1 Symbolic Python

As discussed before, softwares such as MATLAB and numpy are numerical, they don't like to deal with symbols too much. Symbolic Python, sympy, is the first(and the last) symbolic software we will deal with in this class. So let's see what we mean by 'symbolic software'.

```
import numpy as np
import sympy as sp
print(np.pi)
print(sp.pi)
```

You will immediately see the difference between two answers here. 'np.pi' gives the numerical estimation of π which is $3.14159\cdots$. On the other hand, 'sp.pi' gives the symbol 'pi'. This is the exact value of 'pi'. Of course, we sometimes want to see the value of 'pi' rather than seeing 'pi'.

```
sp.pi.evalf(100)
```

gives the first hundred digits of π. So what exactly is 'sp.pi'? Let's try to check the types of two π.

```
type(np.pi)
type(sp.pi)
```

As one expected, 'np.pi' is a float. However, 'sp.pi' is a class called Pi inside sympy package. That is, sympy treats π as a separate object. We will discuss more about what 'object' means later.
In terms of precision, 'sp.pi' does a much better job compared to 'np.pi'. Then why don't we just use 'sp.pi' all the time? As you probably guessed already, working with symbolic objects is much slower. What we found out so far brings up a couple important points.

1. Here's another reason why you want to load packages using 'import package as name' rather than dumping everything into name space. (What would happen if we do?)

2. We need to be smart about which one between estimation and precise representation we want to use. Do we care more about precision or efficiency?

Of course, having the exact values of π, e and such is not the main point of symbolic software. What's important is that you can do calculations with symbolic variables. So let's define symbols first.

```
sp.var('x, y')
```

defines two variables x and y. If you type 'x', Python now returns 'x' rather than giving an error message. Now that we defined variables, let's see how Python deals with them.

```
>>> x-x
0
>>> x-y
x - y
>>> x*x
x**2
>>> x*y
x*y
>>> (x+2*x)**2
9*x**2
>>> (x+y)**2
(x + y)**2
```

We see a couple more important points here.

1. You can do usual arithmetic operations between variables.

2. Python simplifies some expressions.

We can also tell Python to expand or factor certain expressions.

```
>>> sp.expand((x+y)**2)
x**2 + 2*x*y + y**2
>>> z = (x+y)**2
>>> z.expand()
x**2 + 2*x*y + y**2

>>> sp.factor(x**2+2*x*y+y**2)
(x + y)**2
>>> z = x**2 + x*y + y**2
>>> z.factor()
x**2 + x*y + y**2
```

Note we don't need to put 'sp' for 'z.factor()' because 'z' is already an object in sympy. Simplifying is also an option.

```
>>> z = x**2+2*x*y+y**2
>>> z/(x+y)
(x**2 + 2*x*y + y**2)/(x + y)
>>> sp.simplify(z/(x+y))
x + y
>>> (z/(x+y)).simplify()
x + y
```

Of course, we can choose to substitute values for variables as well.

```
>>> z = x**2+2*x*y+y**2
>>> z.subs(x,2)
y**2 + 4*y + 4
>>> z.subs(x,2).factor()
(y + 2)**2
```

Sympy is capable of manipulating fractions as well!

```
>>> z = 1/((x+1)*(x-1)**2)
>>> z.apart()
1/(4*(x + 1)) - 1/(4*(x - 1)) + 1/(2*(x - 1)**2)
>>> z1 = sp.apart(z)
>>> z1.together()
(2*x + (x - 1)**2 - (x - 1)*(x + 1) + 2)/(4*(x - 1)**2*(x + 1))
>>> sp.simplify(z1.together())
1/(x**3 - x**2 - x + 1)
```

'apart' does the partial fraction decomposition and 'together' puts fractions in a common denominator.

2 Doing math

So let's do some math using sympy. Sympy can handle calculus pretty well.

```
>>> z = sp.cos(x)
>>> sp.diff(z,x)
-sin(x)
>>> z.diff(x)
-sin(x)
>>> sp.diff(z,x,2)
-cos(x)
>>> sp.diff(z,y)
0
>>> sp.integrate(z,x)
sin(x)
>>> sp.integrate(z,(x,0,sp.pi))
```

```
0
>>> z.integrate((x,0,sp.pi))
0
```

As you can see, Smypy can tell the difference between different variables and do calculus accordingly! It can also handle limit.

```
>>> z = sp.cos(x)
>>> sp.limit(z,x,0)
1
>>> z.limit(x,sp.oo)
AccumBounds(-1, 1)
```

'sp.oo' is ∞ and 'AccumBounds(-1,1)' means the limit is bounded between -1 and 1 but doesn't converge to a single value. What's even cooler is that Sympy can solve algebraic equations for us!

```
>>> sp.var('x, a, b, c')
(x, a, b, c)
>>> sp.solve(x**2+1)
[-I, I]
>>> sp.solve(a*x**2+b*x+c,x)
[(-b + sqrt(-4*a*c + b**2))/(2*a), -(b + sqrt(-4*a*c + b**2))/(2*a)]
```

Of course, any non-algebraic(transcendental) equations cannot be solved.

```
>>> sp.solve(sp.exp(x)+sp.cos(x))
Traceback (most recent call last):
File "<pyshell#110>", line 1, in <module>
sp.solve(sp.exp(x)+sp.cos(x))
File "C:\Users\shimj\AppData\Roaming\Python\Python36\site-packages\
sympy\solvers\solvers.py", line 1171, in solve
solution = _solve(f[0], *symbols, **flags)
File "C:\Users\shimj\AppData\Roaming\Python\Python36\site-packages\
sympy\solvers\solvers.py", line 1742, in _solve
raise NotImplementedError('\n'.join([msg, not_impl_msg % f]))
NotImplementedError: multiple generators [cos(x), exp(x)]
No algorithms are implemented to solve equation exp(x) + cos(x)
```

In this case, we can tell Sympy to solve such equations numerically.

```
>>> sp.nsolve(sp.exp(x)+sp.cos(x),x,-2)
-1.74613953040801
>>> sp.nsolve(sp.exp(x)+sp.cos(x),x,-4)
-4.70332375945224
```

'nsolve' tries to find an answer near the specified value. Of course, this method can fail and we will investigate this in our final project. 'nsolve' can even solve equations with multiple variables as long as you provide enough number of equations.

```
>>> sp.nsolve((x**2+sp.exp(y)-1, x+y), (x, y), (1, 1))
Matrix([
[ 2.45851271393911e-23],
[-2.45851271393911e-23]])
```

This may seem a little bit weird since the answer to the given system of equations must be $x = 0, y = 0$. However, note that the given answer is really really close to (0,0) as well. This is simply due to round-off error. Remember that we are trying to solve it numerically at this point!

3 Assignment

1. Using sympy, do the partial fraction decomposition of

$$z = \frac{1}{(x+3)(x-5)}$$

2. Use ‘solve’ command to find zero(s) of z.
3. Use ‘nsolve’ command to find zero(s) of z near $x = 0$. What’s happening here?

You don’t need to upload anything this time. Just type your answers directly into Blackboard.