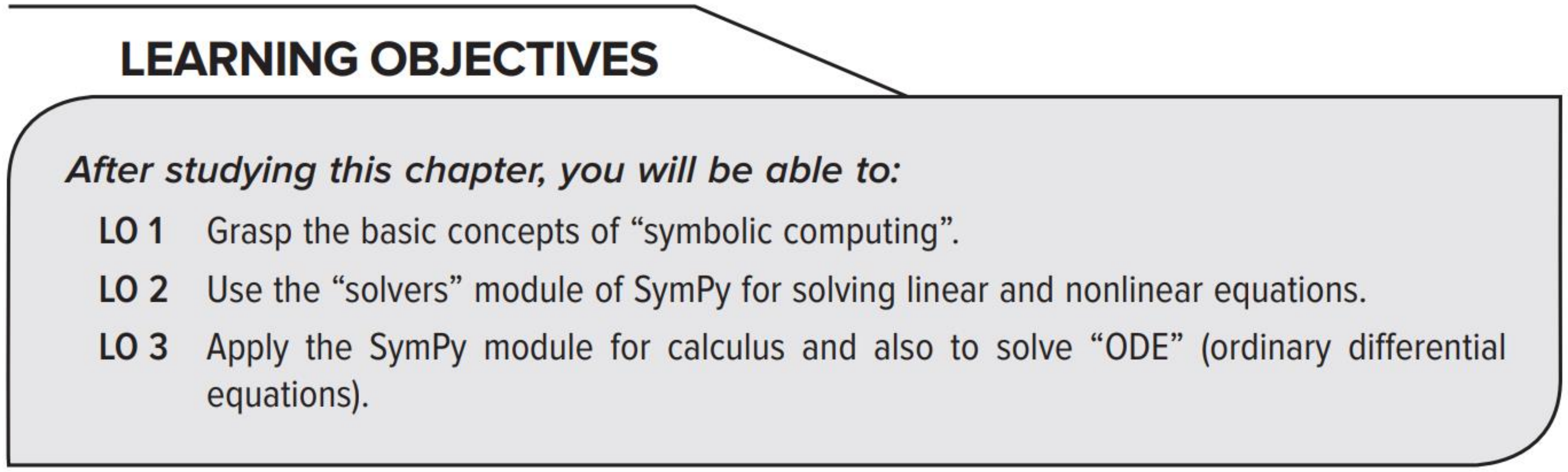
Because learning changes everything.®

TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT Chapter 19: SymPy

SymPy – Learning Objectives



LEARNING OBJECTIVES

After studying this chapter, you will be able to:

LO 1 Grasp the basic concepts of “symbolic computing”.

LO 2 Use the “solvers” module of SymPy for solving linear and nonlinear equations.

LO 3 Apply the SymPy module for calculus and also to solve “ODE” (ordinary differential equations).

Basics of SymPy -- The “symbols()” function -- 1

In SymPy there is a function “symbols()” by which you can declare a variable to be a symbol. A variable “declared to be a symbol” using the symbols() function is like any other Python variable and, therefore, must be “assigned” before they can be used. This will be clear from the following code:

```python
from sympy import symbols
x,y,z = symbols('x y z')
a_expr = (x + y)*(y + z)
print(a_expr)
# Output
(x + y)*(y + z)
```

Note: The delimitation between the symbols can be white space or even comma. So you can write x,y,z = symbols (‘x, y, z’) also.

Basics of SymPy -- The “symbols()” function -- 2

- To help the programmer write the code faster, the symbols() function supports what is called the “range index”.
- The “range index” is indicated by a colon, i.e., (:). Further the “type of range” is determined by the “type of character” to the right of the colon.
- This will be clear from following code:

```python
print(symbols('x:5'))
print(symbols('x10:15'))
```

```
# Output. . .
(x0, x1, x2, x3, x4)
(x10, x11, x12, x13, x14)
```

- Note: C, O, S, I, N, E, and Q are special variables with predefined meanings.
- For instance, I and E stand for the imaginary unit and Euler’s number, respectively.
- Therefore, C, O, S, I, N, E and Q should not be used as symbols in SymPy.

Basics of SymPy -- The “expand()” and “factor()”function

The SymPy library also has expand and factor functions to expand/ factorize the expression as shown in the following code:

```python
from sympy import expand, factor
exp1 = (x + y)*(x + y)*(y + z)
exp2 = expand(exp1)
print('exp1 on expansion->', exp2)
exp3 = factor(exp2)
print('exp2 on factorization->', exp3)
```

```
# Output. . .
exp1 on expansion-> x**2*y + x**2*z + 2*x*y**2 + 2*x*y*z + y**3 + y**2*z
exp2 on factorization-> (x + y)**2*(y + z)
```

Note: The factor function is able to convert (x+y)*(x+y) into (x+y)2**

Importing symbols from module sympy.abc

- **The SymPy library has a module abc and one can directly import symbols from this module also.**
- **This becomes clear if you look at the docstring of the abc module, which is also shown below (On Jupyter):**

```
import sympy.abc
?abc
```

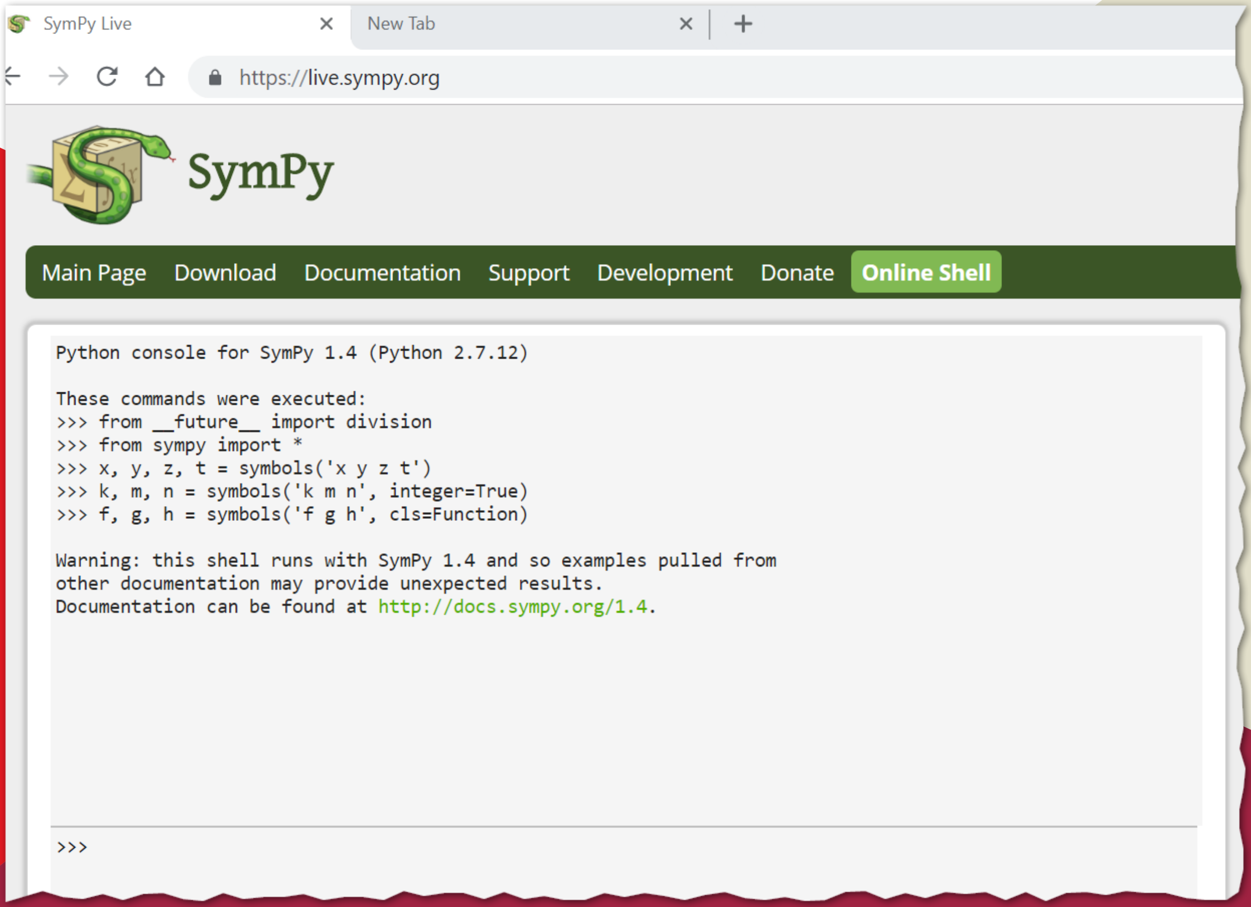
```
# OUTPUT (Truncated and modified)
This module exports all latin and greek letters as Symbols, so you can
conveniently do

    >>> from sympy.abc import x, y

instead of the slightly more clunky-looking

    >>> from sympy import symbols
    >>> x, y = symbols('x y')
```

From above you can see that you can import symbols like x and y directly from the sympy.abc module. So this is a kind of a short cut available to you

SymPy online shell

SymPy is provided online in a browser at https://live.sympy.org/ .

So you can try out SymPy even if you don’t have Python/ SymPy installed on your system.



```
Python console for SymPy 1.4 (Python 2.7.12)

These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)

Warning: this shell runs with SymPy 1.4 and so examples pulled from
other documentation may provide unexpected results.
Documentation can be found at http://docs.sympy.org/1.4.

>>>
```

Equality testing in SymPy using "==" (Not a good idea)

- Just like Python, the symbol "==" is used for equality testing in SymPy.
- But many a times a == b in SymPy may give unpredictable results.
- Therefore, it is better to test whether a - b ==0.
- This is done by a function called simplify. This will be clear from the following example:

```python
from sympy import *
a = (x + y)*(x - y)
b = x**2 - y**2
print(' Is a == b?', a == b)
print('is simplify(a - b) ==0?', simplify(a - b) ==0)
```

```
# Output. . .
is a == b? False
is simplify(a - b) ==0? True
```

Numeric types in SymPy - - 1

- SymPy uses its own “classes” for integers and rational numbers.
- SymPy has two numeric types: (1) rational number and (2) real number. The class Rational(m, n) is used to “construct” a rational number of type m/n. Note that m and n are integers. So Rational (1,2) represents ½ and not 0.5.
- Note that Rational (m,n) is not the only way of constructing rational numbers.
- Table 19.1 gives some common ways in which rational numbers can be constructed using the rational() class.

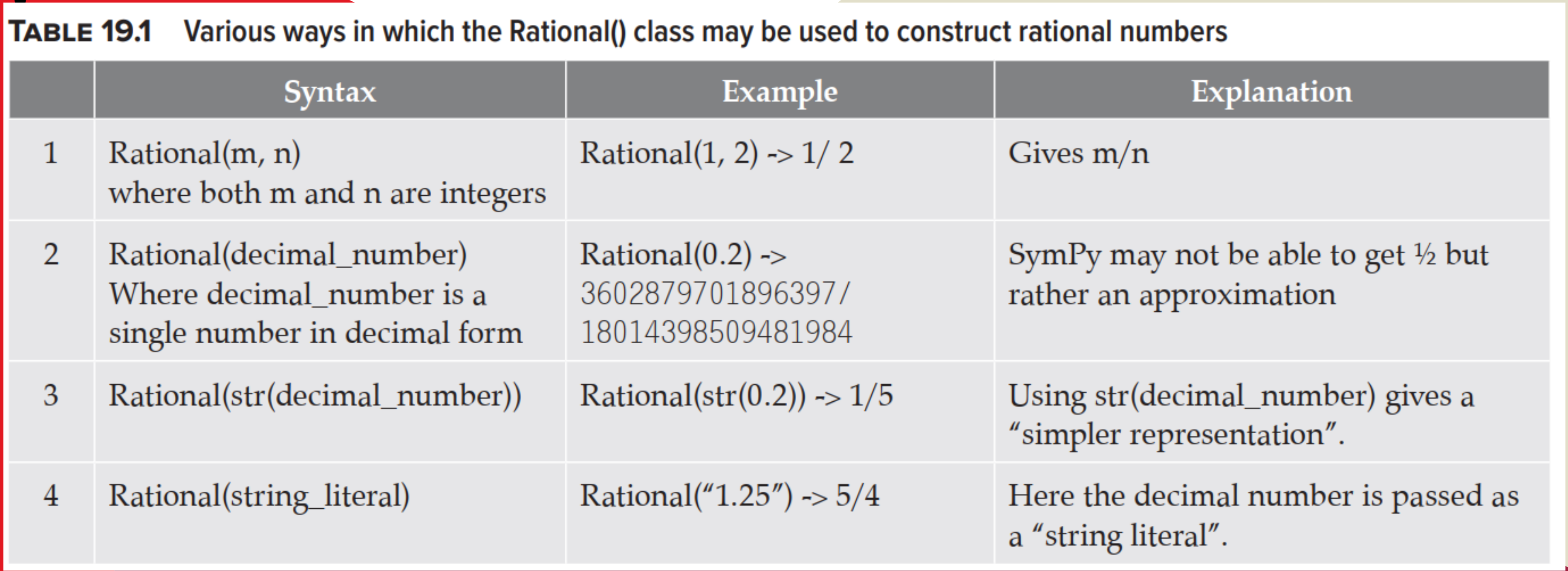


TABLE 19.1 Various ways in which the Rational() class may be used to construct rational numbers

	Syntax	Example	Explanation
1	Rational(m, n) where both m and n are integers	Rational(1, 2) -> 1/ 2	Gives m/n
2	Rational(decimal_number) Where decimal_number is a single number in decimal form	Rational(0.2) -> 3602879701896397/ 18014398509481984	SymPy may not be able to get ½ but rather an approximation
3	Rational(str(decimal_number))	Rational(str(0.2)) -> 1/5	Using str(decimal_number) gives a “simpler representation”.
4	Rational(string_literal)	Rational(“1.25”) -> 5/4	Here the decimal number is passed as a “string literal”.

Numeric types in SymPy - - 2

- When two objects are linked by an operator and they are not of the same type, then some implicit type casting takes place.
- So, you may have two SymPy objects, two Python objects or one of each. The problem arises when you want to write a formula x + ¾. Here, ¾ will evaluate to 0.75.
- SymPy also has an Integer class which converts a Python integer to a SymPy integer. The way to write this is as follows:

```python
from sympy import *
a = x + 3/4
print(a) # Gives x + 0.75
b = x + Rational(3,4)
print(b)# Gives x + 3/4
c = x + Integer(3)/ Integer(4)
print(c) # Gives x + ¾
```

Basics of SymPy 2

This section deals with certain more advanced SymPy concepts such as:-

- **substitution in expression,**
- **evaluation of SymPy strings,**
- **the singleton class,**
- **the various types of functions in SymPy and**
- **the Lambda class.**

Substitution in a SymPy expression

In SymPy it is possible to replace a symbol with another symbol or SymPy object or Python object. The following example will clarify the concept:

```python
from sympy import *
x, y, z = symbols('x, y, z')
a, b, c = symbols('a b c')
m = x**2 + y*4 + sin(z)
n = m.subs([(x, a), (y, 3), (z, 2*c)])
print(n) # Gives a**2 + sin(2*c) + 12
```

In the above example:-

- x is substituted by a
- y by 3 and
- z by 2*c

Convert Python strings to SymPy expression and evaluating it (Functions sympify and evalf)

SymPy has a function sympify (not simplify). Sympify can be used to convert strings to expressions and a function evalf() to get its value. Following example clarifies this:

```python
from sympy import *
#x, y, z = symbols('x y z')
a = 'x**2 + log(y) + cos(z)'
b = sympify(a)
print(b) # a -> x**2 + log(y) + cos(z). Note in sympy log is natural log ie ln
c = b.subs([(x,2), (y, 100), (z, 0)])
print(c) # Prints-> log(100) + 5
d = c.evalf()
print(d) # Output-> 9.60517018598809
```

Using sympify() to “convert” Python data type to SymPy data type

- **sympify can also be used to convert numbers of Python data type to SymPy data type.**
- **For instance, the number 1 in Python is an integer, while in SymPy it may be a rational. ½ in Python is 0.5 while Rational(1,2) or sympify(1)/2 in SymPy is ½.**
- **This is clear from the following example:**

```
from sympy import *
print('1/2 ->', 1/2)
print('Rational(1,2)->', Rational(1,2))
print('sympify(1)/2->', sympify(1)/2)
```

```
# Output. . .
1/2 -> 0.5
Rational(1,2)-> 1/2
sympify(1)/2-> ½
```

Singleton class in SymPy

In SymPy there is a class called SingletonRegistry. Here we would not go into details but touch the essential factors:

- **Common mathematical constants are represented by singleton classes.**
- **This singleton class can be represented by capital letter S.**
- **So, S is an “instance ” of SingletonRegistry class.**
- **These constants are directly available in the SymPy namespace. So, if you do an import sympy*, then S becomes available and through S, the constants also become available. Therefore, constants, such as “pi”,”e” can be accessed as S.pi, S.e, etc.**
- **Infact, S can also act as a shortcut for sympify function. Remember that in SymPy if you want to get ½ as ½ and not 0.5, then use Rational(1,2). You can also use S(1)/2.**
- **The function, “sympy.core.sympify.sympify” can also be used to create singletons. Note that “S(1)” is the same thing as “sympify(1)”.**

[- - - - - -Example code is on next slide]

Singleton class in SymPy -- Example

The following script shows that S(1)/2 is same as sympify(1)/2

```
import sympy
from sympy import *
print('using S(1)/2->', S(1)/2)
print('Using sympify(1)/2->', sympify(1)/2)
```

```
# OUTPUT
using S(1)/2-> 1/2
Using sympify(1)/2-> ½
```

Some common singletons of SymPy

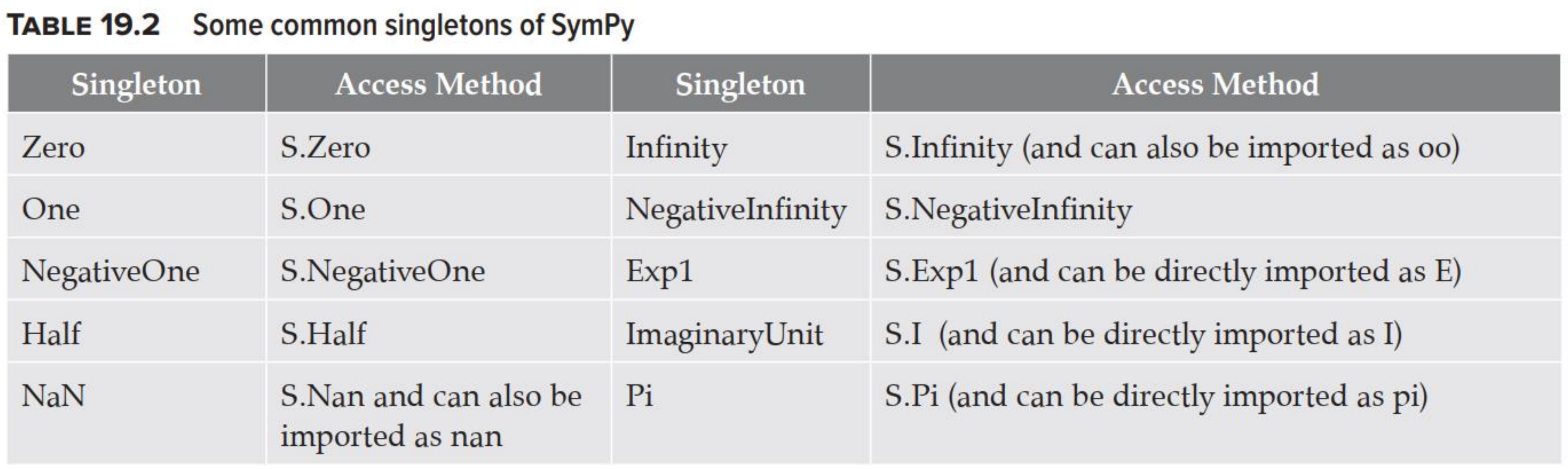


TABLE 19.2 Some common singletons of SymPy

Singleton	Access Method	Singleton	Access Method
Zero	S.Zero	Infinity	S.Infinity (and can also be imported as oo)
One	S.One	NegativeInfinity	S.NegativeInfinity
NegativeOne	S.NegativeOne	Exp1	S.Exp1 (and can be directly imported as E)
Half	S.Half	ImaginaryUnit	S.I (and can be directly imported as I)
NaN	S.Nan and can also be imported as nan	Pi	S.Pi (and can be directly imported as pi)

The following script shows the use of singletons

```
1 from sympy import *
2 print(1/2) # gives 0.5
3 print(Rational(1,2)) # gives 1/2
4 print(S(1)/2) # gives 1/2
5 print(pi, S.Pi) # Both pi and S.Pi give pi
6 print(E, S.Exp1) # Both E and S.Exp1 give E
7 print(oo, S.Infinity)# Both oo and S.Infinity give oo
```

Line4: Here S(1)/2 is Rational and so you get 1/2 and not 0.5

Functions in SymPy

As per the SymPy documentation, there are three types of functions available in SymPy:

- Defined functions (in the sense that they can be evaluated) such as exp or sin; they have a name and a body: f = exp
- Undefined functions which have a name but no body. Undefined functions can be defined using a Function class as follows: f = Function('f'). (the result will be a Function instance).
- Anonymous function (or lambda function) which have a body (defined with dummy variables) but have no name. Example: For 1 variable→ f = Lambda(x, exp(x)*x), for 2 variables→ f = Lambda((x, y), exp(x)*y).
- The fourth type of functions are composites, such as (sin + cos)(x); these work in SymPy core, but are not yet part of SymPy."

You can see how the undefined function is implemented in SymPy in the following code:

[- - - Script on next slide]

Functions in SymPy – Example script

```python
from sympy import *
x,y = symbols('x y')
f = Function('f')
g = Function('g')(y)
h = Function('h')(x,y)
print('f->', f, 'g->', g, 'h->', h)# f-> f g-> g(y) h-> h(x, y)
j = f(x)
print(f(x), f(y))# Prints f(x) f(y)
```

Lambda class in SymPy

- SymPy Lambda class is different from lambda function in Python. A good explanation of Lambda class and its use is given in SymPy docs. (See:- https://docs.sympy.org/0.7.5/_modules/sympy/core/function.html#Lambda)
- "Lambda(x, expr) represents a lambda function similar to Python's lambda x: expr'. A function of several variables is written as Lambda((x, y, ...), expr)."
- The docstring of the Lambda class gives a number of examples on how to use this Lambda class. On Jupyter, the docstring can be accessed using (?).
- A modified version of the docstring of SymPy's Lambda class is shown on next slide and some relevant lines of code are also explained:

[- - - - Docstring of Lambda class of SymPy shown on next slide]

Lambda class in SymPy - - Docstring

```
1   from sympy import Lambda
2   ?Lambda
```

```
3   # OUTPUT (Truncated and modified)
4   Init signature: Lambda(variables, expr)
5   Docstring:
6   Lambda(x, expr) represents a lambda function similar to Python's 'lambda x: expr'. A function of
7   several variables is written as Lambda((x, y, ...), expr).
8   
9   A simple example:
10  >>> from sympy import Lambda
11  >>> from sympy.abc import x
12  >>> f = Lambda(x, x**2)
13  >>> f(4)
14  16
15  
16  For multivariate functions, use:
17  >>> from sympy.abc import y, z, t
18  >>> f2 = Lambda((x, y, z, t), x + y**z + t**z)
19  >>> f2(1, 2, 3, 4)
20  73
21  
22  A handy shortcut for lots of arguments:
23  >>> p = x, y, z
24  >>> f = Lambda(p, x + y*z)
25  >>> f(*p)
26  x + y*z
```

Line10-14:- This shows how a single variable namely x is used in a SymPy Lambda class
Line16-20:- This block of script shows how the Lambda class can be used on more than 1 variables. In fact here 3 variables namely x, y and t are used.
Line22-26:- This block of script shows how a number of variables can be "packed" into a single variable name and then "unpacked" when giving as an argument to an object of the Lambda class.

SymPy sets - - 1

- The SymPy module has a base class called Set.
- This class Set of SymPy is not meant to be used as a container of items like the “set” of python.
- Remember that in Python “set” is a container.
- In SymPy, you subclass this base class Set to get a number of other “types” of Set.
- Some examples of sub-classed sets (From the base class Set) are:- FiniteSet, Interval, ConditionSet and so on.
- These sets (sub-classed from base class Set) can be of following 2 types:-
  - Those which can act as “containers” and
  - Those which cannot act as containers
- Some of these sub-classed sets like FiniteSet can be used as “containers” but others like “Interval” cannot be used as “containers.

[- - - Continued on next slide]

SymPy sets - - 2

- Thus a major difference between a Python “set” and a SymPy Set (or its sub-classed sets) is that some (though not all) of the SymPy sub-classed sets are not containers.
- The reason for having some SymPy sets which are not containers is that in Sympy, many times, the solution to some equation (or set of equations) may be a “finite” or an “infinite” interval.
- However a “container” cannot be used to represent an interval because an interval can have an infinite numbers.
- So in order to cater to this need of representing “intervals”, SymPy has certain types of sets which are not containers. (This will become clear when such types of Sets are discussed later)

SymPy sets - - 3

- Similarly a Python “set” can only be used to contain “discrete” items like numbers.
- A Python set cannot be used to represent an “interval” because an interval say (1,2) would contain all the infinite numbers between 1 and 2.
- Therefore, SymPy has this scheme where it has different “types” of sets (All Subclassed from the base class Set).
- These different types of sets in SymPy do have some common methods also.
- Note: In general SymPy sets are primarily meant to work with numbers and therefore <u>should not be used to “contain” other objects</u>.

SymPy sets - - Table of common types

	Set type	Explanation
1	FiniteSet	The FiniteSet class represents a finite set of discrete numbers. This means that the members are discrete (Not continuous, like in an interval).
2	Interval	The Interval class can be used to represent a real interval in the form of a set. The Interval class returns an interval with end points "start" and "end". The Interval class has a constructor parameter left_open. This parameter by default is false. If however you have left_open = True, then the interval will be open on the left. Similarly, there is a parameter right_open which is also false by default. If however you have right_open = True, then the interval will be open on the right.
3	EmptySet	The EmptySet represents the empty set. The empty set is also available as a singleton,that is, as S.EmptySet.
4	Intersection	The Intersection class of SymPy can be used to represent an intersection of sets as an “Intersection set”.
5	Union	The Union class is used to represent the union of “self” and “other” in the form of a “Union set”. As a shortcut it is possible to use the “+” operator:
6	ConditionSet	The ConditionSet class is used to represent a set of those elements which satisfy a given condition. The representation is done in the form of a “ConditionSet set”.
7	ImageSet	The ImageSet gives the Image of a set under a mathematical function.
8	ProductSet	The ProductSet class is used to represent a Cartesian Product of Sets.
9	ComplexRegion	The ComplexRegion class is used to represent the Set of all Complex Numbers.
10	Complement	The Complement class is used to represent the set difference or relative complement of a set with another set. A−B={x∈A|x not in B}

FiniteSet

The FiniteSet can “contain” numbers. The following code shows how FiniteSet() works:

```
from sympy import *
f1 = FiniteSet(1,2,3,4)  # Create FiniteSet by giving it some items
print(f1)
a_list = [2,3,4,5]
f2 = FiniteSet(*a_list)  # Create FiniteSet from a list
print(f2)
a_tup = (5,6,7,8)
f3 = FiniteSet(*a_tup)  # Create FiniteSet from a tuple
print(f3)
is_member = 5in f3 #Check whether an object member or not
print(is_member) # True
```

- **FiniteSet is important in SymPy because many a times the solutions to equations/ inequalities may be given in the form of a FiniteSet. Suppose you want to access the individual items of a FiniteSet, how do you do this? You cannot in general use an index over a set because the items in a set are not ordered.**
- **The SymPy sets provide an iter method to access individual items in a set. But not all sets are iterable. For instance, the Interval is not iterable because an interval would have infinite points.**

FiniteSet (How to iterate over it)

```python
from sympy import *
s1 = FiniteSet('A', 'B', 'C')
check_s1 = s1.is_iterable
print(check_s1)
if check_s1: #Should check whether iterable
    memb_s1 = iter(s1)
    print(memb_s1)
    len_s1 = len(s1)# Get number of members
    for r in range(len_s1):
        a_memb = next(memb_s1)# all iterables implement next()
        print(a_memb)
```

```
# Output. . .
True
<tuple_iterator object at 0x06343930>
A
B
C
```

Interval

The signature of Interval class, which is an extension of Python set data type is:

```
Interval(start, end, left_open=False, right_open=False)
```

So by default both the left and right intervals are not open, that is, closed. Only real end points are supported (not complex). It must also be kept in mind that Interval(a, b) with a > b will return the empty set. The following code clarifies the concept:

```
from sympy import *
print(Interval(0,1))# Equivalent to [0,1]
print(Interval(0,1, False, False)) # Equivalent to [0,1]
print(Interval(0,1, False, True))# Equivalent to [0,1). Interval.Ropen(0, 1)
print(Interval(0,1, True, True)) # Equivalent to (0,1). Interval.open(0, 1)
print(Interval(0,1, True, False))# Equivalent to (0,1]. Interval.Lopen(0, 1)
print(Interval(1,0))# EmptySet()
```

EmptySet

- Following code shows how EmptySet works.
- Intersection with an EmptySet() is always an EmptySet().

```
from sympy import *
print(S.EmptySet) # Output is EmptySet()
f1 = FiniteSet(1,2,3)
print(f1.intersect(S.EmptySet))# Intersection with EmptySet() is EmptySet()
```

Intersection

- **Intersection(Set_A, Set_B) will return the intersection set of the two sets.**
- **You can also use the method intersect as set_A.intersect(set_B).**
- **This is clear from the following code:**

```
from sympy import *
s1 = Intersection(Interval(2,4), Interval(3,5))# Use Intersection() class
print(s1)
s2 = Interval(2,4).intersect(Interval(3,5))# use intersect() method
print(s2)
```

```
# Output. . .
Interval(3, 4)
Interval(3, 4)
```

Union

Union(set_A, set_B) returns the union of “set_A” and “set_B”.

As a shortcut one can use the “+” operator for “union” of two sets.

```
1 from sympy import *
2 s1 = Union(Interval(0,1), Interval(2,3))
3 pprint(s1) # output is [0, 1] ∪ [2, 3]
4 s2 = Union(Interval(0,1), Interval(1,2))
5 pprint(s2) #Output is [0, 2]
6 s3 = Union(Interval(0,1, True, True), Interval(1,2, True, True))
7 pprint(s3)#Output is (0, 1) ∪ (1, 2)
8 s4 = Interval(0,1, True, True).union(FiniteSet(1,2))
9 pprint(s4)# Output is (0, 1] ∪ {2}
```

Line6: Both the intervals in the union are open intervals (because open is True) and don’t include 1. This is why you get (0, 1) ∪ (1, 2)and not (0,2). So the SymPy library is smart enough to know that the two sets in the union are open sets so 1 is not a member of the union.

Line8: Here the union() method is used.

ConditionSet - - 1

ConditionSet is a set that satisfies a given condition. The signature of ConditionSet on Jupyter is shown in the following example:

```
from sympy import *
?ConditionSet
```

```
Init signature: ConditionSet(sym, condition, base_set)
Docstring:
Set of elements which satisfies a given condition.
{x | condition(x) is True for x in S}
```

Note that the constructor to ConditionSet takes 3 parameters namely:- (1) sym (2) condition and (3) base_set. Each of these 3 parameters are explained below:-

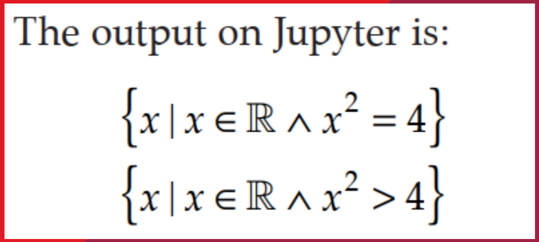
- **sym is the symbol or variable for which the ConditionSet is evaluated.**
- **Condition may be an equality or inequality**
- **base_set is the domain over which the condition is evaluated.**

ConditionSet - - 2

The use of ConditionSet will become clear from the code below :-

```
1 from sympy import *
2 from sympy.abc import x
3 s1 = ConditionSet(sym = x,   # symbol is x
4                   condition = Eq(x**2, 4 ),# equation is x **2 = 4
5                   base_set = S.Reals)  # The solution is over real numbers
6 pprint(s1)
7 s2 = ConditionSet(x, x**2>4 , S.Reals)
8 pprint(s2)
```

- **Explaination:-**
- **Line3-5:- Here the ConditionSet is created. The symbol used is "x". The condition used is an equality ie $x^2 = 4$. The domain used is real numbers**
- **Line7:- Here again a ConditionSet object is created. Here the symbol is again "x". The condition used is an inequality that is $x^2 > 4$. The domain is again real numbers.**
- **The output on Jupyter is**



The output on Jupyter is:

$$\{x \mid x \in \mathbb{R} \wedge x^2 = 4\}$$

$$\{x \mid x \in \mathbb{R} \wedge x^2 > 4\}$$

Complement

The following script shows how the set type Complement is used

```
from sympy import *
s1 = FiniteSet(1,2,3,4)
s2 = FiniteSet(2,3)
s3 = Complement(s1, s2)
pprint(s3) # Output {1, 4}
```

ImageSet (along with imageset function) - - 1

- When a function say f is applied to a set say A = {x| x ∈S} , then a new set say B is created such that B = {f(x)| x ∈S}.
- Then this set B is the ImageSet of the set A.
- One advantage of using a set to represent the image is that it could represent those images which have infinite entries.
- For instance, equations like sin(x) = 0; cos(x) = ½ would have infinite solutions.
- Such images, that is, solutions can be easily represented by a set.

[- - - Continued on next slide]

ImageSet (along with imageset function) - - 2

- **The concept of ImageSet is explained very well in the SymPy docs as follows:**
- **“ Image of a set under a mathematical function. The transformation must be given as a Lambda function which has as many arguments as the elements of the set upon which it operates, for instance, 1 argument when acting on the set of integers or 2 arguments when acting on a complex region. This function is not normally called directly, but is called from `imageset`”.**
- **This means that you can create the image of a set of values using either the ImageSet class or the imageset() function. The signature of ImageSet class is:**

```
Init signature: ImageSet(Lamda, base_set)
```

```
Signature: imageset(*args)
Docstring:Return an image of the set under transformation ``f``.
If this function can't compute the image, it returns an
unevaluated ImageSet object.
.. math::
    { f(x) | x \in self }
```

Imageset Example script

```
from sympy import imageset, Lambda, symbols, S
x,y = symbols('x y')
b = FiniteSet(1,2,3,4,8,16,36,49)# For intersection
# -------------Use sympy Lambda----------------
a = imageset(Lambda(x, 2*x), S.Integers)#Create imageset using Lambda from sympy
print('a->', a)
c = a.intersect(b)
print('c->', c)
print('c type->', type(c))
# ------------Use python lambda-----------------
f = lambda x: x**2# Normal lambda of python
g = imageset(f, S.Integers) # Create imageset using normal python lambda
h = g.intersect(b)
print('g->', g)
print('h->', h)
```

```
# Output. . .
a-> ImageSet(Lambda(x, 2*x), S.Integers)
c-> {2, 4, 8, 16, 36}
c type-><class'sympy.sets.sets.FiniteSet'>
g-> ImageSet(Lambda(x, x**2), S.Integers)
h-> {1, 4, 16, 36, 49}
```

The above script shows use of the imageset() function in 2 different ways (1) Using the Lambda class od SymPy and (2) using a normal Python lambda anonymous function.

Set operations in SymPy

Some common set functions in SymPy are shown in the following code:

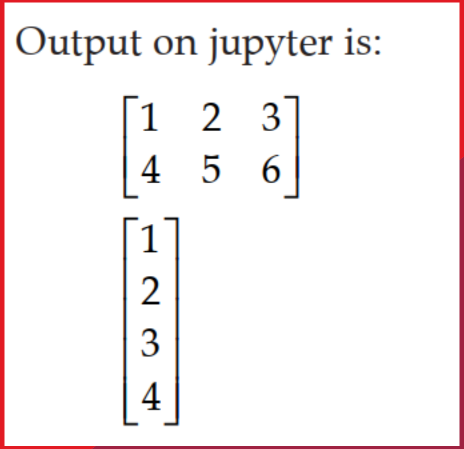
```python
from sympy import imageset, Lambda, symbols, S
x,y = symbols('x y')
s1 = FiniteSet(1,2,3)
s2 = FiniteSet(3,4,5)
s3 = s1.union(s2)
print(s3)
s4 = s1 + s2 # Can use + operator for union
print(s4)
s5 = s1.intersection(s2)
print(s5)
s6 = s1-s2 # Can use - operator for difference
print(s6)
s7 = s1*s2 # Can use * operator for cartesian product
print(s7)
s8 = s1**2# Can use ** for cartesian product with itself
print(s8)
s9 = set(s1**2) # Expand the cartesian product to list all members
print(s9)
```

```
#Output. . .
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
{3}
{1, 2}
{1, 2, 3} x {3, 4, 5}
{1, 2, 3} x {1, 2, 3}
{(1, 2), (3, 2), (1, 3), (3, 3), (3, 1), (2, 1), (2, 3), (2, 2), (1, 1)}
```

Matrices

- SymPy provides extensive support for Matrices.
- It has a Matrix class which can take a list of lists.
- Each inner list is one row of the matrix.
- However, if you give only one list, that is, not a list of lists, then the list will be treated as a column matrix.
- This is clear from the following code:

```
M1 = Matrix([[1,2,3], [4,5,6]]) # 2 x 3 matrix
display(M1)
M2 = Matrix([1,2,3,4]) #  4 x 1 Column matrix
display(M2)
```



Output on jupyter is:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Example code shows how to use various methods of the Matrix class of SymPy

```python
from sympy import *
from sympy.plotting import plot
from IPython.display import display
init_printing(use_latex='mathjax')
M1 = Matrix([[1,2,3], [4,5,6]]) # 2 x 3 matrix
print('M1 shape->', M1.shape) # Shape is 2 x 3
# M1 doesnt change on inserting row. New matrix created
M1_new = M1.row_insert(1, Matrix([[0,0,1]])) #Insert row [0,0,1]at index 1
print('M1_new ->', M1_new)
M2 = Matrix([[4,5,6], [7,8,9]]) # Another 2 x 3 matrix
M3 = M1 + M2
print('M1 + M2->', M3)
M4 = Matrix([[2,4], [3,5], [4,6]])# 3 x 2 matrix
M5 = M1*M4 # Multiply
print('M1*M4->', M5) # Note M5 is a square matrix so can find inverse

M6 = M5**(-1) # Get inverse of M5
print('Inverse of M5->', M6)
print('M5*M6->', M5*M6) #Confirms that M6 is inverse of M5
M7 = M1.T # M1 is 2 x 3. M7 is 3 x 2
print('Transpose of M1->', M7)
```

```
# Output
M1 shape-> (2, 3)
M1_new -> Matrix([[1, 2, 3], [0, 0, 1], [4, 5, 6]])
M1 + M2-> Matrix([[5, 7, 9], [11, 13, 15]])
M1*M4-> Matrix([[20, 32], [47, 77]])
Inverse of M5-> Matrix([[77/36, -8/9], [-47/36, 5/9]])
M5*M6-> Matrix([[1, 0], [0, 1]])
Transpose of M1-> Matrix([[1, 4], [2, 5], [3, 6]])
```

The code on left creates a 2 x 3 Matrix. Then it uses methods/ attributes of Matrix class like:-

- **M1.shape**
- **M1.row_insert()**
- **M1 * M2**
- **M1 ** (-1)**
- **M1.T**

The Equality class and Eq

- **SymPy has a class Equality. This class can compare two objects. It has an alias Eq.**
- **So you may use Equality or Eq.**
- **The signature of Equality (or Eq) on Jupyter is as follows:**

```
Init signature: Eq(lhs, rhs=0, **options)
Docstring:
An equal relation between two objects. Represents that two
objects are equal.  If they can be easily shown to be
definitively equal (or unequal), this will reduce to True (or
False).  Otherwise, the relation is maintained as an
unevaluated Equality object.  Use the ``simplify`` function
on this object for more nontrivial evaluation of the equality
relation. As usual, the keyword argument ``evaluate=False``
can be used to prevent any evaluation.
```

The solvers module of SymPy

The solvers module in SymPy implements methods for solving equations. You should:

- **Use solveset() for solving equations with one variable. Note that it is better to use the solveset() method rather than the earlier method solve().**
- **Use sympy.solvers.solveset.linsolve() to solve system of linear equations.**
- **Use sympy.solvers.solveset.nonlinsolve() method to solve a system of non-linear equations.**
- **It is preferable to use solveset() rather than solve(). To know why see:- https://docs.sympy.org/latest/modules/solvers/solveset.html#module-sympy.solvers.solveset**

solveset()

- SymPy provides a function solveset(). This function is called solveset() because it provides a "set of solutions" and not just a single solution. Also solveset() is able to take care of different types of outputs.
- When you solve an equation or an inequality, for a single variable, you may have following possibilities:
  - No solution (this can be represented by the EmptySet).
  - Finitely many solutions (can be represented by FiniteSet).
  - Solutions may consist of intervals (can be represented by Interval).
  - Infinitely many solutions. The solutions may be countably finite (i.e., discrete infinite) or uncountably infinite (i.e., continuous infinite) (can be represented by using the ImageSet module).
  - Unusual solutions.

Docstring of solveset()

```
?solveset
Signature: solveset(f, symbol=None, domain=S.Complexes)
Docstring:Solves a given inequality or equation with set as output
Parameters
==========
f : Expr or a relational.
    The target equation or inequality
symbol : Symbol
    The variable for which the equation is solved
domain : Set
    The domain over which the equation is solved
Returns
=======
Set
    A set of values for`symbol`for which `f` is True or is equal to
    zero. An `EmptySet` is returned if `f` is False or nonzero.
    A `ConditionSet` is returned as unsolved object if algorithms
    to evaluate complete solution are not yet implemented.
```

- **You can use solveset() to solve both equalities as well as inequalities.**
- **From the above signature of solveset(), it is clear that by default the domain over which the function is solved is complex.**

Using solveset()

Following code shows use of solvset() to solve a quadratic equation of type

$ax^2 + bx + c = 0$

```
from sympy import *
a,b,c,x = symbols('a b c x')
q1 = a*(x**2) + 2*(b*x) + c
q2 = Eq(q1, rhs = 0)
s1 = solveset(q2, x)# Default for domain = S.Complexes
print('General solution->', s1)
q_sub = q.subs([(a,1), (b, 2), (c, 5)])
s2 = solveset(q_sub, x, domain = S.Complexes)
print('Solution over Complex domain->', s2)
s3 = solveset(q_sub, x, domain = S.Reals)
print('Solution over Real domain->', s3)
```

```
# Output. . .
General solution-> {-b/a - sqrt(-a*c + b**2)/a, -b/a + sqrt(-a*c + b**2)/a}
Solution over Complex domain-> {-2 - I, -2 + I}
Solution over Real domain-> EmptySet()
```

Plotting a graph in SymPy -- 1

- You can even plot a graph of function in SymPy because SymPy provides a plot() function.
- The details of the function are not discussed here (those interested may do ?plot on Jupyter). Basic signature of plot is as follows:

```
plot(expr, range, **kwargs)
_``expr``  : Expression representing the function of single variable
``range``: (x, 0, 5), A 3-tuple denoting the range of the free variable.
```

Solving and plotting using SymPy

The following script solves for roots of a cubic $x^3 - 4x^2 + x + 6 = 0$ and plots it

```
% matplotlib inline
from sympy import *
from IPython.display import display
init_printing(use_latex='mathjax')
x = symbols('x')
exp1 = (x**3) - 4*(x**2) +x + 6
exp2 = Eq(exp1, rhs = 0)
sol1 = solveset(exp2, x)
plot(exp1, (x,-2, 4 ))
```

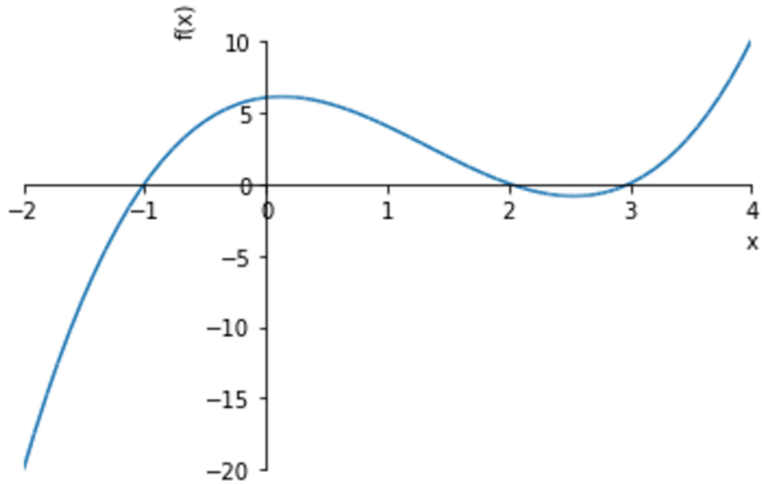


Figure 19.2: Output graph of cubic $x^3 - 4x^2 + x + 6 = 0$ on Jupyter

Using solveset() to solve the equation sin x = 0

```
exp1 = sin(x)
exp2 = 0
s1 = solveset(Eq(exp1, exp2), x)
display(s1)
```

The output on Jupyter is:

$$\{2n\pi \mid n \in Z\} \cup \{2n\pi + \pi \mid n \in Z\}$$

You can see from the output that the solveset() method is able to provide infinite set of solutions.

The linsolve() method

The linsolve() method is used to solve linear equations.

A part of the signature of linsolve() method is as follows:

```
Signature: linsolve(system, *symbols)
Docstring:
•  Solve system of N linear equations with M variables, which means
   both under - and overdetermined systems are supported.
•  The possible number of solutions is zero, one or infinite.
•  Zero solutions throws a ValueError, whereas infinite solutions
   are represented parametrically in terms of given symbols.
•  For unique solution a FiniteSet of ordered tuple is returned.
```

Using linsolve() to solve a system of 3 linear equations in 3 variables - - 1

$a_{11}\,x + a_{12}\,y + a_{13}\,z = b_1$ $a_{21}\,x + a_{22}\,y + a_{23}\,z = b_2$ $a_{31}\,x + a_{32}\,y + a_{33}\,z = b_3$	$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$	A.v = b

The first parameter to the linsolve() method is system
This parameter system can be in one of the following formats:

First format: Here you can convert the coefficients of both LHS and RHS into Matrix A and b and then give them as a tuple to the system as shown in the following code:

```
M = Matrix([a11 + a12 + a13,
            a21 + a22 + a23,
            a31 + a32 + a33])
b = Matrix([b1, b2, b3])
system = (A,b)
```

Using linsolve() to solve a system of 3 linear equations in 3 variables - - 2

Second format: Here you can provide the three linear equations (separated by comma) to the parameter system as shown in the following code:

```
# 3 linear equations are surrounded by square brackets ie [ and ]
# The linear equations are separated by comma
system = [a11*x + a12*y + a13*z = b1,
          a21*x + a22*y + a23*z = b2,
          a31*x + a32*y + a33*z = b3]
```

Further the algorithm used here is Gauss–Jordan elimination. This algorithm results in a matrix in a row echelon form. There are two possible outcomes:

- **The first possibility is that there are no solutions to the problem, that is, the "system is inconsistent". In that case the method returns an EmptySet.**
- **The second possibility is that a solution exists. Here the method returns a FiniteSet of tuple.**

Example using linsolve() - - 1

This is best understood by an example. Consider the following system of three linear equations in three variables: x, y and z
(1) x + y + z = 5, (2) 2x + 3y + 5z = 8 and (3) 4x + 5z = 2.
The process of conversion:

$\begin{aligned}1x+1y+1z&=5\\2x+3y+5z&=8\\4x+0y+5z&=2\end{aligned}$	$\begin{bmatrix}1&1&1\\2&3&5\\4&0&5\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=\begin{bmatrix}5\\8\\2\end{bmatrix}$	$\begin{bmatrix}1&1&1&5\\2&3&5&8\\4&0&5&2\end{bmatrix}$	$A=\begin{bmatrix}1&1&1\\2&3&5\\4&0&5\end{bmatrix}\ b=\begin{bmatrix}5\\8\\2\end{bmatrix}$
3 Linear equations	Form Ax = b	Augmented matrix form	A and b

First method: In the first method the parameters are provided to the linsolve() method in the form of (A,b) where A and b are matrix. This is shown in the following code:

```python
from sympy import Matrix, S, linsolve, symbols
#x + y + z = 5, 2x + 3y + 5z = 8, 4x + 5z = 2
x, y, z = symbols("x, y, z")
A = Matrix([[1, 1, 1], [2, 3, 5], [4, 0, 5]])
b = Matrix([5, 8, 2])
sol = linsolve((A, b), [x, y, z])
print(sol)# Solution is {(3, 4, -2)}
```

Example using linsolve() - - 2

Second method: Here the three linear equations are provided as comma separated values to the system parameter of the linsolve() method as shown in the following code:

```python
from sympy import *
from sympy.solvers.solveset import linsolve
x, y, z = symbols('x y z')
my_sol = linsolve([x + y + z - 5,
                   2*x + 3*y + 5*z - 8,
                   4*x + 5*z - 2], (x, y, z))
print(my_sol)
# OUTPUT
{(3, 4, -2)}
```

Calculus with SymPy

Topics discussed here are:

- **Differentiation**
- **Integration**
- **Finding Limits (Lim)**
- **Ordinary Differential Equations (ODE)**

Calculus with SymPy

Topics discussed here are:

- **Differentiation**
- **Integration**
- **Finding Limits (Lim)**
- **Ordinary Differential Equations (ODE)**

Differentiation - - 1

For differentiation, SymPy has a function

`diff(expression, variable_1, variable_2…)`

Here expression is the expression to be differentiated and variable_1, variable_2 are the variables with respect to which the differentiation is to take place.

So you can do any number of differentiations at once. This will become clear from the following example:

```
from sympy import *
x, y, z = symbols('x y z')
exp1 = x*((x**3) + (y**4) + (z**5))
exp2 = diff(exp1, x)
print(exp2) # exp2-> 4*x**3 + y**4 + z**5
exp3 = diff(exp1, x, y)
print(exp3) # exp3-> 4*y**3
exp4 = diff(exp1, x, y, y)
print(exp4) # exp4-> 12*y**2
```

Differentiation - - 2

You can differentiate an expression with respect to another expression also. See the following code:

```
from sympy import Symbol
x = Symbol('x')
print((sin(x)**2).diff(x))# Differentiate wrt x
print((sin(x)**2).diff(sin(x)))# Differentiate wrt sin(x)
```

```
# Output. . .
2*sin(x)*cos(x)
2*sin(x)
```

Differentiation - - 3

SymPy also provides a Derivative class for derivation.

```
from sympy import *
x, y = symbols('x y')
exp1 = 2*(x**2) + 5*x*y
sol = Derivative(exp1, x)
print(sol)
```

```
# Output. . .
Derivative(2*x**2 + 5*x*y, x)
```

Integration - - 1

Similarly to integrate you can use the function

`integrate(expression, var_1, var_2…)`

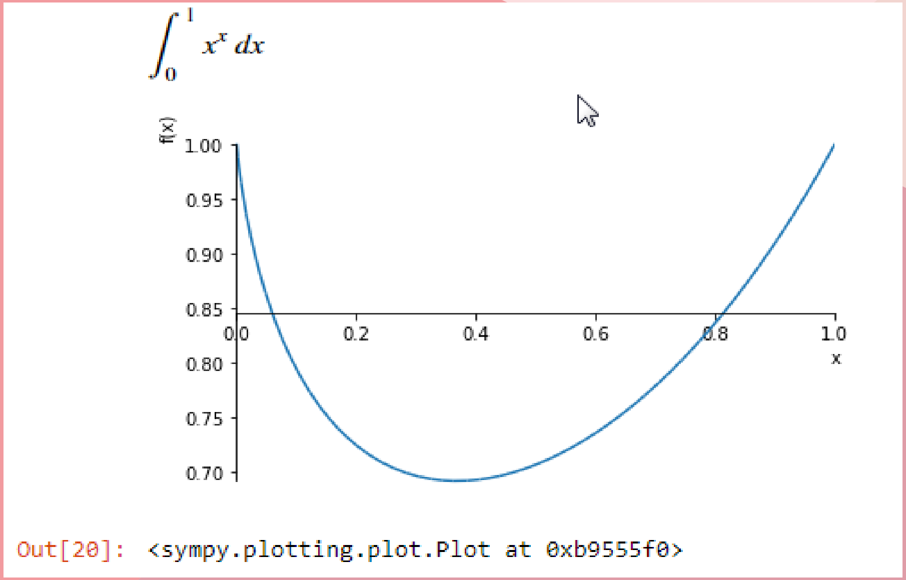
This will be clear from the following code:

```python
from sympy import *
x, y, z = symbols('x y z')
exp1 = sin(x) + log(y) + z
exp2 = integrate(exp1, x)
print(exp2) # exp2-> x*z + x*log(y) - cos(x)
exp3 = integrate(exp1, x, y)
print(exp3) # exp3-> x*y*log(y) + y*(x*z - x - cos(x))
```

Integration - - 2

You can use SymPy for plotting a function also. This will be clear from following code:

```python
from sympy import symbols
from sympy.plotting import plot
from IPython.display import display
init_printing(use_latex='mathjax')
%matplotlib inline
exp1 = integrate(x**x, (x, 0, 1))
display(exp1)
plot(x**x, (x,0,1))
```



$$\int_0^1 x^x\,dx$$

```
Out[20]: <sympy.plotting.plot.Plot at 0xb9555f0>
```

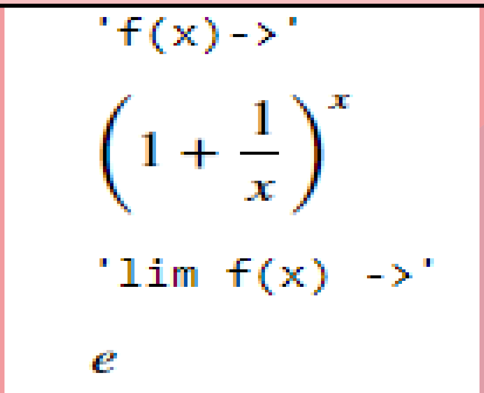
Finding limits (Lim) - - 1

SymPy can also be used to get limits of functions. You can solve an equation of type $\underset{x \to x0}{lim} f(x)$

For instance, it is well known that $\lim_{x \to oo} (x + \frac{1}{x})^{x}$ **=** ***e*****. You can solve this on SymPy as follows:**

```
x = symbols('x')
exp1 = (1 + 1/x)**x # Function whose limit to be found
display('f(x)->', exp1)
exp2 = limit(exp1, x,oo ) # limit at infinity
display('lim f(x) ->',exp2)
```

The output on Jupyter is as follows:-



```
'f(x)->'
```

$$\left(1 + \frac{1}{x}\right)^{x}$$

```
'lim f(x) ->'
```

$$e$$

Finding limits (Lim) - - 2

In SymPy you can evaluate limit from both sides, that is, positive side as well as negative side.

Following example clarifies it:

```
exp1 = (1/(x-3)) # Function whose limit to be found
exp2 = limit(exp1, x,3, '+' ) # limit from positive direction
display('lim f(x) ->',exp2) # Gives oo (+ infinity)
exp3 = limit(exp1, x,3, '-' ) # limit from negative direction
display('lim f(x) ->',exp3) # Gives -oo (- infinity)
```

Ordinary Differential equations (ODE) - - 1

SymPy provides a dsolve() method for solving differential equations. This method is part of the ODE module.

The dsolve() method has many parameters as can be seen by its signature. Here it is not possible (nor necessary) to discuss all the details. The complete signature of dsolve() is:

```
Signature: dsolve(eq, func=None, hint='default', simplify=True,
ics=None, xi=None, eta=None, x0=0, n=6, **kwargs)
```

- **For the present purpose consider just one parameter, that is, the first parameter namely "eq".**
- **This parameter is the "supported differential equation type" which is to be solved.**
- **The dsolve() method cannot solve all types of ODE.**
- **Rather it can solve only the "supported types" of ODE.**
- **A brief discussion on the "supported types of ODE" is given at the end of the chapter.**

Solving ODE for an undamped and damped harmonic oscillator - - 1

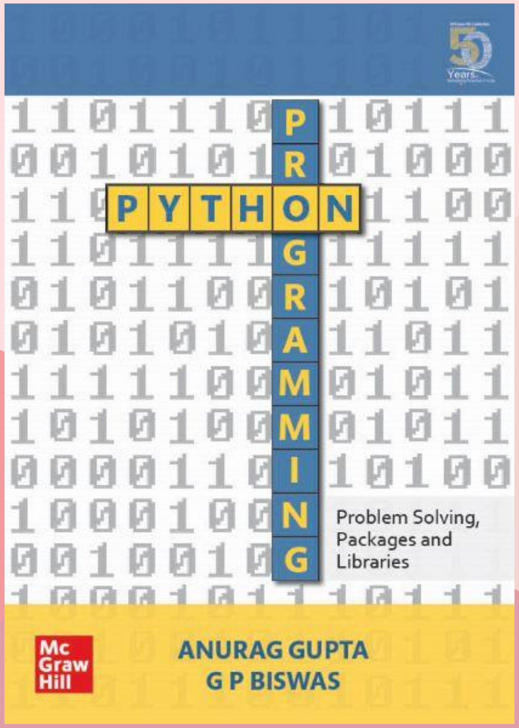
Note: In order to use the dsolve() method of ODE of SymPy, you need to feed the equation in the form:
f(x, y, dy/dx, d^2y/dx^2..) = 0
Those who are familiar with oscillators (in physics) , know that the motion of an undamped and a damped oscillators with respect to time can be represented in form of differential equations.
In general the motion of a damped oscillator is: m*ÿ(t) +c* ẏ(t) + k*y(t) = 0. An undamped oscillator has a slightly simpler form because here the coefficient c = 0.

```
m*ÿ(t) + k*y(t) = 0 # Undamped oscillator ie c = 0
m*ÿ(t) +c* ẏ(t) + k*y(t) = 0 # Damped oscillator
Where
• ÿ(t) = d²y(t)/ dt²,
• ẏ(t) = dy(t)/dt
• t = time,
• m = mass,
• k = spring constant and
• c = friction constant
```

Solving ODE for an undamped and damped harmonic oscillator - - 2

```python
from sympy.solvers.ode import dsolve
from sympy import *
t, m, k, c = symbols('t m k c')
y = Function('y')(t)
y_ = Derivative(y, t) # y_ is first derivative of y(t) wrt to t
y__ = Derivative(y_, t) # y__ is second derivative of y(t) wrt to t
# Undamped oscillator m is mass k is spring constant
eq_ud = m*y__ + k*y
sol_ud = dsolve(eq_ud)
print('Undamped oscillator->', sol_ud)
print('type of differential eq->', classify_ode(eq_ud))  # The classify_ode()
method gives "type" of ODE
# Damped Oscillator force of friction Ff = -cy_
eq_dd = m*y__ + c*y_ + k*y
sol_dd = dsolve(eq_dd)
print('Damped oscillator->', sol_dd)
print('type of differential eq->', classify_ode(eq_dd))
```

```
# Output. . .
Undamped oscillator-> Eq(y(t), C1*exp(-t*sqrt(-k/m)) + C2*exp(t*sqrt(-k/m)))
type of differential eq-> ('nth_linear_constant_coeff_homogeneous',
'2nd_power_series_ordinary')
Damped oscillator-> Eq(y(t), C1*exp(t*(-c - sqrt(c**2 - 4*k*m))/(2*m)) +
C2*exp(t*(-c + sqrt(c**2 - 4*k*m))/(2*m)))
type of differential eq-> ('nth_linear_constant_coeff_homogeneous',
'2nd_power_series_ordinary')
```


PYTHON
PROGRAMMING
Problem Solving, Packages and Libraries
ANURAG GUPTA
G P BISWAS
Mc Graw Hill

Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

support.india@mheducation.com

1800-103-5875

www.mheducation.co.in