

In Python/Sage:

```
x = pi
if x >= 3 and x <= 5:
  print x, "is between 3 and 5"
```

pi is between 3 and 5

oder auch:

```
if 3 <= x <= 5:
  print "{x} is between 3 and 5".format(x=x)
```

pi is between 3 and 5

Einrückung statt Klammern!

Einrückungslevel (Anzahl der Tabs/Spaces) wird durch erste Zeile nach dem if festgelegt und muss für die folgenden Zeilen innerhalb dieses if beibehalten werden.

Richtig:

```
if 3 <= x <= 5:
  sq = x^2
  print "{x} is between 9 and 25".format(x=sq)
```

pi^2 is between 9 and 25

```
if 3 <= x <= 5:
```

```
sq = x^2
print "{x} is between 9 and 25".format(x=sq)
```

pi^2 is between 9 and 25

Falsch:

```
x = 19
if 3 <= x <= 5:
    xsq = x^2
print "{x} is between 9 and 25".format(x=xsq)
```

Traceback (click to the left of this block for traceback)

```
...
IndentationError: unindent does not match any outer indentati
level
```

else und **elif** für alternative Fälle:

```
x=pi
if 3 <= x < 5:
    print "x is in [3,5)"
elif 5 <= x < 7:
    print "x is in [5,7)"
elif 7 <= x < 9:
    print "x is in [7,9)"
else:
    print "x is in (-oo,3) or in [9,+oo)"
```

x is in [3,5)

While-Statement

While

While-Else

In C:

```
while (condition)
{
    statements
}
```

—

In Sage/Python:

```
while condition:
    statements
```

```
while condition:
    statements
else:
    statements
```

Neben der normalen **while**-Schleife unterstützt Python auch eine **while-else**-Schleife: Der **while**-

Teil wird wiederholt, bis die *condition* falsch wird; danach wird einmal der **else**-Block ausgeführt. Die Ausführung findet nur statt, wenn die Schleife "regulär" beendet wird (d.h. sobald die Bedingung falsch wird), aber nicht bei vorzeitigem Abbruch der Schleife (durch **return**, **break** o.ä.).

For-Statement

	For	For-Else
In C:	<pre>for (init; condition; update) { statements }</pre>	—
In Sage/Python:	<pre>for element in sequence: statements</pre>	<pre>for element in sequence: statements else: statements</pre>

Im Unterschied zu C ist die **for**-Schleife in Python/Sage nicht definiert durch Abbruchbedingung, Startwerte und Updatefunktion, sondern iteriert immer über alle Elemente einer *sequence*. Eine *sequence* ist ein aufzählbarer (iterierbarer) Datentyp, beispielsweise eine Liste, ein String, ein Tuple, ein Generator, ...

Wie bei der **while**-Schleife wird der **else**-Block der **for**-Schleife dann ausgeführt, wenn die Schleife am Ende der Liste angekommen ist; nicht jedoch, wenn die Schleife vorzeitig (z.B. durch **break**) beendet wurde.

In C:

```
int i;
for (i = 0; i < 10; i++)
{
    printf("%d^2 = %d\n", i, i*i);
}
```

In Python/Sage wird stattdessen eine Liste der Werte angegeben, für die die Schleife ausgeführt werden soll:

```
for i in [0,1,2,3,4,5]:
    print i, "^2 = ", i^2
```

```
0 ^2 = 0
1 ^2 = 1
2 ^2 = 4
3 ^2 = 9
```

```
4 ^2 = 16
5 ^2 = 25
```

```
for letter in "alternativlos":
    if letter in "aeiou":
        print 3 * letter,
    else:
        print letter,
```

```
aaa l t eee r n aaa t iii v l ooo s
```

```
for x in [1,"b",ZZ,2,3/2]:
    print x
```

```
1
b
Integer Ring
2
3/2
```

```
for x in [1,"b",ZZ,2,3/2]:
    if x in ZZ:
        print x
```

```
1
2
```

Natürlich braucht man diese Sequence nicht immer explizit selbst händisch erzeugen, sondern viele eingebaute Funktionen liefern eine Liste oder andere solche Sequence als Ergebnis zurück:

In Sage/Python:

```
for i in range(10):
    print i, "^2 = ", i^2
```

```
0 ^2 = 0
1 ^2 = 1
2 ^2 = 4
3 ^2 = 9
4 ^2 = 16
5 ^2 = 25
6 ^2 = 36
7 ^2 = 49
8 ^2 = 64
9 ^2 = 81
```

In Sage:

```
for i in [0..9]:
    print i, "^2 = ", i^2
```

```
0 ^2 = 0
1 ^2 = 1
2 ^2 = 4
3 ^2 = 9
4 ^2 = 16
5 ^2 = 25
6 ^2 = 36
7 ^2 = 49
```

```
8 ^2 = 64
9 ^2 = 81
```

Oder mit anderer Schrittweite

```
for i in [0,2..9]:
    print i, "^2 = ", i^2
0 ^2 = 0
2 ^2 = 4
4 ^2 = 16
6 ^2 = 36
8 ^2 = 64
```

Achtung: range erzeugt den Python-Integertyp. Manche Funktionen funktionieren damit nicht!

```
for i in range(3,6):
    print parent(i)
    i.is_prime()
<type 'int'>
Traceback (click to the left of this block for traceback)
...
AttributeError: 'int' object has no attribute 'is_prime'
```

Für solche Zwecke besser srange verwenden.

```
for i in srange(3,6):
    print parent(i)
    i.is_prime()
Integer Ring
True
Integer Ring
False
Integer Ring
True
```

Im Prinzip kann über alle endlichen Kollektionen iteriert werden. Mehr dazu später unter dem Stichwort "generator".

```
for k in Integers(7):
    print k
0
1
2
3
4
5
6
```

Eine `while`-Schleife läuft, solange die Bedingung erfüllt ist.

```
k=0
while not 3<k<5:
    print "trying..."
    k=10*random()
k
```

```
trying...
trying...
trying...
4.210123048901526
```

Was ist mit Switch-Case, Goto, ...?

Switch-case kann man beispielsweise mit **elif** nachbauen.

Die wenigen erträglichen Verwendungen von `goto` werden eigentlich abgedeckt durch

- **return**: zur aufrufenden Funktion zurückkehren
- **break**: Schleife vorzeitig beenden
- **continue**: direkt zum nächsten Schleifendurchlauf
- **raise**: Exception (Fehler) melden
- **try/except**: Exception (Fehler) behandeln

Funktionen

Funktionen definieren und verwenden

In C:

```
int plus(int a, int b)
{
    return a + b;
}
```

In Sage/Python ist es nicht notwendig, Datentypen (insb. Typ des Rückgabewerts) anzugeben:

```
def plus(a, b):
    return a + b
```

Diese Funktion funktioniert also für alle Datentypen, die den `+`-Operator unterstützen, auch Strings, Listen etc:

```
print plus(4, 6)
print plus("hello", "world")
print plus(pi/2, 1/2)

10
helloworld
1/2*pi + 1/2
```

Wenn für Parameter Default-Werte angegeben werden, müssen beim Aufruf nicht alle Parameter angegeben werden:

```
def diff(a, b=1):
```

```
return a - b
```

```
diff(5)
```

```
4
```

```
diff(2,5)
```

```
-3
```

```
diff(b=2, a=5)
```

```
3
```

Rekursive Funktionen mit Gedächtnis

Wir betrachten die Folge der [Fibonaccizahlen](#), die rekursiv definiert ist:

```
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
fib(20)
```

```
10946
```

Bei dieser Definition werden für eine Auswertung unzählige Funktionsaufrufe getätigt, sodaß bald Schluß ist:

```
%time
fib(34)
```

```
9227465
```

```
CPU time: 2.35 s, Wall time: 2.35 s
```

Es ist daher wünschenswert, daß die Funktion sich "merkt", welche Werte bereits bekannt sind. Das läßt sich mit `cached_function` bewerkstelligen:

```
fib1 = cached_function(fib)
```

```
fib1(34)
```

```
9227465
```

Allerdings löst das unser Problem nicht, weil `fib` ebenfalls `fib1` aufrufen müßte, um Zugriff auf das Gedächtnis zu bekommen. Die Lösung ist der "magische" sogenannte *Dekorator* `@cached_function`:

```
@cached_function
def fib(n):
```

```

if n < 2:
    return 1
else:
    return fib(n-1) + fib(n-2)

```

Er bewirkt, daß sofort eine Kopie von `fib` mit Gedächtnis angelegt wird, die den gleichen Namen hat.

Damit geht es jetzt sehr schnell.

```

%time
fib(35)

14930352
CPU time: 0.00 s, Wall time: 0.00 s

```

Der Wertespeicher kann direkt eingesehen werden

```

fib.cache
{((12,), ()): 233, ((20,), ()): 10946, ((35,), ()): 14930352,
((17,), ()): 2584, ((5,), ()): 8, ((0,), ()): 1, ((30,), ()):
1346269, ((25,), ()): 121393, ((31,), ()): 2178309, ((13,), (
377, ((10,), ()): 89, ((18,), ()): 4181, ((3,), ()): 3, ((28,
514229, ((11,), ()): 144, ((8,), ()): 34, ((6,), ()): 13, ((3
)): 5702887, ((7,), ()): 21, ((21,), ()): 17711, ((16,), ())
((29,), ()): 832040, ((26,), ()): 196418, ((4,), ()): 5, ((34
)): 9227465, ((19,), ()): 6765, ((1,), ()): 1, ((27,), ()):
((14,), ()): 610, ((9,), ()): 55, ((15,), ()): 987, ((24,), (
75025, ((22,), ()): 28657, ((23,), ()): 46368, ((32,), ()): 3
((2,), ()): 2}

```

```

fib.cache
{((12,), ()): 233, ((20,), ()): 10946, ((35,), ()): 14930352,
((17,), ()): 2584, ((5,), ()): 8, ((0,), ()): 1, ((30,), ()):
1346269, ((25,), ()): 121393, ((31,), ()): 2178309, ((13,), (
377, ((10,), ()): 89, ((18,), ()): 4181, ((3,), ()): 3, ((28,
514229, ((11,), ()): 144, ((8,), ()): 34, ((6,), ()): 13, ((3
)): 5702887, ((7,), ()): 21, ((21,), ()): 17711, ((16,), ())
((29,), ()): 832040, ((26,), ()): 196418, ((4,), ()): 5, ((34
)): 9227465, ((19,), ()): 6765, ((1,), ()): 1, ((27,), ()):
((14,), ()): 610, ((9,), ()): 55, ((15,), ()): 987, ((24,), (
75025, ((22,), ()): 28657, ((23,), ()): 46368, ((32,), ()): 3
((2,), ()): 2}

```

```

}
```

und gelöscht werden.

```

fib.clear_cache()

```

```
fib.cache
```

```
{}
```

Allerdings muß man darauf achten, die maximale Rekursionstiefe nicht zu überschreiten:

```
fib(500)
```

```
WARNING: Output truncated!
```

```
full\_output.txt
```

```
Exception RuntimeError: 'maximum recursion depth exceeded whi
calling a Python object' in <type 'exceptions.KeyError'>
ignored
```

```
Traceback (click to the left of this block for traceback)
```

```
...
```

```
RuntimeError: maximum recursion depth exceeded
```

```
full\_output.txt
```

Das kann vermieden werden, indem man in kleinen Schritten vorgeht:

```
%time
```

```
fib(250)
```

```
12776523572924732586037033894655031898659556447352249
```

```
CPU time: 0.00 s, Wall time: 0.00 s
```

```
%time
```

```
fib(500)
```

```
2255915161619363308725126950360720720460113249137581905886388
4627738686883405015987052796968498626
```

```
CPU time: 0.00 s, Wall time: 0.00 s
```

Manipulation von Listen

```
l = range(10)
```

```
l
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Teillisten extrahieren werden.

```
l[2:7]
```

```
[2, 3, 4, 5, 6]
```

```
l[2:10:2]
```

```
[2, 4, 6, 8]
```

```
l2 = range(20,30)
```

```
l2
```

```
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

Verketten von Listen mit +

```
l + l2
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

Eine Liste ist im Wesentlichen ein Zeiger auf Speicheradressen.

```
l1 = [1,2,3]
```

Bei Zuweisung an eine andere Variable wird nur die Adresse kopiert.

```
l2 = l1
```

```
l2
```

```
[1, 2, 3]
```

```
l1 == l2
```

```
True
```

Die Listen `l1` und `l2` sind *identisch* und zeigen auf die gleichen Speicheradressen. Wenn eine der Listen verändert wird, dann auch die andere.

```
l1.append(4)
```

```
l2
```

```
[1, 2, 3, 4]
```

Das kann zu Endlosschleifen führen, wenn man nicht aufpaßt.

```
for x in l1:
    l2.append(x)
```

```
^C
```

```
Traceback (click to the left of this block for traceback)
```

```
...
```

```
__SAGE__
```

Um alle Elemente einer Liste einzeln zu kopieren, muß explizit der Befehl `copy` verwendet werden.

```
l1 = [1,2,3]
```

```
l3 = copy(l1)
```

```
l3
```

```
[1, 2, 3]
```

```
l1.append(4)
```

```
l1
```

```
[1, 2, 3, 4]
```

```
l3
```

```
[1, 2, 3]
```

Allerdings wird dabei nur die erste Ebene kopiert.

```
l4 = [l1, l1]
```

```
l4
```

```
[[1, 2, 3, 4], [1, 2, 3, 4]]
```

```
l1.append(6)
```

```
l4
```

```
[[1, 2, 3, 4, 6], [1, 2, 3, 4, 6]]
```

```
l5 = copy(l4)
```

```
l5
```

```
[[1, 2, 3, 4, 6], [1, 2, 3, 4, 6]]
```

```
l1.append(7)
```

```
l5
```

```
[[1, 2, 3, 4, 6, 7], [1, 2, 3, 4, 6, 7]]
```

Dagegen werden beim Vergleich von Listen alle Elemente rekursiv verglichen.

```
l1
```

```
[1, 2, 3, 4, 6, 7]
```

```
l6 = copy(l1)
```

```
l6
```

```
[1, 2, 3, 4, 6, 7]
```

```
l1 == l6
```

```
True
```

Transformationen von Listen: map und filter

map(f, l) erzeugt aus der Liste `l` eine neue Liste, in der alle Elemente `x` durch `f(x)` ersetzt werden.

```
l = srange(1, 15)
```

```
l
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
map(factor, l)
```

```
[1, 2, 3, 2^2, 5, 2 * 3, 7, 2^3, 3^2, 2 * 5, 11, 2^2 * 3, 13,
```

Die ursprüngliche Liste bleibt dabei unverändert.

```
l
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

filter(f, l) behält nur diejenigen Elemente x , für die $f(x)$ wahr ist.

```
filter(is_prime, l)
[2, 3, 5, 7, 11, 13]
```

Wieder bleibt die ursprüngliche Liste unverändert.

```
l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Abfrage, ob ein Element in der Liste vorkommt.

```
4 in l
True
```

Zählen, wie oft ein Element in einer Liste vorkommt.

```
l.count(4)
1
```

```
l.append(4)
l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 4]
```

```
l.count(4)
2
```

Um mit `map` arithmetische Funktionen anwenden zu können, müssen diese erst definiert werden.

```
def quadrat(x):
    return x^2
```

```
map(quadrat, l)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 169]
```

Für solche Einmalfunktionen gibt es eine elegantere Methode.

Funktionen: Lambda

Sogenannte *anonyme Funktionen*, meistens nur einmalig gebrauchte Einzeiler, können mit dem [lambda-Kalkül](#) definiert werden. `return` kann hier entfallen.

```
f = lambda x: x^2
```

```
f(2)
4
```

```
type(f)
```

```
<type 'function'>
```

Das Resultat ist also eine ganz normale Funktion, und kann als solche auch mit mehreren Inputparametern deklariert werden:

```
add = lambda x,y: x+y
```

```
add(1,1)
```

```
2
```

Man kann eine neue Funktion bilden, in der eines der Argumente gebunden wird (**Curry**), z.B. eine Funktion, die 42 addiert:

```
add42 = lambda y: add(y,42)
```

```
add42(1)
```

```
43
```

Man kann auch eine Funktion definieren, die eine Addierfunktion zurückgibt:

```
addy = lambda y: lambda x:add(x,y)
```

```
addy(42)
```

```
<function <lambda> at 0x7fb1d52b1758>
```

```
addy(42)(1)
```

```
43
```

```
type(addy)
```

```
<type 'function'>
```

```
(lambda x,y:(x(y)))(lambda u:u^2,x)
```

```
1
```

Listenverarbeitung: reduce

Um eine Funktion rekursiv mit den Elementen einer Liste zu füttern, gibt es die rekursive Funktion **reduce**. Sie ist rekursiv definiert durch $\text{reduce}(f, [x_0, x_1, \dots]) = f(x_0, \text{reduce}(f, [x_1, x_2, \dots]))$

```
l = [1,2,3,4]
```

Um z.B. die Elemente einer Liste aufzusummieren, kann man die Elemente rekursiv addieren,

```
reduce(+, l)
```

[Traceback \(click to the left of this block for traceback\)](#)

```
...
SyntaxError: invalid syntax
```

allerdings hat der Operator '+' den internen Namen `add`:

```
reduce(operator.add, l)
10
```

alternativ mit einer zweiargumentigen [anonymen Funktion](#)

```
reduce(lambda x,y: x+y, l)
10
```

```
reduce(operator.mul, l)
24
```

List Comprehension

Anstelle von `map` und `filter`, insbesondere wenn beide kombiniert werden, ist es oft eleganter und intuitiver, mit sogenannten *List Comprehensions* zu arbeiten. Anwendung einer Funktion analog zu `map`:

```
l2 = srange(1,13)
[is_prime(k) for k in l2]
[False, True, True, False, True, False, True, False, False,
 True, False]
```

```
[k^2 for k in l2]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

ergibt das gleiche wie

```
map(lambda x:x^2, l2)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

Auswahl von Elementen

```
[k for k in l2 if is_prime(k)]
[2, 3, 5, 7, 11]
```

ergibt das gleiche wie

```
filter(is_prime, l2)
[2, 3, 5, 7, 11]
CPU time: 0.00 s, Wall time: 0.00 s
```

In Kombination: Quadrate der Primzahlen

```
[k^2 for k in l2 if is_prime(k)]
```

```
[4, 9, 25, 49, 121]
```

Als Beispiel erzeugen wir eine Liste aller rationalen Zahlen mit Nenner nicht größer als 5

```
rat5 = [[m/n for m in [1..n]] for n in [1..5]]
rat5
```

```
[[1], [1/2, 1], [1/3, 2/3, 1], [1/4, 1/2, 3/4, 1], [1/5, 2/5, 3/5, 4/5, 1]]
```

```
rat5a = reduce(operator.add, rat5)
rat5a
```

```
[1, 1/2, 1, 1/3, 2/3, 1, 1/4, 1/2, 3/4, 1, 1/5, 2/5, 3/5, 4/5, 1]
```

Um doppelte Elemente hinauszuerwerfen, kann man auf den Datentyp set zurückgreifen.

```
set(rat5a)
```

```
set([1, 1/2, 1/3, 2/3, 1/4, 3/4, 1/5, 2/5, 3/5, 4/5])
```

und in eine Liste zurückverwandeln.

```
ls = list(set(rat5a))
```

```
ls.sort()
ls
```

```
[1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 1]
```

Tupel

Tupel sind unveränderliche Listen.

```
t = 1, 2, 3
t
```

```
(1, 2, 3)
```

```
t.append(5)
```

Traceback (click to the left of this block for traceback)

```
...
AttributeError: 'tuple' object has no attribute 'append'
```

```
for i in t:
    print i
```

```
1
2
3
```

Praktische Anwendung: Austausch der Werte zweier Variablen ("*swap*"):

```
x = 1
y = 2
x,y
```

```
(1,2)
```

```
x,y = y,x
```

```
x,y
```

```
(2,1)
```

Tupel können dazu verwendet werden, mehrere Werte aus einer Funktion zurückzubekommen.
Beispiel: Quotient und Rest einer ganzzahligen Division.

```
n = 17
m = 5
q,r = n.quo_rem(m)
```

```
q
```

```
3
```

```
r
```

```
2
```

Listenverarbeitung: zip

Wir wollen über zwei Listen im Reißverschlußsystem parallel iterieren, dazu müssen sie zunächst mit **zip** aneinandergeheftet werden:

```
l1 = ['a', 'b', 'c']
```

```
l2 = [1,2,3]
```

```
l12 = zip(l1,l2)
l12
```

```
[(a, 1), (b, 2), (c, 3)]
```

Wir haben jetzt eine Liste von Paaren

```
type(l12[0])
```

```
<type 'tuple'>
```

über diese Liste kann jetzt mit einem Paar von Variablen iteriert werden:

```
for x,y in l12:
    print(x+str(y))
```

```
a1
b2
c3
```

Anwendungsbeispiel: wir bilden das Polynom $\sum a_i x^i$. Zunächst erzeugen wir die benötigten Variablen, das geht elegant mit einem Formatstring analog zu print. %s steht als Platzhalter für eine String.

```
print("x%s" % 'a')
```

```
xa
```

```
var("a%s" % 1)
```

```
a1
```

```
aa = [var("a%s" % i) for i in range(10)]
aa
```

```
[a0, a1, a2, a3, a4, a5, a6, a7, a8, a9]
```

```
zip(aa, range(10))
```

```
[(a0, 0), (a1, 1), (a2, 2), (a3, 3), (a4, 4), (a5, 5), (a6, 6), (a7, 7), (a8, 8), (a9, 9)]
```

```
var('x')
```

```
x
```

```
[a*xb for a,b in zip(aa,range(10))]
```

```
[a0, a1x, a2x2, a3x3, a4x4, a5x5, a6x6, a7x7, a8x8, a9x9]
```

```
reduce(operator.add, [a*x^b for a,b in zip(aa,range(10))])
```

$$a_9x^9 + a_8x^8 + a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

Der Datentyp Dictionary

Ein *Dictionary* ist eine Datenstruktur ähnlich einer Liste, die mit beliebigen Objekten anstelle der Zahlen 0...n indiziert werden kann. Die Eingabe erfolgt mit geschwungenen Klammern.

```
d = { 'a':1, 5:3, 'c':ZZ }
d
```

```
{a : 1, c : Z, 5 : 3}
```

```
d['a']
```

```
1
```

Nicht zugewiesene Indices ergeben eine Fehlermeldung.

```
d['xy']
```

```
Traceback (click to the left of this block for traceback)
```

```
...
KeyError: 'xy'
```

können aber durch eine Zuweisung erzeugt

```
d['xy'] = 'z'
d['xy']
```

```
z
```

und danach beliebig verändert werden

```
d['xy'] = 42
d['xy']
```

```
42
```

Ein Dictionary besteht aus *Keys* und *Values*.

```
d.keys()
```

```
[a, c, xy, 5]
```

```
d.values()
```

```
[1, Z, 42, 3]
```

```
d.items()
```

```
[(a, 1), (c, Z), (xy, 42), (5, 3)]
```

Iteration über die Indices.

```
for x in d:
    print x
```

```
a
c
xy
5
```

```
for x in d:
    print d[x]
```

```
1
Integer Ring
42
3
```

Man kann sich die Lösungen von Gleichungen als Dictionary erzeugen lassen.

```
var('x,y')
solve(x^2-3 == 0, x, solution_dict=True)
```

```
[{x: -sqrt(3)}, {x: sqrt(3)}]
```

```
sol=solve([x+y == 1, x-3*y == 5], [x,y], solution_dict=True)
sol
```

```
[{y: -1, x: 2}]
```

```
sol[0][x]
```

```
2
```

Zeitmessung

Es laufen intern zwei Uhren: die *cputime* mißt die Zeit, die im Prozessor verbraucht wird, die *walltime* mißt die Zeit, die insgesamt vergeht.

```
cputime()
```

```
9.312
```

```
walltime()
```

```
1512381744.161634
```

Durch Auswerten der Differenz kann die für einen Vorgang verbrauchte Zeit gemessen werden.

```
t=cputime()  
factor(2^67-1)  
cputime()-t
```

```
0.0080000000000000895
```

Eine genaueren Wert erhält man durch Mittelung über mehrere Versuche. Dabei muß der auszuführende Befehl als String übergeben werden.

```
timeit( 'factor(2^67-1)' )
```

```
125 loops, best of 3: 4.05 ms per loop
```

```
t=cputime()  
for k in [2^10..2^13]:  
    factor(k)  
cputime(t)
```

```
0.096000000000000009
```

Zeilenumbrüche können mit `\n` angegeben werden.

```
timeit( 'for k in [2^10..2^13]:\n    factor(k)'
```

```
5 loops, best of 3: 74.2 ms per loop
```

```
print("a\nb")
```

```
a  
b
```