Vorlesung 04.12.2020

Inhaltsverzeichnis

Einfache Rechnungen

Sage kann alles was ein Taschenrechner kann, und das entweder mit Gleitkommazahlen beliebiger Genauigkeit oder exakt mit ganzen Zahlen, Brüchen und "formalen Ausdrücken". Die Auswertung einer Eingabe erfolgt mit `Shift+Enter`

```
In [1]: 1+1
```

```
Out[1]: 2
```

```
In [2]: 2*3
```

```
Out[2]: 6
```

```
In [3]: 2/3
```

```
Out[3]: 2/3
```

Das Ergebnis der vorhergehenden Rechnung kann mit '_' abgerufen werden.

```
In [10]: _^2
```

```
Out[10]: 4
```

Ganzzahlige Divison mit Rest

```
In [11]: 14 // 3
```

```
Out[11]: 4
```

In [12]:
```
14 % 3
```

Out[12]: 2

Ergebnisse können in Variablen abgelegt werden.

In [13]:
```
a = 1/2
```

In diesem Fall wird die Ausgabe unterdrückt. Anzeige mit **print**:

In [14]:
```
print(a)
```

```
1/2
```

oder **show**

In [15]:
```
show(a)
```

$$\frac{1}{2}$$

Sage kann mit rationalen und algebraischen Zahlen exakt rechnen. Wenn nicht explizit gewünscht, werden irrationale Zahlen nicht ausgewertet, sondern symbolisch weitergeführt.

In [16]:
```
b=sqrt(2)
show(b)
```

$$\sqrt{2}$$

In [17]:
```
b^2
```

Out[17]: 2

Intern werden Irrationalzahlen anders behandelt als Rationalzahlen oder ganze Zahlen. Sage ist wie Python *objektorientiert* und jedes Objekt gehört einem eindeutig definierten Typ an, der die notwendigen Operationen bereitstellt. Den Typ kann man sich mit **type** (zugrundeliegende python-Klasse) oder (leichter interpretierbar) **parent** anzeigen lassen:

In [18]:
```
type(a)
```

Out[18]: <type 'sage.rings.rational.Rational'>

In [19]:
```
parent(a)
```
Out[19]: Rational Field

In [20]:
```
show(parent(a))
```
$$\mathbb{Q}$$

Irrationale Zahlen werden wie **formale Ausdrücke** behandelt.

In [21]:
```
type(b)
```
Out[21]: <type 'sage.symbolic.expression.Expression'>

In [22]:
```
parent(b)
```
Out[22]: Symbolic Ring

Das heißt, b ist ein Objekt, dessen Quadrat 2 ergibt:

In [23]:
```
b^2
```
Out[23]: 2

es wird allerdings nicht automatisch versucht, den einfachsten möglichen Typ zu finden.

In [24]:
```
parent(b^2)
```
Out[24]: Symbolic Ring

Auch komplexe Zahlen sind bekannt, die Variable `I` ist vordefiniert:

In [25]:
```
sqrt(-1)
```
Out[25]: I

In [26]:
```
I^2
```
Out[26]: -1

Vorsicht, diese Variable ist nicht geschützt:

In [27]:
```
I=0
I
```

Out[27]: 0

Sie kann aber im Falle eines Falles mit `restore` wiederhergestellt werden.

In [28]:
```
restore('I')
I^2
```

Out[28]: -1

In [29]:
```
pi
```

Out[29]: pi

In [30]:
```
parent(pi)
```

Out[30]: Symbolic Ring

Gleitkommanäherungen sind in beliebiger Genauigkeit verfügbar. (siehe unten).

In [31]:
```
numerical_approx(pi)
```

Out[31]: 3.14159265358979

In [32]:
```
pi.n()
```

Out[32]: 3.14159265358979

In [33]:
```
pi.n(digits=50)
```

Out[33]: 3.1415926535897932384626433832795028841971693993751

Zu beachten ist der Unterschied zwischen **digits** (=Dezimalstellen) und **precision** (=Rechengenauigkeit, d.h., Anzahl der Stellen in Basis 2)

In [34]:
```
pi.n(prec=160)
```

Out[34]: 3.1415926535897932384626433832795028841971693994

Objekte, Klassen und Methoden.

Wie oben erwähnt, hat jedes Objekt einen Typ bzw gehört einer sogenannten **Klasse** an. Eine Klasse enthält daneben auch ein Arsenal von sogenannten **Methoden,** das sind Funktionen, die nur auf Instanzen der beinhaltenden Klasse angewendet werden können. Diese Methoden werden durch die Syntax `.methode(...)` angesprochen. Je nachdem, welche Methode angesprochen wird, kann der Typ der Ergebnisse verschieden sein (auch wenn sie am Bildschirm exakt gleich ausgegeben werden):

```
In [35]: Mod(42,9)
Out[35]: 6

In [36]: _.parent()
Out[36]: Ring of integers modulo 9

In [37]: mod(42,9)
Out[37]: 6

In [38]: _.parent()
Out[38]: Ring of integers modulo 9

In [39]: 42.mod(9)
Out[39]: 6

In [40]: _.parent()
Out[40]: Integer Ring
```

In [41]:
```
42.Mod(9)
```

```
---------------------------------------------------------------
---------------
AttributeError                            Traceback (most r
ecent call last)
<ipython-input-41-21ae263355cc> in <module>()
----> 1 Integer(42).Mod(Integer(9))

/usr/opt/Sage-8.3-amd64/local/lib/python2.7/site-packages/s
age/structure/element.pyx in sage.structure.element.Elemen
t.__getattr__ (build/cythonized/sage/structure/element.c:45
18)()
    491             AttributeError: 'LeftZeroSemigroup_with
_category.element_class' object has no attribute 'blah_blah
'
    492         """
--> 493         return self.getattr_from_category(name)
    494
    495     cdef getattr_from_category(self, name):

/usr/opt/Sage-8.3-amd64/local/lib/python2.7/site-packages/s
age/structure/element.pyx in sage.structure.element.Elemen
t.getattr_from_category (build/cythonized/sage/structure/el
ement.c:4627)()
    504         else:
    505             cls = P._abstract_element_class
--> 506         return getattr_from_other_class(self, cls,
name)
    507
    508     def __dir__(self):

/usr/opt/Sage-8.3-amd64/local/lib/python2.7/site-packages/s
age/cpython/getattr.pyx in sage.cpython.getattr.getattr_fro
m_other_class (build/cythonized/sage/cpython/getattr.c:253
5)()
    392         dummy_error_message.cls = type(self)
    393         dummy_error_message.name = name
--> 394         raise AttributeError(dummy_error_message)
    395     attribute = <object>attr
    396     # Check for a descriptor (__get__ in Python)

AttributeError: 'sage.rings.integer.Integer' object has no
attribute 'Mod'
```

Hilfe

Die zur Verfügung stehenden *Methoden* kann man sich durch **pi.** und anschließendes Drücken von **TAB** anzeigen lassen.

Information über eine Methode erhält man durch ein angehängtes Fragezeichen

In [42]:
```
pi.n?
```

Will man es noch genauer wissen, macht man zwei Fragezeichen und erhält den Quellcode der Funktion:

In [43]:
```
pi.n??
```

Gleitkommazahlen

Für Zahlen gegebener Genauigkeit gibt es einen eigenen Typ

Ohne weitere Spezifizierung wird IEEE "double precision" (=53bit) verwendet, das entspricht dezimal einer Genauigkeit von 10^{-16}

In [44]:
```
R=RealField()
R
```

Out[44]:
```
Real Field with 53 bits of precision
```

In [45]:
```
R(1)
```

Out[45]:
```
1.00000000000000
```

Eine andere Rechengenauigkeit muß explizit angegeben werden.

In [46]:
```
R100=RealField(100)
R100
```

Out[46]:
```
Real Field with 100 bits of precision
```

Zu beachten ist, daß **Precision** sich auf das intern verwendete Binärsystem bezieht. Das sind bei gleicher Genauigkeit etwa dreimal soviele Stellen wie im Dezimalsystem

In [47]:
```
R100(1)
```

Out[47]:
```
1.0000000000000000000000000000
```

Bei Rechnungen mit Daten verschiedener Genauigkeit wird am Ende die jeweils geringste verwendet, mehr kann nicht garantiert werden.

In [48]:
```
R100(1)/R100(3)
```

Out[48]:
```
0.33333333333333333333333333333
```

```
In [49]: c=R100(1)/R(3)
         c
```

```
Out[49]: 0.333333333333333
```

```
In [50]: R100(  R100(1)/R(3) )
```

```
Out[50]: 0.33333333333333331482961625625
```

```
In [51]: parent(c)
```

```
Out[51]: Real Field with 53 bits of precision
```

Die "Zufallszahlen" am Ende der Dezimalentwicklung rühren daher, daß Gleitkommazahlen intern in Binärentwicklung dargestellt werden. Für unsere Beispielzahl 1/3 ist diese auch zur Basis zwei periodisch:

```
In [52]: c.str(base=2)
```

```
Out[52]: '0.01010101010101010101010101010101010101010101010101010101'
```

Listen

Listen bilden einen wichtigen Datentyp, weil darin verschiedene Objekte gesammelt werden können. Eine Liste kann Objekte beliebiger Art enthalten.

```
In [53]: l = [1,2,3,x,"abc",ZZ,2]
         l
```

```
Out[53]: [1, 2, 3, x, 'abc', Integer Ring, 2]
```

Vorsicht! Wie in der Informatik üblich, beginnen die Indices bei 0 zu laufen!

```
In [54]: l[1]
```

```
Out[54]: 2
```

```
In [55]: len(l)
```

```
Out[55]: 7
```

```
In [56]: l[0]
```

```
Out[56]: 1
```

Mit negativen Indices kann man die Liste von hinten durchlaufen.

In [57]:
```
l[-1]
```

Out[57]: 2

In [58]:
```
l[-2]
```

Out[58]: Integer Ring

Listen können verlängert werden:

In [59]:
```
l.append(5)
```

Dabei wird die Liste direkt modifiziert:

In [60]:
```
l
```

Out[60]: [1, 2, 3, x, 'abc', Integer Ring, 2, 5]

Es können auch Elemente entfernt werden, nach Index

In [61]:
```
del l[2]
l
```

Out[61]: [1, 2, x, 'abc', Integer Ring, 2, 5]

Oder nach Wert:

In [62]:
```
l.remove(x)
l
```

Out[62]: [1, 2, 'abc', Integer Ring, 2, 5]

In letzterem Falle wird nur der erste vorgefundene Eintrag entfernt:

In [63]:
```
l.remove(2)
l
```

Out[63]: [1, 'abc', Integer Ring, 2, 5]

Ersetzen einzelner Elemente

In [64]:
```
l[2]=8
l
```

Out[64]:
```
[1, 'abc', 8, 2, 5]
```

Einfügen neuer Elemente an beliebiger Stelle:

In [65]:
```
l.insert(1,"u")
l
```

Out[65]:
```
[1, 'u', 'abc', 8, 2, 5]
```

Teillisten können wie in `matlab` extrahiert werden:

In [66]:
```
l[1:4]
```

Out[66]:
```
['u', 'abc', 8]
```

Komplexe und ganze Zahlen

Komplexe Zahlen werden durch die imaginäre Einheit `I` erzeugt.

In [67]:
```
a=pi+I
a
```

Out[67]:
```
pi + I
```

numerisch

In [68]:
```
CC(a)
```

Out[68]:
```
3.14159265358979 + 1.00000000000000*I
```

Real- und Imaginärteil

In [69]:
```
real(a)
```

Out[69]:
```
pi
```

In [70]:
```
imag(a)
```

Out[70]:
```
1
```

Einige Zahlentheoretische Funktionen. Der ggT

```
In [71]: gcd(77,335)
```

```
Out[71]: 1
```

Der erweiterte euklidische Algorithmus

```
In [72]: xgcd(77,335)
```

```
Out[72]: (1, -87, 20)
```

kgV

```
In [73]: lcm(77,335)
```

```
Out[73]: 25795
```

Primfaktorisierung

```
In [74]: factor(335)
```

```
Out[74]: 5 * 67
```

Primzahlen

```
In [75]: 77.is_prime()
```

```
Out[75]: False
```

```
In [76]: 77.next_prime()
```

```
Out[76]: 79
```

```
In [77]: 79.next_prime()
```

```
Out[77]: 83
```

Teiler

```
In [78]: divisors(77)
```

```
Out[78]: [1, 7, 11, 77]
```

Restklassenbestimmung

In [79]:
```
mod(7,5)
```

Out[79]: 2

In [80]:
```
mod(7^77, 5)
```

Out[80]: 2