metatable

Release 1.3.0

Reity LLC

Aug 04, 2022

CONTENTS

Extensible table data structure that supports the introduction of user-defined workflow combinators and the use of these combinators in concise workflow descriptions.

CHAPTER

ONE

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install metatable
```

The library can be imported in the usual ways:

```
import metatable
from metatable import *
```

1.1 Examples

This library makes it possible to work with tabular data that is represented as a list of lists (*i.e.*, each row is a list of column values and a table is a list of rows):

```
>>> from metatable import *
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> list(iter(t))
[['a', 0], ['b', 1], ['c', 2]]
```

All rows in a `metatable` instance can be updated in-place using a symbolic representation (implemented using the symbolism library) of the transformation that must be applied to each row. For example, the transformation `{1: column(0)}` indicates that the value in the column having index `1` (*i.e.*, the right-hand column) should be replaced with the value in the column having index `0` (*i.e.*, the left-hand column):

```
>>> t.update({1: column(0)})
[['a', 'a'], ['b', 'b'], ['c', 'c']]
```

It is also possible to perform an update that removes rows based on a condition. The condition in the example below is the symbolic expression `column(1) > symbol(0)` (constructed using the symbolism library):

```
>>> from symbolism import symbol
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> t.update_filter({0: column(1)}, column(1) > symbol(0))
[[1, 1], [2, 2]]
```

There is also support for working with a tabular data set in which there is a header row containing column names:

```
>>> t = metatable([['char', 'num'], ['a', 0], ['b', 1]], header=True)
>>> t.update({1: column(0)})
[['char', 'num'], ['a', 'a'], ['b', 'b']]
```

CHAPTER

TWO

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]
```

2.1 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
```

2.2 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/metatable/metatable.py -v
```

Style conventions are enforced using Pylint:

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

2.3 Contributions

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

2.4 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.

2.5 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/*
```

2.5.1 metatable module

Table data structure that supports the introduction of user-defined workflow combinators and the use of these combinators in concise workflow descriptions.

class metatable.metatable.**metatable**(*iterable*, *name=None*, *header=False*)

Bases: object

Class for the extensible metatable data structure.

Parameters

- **iterable** (Iterable) – Iterable of rows corresponding to the data in this instance.
- **name** (Optional[str]) – Instance name.
- **header** (Optional[bool]) – Header row consisting of column names.

```
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> list(iter(t))
[['a', 0], ['b', 1], ['c', 2]]
```

All rows in an instance can be updated in-place using a symbolic representation of the transformation that must be applied to each row.

```
>>> t = metatable([['char', 'num'], ['a', 0], ['b', 1]], header=True)
>>> t.update({1: column(0)})
[['char', 'num'], ['a', 'a'], ['b', 'b']]
```

Find more examples under the entries for the *update* and *update_filter* methods.

__iter__()
Return this instance as an iterable.

```
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> list(iter(t))
[['a', 0], ['b', 1], ['c', 2]]
```

Return type Iterable

map(*function*, *iterable*, *progress*)
Internal method for mapping over the data in the table. This method can be redefined in derived classes to change how rows are processed (*e.g.*, to introduce multiprocessing).

Parameters

- **function** (Callable) – Function to apply to every item in the iterable.
- **iterable** (Iterable) – Iterable of items to which the function should be applied (this should normally be the instance itself).
- **progress** (Callable) – Function that returns its iterable input and reports progress.

```
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> list(t.map(lambda row: [[row[1], row[0]]], t, lambda _: _))
[[0, 'a'], [1, 'b'], [2, 'c']]
```

Return type Iterable

update_filter(*update*, *filter*, *header=None*, *strict=False*, *progress=lambda *a, **ka: ...*)
Perform update-then-filter operations across the entire table, based on symbolic expressions for the update and filter task(s). The result of the operation is returned.

Parameters

- **update** (symbol) – Symbolic expression that represents an update operation (to be applied to every row).
- **filter** (symbol) – Symbolic expression that represents a filter predicate (to be tested for every row).
- **header** (Optional[list]) – Header row for the overall result of this method.
- **strict** (Optional[bool]) – Drop columns that do not explicitly appear in the update expression.

- **progress** (Optional[Callable]) – Function that returns its iterable input and reports progress.

```
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> t.update_filter({0: column(1)}, column(1) > symbolism.symbol(0))
[[1, 1], [2, 2]]
```

This instance is modified in-place, so iterating over it again yields the updated version.

```
>>> list(t)
[[1, 1], [2, 2]]
```

This method can be used in combination with the *row* class to introduce the row index into a column during the update.

```
>>> t = metatable([['a'], ['b'], ['c']])
>>> t.update_filter({3: row}, column(3) < 2)
[['a', None, None, 0], ['b', None, None, 1]]
>>> list(t)
[['a', None, None, 0], ['b', None, None, 1]]
```

Return type list

update(*update*, *header=None*, *strict=False*, *progress=lambda *a, **ka: ...*)

Update operation across the entire table, based on a symbolic expression for the update task(s).

Parameters

- **update** (symbol) – Symbolic expression that represents an update operation (to be applied to every row).
- **header** (Optional[list]) – Header row for the overall result of this method.
- **strict** (Optional[bool]) – Drop columns that do not explicitly appear in the update expression.
- **progress** (Optional[Callable]) – Function that returns its iterable input and reports progress.

```
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> t.update({0: column(1)}) # Replace first-column value with second-column␣
↪value.
[[0, 0], [1, 1], [2, 2]]
>>> list(t)
[[0, 0], [1, 1], [2, 2]]
```

If a header row is present (and should be preserved when performing the update), this can be indicated using the header argument.

```
>>> t = metatable([['char', 'num'], ['a', 0], ['b', 1]], header=True)
>>> t.update({1: column(0)})
[['char', 'num'], ['a', 'a'], ['b', 'b']]
```

This method can be used in combination with the *drop* class in order to indicate that a column should be dropped during the update.

```
>>> t.update({0: drop})
[['num'], ['a'], ['b']]
>>> t.update({0: drop})
[[], [], []]
>>> t = metatable([['a', 0, True], ['b', 1, True], ['c', 2, False]])
>>> t.update([column(1), column(0), drop])
[[0, 'a'], [1, 'b'], [2, 'c']]
```

If the `strict` argument is assigned the value `True`, then columns that do not explicitly appear in the update task specification are dropped.

```
>>> t = metatable([['c', 'n', 'b'], ['a', 0, True], ['b', 1, True]],␣
↪header=True)
>>> t.update([column(1), column(0)], strict=True, header=['n', 'c'])
[['n', 'c'], [0, 'a'], [1, 'b']]
>>> t.update([column(1)], strict=True)
[['n'], ['a'], ['b']]
>>> t.update([column(0)], strict=True, header=['c'])
[['c'], ['a'], ['b']]
>>> t.update({2: 'x'})
[['c', None, None], ['a', None, 'x'], ['b', None, 'x']]
```

Other common operations (such as the functions pre-defined within the symbolism library) can be used to introduce a new computed column (in which the entry for that column in every row is computed using zero or more of the values from that row found in the existing columns).

```
>>> t = metatable([['a', 0], ['b'], ['c', 2]])
>>> t.update({2: symbolism.is_(column(1), None)})
[['a', 0, False], ['b', None, True], ['c', 2, False]]
```

Return type list

class metatable.metatable.**row**

Bases: object

Symbolic representation of a row index (for use with methods such as *metatable.update*).

```
>>> t = metatable([['a'], ['b'], ['c']])
>>> t.update_filter({3: row}, column(3) < 2)
[['a', None, None, 0], ['b', None, None, 1]]
```

class metatable.metatable.**drop**

Bases: object

Symbolic representation of a column drop operation (for use with methods such as *metatable.update*).

```
>>> t = metatable([['char', 'num'], ['a', 0], ['b', 1]], header=True)
>>> t.update({1: column(0)})
[['char', 'num'], ['a', 'a'], ['b', 'b']]
>>> t.update({0: drop})
[['num'], ['a'], ['b']]
```

class metatable.metatable.**column**(*instance*)

Bases: symbolism.symbolism.symbol

Symbolic representation of a column specifier, such as a numerical index or an attribute name (for use with methods such as *metatable.update*).

```
>>> t = metatable([['a', 0], ['b', 1], ['c', 2]])
>>> t.update_filter({0: column(1)}, column(1) > symbolism.symbol(0))
[[1, 1], [2, 2]]
```

PYTHON MODULE INDEX

INDEX

Symbols

C

D

M

R

U