SympyCore User's Guide

Author: Pearu Peterson
Author: Fredrik Johansson
Created: January 2008
Website: http://sympycore.googlecode.com/
Version: sympycore 0.2-svn.

Contents

1 Introduction

The aim of the SympyCore project is to develop a robust, consistent, and easy to extend Computer Algebra System model for Python.

Editorial notes: - This document is written in reStructuredText format.

2 Getting Started

To use SympyCore from Python, one needs to import the `sympycore` package:

```
>>> from sympycore import *
```

The `sympycore` package provides `Symbol` and `Number` functions to construct symbolic objects and numbers. By default, the symbolic objects are the elements of `Calculus` algebra -- a commutative ring of symbolic expressions where exponent algebra is also `Calculus` algebra.

```
>>> x = Symbol('x')
>>> n = Number(2,5)
>>> x+n
Calculus('x + 2/5')
>>> x,y,z,v,w=map(Symbol,'xyzvw')
```

To construct expression from a string, use the corresponding algebra class with one argument. For example,

```
>>> Calculus('x+y+1/4 + x**2')+x
Calculus('y + x**2 + 1/4 + 2*x')
```

More examples on `sympycore` features can be found in Demo documentation.

3 The CAS model

Symbolic expressions represent mathematical concepts like numbers, constants, variables, functions, operators, and various relations between them. Symbolic objects, on the other hand, represent symbolic expressions in a running computer program. The aim of a Computer Algebra System (CAS) is to provide methods to manipulate symbolic objects and by that manipulate symbolic expressions. These manipulations of symbolic expressions have mathematical meaning when the methods are consistent with the rules and theories from mathematics.

There are many possible ways to represent a mathematical concept as a structure of a computer program. SympyCore mimics mathematical concepts via implementing the corresponding algebra and algebraic operations in a class, say Algebra, that is derived from the BasicAlgebra class. So, a symbolic object is an instance of the Algebra class. This instance contains information about the mathematical operator that when applied to operands forms the corresponding symbolic object. The operator and operands of the given symbolic object can be accessed via atrributes `func` and `args`. The value of `func` is a callable object and `args` is a sequence of symbolic objects. So, if `A` is a `Algebra` instance then:

```
<symbolic object> = A.func(*A.args)
```

The actual value of `func` is defined by the `Algebra` class. For example, in the case of calculus algebra class `Calculus`, the `func` value can be `Add`, `Mul`, `Pow`, `sin`, `log`, etc. If the symbolic object represents a symbol (eg a variable) or a number of the algebra then `func` contains a callable that returns the symbolic object (the `args` in this case will be an empty sequence).

The symbolic objects representing symbols and numbers can be constructed via the `Symbol` and `Number` functions. Such symbolic objects are called atomic. One should note that functions `Add`, `Mul`, `Pow`, `Symbol`, `Number`, etc are always specific to the given algebra (in fact, they are defined as classmethods of the corresponding algebra class).

While most of the algebra operators assume symbolic objects as their operands then `Symbol` and `Number` functions may take various Python objects as arguments. For example, the argument to `Calculus.Symbol` can be any python object that is immutable (this requirement comes from the fact terms of sums and factors of products are internally saved as Python dictionary keys), and the arguments to `Calculus.Number` can be Python number types such as `int`, `long`, `float`, `complex` as well as `mpq`, `mpf`, `mpqc` instances (these are defined in `sympycore.arithmetic` package).

One can construct symbolic objects from Python strings using algebra `convert` class method or algebra constructor with one argument. For example,

```
>>> Calculus.convert('a-3/4+b**2')
Calculus('a + b**2 - 3/4')
>>> Calculus('a-3/4+b**2').func
<bound method type.Add of <class 'sympycore.calculus.algebra.Calculus'>>
>>> Calculus('a-3/4+b**2').args
[Calculus('a'), Calculus('-3/4'), Calculus('b**2')]
```

4 Package structure

SympyCore project provides a python package sympycore that consists of several modules and subpackages:

1. core.py - provides a base class Basic to all symbolic objects. Note that almost any (hashable) python object can be used as an operand to algebraic operations (assuming the corresponding algebra class accepts it) and hence it is not always necessary to derive classes defining some mathematical notion from Basic. Only classes that could be used by other parts of the sympycore should be derived from Basic. In such cases, these classes are available via classes attributes (also defined in core.py). For example,

```
>>> from sympycore.core import classes
>>> classes.Calculus
<class 'sympycore.calculus.algebra.Calculus'>
>>> classes.Unit
<class 'sympycore.physics.units.Unit'>
>>> classes.CollectingField
<class 'sympycore.basealgebra.pairs.CollectingField'>
```

2. arithmetic/ - provides mpq, mpf, mpqc, mpc classes that represent low-level fractions, multiprecision floating point numbers, and complex numbers with rational parts. The package defines also Infinity class to represent extended numbers.

3. basealgebra/ - provides abstract base classes representing algebras: BasicAlgebra, CommutativeRing, etc, and base classes for algebras with implementations: Primitive, CollectingField, etc.

4. calculus/ - provides class Calculus that represents the algebra of symbolic expressions. The Calculus class defines the default algebra in sympycore. For more information, see [section on calculus]. calculus/functions/ - provides symbolic functions like exp, log, sin, cos, tan, cot, sqrt, ...

5. physics/ - provides class Unit that represents the algebra of symbolic expressions of physical quantities. For more information, see [section on physics].

6. polynomials/ - provides classes Polynomial, UnivariatePolynomial, MultivariatePolynomial to represent the algebras of polynomials with symbols, univariate polynomials in (coefficient:exponent) form, and multivariate polynomials in (coefficients:exponents) form, respectively. For more information, see [section on polynomials].

5 Basic methods

In `sympycore` all symbolic objects are assumed to be immutable. So, the manipulation of symbolic objects means creating new symbolic objects from the parts of existing ones.

There are many methods that can be used to retrive information and subexpressions from a symbolic object. The most generic method is to use attribute pair of `func` and `args` as described above. However, many such methods are also algebra specific, for example, classes of commutative rings have methods like `as_Add_args`, `as_Mul_args`, etc for retriving the operands of operations and `Add`, `Mul`, etc for constructing new symbolic objects representing addition, multiplication, etc operations. For more information about such methods, see sections describing the particular algebra classes.

5.1 Output methods

`str(<symbolic object>)` return a nice string representation of the symbolic object. For example,

```
>>> expr = Calculus(’-x + 2’)
>>> str(expr)
’2 - x’
```

`repr(<symbolic object>)` return a string representation of the symbolic object that can be used to reproduce an equal object:

```
>>> expr=Calculus(’-x+2’)
>>> repr(expr)
"Calculus(’2 - x’)"
```

`<symbolic object>.as_tree()` return a tree string representation of the symbolic object. For example,

```
>>> expr = Calculus(’-x + 2+y**3’)
>>> print expr.as_tree()
Calculus:
ADD[
  -1:SYMBOL[x]
  1:MUL[
  1:  3:SYMBOL[y]
  1:]
  2:NUMBER[1]
]
```

where the first line shows the name of a algebra class following the content of the symbolic object in tree form. Note how are represented the coefficients and exponents of the example subexpressions.

5.2 Conversation methods

`<symbolic object>.as_verbatim()` return symbolic object as an instance of `Verbatim` class. All algebra classes must implement `as_verbatim` method as this allows converting symbolic objects from one algebra to another that is compatible with respect to algebraic operations. Also, producing the string representations of symbolic objects is done via converting them to Verbatim that implements the corresponding printing method. For example,

```
>>> expr
Calculus('2 + y**3 - x')
>>> expr.as_verbatim()
Verbatim('2 + y**3 - x')
```

`<symbolic object>.as_algebra(<algebra class>)` return symbolic object as an instance of given algebra class. The transformation is done by first converting the symbolic object to `Verbatim` instance which in turn is converted to the instance of targer algebra class by executing the corresponding target algebra operators on operands. For example,

```
>>> expr = Calculus('-x + 2')
>>> print expr.as_tree()
Calculus:
ADD[
  -1:SYMBOL[x]
  2:NUMBER[1]
]
>>> print expr.as_algebra(Verbatim).as_tree()
Verbatim:
ADD[
  NEG[
    SYMBOL[x]
  ]
  NUMBER[2]
]
>>> print expr.as_algebra(CollectingField).as_tree()
CollectingField:
ADD[
  -1:SYMBOL[x]
  2:NUMBER[1]
]
```

5.3 Substitution of expressions

<symbolic object>.subs(<sub-expr>, <new-expr>) return a copy of <symbolic object> with all occurances of <sub-expr> replaced with <new-expr>. For example,

```
>>> expr = Calculus('-x + 2+y**3')
>>> expr
Calculus('2 + y**3 - x')
>>> expr.subs('y', '2*z')
Calculus('2 + 8*z**3 - x')
```

<symbolic object>.subs([(<subexpr1>, <newexpr1>), (<subexpr2>, <newexpr2>), ...]) is equivalent to <symbolic object>.subs(<subexp1>, <newexpr1>).subs(<subexpr2>, <newexpr2>).subs. For example,

```
>>> expr
Calculus('2 + y**3 - x')
>>> expr.subs([('y', '2*z'),('z', 2)])
Calculus('66 - x')
```

5.4 Pattern matching

<symbolic object>.match(<pattern-expr> [, <wildcard1>, <wildcard2> ...]) check if the give symbolic object matches given pattern. Pattern expression may contain wild symbols that match arbitrary expressions, the wildcard must be then the corresponding symbol. Wild symbols can be matched also conditionally, then the <wildcard> argument must be a tuple (<wild-symbol>, <predicate>), where <predicate> is a single-argument function returning True if wild symbol matches the expression in argument. If the match is not found then the method returns. Otherwise it will return a dictionary object such that the following condition holds:

```
pattern.subs(expr.match(pattern, ...).items()) == expr
```

For example,

```
>>> expr = 3*x + 4*y
>>> pattern = v*x + w*y
>>> d = expr.match(pattern, v, w)
>>> print 'v=',d.get(v)
v= 3
>>> print 'w=',d.get(w)
w= 4
>>> pattern.subs(d.items())==expr
True
```

5.5 Checking for atomic objects

A symbolic object is atomic if `<symbolic object>.args == ()`.

`<symbolic object>.symbols` is a property that holds a set of all atomic symbols in the given symbolic expression.

`<symbolic object>.has(<symbol>)` returns `True` if the symbolic expression contains `<symbol>`.

6 Verbatim algebra

Verbatim algebra elements are symbolic expressions that are not simplified in anyway when performing operatons. For example,

```
>>> s=Verbatim('s')
>>> s+s
Verbatim('s + s')
```

7 Commutative ring

In SympyCore a commutative ring is represented by an abstract class `CommutativeRing`. The `CommutativeRing` class defines support for addition, substraction, multiplication, division, and exponentiation operations.

7.1 Operations

Classes deriving from `CommutativeRing` must define a number of method pairs `(Operation, as_Operation_args)` that satisfy the following condition:

```
cls.Operation(*obj.as_Operation_args()) == obj
```

Here `Operation` can be `Add`, `Mul`, `Terms`, `Factors`, `Pow`, `Log`. For example,

```
>>> print map(str, (2*x+y).as_Add_args())
['y', '2*x']
>>> print map(str, (2*x+y).as_Mul_args())
['y + 2*x']
>>> print map(str, (2*x+y).as_Pow_args())
['y + 2*x', '1']
>>> print (2*x+y).as_Terms_args()
[(Calculus('y'), 1), (Calculus('x'), 2)]
```

7.2 Expanding

Expanding means applying distributivity law to open parenthesis.

<symbolic object>.expand() return an expanded expression. For example,

```
>>> expr = x*(y+x)**2
>>> print expr
x*(x + y)**2
>>> print expr.expand()
x**3 + 2*y*x**2 + x*y**2
```

7.3 Differentation

<symbolic object>.diff(*symbols) return a derivative of symbolic expression with respect to given symbols. The diff methods argument can also be a positive integer after some symbol argument. Then the derivative is computed given number of times with respect to the last symbol. For example,

```
>>> print sin(x*y).diff(x)
y*cos(x*y)
>>> print sin(x*y).diff(x).diff(y)
cos(x*y) - x*y*sin(x*y)
>>> print sin(x*y).diff(x,4)
sin(x*y)*y**4
```

7.4 Integration

<symbolic object>.integrate(<symbol>, integrator=None) return an antiderivative of a symbolic expression with respect to <symbol>. For example,

```
>>> from sympycore import *
>>> print (x**2 + x*y).integrate(x)
1/2*y*x**2 + 1/3*x**3
```

<symbolic object>.integrate((<symbol>, <a>, <b>) return a defined integral of a symbolic expression with respect to <symbol> over the interval [<a>, <b>]. For example,

```
>>> from sympycore import *
>>> print (x**2 + x*y).integrate(x)
1/2*y*x**2 + 1/3*x**3
>>> print (x**2 + x*y).integrate((x, 1, 3))
26/3 + 4*y
```

8 Commutative ring implementation

Commutative ring operations are implemented in the class `CollectingField` (derived from `CommutativeRing`).

The class `CollectingField` holds two attributes, `head` and `data`. The attribute `head` defines the meaning of the attribute `data` content:

1. If `<obj>.head==SYMBOL` then `<obj>.data` is treated as an element of the ring. Usually `<obj>.data` is a Python string object but in general it can be any hashable Python object.

2. If `<obj>.head==NUMBER` then `<obj>.data` is treated as a number element of the ring, that is, an element that can be represented as *one* * n where *one* is unit element of the ring and n is a number saved in `<obj>.data`. Usually `<obj>.data` is a Python `int`, `long`, `float`, `complex` object but it can be also any other number-like object that supports arithmetic operations with Python numbers. An examples are `mpq`, `mpf`, `mpqc` classes defined in `sympycore.arithmetic` package.

3. If `<obj>.head==TERMS` then `<obj>.data` contains a Python dictionary holding the pairs `(<ring element>, <coefficient>)`. The values of `<coefficients>` can be Python numbers or number-like objects or elements of some other ring (for example, see `Unit` class where the coefficients are `Calculus` instances). For example, if `<obj>.data` is `{x:2, y:1}` then `<obj>` represents an expression *y + 2*x*.

4. If `<obj>.head==FACTORS` then `<obj>.data` contains a Python dictionary holding the pairs `(<ring element>, <exponent>)`. The values of `<coefficients>` can be Python numbers of number-like objects or elements of some ring (for exapmle, see `Calculus` class where the exponents can also be `Calculus` instances).

5. If `callable(<obj>.head)` then `<obj>` represents an applied function where `<obj>.head` contains a callable object that performs evaluation and `<obj>.data` contains an argument instance (for example, an instance of some algebra elements) or a Python `tuple` containing argument instances.

The constants `SYMBOL`, `NUMBER`, `TERMS`, `FACTORS` are defined in `sympycore/utils.py`. For example,

```
>>> from sympycore.utils import head_to_string
>>> head_to_string[x.head]
'SYMBOL'
>>> x.data
'x'
>>> head_to_string[(x+y).head]
```

```
'ADD'
>>> (x+y).data == {x:1,y:1}
True
>>> head_to_string[(x**y).head]
'MUL'
>>> (x**y).data
{Calculus('x'): Calculus('y')}
>>> sin(x).head
<class 'sympycore.calculus.functions.elementary.sin'>
>>> sin(x).data
Calculus('x')
```

8.1 Defining functions for CollectingField

The representation of an applied function within the class CollectingField can hold any Python callable object that satisfies the following basic condition: it must return an instance of a algebra class. The instance may represent an evaluated result of applying the function to its arguments, or when evaluation is not possible, then it return `<algebra class>(<arguments>, head=<callable>)`.

For example, let us define a customized sinus function:

```
>>> def mysin(x):
...     if x==0:
...         return x
...     return Calculus(mysin, x)
...
>>> mysin(0)
0
>>> print mysin(x+y)
mysin(x + y)
```

9 Calculus

The default algebra of symbolic expressions with commutative ring operations is represented by the Calculus class (derived from CollectingField). The Calculus class can handle rational numbers represented by the mpq class, multi-precision floating point numbers represented by the mpf class, and rational complex numbers represented by the mpqc class.

The sympycore.calculus.functions package defines the following symbolic functions: sqrt, exp, log, sin, cos, tan, cot. It also provides Calculus based interfaces to constants E, pi, and symbols I, oo, moo, zoo, undefined.

10 Arithemetics

The `sympycore.arithmetic` package is not an algebra package but it implements fractions, multi-precision floating point numbers, rational complex numbers, and extended numbers. In addition, it implements various algorithms from number theory and provides methods to compute the values of constants like pi and Eulers number, etc.

11 Polynomials

The `sympycore.polynomials` package has two different implementations for polynomials: `UnivariatePolynomial` and `PolynomialRing`.

11.1 `UnivariatePolynomial`

The `UnivariatePolynomial` class stores polynomial coefficients in a Python tuple. The exponents are implicitly defined as indices of the list so that the degree of a polynomial is equal to the length of the list minus 1. `UnivariatePolynomial` is most efficient for manipulating low order and dense polynomials. To specify the variable symbol of a polynomial, use `symbol` keyword argument (default variable symbol is `x`).

```
>>> poly([4,3,2,1])
4 + 3*x + 2*x**2 + x**3
>>> poly([4,3,2,1]).degree
3
>>> poly([4,3,2,1],symbol='y')
4 + 3*y + 2*y**2 + y**3
```

Coefficients can be arbitrary symbolic expressions:

```
>>> poly([2,y+1,y+z])
2 + ((1 + y))*x + ((y + z))*x**2
```

11.2 `PolynomialRing`

The `PolynomialRing` based classes store polynomial exponenets and coefficients information in a Python dictionary object where keys are exponenents (in univariate case Python integers, in multivariate case `AdditiveTuple` instances) and values are coefficients. `PolynomialRing` is most efficient for manipulating sparse polynomials. The coefficients belong to specified ring (default ring is `Calculus`).

The `PolynomialRing` class (derived from `CommutativeRing`) is a base class to various polynomial rings with different coefficent rings and different number of variables. To create a class representing a polynomial element with variables `(X, Y, ..)` and with `<ring>` coefficients, use one of the following constructions:

```
PolynomialRing[(X, Y, ..), <ring>]
PolynomialRing[<int>, <ring>]
```

where nonnegative `<int>` specifies the number of variables (default symbols are then `X0`, `X1`, etc). The `<ring>` argument can be omitted, then `Calculus` is used as a default ring. Variables can be arbitrary symbolic expressions.

For example,

```
>>> polyXY = PolynomialRing[('X', 'Y'), Calculus]
>>> polyXY
<class 'sympycore.polynomials.algebra.PolynomialRing[(X, Y), Calculus]'>
```

To create a polynomial with given exponents and coefficients pairs, the `PolynomialRing` constructor accepts dictinary objects containing the corresponding pairs:

```
>>> polyXY.convert({(0,0):4, (2,1):3, (0,3):2})
PolynomialRing[(X, Y), Calculus]('3*X**2*Y + 2*Y**3 + 4')
```

Univariate polynomials can also be constructed from a list in the same way as `UnivariatePolynomial` instances were constructed above:

```
>>> PolynomialRing[1].convert([4,3,2,1])
PolynomialRing[X0, Calculus]('X0**3 + 2*X0**2 + 3*X0 + 4')
```

12 Matrices

The `sympycore.matrices` package defines `MatrixRing` that is base class to matrix algebras. Matrix algebras are represented as classes (derived from `MatrixRing`) parametrized with matrix shape and element ring (default ring is `Calculus`). To create a matrix ring, use the following constructs:

```
MatrixRing[<shape>, <ring>]
SquareMatrix[<size>, <ring>]
PermutationMatrix[<size>]
```

where `<ring>` can be omitted, then `Calculus` is used as a default element ring.

For example,

```
>>> m=MatrixRing[3,4]({})
>>> print m
 0  0  0  0
 0  0  0  0
 0  0  0  0
>>> m[1,2] = 3
>>> m[2,3] = 4
>>> print m
 0  0  0  0
 0  0  3  0
 0  0  0  4
```

The content of the matrix is stored as a dictionary containing pairs `(<rowindex>,<column-index>):<non-zero element>`.

Matrix instances can be constructed from Python dictionary or from a Python list:

```
>>> print MatrixRing[2,2]({(0,0):1,(0,1):2,(1,1):3})
 1  2
 0  3
>>> print MatrixRing[2,2]([[1,2],[3,4]])
 1  2
 3  4
```

Permutation matrices can be constructed from a sequence of integers:

```
>>> print PermutationMatrix([1,0,2])
 0  1  0
 1  0  0
 0  0  1
```

Use `random()` classmethod to construct matrices with random content:

```
>>> print SquareMatrix[2].random()         #doctest: +SKIP
 -1  3
  3  0
>>> print SquareMatrix[2].random((10,20))  #doctest: +SKIP
 15  10
 13  15
```

13 Canonical forms and suppressed evaluation

See also Automatic evaluation rules of symbolic expressions.

The `Calculus` algebra automatically applies some transformations to expressions. The purpose of these transformations is to permit quick recognition of mathematically equivalent expressions. Sums and products of numbers are always evaluated, and multiples/powers of identical subexpressions are automatically collected together. Rational factors are also automatically distributed over sums. For example, the following transformations are performed automatically:

```
2*3 -> 6

x+x -> 2*x

x*x -> x**2

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

An expression to which default transformations have been applied is said to be in canonical or normalized form. The enforcement of canonical forms is important for performance reasons as it ensures that, in many important basic cases, expressions that are mathematically equivalent will be recognized directly as equal no matter in what form they were entered, without the need to apply additional transformations. The default transformations described above ensure that for example the following expressions cancel completely:

```
2*3 - 6 -> 0

x+x - (2*x) -> 0

x*x - x**2 -> 0

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

Ideally we would like the canonical form to be the simplest expression possible, e.g.:

```
cos(x)**2 + sin(x)**2 -> 1
```

Automatically generating the simplest possible form is not always possible, as some expressions have multiple valid representations that may each be useful in different contexts. E.g.: `cos(2*x)` and `cos(x)**2 - sin(x)**2`. In general, detecting whether two expressions are equal is not even algorithmically decidable, and even when it is possible, the required simplifications can be extremely computationally expensive (and unpredictably so).

Default transformations are limited to performing operations cases that are fast and have predictable behavior. To perform more expensive simplifications, one should explicitly invoke `simplify()` or, depending on the desired form, special-purpose rewriting functions like `collect()`, `apart()`, etc (note: these are not yet implemented in Sympy-Core).

It can sometimes be useful to bypass automatic transformations, for example to keep the expression `2*(x+y)` in factored form. The most general way to achieve this is to use the `Verbatim` class (which performs no simplifications whatsoever) instead of `Calculus`.

```
>>> Verbatim('2*(x+pi)')
Verbatim('2*(x + pi)')
```

You can also construct non-canonical `Calculus` instances by manually passing data to the `Calculus` constructor. For example:

```
>>> p = Calculus(utils.TERMS, {(pi+x):2})
>>> print p
2*(pi + x)
```

It is important to note that some `Calculus` functions assume the input to be in canonical form. Although they should never break (i.e. generate invalid results) when given noncanonical input, they may fail to simplify results. For example, `sin` assumes its argument to be flattened such that if it contains an integer multiple of pi that can be eliminated, this term will be available at the top of the expression. Thus:

```
>>> sin(2*(pi+x))  # sin(2*pi + 2*x)
Calculus('sin(2*x)')
>>> sin(p)
Calculus('sin(2*(pi + x))')
```

To canonize an expression, either use the function XXX or convert it to `Verbatim` and then back to `Calculus`.

```
>>> Calculus(Verbatim(p))
Calculus('2*pi + 2*x')
```