symbolism

Release 0.4.0

Reity LLC

Aug 04, 2022

CONTENTS

Extensible combinator library for building symbolic Python expressions that are compatible with serialization and can be evaluated at a later time.

CHAPTER

ONE

PURPOSE

In many scenarios that require some form of lazy evaluation, it is sufficient to employ lambda expressions, generators/iterables, or abstract syntax trees (via the ast and/or inspect modules). However, there are certain cases where none of these are an option (for example, employing lambda expressions precludes serialization and employing the ast or inspect modules usually involves introducing boilerplate that expands the solution beyond one line of code). The purpose of this library is to fill those gaps and make it possible to write concise symbolic expressions that are embedded directly within the concrete syntax of the language.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install symbolism
```

The library can be imported in the usual ways:

```
import symbolism
from symbolism import *
```

2.1 Examples

The library makes it possible to construct symbolic Python expressions (as instances of the `symbol` class) that can be evaluated at a later time. A symbolic expression involving addition of integers is created in the example below:

```
>>> from symbolism import *
>>> addition = symbol(lambda x, y: x + y)
>>> summation = addition(symbol(1), symbol(2))
```

The expression above can be evaluated at a later time:

```
>>> summation.evaluate()
3
```

Instances of `symbol` are compatible with common built-in infix and prefix arithmetic, logical, and relational operators. When an operator is applied to one or more `symbol` instances, a new `symbol` instance is created:

```
>>> summation = symbol(1) + symbol(2)
>>> summation.evaluate()
3
```

Pre-defined constants are also provided for all built-in operators supported by the `symbol` class:

```
>>> conjunction = and_(symbol(True), symbol(False))
>>> conjunction.evaluate()
False
```

CHAPTER

THREE

DEVELOPMENT

All installation and development dependencies are fully specified in `pyproject.toml`. The `project.optional-dependencies` object is used to specify optional requirements for various development tasks. This makes it possible to specify additional options (such as `docs`, `lint`, and so on) when performing installation using pip:

```
python -m pip install .[docs,lint]
```

3.1 Documentation

The documentation can be generated automatically from the source files using Sphinx:

```
python -m pip install .[docs]
cd docs
sphinx-apidoc -f -E --templatedir=_templates -o _source .. && make html
```

3.2 Testing and Conventions

All unit tests are executed and their coverage is measured when using pytest (see the `pyproject.toml` file for configuration details):

```
python -m pip install .[test]
python -m pytest
```

Alternatively, all unit tests are included in the module itself and can be executed using doctest:

```
python src/symbolism/symbolism.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/symbolism
```

3.3 Contributions

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page for this library.

3.4 Versioning

The version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

3.5 Publishing

This library can be published as a package on PyPI by a package maintainer. First, install the dependencies required for packaging and publishing:

```
python -m pip install .[publish]
```

Ensure that the correct version number appears in `pyproject.toml`, and that any links in this README document to the Read the Docs documentation of this package (or its dependencies) have appropriate version numbers. Also ensure that the Read the Docs project for this library has an automation rule that activates and sets as the default all tagged versions. Create and push a tag for this version (replacing `?.?.?` with the version number):

```
git tag ?.?.?
git push origin ?.?.?
```

Remove any old build/distribution files. Then, package the source into a distribution archive:

```
rm -rf build dist src/*.egg-info
python -m build --sdist --wheel .
```

Finally, upload the package distribution archive to PyPI:

```
python -m twine upload dist/*
```

3.5.1 symbolism module

Extensible combinator library for building symbolic expressions that can be evaluated at a later time.

class symbolism.symbolism.**symbol**(*instance: Any*)

Bases: `object`

Instances of this class represent individual symbolic values, as well as entire symbolic expressions (*i.e.*, trees consisting of nested *symbol* instances are represented using the root instance). A symbolic expression involving addition of integers is created in the example below.

```
>>> from symbolism import *
>>> addition = symbol(lambda x, y: x + y)
>>> summation = addition(symbol(1), symbol(2))
```

The expression above can be evaluated at a later time.

```
>>> summation.evaluate()
3
```

Instances are compatible with all built-in infix and prefix operators. When an operator is applied to one or more instances, a new *symbol* instance is created.

```
>>> summation = symbol(1) + symbol(2)
>>> summation.evaluate()
3
```

Pre-defined constants are also provided for all built-in Python operators.

```
>>> conjunction = and_(symbol(True), symbol(False))
>>> conjunction.evaluate()
False
```

__call__(**args*, ***kwargs*) → *symbolism.symbolism.symbol*

Allow creation of a symbolic expression via application of a *symbol* instance to zero or more parameter expressions.

```
>>> add = lambda x, y: x + y
>>> add_ = symbol(add)
>>> e = add_(symbol(1), symbol(2))
>>> isinstance(e, symbol)
True
>>> len(e.parameters)
2
>>> e.parameters[0].instance
1
```

Keyword arguments are also supported. However, note that the keywords are preserved only in the keys of the `parameters` attribute (which in this case is instantiated as a dictionary). The indexing method *__getitem__* and the iteration method *__iter__* only support positional integer indexing and (where applicable) slicing.

```
>>> add = lambda x, y: x + y
>>> add_ = symbol(add)
>>> e = add_(x=symbol(1), y=symbol(2))
>>> isinstance(e, symbol)
True
>>> len(e.parameters)
2
>>> e.parameters['x'].instance
1
```

Positional and keyword arguments cannot be mixed.

```
>>> add = lambda x, y: x + y
>>> add_ = symbol(add)
>>> e = add_(symbol(1), y=symbol(2))
Traceback (most recent call last):
  ...
ValueError: cannot mix positional and keyword arguments
```

__getitem__(*key: Union[int, slice]*) → Union[Any, list, tuple]
Retrieve an instance parameter using an integer index, or retrieve a sequence of instance parameters using a slice.

```
>>> add = lambda x, y: x + y
>>> add_ = symbol(add)
>>> e = add_(symbol(1), symbol(2))
>>> (e[0].instance, e[1].instance)
(1, 2)
>>> [e[i].instance for (i, p) in enumerate(e.parameters)]
[1, 2]
>>> [e[i].instance for (i, p) in enumerate(e)]
[1, 2]
```

Slice notation is also supported when the `parameters` attribute supports it.

```
>>> [s.instance for s in e[0:2]]
[1, 2]
```

__iter__() → Iterable
Allow iteration over instance parameters.

```
>>> add = lambda x, y: x + y
>>> add_ = symbol(add)
>>> e = add_(symbol(1), symbol(2))
>>> [p.instance for p in e]
[1, 2]
>>> 123 in add_(123)
True
```

Even if keyword arguments are used when this instance is instantiated, the iteration returns the actual parameter instances (and **not** the keys of the `parameters` attribute).

```
>>> add = lambda x, y: x + y
>>> add_ = symbol(add)
>>> e = add_(x=symbol(1), y=symbol(2))
>>> [p.instance for p in e]
[1, 2]
>>> 123 in add_(123)
True
```

__len__() → int
The length of an instance corresponds to the number of parameters that it has.

```
>>> add = lambda x, y: x + y
>>> len(symbol(add))
0
>>> e = symbol(add)(symbol(1), symbol(2))
>>> len(e)
2
```

evaluate() → Any
Evaluate a symbolic expression (via recursive evaluation of all subexpressions) and return the result.

```
>>> add = lambda x, y: x + y
>>> e = symbol(add)(symbol(1), symbol(2))
>>> e.evaluate()
3
>>> e = symbol(list.__getitem__)(symbol(['a', 'b', 'c']), symbol(1))
>>> e.evaluate()
'b'
```

__add__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) + symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
5
```

__sub__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) - symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
-1
```

__mul__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) * symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
6
```

__matmul__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> class Test:
...     def __matmul__(self, other):
...         return True
>>> e = symbol(Test()) @ symbol(Test())
>>> isinstance(e, symbol)
True
>>> e.evaluate()
True
```

__truediv__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(5) / symbol(2)
>>> isinstance(e, symbol)
True
```

(continues on next page)

(continued from previous page)

```
>>> e.evaluate()
2.5
```

__floordiv__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(5) // symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
2
```

__mod__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(5) % symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
1
```

__pow__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(5) ** symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
25
```

__lshift__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(4) << symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
16
```

__rshift__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(16) >> symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
4
```

__and__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol({1, 2}) & symbol({2, 3})
>>> isinstance(e, symbol)
```

(continues on next page)

(continued from previous page)

```
True
>>> e.evaluate()
{2}
```

__xor__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol({1, 2}) ^ symbol({2, 3})
>>> isinstance(e, symbol)
True
>>> e.evaluate()
{1, 3}
```

__or__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol({1, 2}) | symbol({2, 3})
>>> isinstance(e, symbol)
True
>>> e.evaluate()
{1, 2, 3}
```

__neg__() → *symbolism.symbolism.symbol*

```
>>> e = -symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
-2
```

__pos__() → *symbolism.symbolism.symbol*

```
>>> e = +symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
2
```

__invert__() → *symbolism.symbolism.symbol*

```
>>> e = ~symbol(2)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
-3
```

__eq__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) == symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
False
```

__ne__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) != symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
True
```

__lt__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) < symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
True
```

__le__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) <= symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
True
```

__gt__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) > symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
False
```

__ge__(*other:* symbolism.symbolism.symbol) → *symbolism.symbolism.symbol*

```
>>> e = symbol(2) >= symbol(3)
>>> isinstance(e, symbol)
True
>>> e.evaluate()
False
```

symbolism.symbolism.**and_ = <symbolism.symbolism.symbol object>**

Symbolic function corresponding to the infix boolean operator and.

symbolism.symbolism.**or_ = <symbolism.symbolism.symbol object>**
Symbolic function corresponding to the infix boolean operator `or`.

symbolism.symbolism.**not_ = <symbolism.symbolism.symbol object>**
Symbolic function corresponding to the prefix boolean operator `not`.

symbolism.symbolism.**in_ = <symbolism.symbolism.symbol object>**
Symbolic function corresponding to the infix operator `in`.

symbolism.symbolism.**is_ = <symbolism.symbolism.symbol object>**
Symbolic function corresponding to the infix operator `is`.

symbolism.symbolism.**add_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__add__*.

symbolism.symbolism.**sub_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__sub__*.

symbolism.symbolism.**mul_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__mul__*.

symbolism.symbolism.**matmul_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__matmul__*.

symbolism.symbolism.**truediv_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__truediv__*.

symbolism.symbolism.**div_ = <symbolism.symbolism.symbol object>**
Concise alias for *symbol.__truediv__*.

symbolism.symbolism.**floordiv_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__floordiv__*.

symbolism.symbolism.**mod_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__mod__*.

symbolism.symbolism.**pow_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__pow__*.

symbolism.symbolism.**lshift_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__lshift__*.

symbolism.symbolism.**rshift_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__rshift__*.

symbolism.symbolism.**bitand_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__and__*.

symbolism.symbolism.**amp_ = <symbolism.symbolism.symbol object>**
Concise alias for *symbol.__and__*.

symbolism.symbolism.**bitxor_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__xor__*.

symbolism.symbolism.**xor_ = <symbolism.symbolism.symbol object>**
Concise alias for *symbol.__xor__*.

symbolism.symbolism.**bitor_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__or__*.

symbolism.symbolism.**bar_ = <symbolism.symbolism.symbol object>**
Concise alias for *symbol.__or__*.

symbolism.symbolism.**invert_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__invert__*.

symbolism.symbolism.**pos_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__pos__*.

symbolism.symbolism.**uadd_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__pos__* (alluding to the name of ast.UAdd).

symbolism.symbolism.**neg_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__neg__*.

symbolism.symbolism.**usub_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__neg__* (alluding to the name of ast.USub).

symbolism.symbolism.**eq_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__eq__*.

symbolism.symbolism.**ne_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__ne__*.

symbolism.symbolism.**lt_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__lt__*.

symbolism.symbolism.**le_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__le__*.

symbolism.symbolism.**gt_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__gt__*.

symbolism.symbolism.**ge_ = <symbolism.symbolism.symbol object>**
Alias for *symbol.__ge__*.

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

D

E

F

G

I

L

M

N

O

P

R

S

T

U

X