Vorlesung 29.11.2024

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 [4]: _^2
```
```
Out[4]: 4/9
```

Ganzzahlige Divison mit Rest

```
In [5]: 14 // 3
```
```
Out[5]: 4
```

```
In [6]: 14 % 3
```
```
Out[6]: 2
```

Quotient und Rest auf einmal:

In [7]:
```
14.quo_rem(3)
```

Out[7]:
```
(4, 2)
```

Ergebnisse können in Variablen abgelegt werden.

In [8]:
```
a = 1/2
```

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

In [9]:
```
print(a)
```

```
1/2
```

oder **show**

In [10]:
```
show(a)
```

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

In diesem Fall wird kein Output produziert. Der letzte Output war das Paar (4,2) in Zelle [7] und das kann nicht quadriert werden:

In [11]:
```
_^2
```

```
---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-11-8e2afc3491b6> in <module>
----> 1 _**Integer(2)

/usr/opt/Sage-9.2-amd64/local/lib/python3.8/site-packages/sage/rin
gs/integer.pyx in sage.rings.integer.Integer.__pow__ (build/cython
ized/sage/rings/integer.c:15223)()
   2206             return coercion_model.bin_op(left, right, oper
ator.pow)
   2207         # left is a non-Element: do the powering with a Py
thon int
-> 2208         return left ** int(right)
   2209
   2210     cpdef _pow_(self, other):

TypeError: unsupported operand type(s) for ** or pow(): 'tuple' an
d 'int'
```

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

In [12]:
```
b=sqrt(2)
show(b)
```

$\sqrt{2}$

In [13]:
```
b^2
```

Out[13]: 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 [14]:
```
type(a)
```

Out[14]:
```
<class 'sage.rings.rational.Rational'>
```

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

Out[15]:
```
Rational Field
```

In [16]:
```
show(parent(a))
```

$\mathbf{Q}$

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

In [17]:
```
b
```

Out[17]:
```
sqrt(2)
```

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

Out[18]:
```
<class 'sage.symbolic.expression.Expression'>
```

In [19]:
```
parent(b)
```

Out[19]:
```
Symbolic Ring
```

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

In [20]:
```
b^2
```

Out[20]: 2

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

In [21]:
```
parent(b^2)
```

Out[21]:
```
Symbolic Ring
```

Der Wert kann aber in eine ganze Zahl umgewandelt werden:

```
In [22]: ZZ(b^2)
Out[22]: 2

In [23]: parent(_)
Out[23]: Integer Ring

In [24]: ZZ
Out[24]: Integer 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 [32]: numerical_approx(pi)
Out[32]: 3.14159265358979

In [33]: pi.n()
Out[33]: 3.14159265358979

In [34]: pi.n(digits=50)
Out[34]: 3.1415926535897932384626433832795028841971693993751
```

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

```
In [35]: pi.n(prec=160)
Out[35]: 3.14159265358979323846264338327950288419716939937
```

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 [36]: Mod(42,9)
Out[36]: 6

In [37]: _.parent()
Out[37]: Ring of integers modulo 9

In [38]: mod(42,9)
Out[38]: 6

In [39]: _.parent()
Out[39]: Ring of integers modulo 9

In [40]: 42.mod(9)
Out[40]: 6

In [41]: _.parent()
Out[41]: Integer Ring
```

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

```
---------------------------------------------------------------------
---------
AttributeError                            Traceback (most recent c
all last)
<ipython-input-42-21ae263355cc> in <module>
----> 1 Integer(42).Mod(Integer(9))

/usr/opt/Sage-9.2-amd64/local/lib/python3.8/site-packages/sage/str
ucture/element.pyx in sage.structure.element.Element.__getattr__
(build/cythonized/sage/structure/element.c:4703)()
    491             AttributeError: 'LeftZeroSemigroup_with_catego
ry.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-9.2-amd64/local/lib/python3.8/site-packages/sage/str
ucture/element.pyx in sage.structure.element.Element.getattr_from_
category (build/cythonized/sage/structure/element.c:4815)()
    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-9.2-amd64/local/lib/python3.8/site-packages/sage/cpy
thon/getattr.pyx in sage.cpython.getattr.getattr_from_other_class
(build/cythonized/sage/cpython/getattr.c:2619)()
    370         dummy_error_message.cls = type(self)
    371         dummy_error_message.name = name
--> 372         raise AttributeError(dummy_error_message)
    373     attribute = <object>attr
    374     # Check for a descriptor (__get__ in Python)

AttributeError: 'sage.rings.integer.Integer' object has no attribu
te '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 [44]:
```
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 [45]:
```
R=RealField()
R
```

Out[45]:
```
Real Field with 53 bits of precision
```

In [46]:
```
R(1)
```

Out[46]:
```
1.00000000000000
```

Eine andere Rechengenauigkeit muß explizit angegeben werden.

In [47]:
```
R100=RealField(100)
R100
```

Out[47]:
```
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 [48]:
```
R100(1)
```

Out[48]:
```
1.0000000000000000000000000000
```

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

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

Out[49]:
```
0.33333333333333333333333333333
```

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

Out[50]:
```
0.333333333333333
```

In [51]:
```
R100(  R100(1)/R(3) )
```

Out[51]:
```
0.33333333333333331482961625625
```

In [52]:
```
parent(c)
```

Out[52]:
```
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 [53]: c.str(base=2)
Out[53]: '0.01010101010101010101010101010101010101010101010101010101'
```

Das hat Seiteneffekte.

```
In [54]: floor(42.1)
Out[54]: 42
```

```
In [55]: floor(1/0.1)
Out[55]: 10
```

```
In [56]: floor(1/(42.1-42))
Out[56]: 9
```

Das liegt daran, daß intern im Dualsystem gerechnet wird und 0.1 in Basis 2 ein periodischer Bruch ist und Auslöschung passiert:

```
In [57]: 2.1-2
Out[57]: 0.100000000000000
```

```
In [58]: _.str(base=2)
Out[58]: '0.00011001100110011001100110011001100110011001100110100000'
```

d.h., die verschwindende 2 vor dem Komma läßt am Ende ein paar Nullen entstehen, sodaß am Ende die Näherung für 2.1-2 echt kleiner ist als diejenige für 0.1

```
In [59]: (0.1).str(base=2)
Out[59]: '0.00011001100110011001100110011001100110011001100110011010'
```

Um die Anzahl der Dezimalstellen festzulegen, `digits` verwenden:

```
In [60]: pi.n(digits=10)
Out[60]: 3.141592654
```

Listen

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

In [61]:
```
l = [1,2,3,x,"abc",ZZ,2]
l
```

Out[61]:
```
[1, 2, 3, x, 'abc', Integer Ring, 2]
```

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

In [62]:
```
l[1]
```

Out[62]:
```
2
```

In [63]:
```
len(l)
```

Out[63]:
```
7
```

In [64]:
```
l[0]
```

Out[64]:
```
1
```

Mit negativen Indices kann man die Liste von hinten durchlaufen.

In [65]:
```
l[-1]
```

Out[65]:
```
2
```

In [66]:
```
l[-2]
```

Out[66]:
```
Integer Ring
```

Listen können verlängert werden:

In [67]:
```
l.append(5)
```

Dabei wird die Liste direkt modifiziert:

In [68]:
```
l
```

Out[68]:
```
[1, 2, 3, x, 'abc', Integer Ring, 2, 5]
```

Es können auch Elemente entfernt werden, nach Index

In [69]:
```
del l[2]
l
```

Out[69]:
```
[1, 2, x, 'abc', Integer Ring, 2, 5]
```

Oder nach Wert:

In [70]:
```
l.remove(x)
l
```

Out[70]:
```
[1, 2, 'abc', Integer Ring, 2, 5]
```

In letzterem Falle wird nur der erste vorgefundene Eintrag entfernt:

In [71]:
```
l.remove(2)
l
```

Out[71]:
```
[1, 'abc', Integer Ring, 2, 5]
```

Ersetzen einzelner Elemente

In [72]:
```
l[2]=8
l
```

Out[72]:
```
[1, 'abc', 8, 2, 5]
```

Einfügen neuer Elemente an beliebiger Stelle:

In [73]:
```
l.insert(1,"u")
l
```

Out[73]:
```
[1, 'u', 'abc', 8, 2, 5]
```

Teillisten können wie in `matlab` extrahiert werden:

In [74]:
```
l[1:4]
```

Out[74]:
```
['u', 'abc', 8]
```

Komplexe und ganze Zahlen

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

In [75]:
```
a=pi+I
a
```

Out[75]:
```
pi + I
```

numerisch

```
In [76]: CC(a)
Out[76]: 3.14159265358979 + 1.00000000000000*I
```

Real- und Imaginärteil

```
In [77]: real(a)
Out[77]: pi
```

```
In [78]: imag(a)
Out[78]: 1
```

Einige Zahlentheoretische Funktionen. Der ggT

```
In [79]: gcd(77,335)
Out[79]: 1
```

Der erweiterte euklidische Algorithmus

```
In [80]: xgcd(77,335)
Out[80]: (1, -87, 20)
```

kgV

```
In [81]: lcm(77,335)
Out[81]: 25795
```

Primfaktorisierung

```
In [82]: factor(335)
Out[82]: 5 * 67
```

Primzahlen

```
In [83]: 77.is_prime()
Out[83]: False
```

In [84]:
```
77.next_prime()
```

Out[84]: 79

In [85]:
```
79.next_prime()
```

Out[85]: 83

Teiler

In [86]:
```
divisors(77)
```

Out[86]: [1, 7, 11, 77]

Restklassenbestimmung

In [87]:
```
mod(7,5)
```

Out[87]: 2

In [88]:
```
mod(7^77, 5)
```

Out[88]: 2

In [ ]: