Symbolic manipulation package for Python

Pearu Peterson
<pearu@cens.ioc.ee>

May 20, 2007

Abstract

Symbolic is a pure Python package that provides tools for performing symbolic manipulations with objects representing numbers, symbols, operations, functions, and operators. Supported operations include logical, relational, arithmetic, and functional operations. The Symbolic package provides a parser tool to create symbolic objects from a string with Python language like syntax.

1 Getting started

To get started using Symbolic package, one must import the module `symbolic.api`. In the following examples we assume that the following import statement has been executed in the beginning of Python session:

```
>>> from symbolic.api import *
```

Simplest way to create symbolic objects is to use *Symbolic* constructor. For an example, to create a polynomial object from a string, execute:

```
>>> poly = Symbolic('2/3 + a * 3 + a ** 2 /4')
>>> poly
```

where *Symbolic* has parsed Python string, constructed symbolic object, and the default string representation of the symbolic object `poly` is displayed. The object `poly` is an *Add* object containing *Number*, *Symbol*, *Mul*, and *Power* objects. To view the internal structure of a symbolic object, use `.torepr()` method:

```
>>> poly.torepr()
```

(here *Integer* and *Rational* are subclasses of the *Number* class). The same polynomial can be constructed using arithmetic operations with symbolic objects:

```
>>> a = Symbol('a')
>>> poly2 = Rational(2,3) + 3*a + a**2 /4
>>> poly2
```

Note that the object representing a rational number 2/3 had to be constructed explicitly as Python integer division does not result in a rational number. The *poly* and *poly2* objects are equal indeed:

```
>>> poly==poly2
>>> bool(poly==poly2)
>>> bool(poly==a)
>>> poly-poly2
```

Here follows more examples on symbolic manipulations with the help of Symbolic package:

```
>>> poly.diff('a')                        # differentation
>>> poly.integrate('a')                   # antiderivative
>>> poly.integrate(Range('a',0,1))        # definite integration
>>> poly + 3 * poly                       # arithmetic operations
>>> (poly ** 2).expand()                  # expansion
>>> poly.substitute('a','(b+1)/3')        # substitution
>>> # elementary functions exp, ln, log, sin, cos etc.
>>> exp(poly)
>>> # elementary propositional calculus
>>> lhs = Symbolic('(a and b).implies(c)')
>>> rhs = Symbolic('a.implies(b.implies(c))')
>>> lhs == rhs
>>> lhs.equiv(rhs).expand()
```

2 Symbolic parser

Symbolic provides a parser for symbolic expressions to ease creating symbolic objects. The syntax of symbolic expressions is borrowed from Python syntax rules with some extensions like parsing rational numbers. The syntax rules are the following:

```
expr          ::=  lambda-test
lambda-test   ::=  [ lambda identifier-list colon ] or-test
or-test       ::=  [ or-test or-op ] xor-test
xor-test      ::=  [ xor-test xor-op ] and-test
and-test      ::=  [ and-test and-op ] not-test
not-test      ::=  [ not-op ] relational
relational    ::=  [ arith rel-op ] arith
arith         ::=  ( [ arith add-op ] term ) | factor
factor        ::=  [ add-op ] term
term          ::=  [ term mult-op ] power
power         ::=  primary [ power-op power ]
primary       ::=  atom | attr-ref | slicing | call
atom          ::=  identifier | literal | parenth
literal       ::=  int-literal | float-literal | logical-literal
parenth       ::=  ( expr-list )
attr-ref      ::=  primary .  identifier
slicing       ::=  primary [ subscript-list ]
subscript     ::=  expr | slice
slice         ::=  [ expr ] colon [ expr ] [ colon expr ]
call          ::=  primary ( [ argument-list ] )
argument      ::=  [ identifier = ] expr
identifier    ::=  ( letter | _ ) [ letter | digit | _ ]...
letter        ::=  lowercase | uppercase
```

where

```
logical-literal  ::=  True | False
or-op            ::=  or | |
xor-op           ::=  xor | ^
and-op           ::=  and | &
not-op           ::=  not | ~
rel-op           ::=  == | <> | != | < | <= | > | >= | in | not in
add-op           ::=  + | -
mult-op          ::=  * | /
power-op         ::=  **
lowercase        ::=  a...z
uppercase        ::=  A...Z
digit            ::=  0...9
colon            ::=  :
```

For each rule the `symbolic.parser` module provides a class to parse a string containing an expression satisfying the particular syntax rule. Parsing always results in a minimal syntax rule object. For example, the most general parser class is `Expr`:

```
>>> from symbolic.parser import *
>>> Expr('a+1')
>>> Expr('a+1').torepr()
>>> Expr('4/5').torepr()
```

Other parser classes are

.

To translate parsed syntax tree to a symbolic object, use `.tosymbolic()` method:

```
>>> Expr('a+1').tosymbolic()
```

3 Expressions

Symbolic objects can represent field values, symbols, function values, differential and integral operators, arithmetic expressions, relational expressions, and boolean expressions.

The following classes are defined to represent numeric fields: `Decimal`, `Rational`, and `Integer`. All these classes are subclasses of the `Number` class. In addition, `Boolean` represents boolean field constisting of two values, `TRUE` and `FALSE`.

To represent symbols of some field, the `Symbol` class is defined. In addition, for symbols representing variables of lambda functions or variables of integration, the `DummySymbol` is defined. Also, the `FunctionSymbol` class is defined to represent abstract functions. These symbol classes are subclasses of the `SymbolBase` class.

For function values the `Apply` is defined containing the information about a function and its arguments. A function can either be a lambda function or an elementary function or an abstract function. For lambda functions the `Lambda` class is defined containing information how its arguments are mapped to some expression. The following classes are defined to represent elementary functions: `Exp`, `Ln`, `Log`, `Sqrt`, `Sin`, `Cos`, etc.

Arithmetic expressions are addition, multiplication, and exponent operations between two or more symbolic objects. For a sum, product, and exponent the `Add`, `Mul`, and `Power` classes are defined, respectively.

Relational expressions are relational operations between symbolic objects. For relational operations the following classes are defined: `Equal`, `NotEqual`, `Less`, `LessEqual`, `Greater`, and `GreaterEqual`. Note that the instances of the last two classes are never created as $a > b$, $a >= b$ can be expressed as $b < a$, $b <= a$, respectively. Relational expressions can have boolean values.

Boolean expressions are boolean operations between symbolic objects. For boolean operations the following classes are defined: `Or`, `XOr`, `And`, and `Not`. Boolean operations operate on expressions with boolean values and result in boolean values.

Differential and integral operators...

Expressions of symbolic objects are all instances of `Symbolic` subclasses.

4 Symbolic classes

All symbolic objects are instances of subclasses of `Symbolic` class. The `Symbolic` can be used to parse a string for symbolic objects:

```
symbolic_object = Symbolic('<string>')
```

The set of all symbolic objects is ordered and symbolic object are hashable, that is, they can be used as dictionary keys. `Symbolic` class defines the following default methods:

Method call	Result	Description
`.is_equal(other)`	Python boolean or `None`	formal equality, the meaning of `False` value depends on the contex; `None` is returned when there is not enough information to deside
`.compare(other)`	-1, 0, or 1	determine canonical order of symbolic objects
`.expand()`	symbolic object	open parenthesis, expand integer powers, etc
`.substitute(expr, repl)`	symbolic object	substitute all occurrences of `expr` with `repl`
`.to_decimal()`	symbolic object	transform all number values to `Decimal` objects; use `Symbolic.set_precision(prec)` to set the precision (default is 28)

4.0.1 Implementation notes

Subclassing — `Symbolic` subclasses must be the attributes of `Symbolic` class. This ensures that all `Symbolic` subclasses are available to all modules as Symbolic.<subclassname> by single import statement `from symbolic.api import Symbolic`. This resolves the issue of cycling imports of `symbolic` modules.

Initialisation — `Symbolic` subclasses call `Symbolic.__new__(cls. *args, **kws)` to initialize a symbolic object. The `Symbolic.__new__()` method carries out two tasks. First, it calls the `.init(*args)` method of the corresponding `Symbolic` subclass with the same arguments that are used in constructing the symbolic object. The `.init()` method initializes the internal state of a symbolic object. Second, the `.flags` attribute is set to hold the `AttributeHolder` instance. The `AttributeHolder` object is used to restore the results of `.calc_*()` methods for future usage.

Constructor methods — are related to constructing symbolic objects:

`.__new__(cls, *args, **kws)` — construct *cls* instance, calls `.init()` method and sets `.flags` attribute.

`.init(*args)` — save the internal state of a symbolic object.

`.astuple()` — return the class name of a symbolic object and the arguments used to construct the symbolic object, the returned tuple must be hashable. For example, for `t = obj.astuple()` the following code `getattr(Symbolic,t[0])(*t[1:])` must return the equal symbolic object to *obj*.

`.eval_power(exponent)` — evaluate power to exponent if possible, otherwise return `None`

Informational methods — require information about symbolic objects:

`.get_precedence()` — return the precedence order of a symbolic object, must be Python integer. Background: symbolic objects may be childs of a parent object and in order to place parenthesis correctly around child objects, the precedece orders of the parent object and its child object are compared: if *child*`.get_precedence()` <= *parent*`.get_precedence()` then parenthesis are placed around *child* string representation. The following table shows precedence valuets for symbolic objects:

Precedence	Classes
0	`Symbol`, `Base`, `TRUE`, `FALSE`
10	`Equal`
20	`Relational`
22	`Or`
23	`XOr`
25	`And`
27	`Not`
30	`NegativeImaginaryUnit`, `Number`
40	`Add`
50	`Mul`, `NcMul`, `Rational`
60	`Power`
70	`Apply`
71	`SymbolicOperator`
72	`UndefinedFunction`

5 Fields

The `symbolic.number` module defines three number classes, `Decimal`, `Rational`, and `Integer`, to represent arbitrary precision floating point numbers, rational numbers, and integers, respectively. All number classes are subclasses of `Number` class which can be used to construct number instances of the three number classes. For example:

```
>>> Number(2).torepr()
>>> Number(2,3).torepr()
>>> Number("1.2").torepr()
```

Complex numbers are defined through arithmetic operations with a predefined imaginary unit object, `I`. The object `I` is an instance of a singleton class `ImaginaryUnit`. For example,

```
>>> (2*I+3).torepr()
```

5.1 Decimal numbers

The `Decimal` is used to represent arbitrary precision floating point numbers. The constructor for `Decimal` class is

```
decimal_object = Decimal(<str> | <int> | <long> | <float> | <decimal.Decimal>)
```

Arithmetic operations with a `Decimal` object and any other number object results in a `Decimal` object. For example,

```
>>> Decimal("2")/3
```

The precision of decimal operations can be controlled via `Symbolic.set_precision(prec=None)` static method:

```
>>> Symbolic.set_precision()                 # get the current precision
>>> 1/Decimal(3)
>>> prev_precision = Symbolic.set_precision(8) # decrease the precision
>>> 1/Decimal(3)
>>> Symbolic.set_precision(prev_precision)    # restore the original precision
```

Note that decimal numbers `"-1"`, `"0"`, `"1"`, `"Infinity"`, `"-Infinity"`, and `"NaN"` are mapped to singletons `NegativeOne()`, `Zero()`, `One()`, `Infinity()`, `NegativeInfinity()`, and `NaN()`, respectively. So, `Decimal("1")/3` results in `Rational(1, 3)`, for instance.

5.1.1 Implementation notes

Other number classes must define `._todecimal()` method returning the corresponding `decimal.Decimal` object.

Internally the `Decimal` class uses Python `decimal.Decimal` object for holding a floating point number, this number is stored in the `.num` attribute.

5.2 Rational numbers

The `Rational` class is used to represent rational numbers:

```
rational_object = Rational(<int> | <long>, <int> | <long>)
```

5.2.1 Implementation notes

The `Rational` class uses Python `int` type to store the nominator and denominator of a rational number in `.numer` and `.denom` attributes. Nominators and denominators are normalized.

Certain rational numbers are mapped to singletons:

```
>>> Rational(1,2).torepr()
>>> Rational(3,0).torepr()
>>> Rational(-2,0).torepr()
>>> Rational(0,0).torepr()
```

5.3 Integers

The `Integer` class is used to represent integers:

```
integer_object = Integer(<int> | <long>)
```

Certain integer are mapped to singletons:

```
>>> Integer(-1).torepr()
>>> Integer(0).torepr()
>>> Integer(1).torepr()
```

5.3.1 Implementation notes

`Integer` is a subclass of `Rational` and integer objects are rational objects with `.denom` equal to `1`.

5.4 Singletons

`Singleton` subclasses are classes that can have exactly one instance. The main advantage of defining certain symbolic objects as singletons is efficiency.

For example, the imaginary unit is defined as `Power(-1, Rational(1,2))` then checking if some object *obj* is an imaginary unit would require two symbolic object comparisons. Much more efficient would be executing `isinstance(obj, ImaginaryUnit)` or `obj is I`, where `ImaginaryUnit` is subclass of `Singleton` and `I` is its only instance.

The following singletons are defined:

Singleton class	**Predefined instance name**	**Description**
`ImaginaryUnit`	`I`	imaginary unit $\sqrt{-1}$
`NegativeImaginary`	-	negative imaginary unit $-\sqrt{-1}$
`One`	-	integer 1
`NegativeOne`	-	integer -1
`Zero`	-	integer 0
`Half`	-	rational 1/2
`Exp1`	`E`	exponent $\exp(0)$
`Pi`	`Pi`	real number π
`Infinity`	`infinity`	infinity ∞
`NegativeInfinity`	-	negative infinity $-\infty$
`NaN`	`NaN`	Not-A-Number or Undefined
`TRUE`	`TRUE`	boolean truth value
`FALSE`	`FALSE`	boolean false value
`Exp`	`exp`	exponent function
`Ln`	`ln`	logarithmic function with base e
`Log`	`log`	logarithmic function with given base
`Sqrt`	`sqrt`	square root function
`Sin`	`sin`	sine function
`Cos`	`cos`	cosine function