fractions

February 24, 2024

1 Arithmetic: Numerical Fractions, continued fractions, accuracy issues, and other topics:

Dealing with infinite precision

1.1 Getting help

Always start by first importing `sympy`

```
import sympy
```

If your system supports online help, then you can type `sympy`, followed by a dot to receive online hints on what options and attributes are available for completion.

```
import sympy
sympy.
```

This will show all attributes of the `sympy` class. You can also inquire programatically with the command

```
dir(sympy)
```

Once you find the attribute that you want to learn about, say, for example,

`AlgebraicNumber` then you can type `help(sympy.AlgebraicNumber)` or `?sympy.AlgebraicNumber` or `??sympy.AlgebraicNumber`

You can (if in Jupyter od JupyterLab) get to the `Help` tab, at the end you will find the entry `SymPy reference`.

In addition to the default refereences, you can browse the internet with Google, ChapGPT, etc.

[2]:
```
import sympy
# dir(sympy)
```

[3]:
```
# dir(sympy) # see that this has more than 900 attributes and methods
```

[4]:
```
# help(sympy.AlgebraicNumber)
# ?sympy.AlgebraicNumber
# ??sympy.AlgebraicNumber
```

[5]:
```
# import sympy
from sympy import Rational
a = Rational(1,2)
b = Rational(3,4)
c = a+b
print(a,b,c)
display("solution after adding 1/2 + 3/4",c)
```

```
1/2 3/4 5/4
```

```
'solution after adding 1/2 + 3/4'
```

$$\frac{5}{4}$$

[6]:
```
a=1/2  # fraction are taken as floating point numbers
b=3/4
c=a+b
print(a,b,c)
```

```
0.5 0.75 1.25
```

[7]:
```
a=1/3
b=Rational(1,3)
a-b
```

[7]: 0

[8]:
```
print(a)
display(b)
```

```
0.3333333333333333
```

$$\frac{1}{3}$$

[9]:
```
d = Rational(.5)
display(d)
```

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

[10]:
```
# Use help(sympy.Rational) to understand what happens if a float is passed as␣
↪argument
```

[11]:
```
e = Rational(0.2)
display(e)
```

$$\frac{3602879701896397}{18014398509481984}$$

[12]:
```
# Use help(sympy.Rational) to predict what happens if a String is passed as␣
↪argument
```

[13]:
```
a = Rational("0.2")
display(a)
```

$\frac{1}{5}$

[14]:
```
# Rational is handy for powers.abs
from sympy import symbols
x = symbols('x')
x**(3/2)
```

[14]: $x^{1.5}$

[15]:
```
x**(Rational(1.5))
```

[15]: $x^{\frac{3}{2}}$

[16]:
```
from sympy import simplify
simplify(e)
```

[16]: $\frac{3602879701896397}{18014398509481984}$

[17]:
```
from fractions import Fraction
print(Fraction(0.2))
```

```
3602879701896397/18014398509481984
```

[18]:
```
float(e)
```

[18]: 0.2

[19]:
```
print(1/5)
```

```
0.2
```

Homework 1: Please explain why the parameter e is such a huge fraction. check here.

[21]:
```
a=Rational(1, 2)
display(a)
```

$\frac{1}{2}$

[23]:
```
import sympy as sp # this is better when ambiguity. There is sqrt in numpy, and␣
↪math also
b=sp.sqrt(2)
display(a*b)
```

$$\frac{\sqrt{2}}{2}$$

[24]:
```
display( (a*b)**2 )
```

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

[25]:
```
print(a,b)
```

```
1/2 sqrt(2)
```

1.2 Operations with numerical radicals

[26]:
```
sp.sqrt(4)
```

[26]: 2

$$\sqrt{27} - \sqrt{12} = 3\sqrt{3} - 2\sqrt{3} = \sqrt{3}. \tag{1}$$

[27]:
```
sp.sqrt(27)-sp.sqrt(12) #note that it simplifies automatically
```

[27]: $\sqrt{3}$

$$2\sqrt{18} + 3\sqrt{8} = 2 \times 3\sqrt{2} + 3 \times 2 \times \sqrt{2} = 12\sqrt{2}.$$

[28]:
```
2*sp.sqrt(18) + 3*sp.sqrt(8)
```

[28]: $12\sqrt{2}$

1.3 Continued fractions

We construct a continued fraction of the form

$$a_0 + \cfrac{x}{a_1 + \cfrac{x}{a_2 + \cfrac{x}{a_3 + \cfrac{x}{a_4 + \cfrac{x}{a_5 + \cdots}}}}} \tag{2}$$

[30]:
```
from sympy import symbols
a = symbols('a0:6')
```

[31]:
```
a
```

[31]:
```
(a0, a1, a2, a3, a4, a5)
```

[32]:
```
x = symbols('x')
```

[33]:
```python
def continued_fraction(depth):
    frac = a[0]
    for d in range(0, depth):
        frac = frac.replace(a[d],  a[d] + x/a[d+1])
    return frac
```

[34]:
```python
cf = continued_fraction(5)
cf
```

[34]: $a_0 + \frac{x}{a_1 + \frac{x}{a_2 + \frac{x}{a_3 + \frac{x}{a_4 + \frac{x}{a_5}}}}}$

1.4 Convert expression to LaTeX

[35]:
```python
import sympy
sympy.latex(cf)
```

[35]:
```
'a_{0} + \\frac{x}{a_{1} + \\frac{x}{a_{2} + \\frac{x}{a_{3} + \\frac{x}{a_{4} +
\\frac{x}{a_{5}}}}}}'
```

Homework 2: Find a way such that `sympy.latex` prints only one back-slash Change the double backslash for a simple slash

$$a_0 + \frac{x}{a_1 + \frac{x}{a_2 + \frac{x}{a_3 + \frac{x}{a_4 + \frac{x}{a_5}}}}}. \tag{3}$$

[36]:
```python
# cfPI = cf.substitute( [a, 3,7,15,1,292,1])
cfPI = cf.subs( [ (a[0], 3), (a[1], 7), (a[2],15), (a[3],1),
↪(a[4],292),(a[5],1),(x,1)])
```

[37]:
```python
print(cfPI)
```

```
104348/33215
```

[38]:
```python
float(cfPI)
```

[38]: `3.141592653921421`

Homework 3: Approximate the Euler number e, and the Golden Ratio $(1+\sqrt{5})/2$ with up to 30 coefficients a_i of continued fractions

1.5 Other ways to evaluate

[39]:
```
import sympy as sp
display(cfPI) # an exact fraction
print(float(cfPI))  # convert to float
print(sp.N(cfPI)) # numerical evaluation from SymPy
```

$$\frac{104348}{33215}$$

```
3.141592653921421
3.14159265392142
```

1.5.1 Display with n decimal places.

We illustrate the operators `N()`, `float()`, `print()`, `Float()`, and `evalf()`. They do not have the same precision. The lower precision are `print()`, and `float()`.

`N()`

[40]:
```
# one more example
sp.N(sp.pi, 30)  # indicates the number of decimal places
```

[40]: 3.14159265358979323846264338328

[41]:
```
sp.N(sp.pi, 300) # try higher to see the limits in your machine
```

[41]: 3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117006

`float()`

[42]:
```
float(sp.pi) #
```

[42]: 3.141592653589793

`print()`

[43]:
```
print("%.30f"%(sp.pi)) # indicates number of decimal places
```

```
3.141592653589793115997963468544
```

`Float()`

[44]:
```
sp.Float(sp.pi, 30)
```

[44]: 3.14159265358979323846264338328

`evalf()`

[45]:
```
sp.pi.evalf(30)
```

[45]: 3.14159265358979323846264338328

Homework 4: Why is this different that `print("%.30f"%(pi))`?

1.5.2 Accuracy:

Some ideas here are taken from the sympy manual.

We commented that the operators `evalf`, `N`, and `Float` have higher precision. We illustrate this with an example taken from the SymPy manual.

The example illustrates how the 100'th term of the Fibonacci series , which is a big number, should be exactly

$$\frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}.$$

with φ equal to the GoldenRatio.

[46]:
```
from sympy import GoldenRatio, fibonacci
n=1000
a,b = (GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5), fibonacci(n)
display(float(a))
display(float(b))
float(a)-float(b)
```

```
4.3466557686937455e+208
```

```
4.3466557686937455e+208
```

[46]: 0.0

[47]:
```
n=100
sp.N((GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5) - fibonacci(n))
```

[47]: $3.0 \cdot 10^{-104}$

Observe how a small error does not mean things are good when scaling them.

[48]:
```
n=1000
sp.N((GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5) - fibonacci(n))
```

[48]: $2.0 \cdot 10^{85}$

Homework 5: Why is this error so huge, while using `float` there was no error. See how the `maxn` increases the precision.

[49]:
```
n=1000
# here 500, try 5000 and see the change
sp.N((GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5) - fibonacci(n), maxn=500)
```

[49]: $8.0 \cdot 10^{-528}$

We can force the operation to exit when precision is getting bad.

[50]:
```python
sp.N((GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5) - fibonacci(n),␣
↪strict=True)
```

```
---------------------------------------------------------------------------
PrecisionExhausted                        Traceback (most recent call last)
Cell In[50], line 1
----> 1␣
 ↪sp.N((GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5) - fibonacci(n), strict=True)

File ~/.local/lib/python3.10/site-packages/sympy/core/evalf.py:1749, in N(x, n,␣
 ↪**options)
   1727 r"""
   1728 Calls x.evalf(n, \*\*options).
   1729
   (...)
   1745
   1746 """
   1747 # by using rational=True, any evaluation of a string
   1748 # will be done using exact values for the Floats
-> 1749 return sympify(x, rational=True).evalf(n, **options)

File ~/.local/lib/python3.10/site-packages/sympy/core/evalf.py:1647, in EvalfMixin.
 ↪evalf(self, n, subs, maxn, chop, strict, quad, verbose)
   1645     options['quad'] = quad
   1646 try:
-> 1647     result = evalf(self, prec + 4, options)
   1648 except NotImplementedError:
   1649     # Fall back to the ordinary evalf
   1650     if hasattr(self, 'subs') and subs is not None:  # issue 20291

File ~/.local/lib/python3.10/site-packages/sympy/core/evalf.py:1532, in evalf(x,␣
 ↪prec, options)
   1530     r = chop_parts(r, chop_prec)
   1531 if options.get("strict"):
-> 1532     check_target(x, r, prec)
   1533 return r

File ~/.local/lib/python3.10/site-packages/sympy/core/evalf.py:358, in␣
 ↪check_target(expr, result, prec)
    356 a = complex_accuracy(result)
    357 if a < prec:
--> 358     raise PrecisionExhausted("Failed to distinguish the expression:␣
 ↪\n\n%s\n\n"
    359         "from zero. Try simplifying the input, using chop=True, or␣
 ↪providing "
    360         "a higher maxn for evalf" % (expr))
```

```
PrecisionExhausted: Failed to distinguish the expression:

-434665576869374564356885276750406258025646605173717804024817290895365555417949051̶3904038798400
 ↪+ sqrt(5)*(-(1 - GoldenRatio)**1000 + GoldenRatio**1000)/5

from zero. Try simplifying the input, using chop=True, or providing a higher maxn␣
 ↪for evalf
```

[51]:
```
sp.N((GoldenRatio**n - (1 - GoldenRatio)**n)/sp.sqrt(5) - fibonacci(n),␣
 ↪chop=True)
```

[51]: $2.0 \cdot 10^{85}$

1.5.3 nsimplify() (from the sympy manual)

Sometimes we have a number and want to find a formula for it. Let the examples speak. While `N()`, `Float()`, `float()`, etc provide a float approximation, `nsimplify()` goes in the reverse direction. It takes a float and convert it into a fraction or a formula.

[52]:
```
from sympy import nsimplify
nsimplify(0.1)
```

[52]: $\frac{1}{10}$

[53]:
```
nsimplify(6.28, [sp.pi], tolerance=0.01) # the pi in square brackets is a hint␣
 ↪for the formula we are looking for
```

[53]: 2π

[54]:
```
nsimplify(sp.pi, tolerance=0.01)
```

[54]: $\frac{22}{7}$

[55]:
```
nsimplify(sp.pi, tolerance=0.001)
```

[55]: $\frac{355}{113}$

[56]:
```
nsimplify(4/(1+sp.sqrt(5)), [GoldenRatio])  # in terms of GoldenRatios
```

[56]: $-2 + 2\phi$

[57]:
```
from sympy import I
nsimplify(I**I, [sp.pi])   # from the manual. Check this
```

[57]: $e^{-\frac{\pi}{2}}$

Homework 6: Explain the previous result using complex analysis.

[58]:
```
nsimplify(0.333333, rational='True') #
```

[58]: $\frac{333333}{1000000}$

Evaluation of this expression using sums.

$$0.333333 = \frac{3}{10} + \frac{3}{10^2} + \cdots + \frac{3}{10^6} = 3\sum_{n=1}^{6} \frac{1}{10^n}.$$

We know that

$$1 + x + x^2 + \cdots + x^6 = \frac{1-x^7}{1-x}.$$

so

$$x + x^2 + \cdots + x^6 = \frac{1-x^7}{1-x} - 1 = \frac{1-x^7-1+x}{1-x} = \frac{x(1-x^6)}{1-x}.$$

Now for $x = 1/10$ we find

$$0.333333 = 3\frac{10^{-1}(1-10^{-6})}{1-10^{-1}} = 3\frac{1-1/10^6}{10-1} = 3\frac{10^6-1}{10^6\times 9} = 3\frac{999999}{9\times 10^6} = \frac{333333}{1000000}.$$

What is the big deal?

$$0.333333 \times \frac{1000000}{1000000} = \frac{333333}{1000000}.$$

However, the method shown here is general for any decimal and in any base. For example, in base 2 the powers would be of the form 2^{-n} , instead of 10^{-n}.

[59]:
```
from sympy import Sum
k=symbols('k')
s = 3*Sum(1/10**k, (k, 1,6))
```

[60]:
```
simplify(s, rational="True")
```

[60]: $\frac{333333}{1000000}$

[61]:
```
s.evalf()
```

[61]: 0.333333

[62]:
```
# one last point
0**0
```

[62]: 1

Homework: why $0^0 = 1$?