nfa

Release 3.1.0

Reity LLC

Aug 04, 2022

CONTENTS

Pure-Python library for building and working with nondeterministic finite automata (NFAs).

CHAPTER

ONE

PURPOSE

This library makes it possible to concisely construct nondeterministic finite automata (NFAs) using common Python data structures and operators, as well as to perform common operations involving NFAs. NFAs are represented using a class derived from the Python dictionary type, wherein dictionary objects serve as individual states and dictionary entries serve as transitions (with dictionary keys representing transition labels).

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install nfa
```

The library can be imported in the usual way:

```
import nfa
from nfa import nfa
```

2.1 Examples

This library makes it possible to concisely construct an NFA by using one or more instances of the `nfa` class. In the example below, an NFA is defined in which transition labels are strings:

```
>>> from nfa import nfa
>>> n = nfa({'a': nfa({'b': nfa({'c': nfa()})})})
```

The `nfa` object can be applied to a sequence of symbols (represented as an iterable of transition labels). This returns the length (as an integer) of the longest path that (1) traverses an ordered sequence of the NFA's transitions whose labels match the sequence of symbols supplied as the argument and (2) terminates at an accepting state:

```
>>> n(['a', 'b', 'c'])
3
```

By default, an empty NFA object `nfa()` is an accepting state and a non-empty object is *not* an accepting state. When an NFA is applied to an iterable of labels that does not traverse a path that leads to an accepting state, `None` is returned:

```
>>> n(['a', 'b']) is None
True
```

To ensure that a state is not accepting (even if it is empty), the built-in prefix operator `-` can be used:

```
>>> n = nfa({'a': nfa({'b': nfa({'c': -nfa()})})})
>>> n(['a', 'b', 'c']) is None
True
```

The prefix operator `+` returns an accepting state and the prefix operator `~` reverses whether a state is accepting:

```
>>> n = nfa({'a': ~nfa({'b': +nfa({'c': nfa()})})})
>>> n(['a'])
1
>>> n(['a', 'b'])
2
```

Applying the built-in `bool` function to an `nfa` object returns a boolean value indicating whether *that specific object* (and *not* the overall NFA within which it may be an individual state) is an accepting state:

```
>>> bool(n)
False
>>> bool(nfa())
True
>>> bool(-nfa())
False
```

Epsilon transitions can be introduced using the `epsilon` object:

```
>>> from nfa import epsilon
>>> n = nfa({'a': nfa({epsilon: nfa({'b': nfa({'c': nfa()})})})})
>>> n(['a', 'b', 'c'])
3
```

If an NFA instance is applied to an iterable that yields enough symbols to reach an accepting state but has additional symbols remaining, `None` is returned:

```
>>> n(['a', 'b', 'c', 'd', 'e']) is None
True
```

If the length of the longest path leading to an accepting state is desired (even if additional symbols remain in the iterable), the `full` parameter can be set to `False`:

```
>>> n(['a', 'b', 'c', 'd', 'e'], full=False)
3
```

It is possible to retrieve the set of all transition labels that are found in the overall NFA (note that this does not include instances of `epsilon`):

```
>>> n.symbols()
{'c', 'a', 'b'}
```

Because the `nfa` class is derived from `dict`, it supports all operators and methods that are supported by `dict`. In particular, the state reachable from a given state via a transition that has a specific label can be retrieved by using index notation:

```
>>> n.keys()
dict_keys(['a'])
>>> m = n['a']
>>> m(['b', 'c'])
2
```

To retrieve the collection of *all* states that can be reached via paths that involve zero or more epsilon transitions (and no labeled transitions), the built-in infix operator `%` can be used (note that this also includes *all* intermediate states along the paths to the first labeled transitions):

```
>>> b = nfa({epsilon: nfa({'b': nfa()})})
>>> c = nfa({'c': nfa()})
>>> n = nfa({epsilon: [b, c]})
>>> for s in (n % epsilon):
...     print(s)
...
nfa({epsilon: [nfa({epsilon: nfa({'b': nfa()})}), nfa({'c': nfa()})]})
nfa({epsilon: nfa({'b': nfa()})})
nfa({'c': nfa()})
nfa({'b': nfa()})
```

Other methods make it possible to retrieve all the states found in an NFA, to compile an NFA (enabling more efficient processing of iterables), and to transform an NFA into a deterministic finite automaton (DFA). Descriptions and examples of these methods can be found in the documentation for the main library module.

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/nfa/nfa.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/nfa test/test_nfa.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

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 nfa module

Pure-Python data structure derived from the built-in `dict` type that can represent nondeterministic finite automata (NFAs) as an ensemble of dictionaries (where dictionary instances serve as nodes, dictionary keys serve as edge labels, and dictionary values serve as edges).

class nfa.nfa.**nfa**(*argument=None*)

Bases: `dict`

An instance of this class can represent an individual state within a nondeterministic finite automaton (NFA). When a state represented by an instance of this class is understood to be a starting state, it also represents an NFA as a whole that consists of all states that are reachable from that starting state.

While instances of this class serve as individual NFA states, entries within the instances represent transitions between states (with keys serving as transition labels). In the example below, an NFA with four states and three transitions is defined. The transition labels are `'a'`, `'b'`, and `'c'`.

```
>>> n = nfa({'a': nfa({'b': nfa({'c': nfa()})})})
```

Strings of symbols (in the formal sense associated with the formal definition of NFAs) are represented using iterable sequences of Python values or objects that can serve as dictionary keys. Applying an instance of this class to an iterable sequences of symbols returns the length (as an integer) of the longest path that (1) traverses an ordered sequence of transitions whose labels match the sequence of symbols supplied as the argument and (2) terminates at an accepting state.

```
>>> n(['a', 'b', 'c'])
3
```

The *epsilon* object defined within this module can be used to represent unlabeled transitions.

```
>>> a = nfa({'a': nfa({epsilon: nfa()})})
>>> a('a', full=False)
1
>>> a = nfa({'a': nfa({epsilon: nfa({'b': nfa()})})})
>>> a('a', full=False) is None
True
```

Increasingly complex NFAs can be represented by building up instances of this class; cycles can be introduced by adding references to instances that have already been defined.

```
>>> accept = nfa()
>>> abc = nfa({'a':accept, 'b':accept, 'c':accept})
>>> abc('a')
1
>>> (abc('ab'), abc(iter('ab')))
(None, None)
>>> d_abc = nfa({'d': abc})
>>> d_abc('db')
2
>>> d_abc('d') is None
True
>>> d_abc('c') is None
True
>>> d_abc('b') is None
True
>>> f_star_e_d_abc = nfa({'e': d_abc})
>>> f_star_e_d_abc('edb')
3
>>> f_star_e_d_abc['f'] = f_star_e_d_abc
>>> all(f_star_e_d_abc(('f'*i) + 'edb') == i + 3 for i in range(5))
True
>>> f_star_e_d_abc['f'] = [f_star_e_d_abc]
>>> all(f_star_e_d_abc(('f'*i) + 'edb') == i + 3 for i in range(5))
True
>>> f_star_e_d_abc['f'] = [f_star_e_d_abc, abc]
>>> all(f_star_e_d_abc(('f'*i) + x) == i + 1 for i in range(1,5) for x in 'abc')
True
>>> set(f_star_e_d_abc(iter(('f'*5) + x))  for _ in range(1,5) for x in 'abc')
{6}
>>> b_star_c = nfa({'c':accept})
```

(continues on next page)

(continued from previous page)

```
>>> b_star_c['b'] = b_star_c
>>> (b_star_c('bbbbc'), b_star_c(iter('bbbbc')))
(5, 5)
>>> b_star_c(['b', 'b', 'b', 'b', 'c'])
5
>>> b_star_c((c for c in ['b', 'b', 'b', 'b', 'c']))
5
>>> b_star_c[epsilon] = abc
>>> (b_star_c('a'), b_star_c('b'), b_star_c('c'), b_star_c('d'))
(1, 1, 1, None)
>>> abc[epsilon] = abc
>>> (abc('a'), abc('b'), abc('c'), abc('d'))
(1, 1, 1, None)
>>> b_star_c[epsilon] = [abc, b_star_c]
>>> (b_star_c('a'), b_star_c('b'), b_star_c('c'), b_star_c('d'))
(1, 1, 1, None)
```

static __new__(*cls*, *argument=None*)

Constructor for an instance that enforces constraints on argument types (*i.e.*, NFA instances can only have other NFA instances or lists/tuples thereof as values).

```
>>> nfa()
nfa()
>>> n = nfa({'a': nfa()})
>>> n = nfa({'a': [nfa()]})
>>> n = nfa({'a': (nfa(),)})
>>> n = nfa([('x', nfa())])
>>> n = nfa(list(zip(['a', 'b'], [nfa(), nfa()])))
```

Any attempt to construct an *nfa* instance that does not contain other *nfa* instances (or lists/tuples of *nfa* instances) raises an exception.

```
>>> len(nfa(zip(['a', 'b'], [nfa(), nfa()])))
Traceback (most recent call last):
  ...
TypeError: argument must be a collection
>>> nfa({'a': []})
Traceback (most recent call last):
  ...
TypeError: values must be nfa instances or non-empty lists/tuples of nfa␣
↪instances
>>> nfa({'a': [123]})
Traceback (most recent call last):
  ...
TypeError: values must be nfa instances or non-empty lists/tuples of nfa␣
↪instances
>>> nfa([1, 2])
Traceback (most recent call last):
  ...
TypeError: values must be nfa instances or non-empty lists/tuples of nfa␣
↪instances
>>> nfa({'x': 123})
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
TypeError: values must be nfa instances or non-empty lists/tuples of nfa␣
↪instances
>>> nfa({'x': [123]})
Traceback (most recent call last):
  ...
TypeError: values must be nfa instances or non-empty lists/tuples of nfa␣
↪instances
```

__bool__() → bool

Return a boolean indicating whether the state represented by this *nfa* instance is an accepting state.

Be default, a non-empty instance *is not* an accepting state and an empty instance *is* an accepting state.

```
>>> bool(nfa())
True
>>> bool(nfa({'a': nfa()}))
False
```

Modifying an empty instance by adding a transition entry to it causes it to become a non-accepting state. In the example below, `cycle` does not accept any string because it contains no accepting states.

```
>>> cycle = nfa()
>>> cycle['a'] = cycle
>>> bool(cycle)
False
>>> cycle('a') is None
True
```

__pos__() → *nfa.nfa.nfa*

Return a shallow copy of this instance with the state represented by this *nfa* instance marked as an accepting state.

```
>>> n = nfa({'a': nfa({'b': nfa()})})
>>> n('a') is None
True
>>> n = nfa({'a': +nfa({'b': nfa()})})
>>> n('a')
1
```

__neg__() → *nfa.nfa.nfa*

Return a shallow copy of this instance with the state represented by this *nfa* instance marked as a non-accepting state.

```
>>> none = nfa({'a': nfa({'b': -nfa()})})
>>> none('a') is None
True
>>> none(['a', 'b'], full=False) is None
True
```

__invert__() → *nfa.nfa.nfa*

Return a shallow copy of this instance that has an accepting status that is the opposite of the accepting status of this instance.

```
>>> none = nfa({'a': nfa({'b': ~nfa()})})
>>> none('a') is None
True
>>> none(['a', 'b'], full=False) is None
True
>>> none['a']['b'] = ~none['a']['b']
>>> none(['a', 'b'])
2
```

__mod__(*argument: Any*) → Iterable[*nfa.nfa.nfa*]

Return an iterable of zero or more *nfa* instances reachable using any path that has exactly one transition labeled with the supplied argument (and any number of *epsilon* transitions). If the supplied argument is itself *epsilon*, then all states reachable via zero or more *epsilon* transitions are returned.

```
>>> a = nfa({'a': nfa()})
>>> b = nfa({epsilon: [a]})
>>> c = nfa({epsilon: [a]})
>>> b[epsilon].append(c)
>>> c[epsilon].append(b)
>>> n = nfa({epsilon: [b, c]})
>>> len(n % epsilon)
4
>>> a = nfa({'a': nfa()})
>>> b = nfa({epsilon: [a]})
>>> c = nfa({epsilon: [a]})
>>> b[epsilon].append(c)
>>> c[epsilon].append(b)
>>> n = nfa({epsilon: [b, c]})
>>> len(n % 'a')
1
```

__matmul__(*argument: Any*) → Sequence[*nfa.nfa.nfa*]

Return a list of zero or more *nfa* instances reachable using a single transition that has a label (either *epsilon* or a symbol) matching the supplied argument.

```
>>> n = nfa({'a': nfa({'b': nfa({'c': nfa()})})})
>>> n @ 'a'
[nfa({'b': nfa({'c': nfa()})})]
>>> n @ 'b'
[]
>>> n = nfa({epsilon: [nfa({'a': nfa()}), nfa({'b': nfa()})]})
>>> n @ epsilon
[nfa({'a': nfa()}), nfa({'b': nfa()})]
```

compile()

Compile the NFA represented by this instance (*i.e.*, the NFA in which this instance is the starting state) into a transition table and store the table within a private attribute of this instance.

```
>>> final = nfa()
>>> middle = +nfa({456: final})
>>> first = nfa({123: middle})
>>> (first([123]), first([123, 456]), first([456]))
(1, 2, None)
```

(continues on next page)

(continued from previous page)

```
>>> first = first.compile()
>>> (first([123]), first([123, 456]), first([456]))
(1, 2, None)
```

Compilation can improve performance when applying instances to iterable sequences of symbols.

```
>>> zeros = nfa({epsilon: nfa()})
>>> zeros[0] = [zeros]
>>> zeros = zeros.compile()
>>> all(zeros([0] * i) == i for i in range(10))
True
```

Compilation does not affect what sequences are accepted, and invoking this method multiple times for the same instance has no new effects.

```
>>> empty = nfa({epsilon: nfa()})
>>> (empty(''), empty('', full=False))
(0, 0)
>>> empty = empty.compile()
>>> (empty(''), empty('', full=False))
(0, 0)
>>> (empty('a'), empty('a', full=False))
(None, 0)
>>> empty = empty.compile()
>>> (empty('a'), empty('a', full=False))
(None, 0)
>>> a = nfa({'a': nfa()})
>>> (a(''), a('', full=False))
(None, None)
>>> a = a.compile()
>>> (a(''), a('', full=False))
(None, None)
>>> a = +a
>>> a('', full=False)
0
>>> a = a.compile()
>>> a('', full=False)
0
>>> cycle = nfa()
>>> cycle['a'] = cycle
>>> bool(cycle)
False
>>> (cycle('a'), cycle('a', full=False))
(None, None)
>>> cycle = cycle.compile()
>>> (cycle('a'), cycle('a', full=False))
(None, None)
>>> reject = nfa({epsilon: ~nfa()})
>>> (reject(''), reject('', full=False))
(None, None)
>>> reject = reject.compile()
>>> (reject(''), reject('', full=False))
(None, None)
```

states(*argument: Optional[Any] = None*) → Sequence[*nfa.nfa.nfa*]

Return list of all states (*i.e.*, the corresponding *nfa* instances) reachable from this instance, or the set of states reachable via any *one* transition that has a label matching the supplied argument.

```
>>> abcd = nfa({'a': nfa({'b': nfa({'c': nfa()})})})
>>> abcd['a']['b']['d'] = nfa()
>>> [list(sorted(state.symbols())) for state in abcd.states()]
[['a', 'b', 'c', 'd'], ['b', 'c', 'd'], ['c', 'd'], [], []]
>>> [
...     [list(sorted(s.symbols())) for s in state.states(list(state.keys())[0])]
...     for state in abcd.states()
...     if len(state.keys()) > 0
... ]
[[['b', 'c', 'd']], [['c', 'd']], [[]]]
```

All states (including accepting empty states and states that have only epsilon transitions) are included.

```
>>> none = nfa({epsilon: nfa()})
>>> len([s for s in none.states()])
2
```

symbols() → set

Return set of all symbols found in transitions of this *nfa* instance.

```
>>> nfa({'a': nfa({'b': nfa({'c': nfa()})})}).symbols() == {'a', 'b', 'c'}
True
```

to_dfa() → *nfa.nfa.nfa*

Compile the NFA represented by this instance (*i.e.*, the NFA in which this instance is the starting state) into a deterministic finite automaton (DFA) that accepts the same set of symbol sequences. The DFA is represented as an *nfa* instance but has an internal structure that conforms to the constraints associated with DFAs (*i.e.*, no nondeterministic collections of transitions and no *epsilon* transitions).

```
>>> final = nfa()
>>> middle = +nfa({456:final})
>>> first = nfa({123:middle})
>>> first = first.to_dfa()
>>> (first([123]), first([123, 456]), first([456]))
(1, 2, None)
>>> first = first.compile()
>>> (first([123]), first([123, 456]), first([456]))
(1, 2, None)
>>> accept = nfa()
>>> three = nfa({3:accept})
>>> two = nfa({2:three})
>>> one = nfa({1:two})
>>> zero = nfa({0:one, 2:three})
>>> zero = zero.to_dfa()
>>> (zero([0, 1, 2, 3]), zero([2, 3]), zero([2, 2, 3]))
(4, 2, None)
>>> (zero([0, 1, 2, 3, 4], full=False), zero([2, 3, 4], full=False), zero([2],␣
↪full=False))
(4, 2, None)
>>> accept = nfa().compile()
```

(continues on next page)

(continued from previous page)

```
>>> accept('')
0
>>> accept('a') is None
True
>>> a_star = +nfa()
>>> a_star['a'] = a_star
>>> a_star = a_star.compile()
>>> all(a_star('a'*i) == i for i in range(10))
True
>>> a_star('b') is None
True
>>> n = nfa()
>>> n[epsilon] = n
>>> n([]) is None
True
>>> d = n.to_dfa()
>>> d([]) is None
True
```

is_dfa() → bool

Return a boolean that indicates whether the NFA represented by this instance satisfies the definition of a deterministic finite automaton (DFA).

```
>>> m = nfa({'a': nfa({'b': nfa({'c': nfa()})})})
>>> m.is_dfa()
True
>>> n = nfa({'d': [m, nfa()]})
>>> n.is_dfa()
False
>>> o = nfa({'e': [nfa({epsilon: m})]})
>>> o.is_dfa()
False
```

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

Determine whether a supplied *string* – in the formal sense (*i.e.*, any iterable sequence of symbols) – is accepted by this *nfa* instance.

```
>>> final = nfa()
>>> middle = +nfa({456:final})
>>> first = nfa({123:middle})
>>> (first([123]), first([123, 456]), first([456]))
(1, 2, None)
>>> first = first.compile()
>>> (first([123]), first([123, 456]), first([456]))
(1, 2, None)
>>> accept = nfa()
>>> three = nfa({3:accept})
>>> two = nfa({2:three})
>>> one = nfa({1:two})
>>> zero = nfa({0:one, 2:three})
>>> (zero([0, 1, 2, 3]), zero([2, 3]), zero([2, 2, 3]))
(4, 2, None)
```

(continues on next page)

(continued from previous page)

```
>>> (zero([0, 1, 2, 3, 4], full=False), zero([2, 3, 4], full=False), zero([2],␣
↪full=False))
(4, 2, None)
>>> zero = nfa({0:one, epsilon:[two, three]}).compile()
>>> (zero([0, 1, 2, 3]), zero([2, 3]), zero([3]), zero([2, 2, 3]))
(4, 2, 1, None)
>>> (zero([0, 1, 2, 3, 4], full=False), zero([2, 3, 4], full=False), zero([2],␣
↪full=False))
(4, 2, None)
>>> zeros = nfa({epsilon:[accept]})
>>> zeros[0] = [zeros]
>>> all(zeros([0]*i) == i for i in range(10))
True
>>> zeros = nfa({0:[accept]})
>>> zeros[0].append(zeros)
>>> all(zeros([0]*i) == i for i in range(1, 10))
True
>>> all(zeros([0]*i, full=False) == i for i in range(1, 10))
True
>>> zeros = zeros.compile()
>>> all(zeros([0]*i) == i for i in range(1, 10))
True
>>> all(zeros([0]*i, full=False) == i for i in range(1, 10))
True
```

A sequence of symbols of length zero is accepted if only epsilon transitions are traversed to reach an accepting state. If no accepting state can be reached, it is rejected.

```
>>> none = nfa({epsilon: nfa()})
>>> none('')
0
>>> none = nfa({epsilon: -nfa()})
>>> none('') is None
True
```

Any attempt to apply an instance to a non-sequence or other invalid argument raises an exception.

```
>>> accept = nfa()
>>> accept(123)
Traceback (most recent call last):
  ...
ValueError: input must be an iterable
>>> accept([epsilon])
Traceback (most recent call last):
  ...
ValueError: input cannot contain epsilon
```

__str__() → str

Return a string representation of this instance.

```
>>> nfa()
nfa()
>>> nfa({'a': nfa({'b':(nfa(),)})})
```

(continues on next page)

(continued from previous page)

```
nfa({'a': nfa({'b': (nfa(),)})})
>>> nfa({'a': [nfa({'b': [nfa()]})]})
nfa({'a': [nfa({'b': [nfa()]})]})
```

Assuming that this module has been imported in a manner such that the *nfa* class is associated with the variable `nfa`, instances that represent small NFAs that do not contain cycles yield strings that can be evaluated successfully to reconstruct the instance.

```
>>> n = nfa({'a': nfa({'b':(nfa(),)})})
>>> eval(str(n))
nfa({'a': nfa({'b': (nfa(),)})})
```

Instances that have cycles are not converted into a string beyond any depth at which a state repeats.

```
>>> cycle = nfa({'a': nfa()})
>>> cycle['a']['b'] = cycle
>>> cycle
nfa({'a': nfa({'b': nfa({...})})})
```

Instances that have been designated as accepting (or as non-accepting) in a manner that deviates from the default are marked as such with the appropriate prefix operator.

```
>>> +nfa({'a': nfa()})
+nfa({'a': nfa()})
>>> +nfa({'a': -nfa()})
+nfa({'a': -nfa()})
```

Any attempt to convert an instance that contains invalid values into a string raises an exception.

```
>>> cycle['a'] = 123
>>> cycle
Traceback (most recent call last):
  ...
TypeError: values must be nfa instances or non-empty lists/tuples of nfa␣
↪instances
```

__repr__() → str
Return string representation of instance. Instances that represent small NFAs that do not contain cycles yield strings that can be evaluated.

copy() → *nfa.nfa.nfa*
Return a deep copy of this instance in which all reachable instances of *nfa* are new copies but references to all other objects are not copies.

```
>>> m = nfa({'a': nfa({'b': nfa({'c': nfa()})})})
>>> n = m.copy()
>>> n('abc')
3
>>> set(map(id, m.states())) & set(map(id, n.states()))
set()
```

This method behaves as expected when cycles exist.

```
>>> a_star = +nfa()
>>> a_star['a'] = a_star
>>> a_star['b'] = [nfa()]
>>> a_star_ = a_star.copy()
>>> all(a_star_('a' * i) == i for i in range(10))
True
>>> a_star_('b')
1
>>> a_star_('c') is None
True
```

nfa.nfa.**epsilon = epsilon**

Constant representing an *epsilon* transition when used as an edge label (*i.e.*, as a key within an *nfa* instance).

```
>>> a = nfa({'a': nfa({epsilon: nfa({'b': nfa()})})})
>>> a('ab')
2
```

This object is hashable, is equivalent to itself, and can be converted into a string.

```
>>> {epsilon, epsilon}
{epsilon}
>>> epsilon == epsilon
True
>>> str(epsilon)
'epsilon'
>>> epsilon is not None
True
```

PYTHON MODULE INDEX

INDEX

Symbols

C

E

I

M

N

S

T