are

Release 2.1.0

Reity LLC

Aug 04, 2022

CONTENTS

Library for defining and working with abstract regular expressions that support strings/sequences with elements of any symbol type, with an emphasis on supporting scenarios in which it is necessary to work with regular expressions as abstract mathematical objects.

CHAPTER

ONE

PURPOSE

This library provides classes that enable concise construction of abstract regular expressions. In the case of this library, the term *abstract* refers to the fact that the symbols that constitute the abstract *strings* (*i.e.*, iterable sequences) that satisfy an abstract regular expression can be values or objects of any immutable type. Thus, this library also makes it possible to determine whether an iterable containing zero or more objects satisfies a given abstract regular expression. Any abstract regular expression can also be converted into a nondeterministic finite automaton (as implemented within another package) that accepts exactly those iterables which satisfy that abstract regular expression.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install are
```

The library can be imported in the usual way:

```
import are
from are import *
```

2.1 Examples

This library makes it possible to construct abstract regular expressions (represented as instances of the `are` class) that work with a chosen symbol type. In the example below, a regular expression is defined (using the literal operator `lit` and concatenation operator `con`) in which symbols are integers. It is then applied to an iterable of integers. This returns the iterable's length (as an integer) if that iterable satisfies the abstract regular expression:

```
>>> from are import *
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a([1, 2, 3])
3
```

If the longest prefix of an iterable that satisfies an abstract regular expression is desired, the `full` parameter can be set to `False`:

```
>>> a([1, 2, 3, 4, 5], full=False)
3
```

Operators for alternation and repetition of abstract regular expressions are also available:

```
>>> a = rep(con(lit(1), lit(2)))
>>> a([1, 2, 1, 2, 1, 2])
6
>>> a = alt(rep(lit(2)), rep(lit(3)))
>>> a([2, 2, 2, 2, 2])
5
>>> a([3, 3, 3, 3])
4
```

The `emp` constructor can be used to create an abstract regular expression that is satisfied by the empty iterable:

```
>>> a = emp()
>>> a([])
∅
```

The nul constructor can be used to create an abstract regular expression that cannot be satisfied:

```
>>> a = nul()
>>> a([]) is None
True
>>> a([1, 2, 3]) is None
True
```

An abstract regular expression that has only string symbols can be converted into a regular expression string that is compatible with the built-in re library:

```
>>> a = alt(lit('x'), rep(lit('y')))
>>> r = a.to_re()
>>> r
'(((x)*)|((y)*))'
>>> import re
>>> r = re.compile(a.to_re())
>>> r.fullmatch('yyy')
<re.Match object; span=(0, 3), match='yyy'>
```

An abstract regular expression can also be converted into an NFA representation (as implemented within the PyPI package):

```
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a.to_nfa()
nfa({1: nfa({2: nfa({3: nfa()})})})
```

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
```

The subset of the unit tests included in the module itself can be executed using doctest:

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

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/are test/test_are.py
```

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

Beginning with version 0.1.0, 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 are module

Library for defining and operating on abstract regular expressions that work with any symbol type, with an emphasis on supporting scenarios in which it is necessary to work with regular expressions as abstract mathematical objects.

class are.are.**are**(*iterable=()*, */*)

Bases: tuple

Base class for abstract regular expression instances (and the individual nodes found within an abstract syntax tree instance). Abstract regular expressions can contain symbols of any immutable type and can be built up using common operators such as concatenation, alternation, and repetition.

```
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a([1, 2, 3])
3
```

This class is derived from the built-in `tuple` type. Each instance of this class acts as a node within the abstract syntax tree representing an abstract regular expression. The elements inside the instance are the child nodes of the node represented by the instance and can be accessed in the usual manner supported by the `tuple` type.

```
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a[1]
con(lit(2), lit(3))
```

to_nfa() → nfa.nfa.nfa

Convert this abstract regular expression instance into a nondeterministic finite automaton (NFA) that accepts the set of iterables that satisfies this instance.

```
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a.to_nfa()
nfa({1: nfa({2: nfa({3: nfa()})})})
```

compile() → *are.are.are*

Convert this instance into an equivalent NFA and store it internally as an attribute (to enable more efficient matching). Return the original abstract regular expression instance.

```
>>> a = alt(lit('x'), rep(con(lit('y'), lit('z'))))
>>> a = a.compile()
>>> a(['x'])
1
>>> a(['y', 'z', 'y', 'z'])
4
```

to_re() → str

If this instance has string symbols (and no other symbols of any other type), convert it to an equivalent regular expression string that is compatible with the built-in `re` module.

```
>>> rep(alt(con(lit('a'), lit('b')), emp())).to_re()
'((((a)(b))|)*)'
>>> rep(alt(con(lit('a'), con(lit('b'), nul())), emp())).to_re()
'((((a)((b)[^\\w\\W]))|)*)'
```

Any attempt to convert an instance that has non-string symbols raises an exception.

```
>>> rep(alt(con(lit(123), lit(456)), emp())).to_re()
Traceback (most recent call last):
  ...
TypeError: all symbols must be strings
```

__call__(*string: Iterable*, *full: bool = True*) → Optional[int]

Determine whether an iterable of symbols (*i.e.*, an abstract *string* in the formal sense associated with the mathematical definition of a regular expression) is in the formal language represented by this instance. By default, the length of the abstract string is returned if the abstract string satisfies this instance.

```
>>> a = rep(con(lit(1), lit(2)))
>>> a([1, 2, 1, 2, 1, 2])
6
>>> a = alt(rep(lit(2)), rep(lit(3)))
>>> a([2, 2, 2, 2, 2])
5
```

(continues on next page)

(continued from previous page)

```
>>> a([3, 3, 3, 3])
4
```

If the supplied abstract string does not satisfy this instance, then `None` is returned.

```
>>> a([1, 1, 1]) is None
True
```

If the optional parameter `full` is set to `False`, then the length of the longest prefix of the abstract string that satisfies this instance is returned. If no prefix satisfies this instance, then `None` is returned.

```
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a([1, 2, 3, 4, 5], full=False)
3
>>> a = con(lit(1), con(lit(2), lit(3)))
>>> a([4, 4, 4], full=False) is None
True
```

If an instance is satisfied by the empty abstract string and `full` is set to `False`, then the empty prefix of any abstract string satisfies the abstract regular expression instance (and, thus, a successful integer result of `0` is returned in such cases).

```
>>> a = alt(lit(2), emp()) # Satisfied by the empty abstract string.
>>> a([1, 1, 1], full=False) # Empty string is a prefix of ``[1, 1, 1]``.
0
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> nul()(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

__str__() → str

Return string representation of instance.

```
>>> a = rep(con(lit(1), alt(lit(2), lit(3))))
>>> str(a)
'rep(con(lit(1), alt(lit(2), lit(3))))'
```

Assuming that this module has been imported in a manner such that the *are* subclasses are associated with the variables as they appear in this module (*e.g.*, *con* is associated with the variable `con` in the relevant scope), the strings returned by this method can be evaluated to reconstruct the instance.

```
>>> a = rep(con(lit(1), alt(lit(2), lit(3))))
>>> eval(str(a))
rep(con(lit(1), alt(lit(2), lit(3))))
```

__repr__() → str

Return string representation of instance.

```
>>> rep(alt(con(lit('a'), lit('b')), emp()))
rep(alt(con(lit('a'), lit('b')), emp()))
```

class are.are.**nul**

Bases: *are.are.are*

Singleton class containing an object that corresponds to the sole abstract regular expression instance that cannot be satisfied by any iterable (*i.e.*, that cannot be satisfied by any abstract string).

```
>>> (nul()(iter('ab')), nul()(iter('abc'), full=False))
(None, None)
>>> r = nul()
>>> (r(''), r('abc'), r('', full=False), r('abc', full=False))
(None, None, None, None)
```

More usage examples involving compilation of *are* instances that contain instances of this class are presented below.

```
>>> r = r.compile()
>>> (r(''), r('abc'), r('', full=False), r('abc', full=False))
(None, None, None, None)
>>> ((con(nul(), lit('a')))('a'), (con(nul(), lit('a'))).compile()('a'))
(None, None)
>>> ((con(lit('a'), nul()))('a'), (con(lit('a'), nul())).compile()('a'))
(None, None)
>>> ((alt(nul(), lit('a')))('a'), (alt(nul(), lit('a'))).compile()('a'))
(1, 1)
>>> ((alt(lit('a'), nul()))('a'), (alt(lit('a'), nul())).compile()('a'))
(1, 1)
>>> ((alt(nul(), nul()))('a'), (alt(nul(), nul())).compile()('a'))
(None, None)
>>> (con(rep(nul()), lit('a')).compile())('a')
1
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> nul()(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

class are.are.**emp**

Bases: *are.are.are*

Singleton class containing an object that corresponds to the sole abstract regular expression instance that is satisfied only by an empty iterable (*i.e.*, an abstract string with a length of zero).

```
>>> (emp()(''), emp()('ab'))
(0, None)
>>> emp()(iter('ab')) is None
True
>>> emp()('abc', full=False)
0
>>> emp()(iter('abc'), full=False)
0
>>> r = emp().compile()
>>> (r(''), r('abc'))
(0, None)
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> emp()(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

class are.are.**lit**(*argument*)

Bases: *are.are.are*

Abstract regular expression instances that are satisfied by exactly one symbol. Instances of this class also serve as the leaf nodes (*i.e.*, base cases) corresponding to abstract string *literals* (in the formal sense associated with the mathematical definition of a regular expression).

```
>>> (lit('a')(''), lit('a')('a'), lit('a')('ab'))
(None, 1, None)
>>> (lit('a')('', full=False), lit('a')('ab', full=False))
(None, 1)
>>> lit('a')(iter('ab'), full=False)
1
>>> r = lit('a').compile()
>>> (r('a'), r(''))
(1, None)
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> lit('a')(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

class are.are.**con**(**arguments*)

Bases: *are.are.are*

Concatenation operation for two *are* instances. Instances of this class also serve as the internal nodes of the tree data structure representing an abstract regular expression.

```
>>> r = con(lit('a'), lit('b'))
>>> (r('ab'), r('a'), r('abc'), r('cd'))
(2, None, None, None)
>>> (r(iter('ab')), r(iter('a')), r(iter('abc')), r(iter('cd')))
(2, None, None, None)
>>> (r('a', full=False), r('abc', full=False), r('cd', full=False))
(None, 2, None)
>>> (r(iter('a'), full=False), r(iter('abc'), full=False), r(iter('cd'),␣
↪full=False))
(None, 2, None)
>>> r = con(lit('a'), con(lit('b'), lit('c')))
>>> (r('abc'), r('abcd', full=False), r('ab'))
(3, 3, None)
>>> (r(iter('abc')), r(iter('abcd'), full=False), r(iter('ab')))
(3, 3, None)
>>> r = con(con(lit('a'), lit('b')), lit('c'))
>>> r('abc')
3
```

(continues on next page)

(continued from previous page)

```
>>> r(iter('abc'))
3
>>> r = con(lit('a'), lit('b')).compile()
>>> (r('ab'), r('a'), r('abc'), r('cd'))
(2, None, None, None)
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> r(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

class are.are.**alt**(**arguments*)

Bases: *are.are.are*

Alternation operation for two *are* instances. Instances of this class also serve as the internal nodes of the tree data structure representing an abstract regular expression.

```
>>> r = alt(con(lit('a'), lit('a')), lit('a'))
>>> r('aa')
2
>>> r = alt(lit('b'), con(lit('a'), lit('a')))
>>> r('aa')
2
>>> r = con(alt(lit('a'), lit('b')), alt(lit('c'), lit('d')))
>>> (r('ac'), r('ad'), r('bc'), r('bd'))
(2, 2, 2, 2)
>>> r = con(alt(lit('a'), lit('b')), lit('c'))
>>> (r('ac'), r('bc'), r('c'), r('a'), r('b'))
(2, 2, None, None, None)
>>> r = alt(con(lit('a'), lit('b')), lit('a'))
>>> r('abc', full=False)
2
>>> r = alt(lit('a'), con(lit('a'), lit('a')))
>>> r('aaa', full=False)
2
>>> r = alt(lit('a'), con(lit('a'), lit('a')))
>>> r('aaa') is None
True
>>> r = alt(con(lit('a'), lit('a')), lit('a'))
>>> r('aa', full=False)
2
>>> r = alt(lit('a'), lit('a'))
>>> r('a')
1
>>> r = alt(lit('a'), lit('b'))
>>> r('ac') is None
True
>>> (r('a'), r('b'), r('c'))
(1, 1, None)
>>> r('ac', full=False)
1
```

(continues on next page)

(continued from previous page)

```
>>> r0 = alt(lit('a'), alt(lit('b'), lit('c')))
>>> r1 = con(r0, r0)
>>> {r1(x + y) for x in 'abc' for y in 'abc'}
{2}
>>> r = alt(lit('b'), con(lit('c'), lit('a')))
>>> r('aab') is None
True
>>> r(iter('aab')) is None
True
>>> r = alt(con(lit('a'), lit('a')), con(lit('a'), con(lit('a'), lit('a'))))
>>> (r('aaa'), r('aa'))
(3, 2)
>>> r = alt(con(lit('a'), lit('a')), con(lit('a'), con(lit('a'), lit('a'))))
>>> (r('aaa', full=False), r('aa', full=False))
(3, 2)
>>> (r(iter('aaa'), full=False), r(iter('aa'), full=False))
(3, 2)
>>> r = alt(con(lit('a'), lit('a')), con(lit('a'), con(lit('a'), lit('a'))))
>>> r = r.compile()
>>> (r('aaa'), r('aa'), r('a'))
(3, 2, None)
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> r(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

class are.are.**rep**(*argument*)

Bases: *are.are.are*

Repetition operation (zero or more times) for an *are* instance. Instances of this class also serve as the internal nodes of the tree data structure representing an abstract regular expression.

```
>>> r = rep(lit('a'))
>>> all([r('a'*i) == i for i in range(100)])
True
>>> {r('a'*i + 'b') for i in range(10)}
{None}
>>> {r(iter('a'*i + 'b')) for i in range(10)}
{None}
>>> r = con(lit('a'), rep(lit('b')))
>>> r('a' + 'b'*10)
11
>>> r(iter('a' + 'b'*10))
11
>>> r = con(rep(lit('a')), lit('b'))
>>> r('aaab')
4
>>> r(iter('aaab'))
4
>>> r = con(rep(lit('a')), lit('b')).compile()
```

(continues on next page)

(continued from previous page)

```
>>> r('aaab')
4
>>> r = rep(lit('a')).compile()
>>> all([r('a'*i) == i for i in range(100)])
True
```

Note that the empty abstract string satisfies any instance of this class.

```
>>> r('')
0
>>> r('bbbb', full=False)
0
>>> all([r('a'*i + 'b', full=False) == i for i in range(100)])
True
>>> all([r(iter('a'*i + 'b'), full=False) == i for i in range(100)])
True
```

Any attempt to apply an abstract regular expression instance to a non-iterable raises an exception.

```
>>> r(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
```

PYTHON MODULE INDEX

INDEX

Symbols

A

C

E

L

M

N

R

T