Generatoren

Generatoren sind ähnlich wie Listen. Sie können allerdings jedes ihrer Elemente nur einmal ausgeben und sind danach leer. Die Funktion next(g) ruft das nächste Element des Generators g auf.

Vorteile: Effizienz! Elemente werden erst dann berechnet, wenn sie gebraucht werden ('lazy evaluation'). Es wird kein Speicher durch nicht mehr oder noch nicht benötigte Elemente belegt.

Generatoren können wie Listen erzeugt werden, nur mit **runden statt eckigen Klammern**.

In [1]:
```
g=(1..3)
g
```

Out[1]:
```
<_cython_3_0_11.generator object at 0x71388ef2e320>
```

In [2]:
```
next(g)
```

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

In [3]:
```
next(g)
```

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

In [4]:
```
next(g)
```

Out[4]:
```
3
```

In [5]:
```
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[5], line 1
----> 1 next(g)

StopIteration:
```

Der Generator g ist nun leer und hat keinen Nutzen mehr. Durch ernezte zuweisung kann er "neu befüllt" werden.

In [6]:
```
g=(1..4)
```

In [7]:
```
next(g)
```

Out[7]:
```
1
```

Restliche Elemente des Generators als Liste ausgeben:

In [8]:
```
list(g)
```

Out[8]:
```
[2, 3, 4]
```

In [9]:
```
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[9], line 1
----> 1 next(g)

StopIteration:
```

Hauptanwendung: über Generatoren kann in Schleifen iteriert werden.

In [10]:
```
for i in (1..4):
    print(i)
```

```
1
2
3
4
```

Unendlicher Generator, der alle natürlichen Zahlen ausgibt (natürlich innerhalb technischer Limits).

In [11]:
```
nn=(1..)
```

In [12]:
```
next(nn)
```

Out[12]:
```
1
```

In [13]:
```
next(nn)
```

Out[13]:
```
2
```

In [14]:
```
for i in nn:
    print(i)
    if i>10:
        break
```

```
3
4
5
6
7
8
9
10
11
```

Die folgende Zelle führt zu einer Endlosschleife. Manueller Abbruch über Menü **Kernel -> Interrupt Kernel**

In [ ]:
```
for i in nn:
    print(i)
```

In [ ]:
```
next(nn)
```

Erschaffung neuer Generatoren aus vorhandenen mittels **generator comprehensions** (analog zu list comprehensions).

In [15]:
```
a=(1..5)
```

In [16]:
```
b=(i^2 for i in a)
```

In [17]:
```
next(b)
```

Out[17]: 1

In [ ]:
```

```

Achtung: Bei Aufruf des äußeren Generators b wird intern a aufgerufen. D.h. auch a zählt einen Schritt weiter.

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

Out[18]: 2

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

Out[19]: 9

In [20]:
```
next(b)
```

Out[20]: 16

In [21]:
```
next(a)
```

Out[21]: 5

In [22]:
```
next(b)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[22], line 1
----> 1 next(b)

StopIteration:
```

Programmierung von Generatoren

Komplexere Generatoren lassen sich wie Funktionen programmieren. Das Schlüsselwort **yield** gibt an, dass an dieser Stelle die Ausführung unterbrochen und der angegebene Wert zurückgegeben werden soll. Beim nächsten Aufruf mittels next() wird die Ausführung an genau dieser Stelle fortgesetzt.

In [23]:
```
def abc():
```

```
    yield 'a'
    yield 'b'
    yield 'c'
```

In [24]:
```
abc
```

Out[24]:
```
<function abc at 0x71388e5c9bc0>
```

Die Funktion abc enthält das Schlüsselwort **yield**. Bei Aufruf liefert sie also einen Generator zurück.

In [25]:
```
g=abc()
```

In [26]:
```
g
```

Out[26]:
```
<generator object abc at 0x71388e031a60>
```

Beim ersten Aufruf wird der Code in obiger Definition von abd() bis zum ersten **yield** durchlaufen und dort unterbrochen.

In [27]:
```
next(g)
```

Out[27]:
```
'a'
```

Beim nächsten Aufruf wird der Code von dort weiter bis zum nächsten **yield** durchlaufen, und so weiter.

In [28]:
```
next(g)
```

Out[28]:
```
'b'
```

In [29]:
```
next(g)
```

Out[29]:
```
'c'
```

In [30]:
```
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[30], line 1
----> 1 next(g)

StopIteration:
```

Generatoren werden üblicherweise mittels Schleifen definiert. Hier ist ein Generator, der alle Primzahlzwillinge liefert.

In [31]:
```
def primzw(n=oo):
    p=3
    while p<n:
        q=next_prime(p)
        if q==p+2:
            yield (p,q)
        p=q
```

In [32]:
```
g=primzw()
```

In [33]:
```
next(g)
```

Out[33]: (3, 5)

In [34]:
```
next(g)
```

Out[34]: (5, 7)

In [35]:
```
next(g)
```

Out[35]: (11, 13)

Primzahlzwillinge bis 1000:

In [36]:
```
pp=list(primzw(1000))
pp
```

Out[36]:
```
[(3, 5),
 (5, 7),
 (11, 13),
 (17, 19),
 (29, 31),
 (41, 43),
 (59, 61),
 (71, 73),
 (101, 103),
 (107, 109),
 (137, 139),
 (149, 151),
 (179, 181),
 (191, 193),
 (197, 199),
 (227, 229),
 (239, 241),
 (269, 271),
 (281, 283),
 (311, 313),
 (347, 349),
 (419, 421),
 (431, 433),
 (461, 463),
 (521, 523),
 (569, 571),
 (599, 601),
 (617, 619),
 (641, 643),
 (659, 661),
 (809, 811),
 (821, 823),
 (827, 829),
 (857, 859),
 (881, 883)]
```

Man kann aus gegebenen Generatoren auch neue programmieren. Hier ist ein Generator, der die ersten n Elemente des gegebenen Generators g liefert.

In [37]:
```
def firstn(g,n):
    for i in range(n):
        yield next(g)
```

Die ersten 20 Primzahlzwillinge.

In [38]:
```
list(firstn(primzw(),20))
```

Out[38]:
```
[(3, 5),
 (5, 7),
 (11, 13),
 (17, 19),
 (29, 31),
 (41, 43),
 (59, 61),
 (71, 73),
 (101, 103),
 (107, 109),
 (137, 139),
 (149, 151),
 (179, 181),
 (191, 193),
 (197, 199),
 (227, 229),
 (239, 241),
 (269, 271),
 (281, 283),
 (311, 313)]
```

Rekursive Generatoren

Generatoren können auch rekursiv definiert werden.

Im folgenden Beispiel schreiben wir einen Generator, der die Elemente der n-ten kartesischen Potenz einer Menge S liefert. Diese kann induktiv definiert werden durch $S^n = S \times S^{n-1}$. Das lässt sich direkt in einen Generator übersetzen.

In [39]:
```
def cart(S,n):
    if n==0:
        yield []
    else:
        for s in S:
            for p in cart(S,n-1):
                yield [s]+p
```

In [40]:
```
list(cart([1,2],4))
```

Out[40]:
```
[[1, 1, 1, 1],
 [1, 1, 1, 2],
 [1, 1, 2, 1],
 [1, 1, 2, 2],
 [1, 2, 1, 1],
 [1, 2, 1, 2],
 [1, 2, 2, 1],
 [1, 2, 2, 2],
 [2, 1, 1, 1],
 [2, 1, 1, 2],
 [2, 1, 2, 1],
 [2, 1, 2, 2],
 [2, 2, 1, 1],
 [2, 2, 1, 2],
 [2, 2, 2, 1],
 [2, 2, 2, 2]]
```

In [41]:
```
list(cart([1,2],3))
```

Out[41]:
```
[[1, 1, 1],
 [1, 1, 2],
 [1, 2, 1],
 [1, 2, 2],
 [2, 1, 1],
 [2, 1, 2],
 [2, 2, 1],
 [2, 2, 2]]
```

In [42]:
```
list(cart([1,2],2))
```

Out[42]:
```
[[1, 1], [1, 2], [2, 1], [2, 2]]
```

In [43]:
```
list(cart([1,2],1))
```

Out[43]:
```
[[1], [2]]
```

Hier schreiben wir einen Generator, der alle Permutationen einer Menge S liefert (hier dargestellt als alle Anordnungen der Elemente von S). Jede Permutation besteht aus einem Element, das an den Anfang gestellt wird, und einer Permutation der restlichen Elemente. Daraus ergibt sich folgender rekursiver Generator.

In [44]:
```
def perm(S):
    if S==Set():
        yield []
    else:
        for s in S:
            for p in perm(S-{s}):
                yield [s]+p
```

In [45]:
```
list(perm(Set({1,2,3,4})))
```

Out[45]:
```
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 3, 2, 4],
 [1, 3, 4, 2],
 [1, 4, 2, 3],
 [1, 4, 3, 2],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [2, 3, 1, 4],
 [2, 3, 4, 1],
 [2, 4, 1, 3],
 [2, 4, 3, 1],
 [3, 1, 2, 4],
 [3, 1, 4, 2],
 [3, 2, 1, 4],
 [3, 2, 4, 1],
 [3, 4, 1, 2],
 [3, 4, 2, 1],
 [4, 1, 2, 3],
 [4, 1, 3, 2],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

Fehlerquelle: leere Menge als Set vs set vs dict:

In [46]:
```
Set()==set()
```

Out[46]:
```
True
```

In [47]:
```
Set()=={}
```

Out[47]:
```
False
```

In [48]:
```
parent({})   # {} gibt das leere dictionary, keine Menge!
```

Out[48]:
```
<class 'dict'>
```

Fehlerbehandlungen: Exceptions

Wir haben schon verschiedene Fehlermeldungen gesehen. Diese sind in Python eigene Objekte, sogenannte **Exceptions**. Exceptions können je nach Art des Fehlers verschiedenen Klassen angehören, z.B. NameError, ValueError etc.

In [49]:
```
y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[49], line 1
----> 1 y

NameError: name 'y' is not defined
```

In [50]:
```
factorial(-1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last
t)
Cell In[50], line 1
----> 1 factorial(-Integer(1))

File /opt/sagemath/sage-10.7/src/sage/symbolic/function.pyx:1061, in sag
e.symbolic.function.BuiltinFunction.__call__()
   1059 res = self._evalf_try_(*args)
   1060 if res is None:
-> 1061     res = super().__call__(
   1062             *args, coerce=coerce, hold=hold)
   1063

File /opt/sagemath/sage-10.7/src/sage/symbolic/function.pyx:558, in sage.s
ymbolic.function.Function.__call__()
    556
    557         from .expression import call_registered_function
--> 558         return call_registered_function(self._serial, self._nargs,
args, hold,
    559                                         not symbolic_input, SR)
    560

File /opt/sagemath/sage-10.7/src/sage/symbolic/pynac_function_impl.pxi:1,
in sage.symbolic.expression.call_registered_function()
----> 1 cpdef call_registered_function(unsigned serial,
      2                                int nargs,
      3                                list args,

File /opt/sagemath/sage-10.7/src/sage/symbolic/pynac_function_impl.pxi:49,
in sage.symbolic.expression.call_registered_function()
     47     res = g_function_evalv(serial, vec, hold)
     48 elif nargs == 1:
---> 49     res = g_function_eval1(serial,
     50                            (<Expression>args[0])._gobj, hold)
     51 elif nargs == 2:

File /opt/sagemath/sage-10.7/src/sage/functions/other.py:1551, in Function
_factorial._eval_(self, x)
   1549 if isinstance(x, (int, Integer)):
   1550     try:
-> 1551         return x.factorial()
   1552     except OverflowError:
   1553         return

File /opt/sagemath/sage-10.7/src/sage/rings/integer.pyx:4575, in sage.ring
s.integer.Integer.factorial()
   4573 """
   4574 if mpz_sgn(self.value) < 0:
-> 4575     raise ValueError("factorial only defined for nonnegative integ
ers")
   4576
   4577 if not mpz_fits_ulong_p(self.value):

ValueError: factorial only defined for nonnegative integers
```

In [51]:
```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[51], line 1
----> 1 Integer(1)/Integer(0)

File /opt/sagemath/sage-10.7/src/sage/rings/integer.pyx:2040, in sage.rings.integer.Integer.__truediv__()
   2038 if type(left) is type(right):
   2039     if mpz_sgn((<Integer>right).value) == 0:
-> 2040         raise ZeroDivisionError("rational division by zero")
   2041     x = <Rational> Rational.__new__(Rational)
   2042     mpq_div_zz(x.value, (<Integer>left).value, (<Integer>right).value)

ZeroDivisionError: rational division by zero
```

In [52]:
```
1/[1,2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[52], line 1
----> 1 Integer(1)/[Integer(1),Integer(2)]

File /opt/sagemath/sage-10.7/src/sage/rings/integer.pyx:2055, in sage.rings.integer.Integer.__truediv__()
   2053         return y
   2054
-> 2055     return coercion_model.bin_op(left, right, operator.truediv)
   2056
   2057 cpdef _div_(self, right):

File /opt/sagemath/sage-10.7/src/sage/structure/coerce.pyx:1288, in sage.structure.coerce.CoercionModel.bin_op()
   1286     # We should really include the underlying error.
   1287     # This causes so much headache.
-> 1288     raise bin_op_exception(op, x, y)
   1289
   1290 cpdef canonical_coercion(self, x, y):

TypeError: unsupported operand parent(s) for /: 'Integer Ring' and '<class 'list'>'
```

Fehlermeldungen ausgeben

Wir sehen nun, wie man in Python professionell Fehlermeldungen ausgibt.

In [63]:
```
def fac(n):
    if n==0:
        return 1
    return n*fac(n-1)
```

In [64]:
```
fac(6)
```

Out[64]: 720

In [65]:
```
fac(5)
```

Out[65]: 120

In [66]:
```
fac(-1)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last
t)
Cell In[66], line 1
----> 1 fac(-Integer(1))

Cell In[63], line 4, in fac(n)
      2 if n==Integer(0):
      3     return Integer(1)
----> 4 return n*fac(n-Integer(1))

Cell In[63], line 4, in fac(n)
      2 if n==Integer(0):
      3     return Integer(1)
----> 4 return n*fac(n-Integer(1))

    [... skipping similar frames: fac at line 4 (2970 times)]

Cell In[63], line 4, in fac(n)
      2 if n==Integer(0):
      3     return Integer(1)
----> 4 return n*fac(n-Integer(1))

Cell In[63], line 2, in fac(n)
      1 def fac(n):
----> 2     if n==Integer(0):
      3         return Integer(1)
      4     return n*fac(n-Integer(1))

RecursionError: maximum recursion depth exceeded while calling a Python ob
ject
```

Hier hat fac(-1) zu einer Endlosrekursion geführt. Um das zu vermeiden, müssen wir abfragen, ob das Argument negativ ist. Hier zuerst die unprofessionelle Variante, eine Fehlermeldung als String zurückzugeben.

In [67]:
```
def fac(n): # unprofessionell
    if n<0:
        return "n ist negativ"
    if n==0:
        return 1
    return n*fac(n-1)
```

In [68]:
```
fac(-1)
```

Out[68]: 'n ist negativ'

Nachteil: im weiteren Programmverlauf kann es zu Fehlern oder seltsamem Verhalten kommen, wenn der zurückgegebene Wert statt der erwarteten Zahl ein String ist.

In [69]:
```
m=fac(-1)
```

In [70]:
```
m^2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[70], line 1
----> 1 m**Integer(2)

File /opt/sagemath/sage-10.7/src/sage/rings/integer.pyx:2214, in sage.rings.integer.Integer.__pow__()
   2212         return coercion_model.bin_op(left, right, operator.pow)
   2213     # left is a non-Element: do the powering with a Python int
-> 2214     return left ** int(right)
   2215
   2216 cpdef _pow_(self, other):

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

In [71]:
```
m+m
```

Out[71]:
```
'n ist negativn ist negativ'
```

Hier die professionelle Variante: mit dem Schlüsselwort **raise** wird eine Exception der Klasse ValueError erstellt.

In [72]:
```
def fac(n):
    if n<0:
        raise ValueError('n ist negativ')
    if n==0:
        return 1
    return n*fac(n-1)
```

Das Ergebnis sieht nun aus wie die Fehlermeldungen der internen Python- oder Sage-Funktionen, die wir weiter oben gesehen haben.

In [73]:
```
fac(-1)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[73], line 1
----> 1 fac(-Integer(1))

Cell In[72], line 3, in fac(n)
      1 def fac(n):
      2     if n<Integer(0):
----> 3         raise ValueError('n ist negativ')
      4     if n==Integer(0):
      5         return Integer(1)

ValueError: n ist negativ
```

Der Vorteil von Exceptions ist, dass diese im umgebenden Code gezielt abgefangen werden können. Mit **try** wird versucht, einen Codeblock auszuführen. Darauf folgende **except**-Blöcke werden dann ausgeführt, wenn im **try**-Block eine entsprechende Exception auftritt. Damit können Fehler gezielt behandelt werden.

Hier als Beispiel eine Funktion, die bei negativen Zahlen n die Faktorielle von $-n$ zurückgeben soll.

In [74]:
```
def fac1(n):
    try:
        return fac(n)
    except ValueError:
        print('failed')
        return fac(-n)
```

In [75]:
```
fac1(6)
```

Out[75]: 720

In [76]:
```
fac1(-3)
```

```
failed
```

Out[76]: 6

Achtung: das war nur ein einfaches Besipiel zur Erklärung der Grundfunktionen. Die Fehlerbehandlung ist hier immer noch unzureichend. Z.B.:

In [77]:
```
fac(-pi)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[77], line 1
----> 1 fac(-pi)

Cell In[72], line 3, in fac(n)
      1 def fac(n):
      2     if n<Integer(0):
----> 3         raise ValueError('n ist negativ')
      4     if n==Integer(0):
      5         return Integer(1)

ValueError: n ist negativ
```

In der Definition von fac1(n) gibt es bisher nur einen **except**-Block für ValueError. Andere Exceptions werden nicht abgefangen und kommen daher durch bis zum Benutzer.

In [78]:
```
fac1([1,2])
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[78], line 1
----> 1 fac1([Integer(1),Integer(2)])

Cell In[74], line 3, in fac1(n)
      1 def fac1(n):
      2     try:
----> 3         return fac(n)
      4     except ValueError:
      5         print('failed')

Cell In[72], line 2, in fac(n)
      1 def fac(n):
----> 2     if n<Integer(0):
      3         raise ValueError('n ist negativ')
      4     if n==Integer(0):

File /opt/sagemath/sage-10.7/src/sage/rings/integer.pyx:925, in sage.ring
s.integer.Integer.__richcmp__()
    923     c = mpz_cmp_d((<Integer>left).value, d)
    924 else:
--> 925     return coercion_model.richcmp(left, right, op)
    926
    927 return rich_to_bool_sgn(op, c)

File /opt/sagemath/sage-10.7/src/sage/structure/coerce.pyx:2064, in sage.s
tructure.coerce.CoercionModel.richcmp()
   2062     raise bin_op_exception('<=', x, y)
   2063 elif op == Py_GT:
-> 2064     raise bin_op_exception('>', x, y)
   2065 else:
   2066     raise bin_op_exception('>=', x, y)

TypeError: unsupported operand parent(s) for >: 'Integer Ring' and '<class
'list'>'
```

Man kann auch mehrere Exceptions abfangen. Mit **as** kann man die abgefangene Exception einer Variable zuweisen, um im darauffolgenden Block auf sie zuzugreifen. **except:** ohne Angabe einer Klasse fängt alle Exceptions ab.

In [79]:
```
def fac1(n):
    try:
        return fac(n)
    except ValueError:
        print('failed')
        return fac(-n)
    except TypeError as e:
        print('sinnloses Argument')
        print(e.args)
    except:
        print('etwas ging schief')
```

In [81]:
```
fac1([1,2])
```

```
sinnloses Argument
("unsupported operand parent(s) for >: 'Integer Ring' and '<class 'lis
t'>'",)
```

In [82]:
```
fac1(oo)
```

```
etwas ging schief
```

Klassen programmieren

Wir sehen und kurz an, wie man in Python eigene Klassen programmieren kann. Eine Klasse stellt Funktionen (sogenannte Methoden) bereit, die man mit ihren Objekten ausführen kann.

Wir wollen als Beispiel eine Klasse MaxPlus definieren, die die Max-Plus-Algebra $(\mathbb{R}\cup -\infty, \oplus,\odot)$ implementiert, also den Halbring mit Operationen $a\oplus b=\max\{a,b\}$ und $a\odot b = a+b$. Die Operationen sollen wie gewohnt direkt mit + und * aufgerufen werden.

Intern wird bei a+b die Methode **.__add__()** der jeweiligen Klasse aufgerufen.

In [83]:
```
1+3
```

Out[83]:
```
4
```

In [84]:
```
1.__add__(3)
```

Out[84]:
```
4
```

Jede Klasse soll zumindest zwei Methoden haben: **__init__()**, die bei der Erstellung neuer Objekte der Klasse aufgerufen wird, und **__repr__()**, die angibt, wie Objekte im Output dargestellt werden.

Hier erhält das Objekt bei der Erstellung ein Attribut **.val**, in dem der übergebene Wert **v** (= die Zahl, die das Objekt darstellen soll) gespeichert wird.

Zur Darstellung wird der Wert in eckigen Klammern ausgegeben, um Elemente der Max-Plus-Algebra von herkömmlichen Zahlen zu unterscheiden.

In [85]:
```
class MaxPlus:
    def __init__(self,v):
        self.val=v
    def __repr__(self):
        return f"[{self.val}]"
```

In [86]:
```
a=MaxPlus(-1)
```

In [87]:
```
a
```

Out[87]:
```
[-1]
```

In [88]:
```
a.val
```

Out[88]: -1

In [89]:
```
b=MaxPlus(-3)
```

In [90]:
```
b
```

Out[90]: [-3]

In [91]:
```
a+b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[91], line 1
----> 1 a+b

TypeError: unsupported operand type(s) for +: 'MaxPlus' and 'MaxPlus'
```

Die Operationen Addition und Multiplikation müssen wir erst implementieren.

In [92]:
```
class MaxPlus:
    def __init__(self,v):
        self.val=v
    def __repr__(self):
        return f"[{self.val}]"
    def __add__(self,b):
        return MaxPlus(max(self.val,b.val))
    def __mul__(self,b):
        return MaxPlus(self.val+b.val)
```

In [93]:
```
a=MaxPlus(-1)
b=MaxPlus(-3)
```

In [94]:
```
a+b
```

Out[94]: [-1]

In [95]:
```
a*b
```

Out[95]: [-4]

In [96]:
```
a+MaxPlus(-oo)
```

Out[96]: [-1]

In [97]:
```
a*MaxPlus(-oo)
```

Out[97]: [-Infinity]

Tatsächlich ist Max-Plus-Algebra in Sage bereits implementiert!

In [98]:
```
TropicalSemiring?
```

```
Init signature: TropicalSemiring(self, x=0, *args, **kwds)
Docstring:
   The tropical semiring.

   Given an ordered additive semigroup R, we define the tropical
   semiring T = R \cup \{+\infty\} by defining tropical addition and
   multiplication as follows:

      a \oplus b = \min(a, b), \quad \quad a \odot b = a + b.

   In particular, note that there are no (tropical) additive inverses
   (except for \infty), and every element in R has a (tropical)
   multiplicative inverse.

   There is an alternative definition where we define T = R \cup
   \{-\infty\} and alter tropical addition to be defined by

      a \oplus b = \max(a, b).

   To use the \max definition, set the argument "use_min = False".

   Warning:

     "zero()" and "one()" refer to the tropical additive and
     multiplicative identities respectively. These are **not** the
     same as calling "T(0)" and "T(1)" respectively as these are
     **not** the tropical additive and multiplicative identities
     respectively.Specifically do not use "sum(...)" as this converts
     0 to 0 as a tropical element, which is not the same as "zero()".
     Instead use the "sum" method of the tropical semiring:

        sage: T = TropicalSemiring(QQ)

        sage: sum([T(1), T(2)]) # This is wrong
        0
        sage: T.sum([T(1), T(2)]) # This is correct
        1

     Be careful about using code that has not been checked for
     tropical safety.

   INPUT:

   * "base" -- the base ordered additive semigroup R

   * "use_min" -- boolean (default: "True"); if "True", then the
     semiring uses a \oplus b = \min(a, b). Otherwise uses a \oplus b
     = \max(a, b).

   EXAMPLES:

      sage: T = TropicalSemiring(QQ)
      sage: elt = T(2); elt
      2

   Recall that tropical addition is the minimum of two elements:

      sage: T(3) + T(5)
      3
```

```
    Tropical multiplication is the addition of two elements:

       sage: T(2) * T(3)
       5
       sage: T(0) * T(-2)
       -2

    We can also do tropical division and arbitrary tropical
    exponentiation:

       sage: T(2) / T(1)
       1
       sage: T(2)^(-3/7)
       -6/7

    Note that "zero" and "one" are the additive and multiplicative
    identities of the tropical semiring. In general, they are **not**
    the elements 0 and 1 of R, respectively, even if such elements
    exist (e.g., for R = \ZZ), but instead the (tropical) additive and
    multiplicative identities +\infty and 0 respectively:

       sage: T.zero() + T(3) == T(3)
       True
       sage: T.one() * T(3) == T(3)
       True
       sage: T.zero() == T(0)
       False
       sage: T.one() == T(1)
       False
Init docstring:    Initialize "self".
File:           /opt/sagemath/sage-10.7/src/sage/rings/semirings/tropical_
semiring.pyx
Type:           ClasscallMetaclass
Subclasses:
```

In [99]:
```
MP=TropicalSemiring(QQ,use_min=False)
MP
```

Out[99]:
```
Tropical semiring over Rational Field
```

In [100…
```
a=MP(-1)
b=MP(-3)
```

In [101…
```
a+b
```

Out[101…
```
-1
```

In [102…
```
a*b
```

Out[102…
```
-4
```

Achtung: nicht alle Sage-Funktionen verhalten sich wie erwartet:

In [103…
```
add([a,b])
```

Out[103…
```
0
```

In [104…
```
add?
```

```
Signature:      add(*args, **kwds)
Docstring:
Return the sum of a 'start' value (default: 0) plus an iterable of
numbers

When the iterable is empty, return the start value. This function is
intended specifically for use with numeric values and may reject non-
numeric types.
Init docstring: Initialize self.  See help(type(self)) for accurate signat
ure.
File:
Type:           builtin_function_or_method
```

In [105…
```
0+a
```

Out[105… 0

Das Problem ist, dass **add** die Summe standardmäßig mit 0 beginnt, aber 0 nicht das neutrale Element in MP ist.

In [106…
```
MP.zero()
```

Out[106… -infinity

In [107…
```
add([a,b],start=MP.zero())
```

Out[107… -1

Andere Möglichkeit, den Startwert zu vermeiden (wenn die übergebene Liste sicher nicht leer ist!):

In [108…
```
reduce(operator.add,[a,b])
```

Out[108… -1