13.1_Introduction_to_Sympy

December 2, 2014

Figure 1: BY-SA

Authors : *Sonia Estradé*
José M. Gómez
Ricardo Graciani
Manuel López
Xavier Luri
Josep Sabater

1 13. Introduction to sympy

The `sympy` python module allows to make *symbolic* calculations:

- solve equations,
- integrate functions,
- derivate functions,
- ...

Python variables will not represent numeric values but fuctions, and sympy allows us to calculate with them. You can find all the details about `sympy` at: Sympy tutorial

It provides a functionality simmilar to Mathematica, Matlab, GNU Octave, etc.

Like any other python module, we just need to import it:

```
import sympy
```

1.1 13.1 Basic functionality

1.1.1 Symbols and Expressions

The first step is to create the symbols that we will use. This can be done in two ways:

- using the function `var`

```
In [1]: import sympy as sp

        sp.var('x')                   # This creates a new "x" symbol which gets
                                      # assigned to the x python variable


        help( sp.var )
```

```
Help on function var in module sympy.core.symbol:

var(names, **args)
    Create symbols and inject them into the global namespace.

    This calls :func:`symbols` with the same arguments and puts the results
    into the *global* namespace. It's recommended not to use :func:`var` in
    library code, where :func:`symbols` has to be used::

        >>> from sympy import var

        >>> var('x')
        x
        >>> x
        x

        >>> var('a,ab,abc')
        (a, ab, abc)
        >>> abc
        abc

        >>> var('x,y', real=True)
        (x, y)
        >>> x.is_real and y.is_real
        True

    See :func:`symbol` documentation for more details on what kinds of
    arguments can be passed to :func:`var`.
```

- using the function `symbols`

```
In [2]: a, b, c = sp.symbols('a b c')     # In this case we define 3 new symbols
                                          # and assign them to a, b and c variables

        help( sp.symbols )
```

```
Help on function symbols in module sympy.core.symbol:

symbols(names, **args)
    Transform strings into instances of :class:`Symbol` class.

    :func:`symbols` function returns a sequence of symbols with names taken
    from ``names`` argument, which can be a comma or whitespace delimited
    string, or a sequence of strings::

        >>> from sympy import symbols, Function

        >>> x, y, z = symbols('x,y,z')
        >>> a, b, c = symbols('a b c')

    The type of output is dependent on the properties of input arguments::

        >>> symbols('x')
        x
        >>> symbols('x,')
```

```
    (x,)
    >>> symbols('x,y')
    (x, y)
    >>> symbols(('a', 'b', 'c'))
    (a, b, c)
    >>> symbols(['a', 'b', 'c'])
    [a, b, c]
    >>> symbols(set(['a', 'b', 'c']))
    set([a, b, c])

If an iterable container is needed for a single symbol, set the ``seq``
argument to ``True`` or terminate the symbol name with a comma::

    >>> symbols('x', seq=True)
    (x,)

To reduce typing, range syntax is supported to create indexed symbols.
Ranges are indicated by a colon and the type of range is determined by
the character to the right of the colon. If the character is a digit
then all continguous digits to the left are taken as the nonnegative
starting value (or 0 if there are no digit of the colon) and all
contiguous digits to the right are taken as 1 greater than the ending
value::

    >>> symbols('x:10')
    (x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

    >>> symbols('x5:10')
    (x5, x6, x7, x8, x9)
    >>> symbols('x5(:2)')
    (x50, x51)

    >>> symbols('x5:10,y:5')
    (x5, x6, x7, x8, x9, y0, y1, y2, y3, y4)

    >>> symbols(('x5:10', 'y:5'))
    ((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))

If the character to the right of the colon is a letter, then the single
letter to the left (or 'a' if there is none) is taken as the start
and all characters in the lexicographic range *through* the letter to
the right are used as the range::

    >>> symbols('x:z')
    (x, y, z)
    >>> symbols('x:c')  # null range
    ()
    >>> symbols('x(:c)')
    (xa, xb, xc)

    >>> symbols(':c')
    (a, b, c)

    >>> symbols('a:d, x:z')
```

```
        (a, b, c, d, x, y, z)

        >>> symbols(('a:d', 'x:z'))
        ((a, b, c, d), (x, y, z))

    Multiple ranges are supported; contiguous numerical ranges should be
    separated by parentheses to disambiguate the ending number of one
    range from the starting number of the next::

        >>> symbols('x:2(1:3)')
        (x01, x02, x11, x12)
        >>> symbols(':3:2')  # parsing is from left to right
        (00, 01, 10, 11, 20, 21)

    Only one pair of parentheses surrounding ranges are removed, so to
    include parentheses around ranges, double them. And to include spaces,
    commas, or colons, escape them with a backslash::

        >>> symbols('x((a:b))')
        (x(a), x(b))
        >>> symbols('x(:1\,:2)')  # or 'x((:1)\,(:2))'
        (x(0,0), x(0,1))

    All newly created symbols have assumptions set according to ``args``::

        >>> a = symbols('a', integer=True)
        >>> a.is_integer
        True

        >>> x, y, z = symbols('x,y,z', real=True)
        >>> x.is_real and y.is_real and z.is_real
        True

    Despite its name, :func:`symbols` can create symbol-like objects like
    instances of Function or Wild classes. To achieve this, set ``cls``
    keyword argument to the desired type::

        >>> symbols('f,g,h', cls=Function)
        (f, g, h)

        >>> type(_[0])
        <class 'sympy.core.function.UndefinedFunction'>
```

In [3]:
```python
print( x, 'variable of type', type(x) )
print( a, 'variable of type', type(a) )
```

```
x variable of type <class 'sympy.core.symbol.Symbol'>
a variable of type <class 'sympy.core.symbol.Symbol'>
```

It is important to notice that in the second case, the symbol that has been created can be assigned to any variable. The name of the variable does not need to be the same as the symbol. However, it can be very confussing to use code like:

```
b, a = sp.symbols( 'a, b' )
```

Sympy **symbols** can be combined to define **Expresions** that can be used, for instance, to find a solution to an equation

```
In [4]: myFirstExpression = a*x**2 + b*x + c
        print( myFirstExpression )
```

```
a*x**2 + b*x + c
```

Sympy can be configure to print Expressions in different formats. In particular, using Latex formatting allows nice integration with notebook.

```
In [5]: sp.init_printing()          # This will make sympy show results as
                                    # Latex equations
        myFirstExpression
```

Out[5]:

$$ax^2 + bx + c$$

We can now use this expresion to symbolically solve the corresponding equation (the usage of `solve()` will be discussed later):

```
In [6]: sp.solve(myFirstExpression, x)
```

Out[6]:

$$\left[\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right), \quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right]$$

We have told sympy to use x as a variable and a, b, c as constants. But we can "solve" it for any of the symbols we have used when defining the Expression:

```
In [7]: solutionA = sp.solve(myFirstExpression, a)
        solutionB = sp.solve(myFirstExpression, b)
        solutionC = sp.solve(myFirstExpression, c)
        { a: solutionA, b: solutionB, c: solutionC }
```

Out[7]:

$$\left\{a : \left[-\frac{1}{x^2}\left(bx + c\right)\right], \quad b : \left[-ax - \frac{c}{x}\right], \quad c : \left[-x\left(ax + b\right)\right]\right\}$$

We can also use greek leters to define symbols by using their english names. For example:

```
In [8]: sp.var('theta')       # creates a symble and assgins it to a variable with the same name
        sp.sin( theta )
```

Out[8]:

$$\sin\left(\theta\right)$$

And we can also create expressions from a `string` using `sympify()`:

```
In [9]: mySecondExpression = sp.sympify('A*cos(phi)')
        mySecondExpression
```

Out[9]:

$$A\cos\left(\phi\right)$$

```
In [10]: mySecondExpression.args
```

Out[10]:

$$\left(A, \quad \cos{\left(\phi \right)}\right)$$

```
In [11]: print( mySecondExpression, type(mySecondExpression) )
         print( mySecondExpression.args[0], type(mySecondExpression.args[0]))
         print( mySecondExpression.args[1], type(mySecondExpression.args[1]))
         print( mySecondExpression.args[1].args[0], type(mySecondExpression.args[1].args[0]))
```

```
A*cos(phi) <class 'sympy.core.mul.Mul'>
A <class 'sympy.core.symbol.Symbol'>
cos(phi) cos
phi <class 'sympy.core.symbol.Symbol'>
```

1.2 13.2 How to plot an expression?

Let's consider a simple expression like:

```
In [12]: import sympy as sp

         sp.var('x')
         sp.var('a')
         sp.var('b')

         y = a*x + b
```

This is the general equation of a straight line. We can assign values to `a` and `b`, and plot the result.

```
In [13]: y_plot = y.subs(a, 2).subs(b, 1)
         y_plot
```

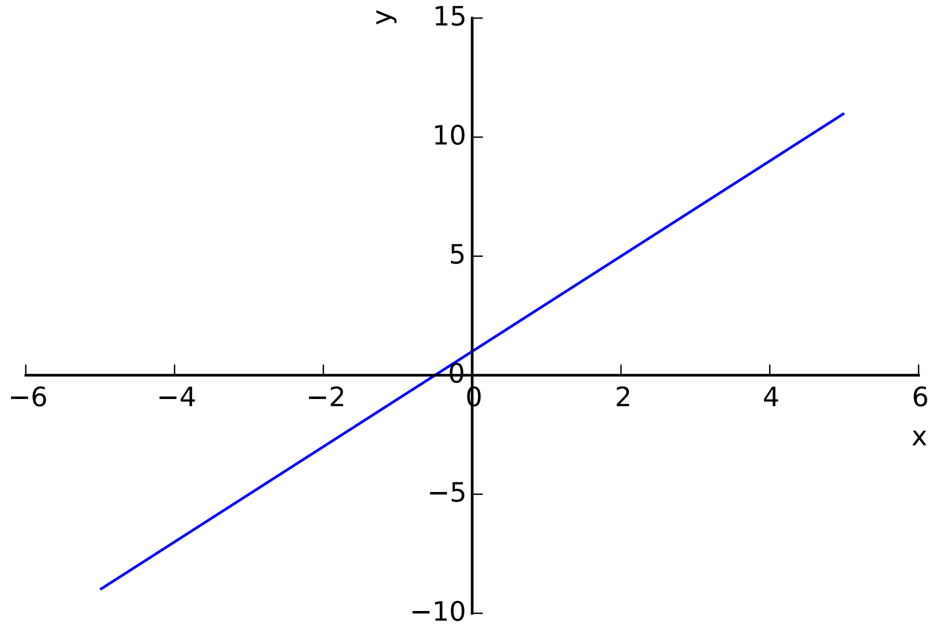
Out[13]:

$$2x + 1$$

```
In [14]: %pylab inline
         %config InlineBackend.figure_format = 'svg'

         import sympy.plotting as symplot
         drawing = symplot.plot(y_plot, (x, -5, 5), xlabel='x', ylabel='y')
```

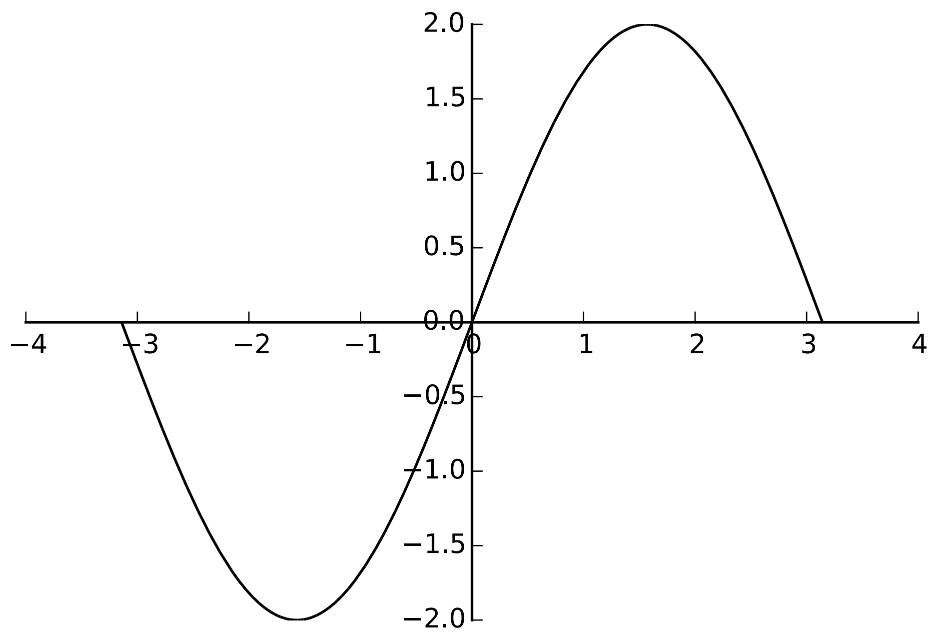
```
Populating the interactive namespace from numpy and matplotlib
```



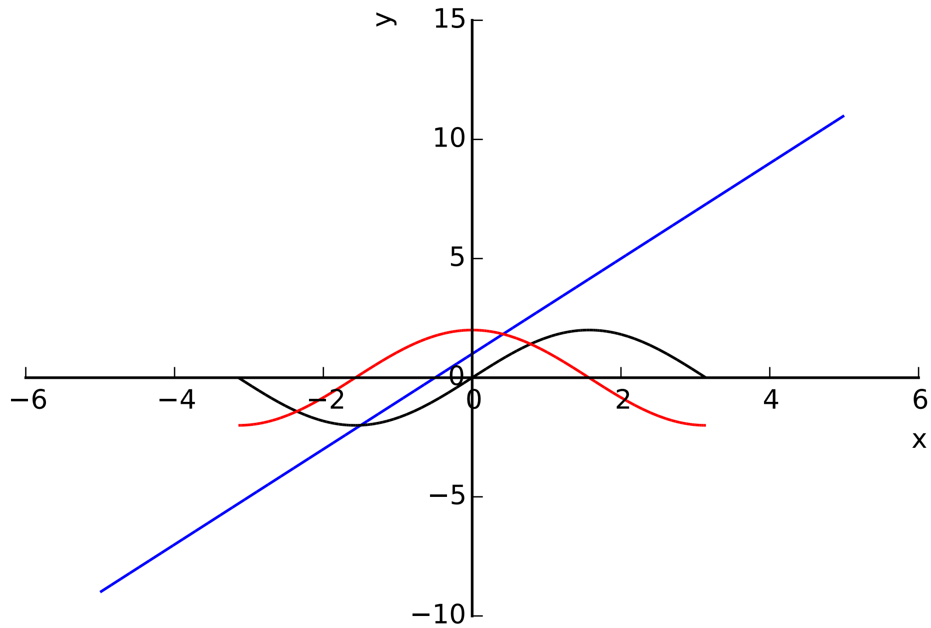
And we can draw several functions.

```
In [15]: y_sin = 2*sp.sin(x)
         y_cos = 2*sp.cos(x)

In [16]: drawing.extend( symplot.plot( y_sin, (x,-sp.pi,sp.pi), line_color='black') )
```



In [17]:
```
drawing.extend( symplot.plot( y_cos, (x,-sp.pi,sp.pi), line_color='red', show=False) )
drawing.show()
```



1.3 13.3 Working with symbolic expressions

Using sympy we can manipulate symbolic expressions in many different ways.

For instance, we can ask *sympy* to `simplify()` expressions, this is a shortterm for a number of algorithms that will try to simplify the form of a given expression:

In [18]:
```
import sympy as sp

# Trigonometric simplifications
expr = sp.sympify('sin(x)**2 + cos(x)**2')
```

In [19]:
```
print('Expression', expr)
print('Simplification:', sp.trigsimp(expr))
```

```
Expression sin(x)**2 + cos(x)**2
Simplification: 1
```

In [20]:
```
# Rational simplification
expr = sp.sympify('(x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)')
```

In [21]:
```
print('Expression', expr)
print('Simplification:', sp.ratsimp(expr))
```

```
Expression (x**3 + x**2 - x - 1)/(x**2 + 2*x + 1)
Simplification: x - 1
```

In [22]:
```
# Function simplification
expr = sp.sympify('gamma(x)/gamma(x - 2)')
```

```
In [23]: print('Expression', expr)
         print('Simplification:', sp.combsimp(expr))
```

```
Expression gamma(x)/gamma(x - 2)
Simplification: (x - 2)*(x - 1)
```

Use `help( sp.simplify )` to get more details, or Sympy tutorial: simplification

1.3.1 Factorization of polynomials

Polynomial factorization can be achieved using the fucntion `factor()`.

```
In [24]: import sympy as sp

         # Factorization of a polynomial
         p = sp.sympify("x**2 + 2*x + 1")
         print( "Polinomial: ", p )
         print( "Factoritzation: ", sp.factor(p) )

         # And also with more than one variable
         p = sp.sympify("x**2*z + 4*x*y*z + 4*y**2*z")
         print( "Polinomial: ", p )
         print( "Factoritzation: ", sp.factor(p) )
```

```
Polinomial:  x**2 + 2*x + 1
Factoritzation:  (x + 1)**2
Polinomial:  x**2*z + 4*x*y*z + 4*y**2*z
Factoritzation:  z*(x + 2*y)**2
```

For more complex polynomial, it is also possible to define the order of precedence of the symbols for factorization:

```
In [25]: sp.var('x,y,z,t')
         p = sp.sympify("x**2*z + 4*x*y*z + 4*y**2*z + x*y + t*z*x")
         print( "Polynomial: ", p )
         print( "Factoritzation (x, t): ", sp.factor(p, [x, t]) )
         print( "Factoritzation (y, t): ", sp.factor(p, [y, t]) )
         print( "Factoritzation (z, t): ", sp.factor(p, [z, t]) )
```

```
Polynomial:  t*x*z + x**2*z + 4*x*y*z + x*y + 4*y**2*z
Factoritzation (x, t):  t*x*z + x**2*z + x*(4*y*z + y) + 4*y**2*z
Factoritzation (y, t):  t*x*z + x**2*z + 4*y**2*z + y*(4*x*z + x)
Factoritzation (z, t):  t*x*z + x*y + z*(x**2 + 4*x*y + 4*y**2)
```

1.3.2 Expansion of polynomials

And vice-versa, We can start from some factorized form and ask sympy to `expand` all terms.

```
In [26]: import sympy as sp
         # polynomial expansion
         p = sp.sympify("(x + 1)**2")
         print( "Product: ", p )
         print( "Polynomial: ", sp.expand(p) )
```

```
Product:  (x + 1)**2
Polynomial:  x**2 + 2*x + 1
```

In some cases, `expand` may simplify the resulting expression if there are terms that cancel:

```
In [27]: p = sp.sympify("(x + 1)*(x - 2) - (x - 1)*x")
         print( "Product: ", p )
         print( "Polynomial: ", sp.expand(p) )
```

```
Product:  -x*(x - 1) + (x - 2)*(x + 1)
Polynomial:  -2
```

1.4 13.4 Substitution of symbols

One of the advantage of symbolic calculation is that we can substitute `symbols` from its current value to new ones. In sympy, this is done using the method `subs()`. The new value can be symbolic or numeric.

```
In [28]: import sympy as sp

         # We define a polynomial and then subtitute x by cos(x)
         p = sp.sympify("(x + 1)**2")
         print( "p = ", p )
         print( "p subs. = ", p.subs(x,sp.cos(x)) )

         # We can also do a numerical substitution
         q = sp.sympify("y*(x + 1)**2")
         print( "q = ", q )
         print( "q subs. = ", q.subs(x,3.) )
```

```
p =  (x + 1)**2
p subs. =  (cos(x) + 1)**2
q =  y*(x + 1)**2
q subs. =  16.0*y
```

1.5 Numeric evaluation of expressions

Given an expression, it is possible to evaluate it by assigning values to the variables (or symbols) it depends on. This is done using the `evalf()` method:

```
In [29]: import sympy as sp

         # Define an expression and evaluate it
         sp.var("x")
         expr = 2*x**2+2*x-1
         print( "Expression: ", expr )
         print( "Value for x=2: ",expr.evalf(subs={x:2}) )
```

```
Expression:  2*x**2 + 2*x - 1
Value for x=2:  11.0000000000000
```

```
In [30]: # It can also be done for expressions that depend on multiple symbols
         sp.var("y")
         expr = y + 2.*x**2
         print( "Expression: ", expr )
         print( "Value for x=2 y=5: ",expr.evalf(subs={x:2,y:5}) )
```

```
Expression:  2.0*x**2 + y
Value for x=2 y=5:  13.0000000000000
```

The method `evalf()` allows to request a given precission on the evaluation, since *sympy* accepts evaluations with arbitrary flotaing point precission.

```
In [31]: import sympy as sp

         # Some square roots with 100 digit precission
         sp.var('z')
         expr = sp.sqrt(z)
         print( "Expression: ", expr )
         print( "Value for sqrt(2) with 100 decimals: ", expr.evalf(100,subs={z:2}))
         print( "Value for sqrt(3) with 100 decimals: ", expr.evalf(100,subs={z:3}))
```

```
Expression:  sqrt(z)
Value for sqrt(2) with 100 decimals:  1.414213562373095048801688724209698078569671875376948073176679737
Value for sqrt(3) with 100 decimals:  1.732050807568877293527446341505872366942805253810380628055806979
```

```
In [32]: print(type(expr.evalf(100,subs={z:3})))
```

```
<class 'sympy.core.numbers.Float'>
```

Using `evalf()` one can calculate some mathematical constants with arbitrary precission. For instance, the value of π or e with 1000 digits by using their corresponding *sympy* objects.

```
In [33]: import sympy as sp

         print( "pi = ", sp.pi.evalf(1000) )
         print( "e = ", sp.exp(1).evalf(1000) )
```

```
pi =  3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211
e =  2.71828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516
```

1.6 13.5 Solving equations

Sympy provides algebraic solution of equations by using the `solve()` method.

Note: `solve()` will assume that the given expression is made equal to zero to produce the equation to solve.

1.6.1 Example 1: polynomial equations

$$x^3 + 2x - 2 = 0$$

```
In [34]: import sympy as sp

         p = sp.sympify("x**3 + 2*x - 2")

         # Solving
         solutions = sp.solve(p, x)
         print( "Polinomi: ",p )
         print( "Solution: ", solutions )
         solutions
```

```
Polinomi:  x**3 + 2*x - 2
Solution:  [(-1/2 - sqrt(3)*I/2)*(1 + sqrt(105)/9)**(1/3) - 2/(3*(-1/2 - sqrt(3)*I/2)*(1 + sqrt(105)/9)
```

Out[34]:

$$\left[\left(-\frac{1}{2} - \frac{\sqrt{3}i}{2}\right)\sqrt[3]{1 + \frac{\sqrt{105}}{9}} - \frac{2}{3\left(-\frac{1}{2} - \frac{\sqrt{3}i}{2}\right)\sqrt[3]{1 + \frac{\sqrt{105}}{9}}},\quad -\frac{2}{3\left(-\frac{1}{2} + \frac{\sqrt{3}i}{2}\right)\sqrt[3]{1 + \frac{\sqrt{105}}{9}}} + \left(-\frac{1}{2} + \frac{\sqrt{3}i}{2}\right)\sqrt[3]{1 + \frac{\sqrt{105}}{9}},\quad -\frac{2}{3\sqrt[3]{1 + \frac{\sqrt{105}}{9}}} + \sqrt[3]{1 + \ldots}\right.$$

We can use the `simplify()`:

```
In [35]: [ i.simplify() for i in solutions ]
```

Out[35]:

$$\left[\frac{\sqrt[3]{3}\left(8\sqrt[3]{3}-\left(1+\sqrt{3}i\right)^2\left(9+\sqrt{105}\right)^{\frac{2}{3}}\right)}{6\left(1+\sqrt{3}i\right)\sqrt[3]{9+\sqrt{105}}}, \quad \frac{\sqrt[3]{3}\left(8\sqrt[3]{3}-\left(1-\sqrt{3}i\right)^2\left(9+\sqrt{105}\right)^{\frac{2}{3}}\right)}{6\left(1-\sqrt{3}i\right)\sqrt[3]{9+\sqrt{105}}}, \quad \frac{\sqrt[3]{3}\left(-2\sqrt[3]{3}+\left(9+\sqrt{105}\right)^{\frac{2}{3}}\right)}{3\sqrt[3]{9+\sqrt{105}}} \right]$$

Or we can get its numeric value using `evalf()`:

```
In [36]: [ i.evalf() for i in solutions ]
```

Out[36]:

$$[-0.385458498529624 - 1.56388451052696i, \quad -0.385458498529624 + 1.56388451052696i, \quad 0.770916997059248]$$

1.6.2 Example 2: trigonometric equations

$$\cos(x) + \sin(x) = 0$$

```
In [37]: import sympy as sp
         sp.init_printing()

         # Let's define the expression
         e = sp.sympify("cos(x)+sin(x)")

         # and let sympy solve it
         print( "Polinomi: ", e )
         print( "Solution: ", sp.solve(e, x) )
         sp.solve(e, x)
```

```
Polinomi:  sin(x) + cos(x)
Solution:  [-pi/4, 3*pi/4]
```

Out[37]:

$$\left[-\frac{\pi}{4}, \quad \frac{3\pi}{4} \right]$$

1.6.3 Example 3: Systems of equations

If we want to solve a system of equations, they must be passed to `solve()` as a list, both for the equations and the variables to solve.

```
In [38]: import sympy as sp
         sp.init_printing()

         sp.var("x")
         sp.var("y")

         # We solve 2 systems, a lineal and a trigonometric one

         sol1 = sp.solve([x + y - 3, x - y - 1], [x, y])
         sol2 = sp.solve([sp.sin(x) + y , sp.cos(x) - y - 1], [x, y])
         sol1,sol2
```

Out[38]:

$$\left(\left\{x:2,\quad y:1\right\},\quad \left[\left(0,\quad 0\right),\quad \left(\frac{\pi}{2},\quad -1\right),\quad \left(\frac{\pi}{2},\quad -1\right)\right]\right)$$

Currently supported equations by *sympy* are: - univariate polynomial, - transcendental - piecewise combinations of the above - systems of linear and polynomial equations - sytems containing relational expressions.

Note: If `solve` returns `[]` or raises NotImplementedError, it doesn't mean that the equation has no solutions. It just means that it couldn't find any. Often this means that the solutions cannot be represented symbolically. For example, the equation $x = cos(x)$ has a solution, but it cannot be represented symbolically using standard functions.

In [39]:
```
import sympy as sp
sp.init_printing()

sp.var('x')

sp.solve(x - sp.cos(x), 'x')
```

```
    ---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)

    <ipython-input-39-7a2e0ba318ea> in <module>()
      4 sp.var('x')
      5
----> 6 sp.solve(x - sp.cos(x), 'x')


    /opt/local/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages/sympy/s
    899     ###########################################################################
    900     if bare_f:
--> 901         solution = _solve(f[0], *symbols, **flags)
    902     else:
    903         solution = _solve_system(f, symbols, **flags)


    /opt/local/Library/Frameworks/Python.framework/Versions/3.4/lib/python3.4/site-packages/sympy/s
   1403     if result is False:
   1404         raise NotImplementedError(msg +
-> 1405         "\nNo algorithms are implemented to solve equation %s" % f)
   1406
   1407     if flags.get('simplify', True):


    NotImplementedError: multiple generators [x, cos(x)]
No algorithms are implemented to solve equation x - cos(x)
```