marbles Documentation

Release 0.12.1

Jane Adams and Leif Walsh

Aug 03, 2020

User Guide

Marbles is a `unittest` extension that allows test authors to write richer tests that expose more information on test failure. You can use marbles anywhere you use unittest to get better failure messages that help you debug failing tests faster.

```
$ python -m marbles hello_world.py
F
======================================================================
FAIL: test_hello_world (hello_world.HelloWorldTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 'Hello, world!' != 'Bonjour, le monde!'
- Hello, world!
+ Bonjour, le monde!


Source (hello_world.py):
     11
 >   12 self.assertEqual(english, french, note=note)
     13
Locals:
        english=Hello, world!
        french=Bonjour, le monde!
Note:
        Welcome to your first marbles test!


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

CHAPTER 1

Philosophy

The main idea behind marbles is that test failures are documentation. We wanted failure messages that put failures in context and clearly communicate the author's intent with each test. We wanted failure messages that give test consumers enough information that they don't have to dig through test code to understand what's going on. And, we wanted test authors to start thinking about test failures as documentation *for test consumers*, whether that's them in a few months or someone completely new to to the test suite, to help everyone get in the habit of writing better, clearer tests.

CHAPTER 2

Contents

2.1 User Guide

2.1.1 Installation

marbles can be installed from PyPI.

Note: We don't care what tool you use (we'll leave that to the Python Packaging Guide), but we do highly recommend that you only install marbles into a virtual environment.

marbles contains two namespace packages: *marbles.core* and *marbles.mixins*. You can install them together or separately. For example, if you don't need any of the custom assertions in *marbles.mixins*, you can install and depend on *marbles.core* by itself.

pip

To install marbles with pip

```
pip install marbles
# -or-
pip install marbles.core
pip install marbles.mixins
```

conda

To install marbles with conda

```
conda install -c conda-forge marbles
```

Source

If you need a copy of the source, you can clone the GitHub repository or download the *tarball*.

GitHub

```
git clone https://github.com/twosigma/marbles.git
cd marbles
pip install .
```

Tarball

```
curl -OL https://github.com/twosigma/marbles/tarball/master
tar xvzf /path/to/archive.tar.gz
cd marbles
pip install .
```

2.1.2 Quickstart

Once you have marbles installed, you can run your existing `unittest` tests with marbles and get marbles failure messages, without changing any code. This lets you switch between marbles and unittest failure messages. For instance, you might want to see marbles failure messages during interactive development and debugging, but see unittest failure messages in CI. To do this

```
$ python -m marbles docs/examples/getting_started.py  # development
$ python -m unittest docs/examples/getting_started.py  # CI
```

For example, let's say we have the following `unittest` test case

```
import requests
import unittest


class ResponseTestCase(unittest.TestCase):

    def test_create_resource(self):
        endpoint = 'http://example.com/api/v1/resource'
        data = {'id': 1, 'name': 'Little Bobby Tables'}

        res = requests.put(endpoint, data=data)
        self.assertEqual(
            res.status_code,
            201
        )


if __name__ == '__main__':
    unittest.main()
```

If we run this test case with unittest, we'll see a normal unittest failure message

```
$ python -m unittest docs/examples/getting_started.py
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/docs/examples/getting_started.py", line 42, in test_create_resource
    201
AssertionError: 409 != 201

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

But, if we run this same test case with marbles instead, we get a marbles failure message, without changing any test code

```
$ python -m marbles docs/examples/getting_started.py
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201
     43 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Reading marbles failure messages

In this section we'll go over the different parts of a marbles failure message.

Source

Python tracebacks only show the last line of the statement that failed, which can be confusing if the statement that failed spans multiple lines. In a marbles failure message, the "Source" section contains the full assertion statement that failed

```
$ python -m marbles docs/examples/getting_started.py
F
======================================================================
```

(continues on next page)

(continued from previous page)

```
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201
     43 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

This doesn't look like a traceback, it looks like code, perhaps even code that I wrote. And, it's a lot easier to recognize than when it's inside a traceback

```
$ python -m unittest docs/examples/getting_started.py
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/docs/examples/getting_started.py", line 42, in test_create_resource
    201
AssertionError: 409 != 201

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Traceback

Speaking of the traceback, where is it? Marbles failure messages contain all of the information you would normally find in a traceback (and more), so we can hide the traceback to make failure messages easier to read without losing any information. If you still want to see the traceback, you can run your tests in verbose mode

```
$ python -m marbles docs/examples/getting_started.py --verbose
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/docs/examples/getting_started.py", line 42, in test_create_resource
    201
  File "/path/to/docs/examples/getting_started.py", line 520, in wrapper
    return attr(*args, msg=annotation, **kwargs)
```

(continues on next page)

(continued from previous page)

```
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201
     43 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Locals

The "Locals" section of a marbles failure messages contains any variables that are in scope at the time the test failed

```
$ python -m marbles docs/examples/getting_started.py
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201
     43 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

This helps you recover the "state of the world" at the time the test failed and see what the actual and expected runtime values were, without having to put debugging statements in your test code (or even reading or changing your test code at all).

See *Curating Locals* to see how to control which local variables show up in this section.

Notes

Assertions on *marbles.core.TestCase*s accept an optional annotation provided by the author. This annotation, if provided, will be included in the failure message.

Warning: You can provide annotations to assertions on vanilla `unittest.TestCase`s *only if* you run them with marbles. If you try to run annotated `unittest.TestCase` tests with unittest they will break.

Let's add an annotation to our example and see what it looks like

```
--- /home/docs/checkouts/readthedocs.org/user_builds/marbles/checkouts/stable/docs/
↪examples/getting_started.py
+++ /home/docs/checkouts/readthedocs.org/user_builds/marbles/checkouts/stable/docs/
↪examples/getting_started.py.annotated
@@ -39,7 +39,8 @@
         res = requests.put(endpoint, data=data)
         self.assertEqual(
             res.status_code,
-            201
+            201,
+            note=res.reason
         )
```

Now when we run our test, we see an additional section

```
$ python -m marbles docs/examples/getting_started.py
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201,
     43     note=res.reason
     44 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}
Note:
        The request could not be completed due to a conflict with the current
        state of the target resource.


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

We go into the *Note* annotation in more detail in *How to win at marbles*.

Writing marbles tests

This section will cover how to write *marbles.core.TestCase*s and how to port `unittest.TestCase`s to marbles.

To write marbles tests, all you need to do is inherit from *marbles.core.TestCase* wherever you would normally inherit from unittest.TestCase and write your test methods exactly as you would normally. Nothing else about your test cases or test methods needs to change (unless you want to add *annotations*).

For example, let's take our example test case from earlier

```
import requests
import unittest


class ResponseTestCase(unittest.TestCase):

    def test_create_resource(self):
        endpoint = 'http://example.com/api/v1/resource'
        data = {'id': 1, 'name': 'Little Bobby Tables'}

        res = requests.put(endpoint, data=data)
        self.assertEqual(
            res.status_code,
            201
        )


if __name__ == '__main__':
    unittest.main()
```

To turn this into a marbles test case

```
--- /home/docs/checkouts/readthedocs.org/user_builds/marbles/checkouts/stable/docs/
↪examples/getting_started.py.original
+++ /home/docs/checkouts/readthedocs.org/user_builds/marbles/checkouts/stable/docs/
↪examples/getting_started.py
@@ -1,5 +1,5 @@
 import requests
-import unittest
+import marbles.core


 # We stub out a put request for the purposes of creating an example
@@ -30,7 +30,7 @@
                                  'state of the target resource.'))


-class ResponseTestCase(unittest.TestCase):
+class ResponseTestCase(marbles.core.TestCase):

     def test_create_resource(self):
         endpoint = 'http://example.com/api/v1/resource'
@@ -44,4 +44,4 @@


 if __name__ == '__main__':
-    unittest.main()
+    marbles.core.main()
```

Now, we get the following output, whether we run our tests with unittest or with marbles

```
$ python -m marbles docs/examples/getting_started.py
```

(continues on next page)

(continued from previous page)

```
F
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201
     43 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Note: If you run your tests with `-m unittest`, the failure message will always include the traceback, even if you don't run your tests in verbose mode. To hide the traceback, you need to run your tests with `-m marbles`.

Porting unittest tests

To replace all of your unittest test cases with marbles test cases

```
find /path/to/files -type f -exec sed -i 's/unittest/marbles.core/g' {} \;
```

Warning: This may not be safe. For example, it will replace `unittest.mock` with `marbles.core.mock`, which doesn't exist. If you use this command, be sure to review the diff.

Don't forget to *declare marbles as a dependency*.

Running tests

You can run marbles tests exactly like you run vanilla unit tests

```
python -m marbles /path/to/marbles_tests.py
# -or-
python -m unittest /path/to/marbles_tests.py
```

As we saw *above*, you can also run vanilla unit tests with marbles and get marbles failure messages, without changing the base class of you test cases

```
python -m marbles /path/to/unittest_tests.py
```

Marbles also creates a setuptools command so if you are used to running `python setup.py test`, you can now run:

```
python setup.py marbles
```

You can go one step further and alias the command test to run marbles by adding the following to `setup.cfg`:

```
[aliases]
test = marbles
```

Declaring marbles as a dependency

To ensure that marbles is available wherever you need to run your package's unit tests you need to declare *marbles.core* as a test dependency in your `setup.py` script

```
setup(
    ...
    tests_require=[
        'marbles.core'
    ],
    ...
)
```

2.1.3 How to win at marbles

Out of the box, marbles gives you better failure messages, but it also gives you control over what information your failure messages contain. In this section, we'll cover how to write your tests to get the most out of your failure messages.

Curating Locals

Local variables defined within the test are included in the "Locals" section of the failure message. This helps the test consumer to reconstruct the "state of the world" at the time the test failed. Marbles lets you control which locals will be included in this section.

Excluding Locals

Not all local variables will be relevant to the test consumer, and exposing too many locals could be as confusing as exposing too few. If you need to define variables in your test but don't want them to show up in the output, you can exclude them from the "Locals" section by making them internal or name-mangled (prepending them with one or two underscores).

Note: The local variables `self`, `msg`, and `note` are automatically excluded from the "Locals" section.

```
import marbles.core


class IntermediateStateTestCase(marbles.core.TestCase):

```

(continues on next page)

(continued from previous page)

```
    def test_foo(self):
        start_str = 'foo'

        # Capitalize our string one character at a time
        _intermediate_state_1 = start_str.replace('f', 'F')
        __intermediate_state_2 = _intermediate_state_1.replace('o', 'O')

        stop_str = __intermediate_state_2.lower()

        self.assertNotEqual(start_str, stop_str)


if __name__ == '__main__':
    marbles.core.main()
```

This will produce the following output. Notice that the variables `_intermediate_state_1` and `__intermediate_state_2` don't appear in "Locals".

```
$ python -m marbles docs/examples/exclude_locals.py
F
======================================================================
FAIL: test_string_case (docs.examples.exclude_locals.IntermediateStateTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 'foo' == 'foo'

Source (/path/to/docs/examples/exclude_locals.py):
     14
 >   15 self.assertNotEqual(start_str, stop_str)
     16
Locals:
        start_str=foo
        stop_str=foo


----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Locals-only Locals

Conversely, there may be some local state that you want to expose to the test consumer that your test doesn't actually need to use. We recommend creating local variables for these anyway.

Note: Python linters like flake8 will complain about variables that are assigned but never used, but most linters provide ways of ignoring specific lines.

In the example below, even though we don't need to define `file_name`, it's useful for the test consumer to know *what* file has a size we don't expect. We sidestep the flake8 warning with the comment `# noqa: F841` (F841 is the code for "local variable is assigned but never used")

```
import os
import marbles.core
```

(continues on next page)

(continued from previous page)

```
class FileTestCase(marbles.core.TestCase):

    def test_file_size(self):
        file_name = __file__  # noqa: F841

        self.assertEqual(os.path.getsize(__file__), 0)


if __name__ == '__main__':
    marbles.core.main()
```

When we run this test, we'll see `file_name` in locals

```
$ python -m marbles docs/examples/extra_locals.py
F
======================================================================
FAIL: test_file_size (docs.examples.extra_locals.FileTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 288 != 0

Source (/path/to/docs/examples/extra_locals.py):
      9
 >   10 self.assertEqual(os.path.getsize(__file__), 0)
     11
Locals:
        file_name=/path/to/docs/examples/extra_locals.py


----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Notes

Note annotations are intended to help the test author communicate any context or background information they have about the test. For example, what's the context of the edge case this particular test method is exercising? The note annotation is a good place to put information that doesn't fit into the test method name or into the assertion statement.

Note annotations are accepted in addition to the `msg` argument accepted by all assertions. If specified, the `msg` is used as the error message on failure, otherwise it will be the standard message provided by the assertion.

The `msg` should be used to explain exactly what the assertion failure was, e.g., `x was not greater than y`, while the `note` can provide more information about why it's important that `x` be greater than `y`, why we expect `x` to be greater than `y`, what needs to happen if `x` isn't greater than `y`, etc. The `note` doesn't (and in fact shouldn't) explain what the assertion failure *is* because the `msg` already does that well.

For example, in the failure message below, the standard message (`409 != 201`) and the note annotation complement each other. The standard message states that the status code we got (`409`) doesn't equal the status code we expected (`201`), while the note provides context about the status code `409`.

```
$ python -m marbles docs/examples/getting_started.py
F
```

(continues on next page)

(continued from previous page)

```
======================================================================
FAIL: test_create_resource (docs.examples.getting_started.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     39 res = requests.put(endpoint, data=data)
 >   40 self.assertEqual(
     41     res.status_code,
     42     201,
     43     note=res.reason
     44 )
Locals:
        endpoint=http://example.com/api/v1/resource
        data={'name': 'Little Bobby Tables', 'id': 1}
Note:
        The request could not be completed due to a conflict with the current
        state of the target resource.


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Note: We recommend that you bind note annotations to a variable named `note`, or pass them to assertions directly, so that they're not repeated in the "Locals" section. Otherwise, you'll need to manually exclude them from the "Locals" section. See *Excluding Locals* for how to do this.

Dynamic Note

Note annotations can contain format string fields that will be expanded with local variables if/when the test fails. They're similar to f-strings in that you don't have to call `str.format()` yourself, but they differ in that they're only expanded if and when your test fails.

Let's add a format string field to our note annotation

```
--- /home/docs/checkouts/readthedocs.org/user_builds/marbles/checkouts/stable/docs/
↪examples/getting_started.py.annotated
+++ /home/docs/checkouts/readthedocs.org/user_builds/marbles/checkouts/stable/docs/
↪examples/getting_started.py.dynamic
@@ -36,11 +36,13 @@
         endpoint = 'http://example.com/api/v1/resource'
         data = {'id': 1, 'name': 'Little Bobby Tables'}

+        note = '{res.reason} at {endpoint}'
+
         res = requests.put(endpoint, data=data)
         self.assertEqual(
             res.status_code,
             201,
-            note=res.reason
+            note=note
```

(continues on next page)

(continued from previous page)

```
        )
```

When this test fails, `endpoint` will be formatted into our note string

```
$ python -m marbles docs/examples/getting_started.py
F
======================================================================
FAIL: test_create_resource (docs.examples.ResponseTestCase)
----------------------------------------------------------------------
marbles.core.marbles.ContextualAssertionError: 409 != 201

Source (/path/to/docs/examples/getting_started.py):
     42 res = requests.put(endpoint, data=data)
 >   43 self.assertEqual(
     44     res.status_code,
     45     201,
     46     note=note
     47 )
Locals:
        data={'name': 'Little Bobby Tables', 'id': 1}
        endpoint=http://example.com/api/v1/resource
Note:
        The request could not be completed due to a conflict with the current
        state of the target resource at http://example.com/api/v1/resource.


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

Required Note

Depending on the complexity of what you're testing, you may want to require that `note` be provided for all assertions. If you want to require notes, your test cases should inherit from *`marbles.core.AnnotatedTestCase`* instead of from *`marbles.core.TestCase`*. The only difference is that, while `note` is optional for assertions on *`TestCase`*s, it's required for all assertions on *`AnnotatedTestCase`*s.

If you don't provide notes to an assertion on an *`AnnotatedTestCase`* you'll see an error

```
E
======================================================================
ERROR: test_for_edge_case (docs.examples.required_note.ComplexTestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/path/to/docs/examples/required_note.py", line 7, in test_for_edge_case
    self.assertTrue(False)
  File "/path/to/marbles/core/marbles/core/marbles.py", line 535, in wrapper
    list(args) + list(rem_args), kwargs) as annotation:
  File "/path/to/marbles/core/marbles/core/marbles.py", line 407, in __enter__
    self._validate_annotation(annotation)
  File "/path/to/marbles/core/marbles/core/marbles.py", line 397, in _validate_
↪annotation
```

(continues on next page)

(continued from previous page)

```
    raise AnnotationError(error)
marbles.core.marbles.AnnotationError: Annotation missing required fields: {'note'}

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Custom assertions

`unittest.TestCase`s expose several assert methods for use in unit tests. These assert methods range from very straightforward assertions like `assertTrue()` to the more detailed assertions like `assertWarnsRegex()`. These assertions allow the test author to clearly and concisely assert their expectations.

marbles.mixins

The *`marbles.mixins`* package adds even more assertion methods that you can use, including assertions about betweenness, monotonicity, files, etc. For the most part, *`marbles.mixins`* assertions trivially wrap `unittest` assertions. The reason to use specific assertions is that the semantically-richer method names can give the test consumer valuable information about the predicate being tested, the types of the objects being tested, etc. For example, `assertRegex()` doesn't tell you anything about the string being tested, *`assertFileNameRegex()`* immediately tells you that the string being tested is a file name.

For example, let's say we've written a function that sorts a list of numbers (which we shouldn't have done because `sorted()` is included in the standard library). We can write a concise unit test for this function using mixin assertions about monotonicity

```
import marbles.core
import marbles.mixins


def my_sort(i, reverse=False):
    '''Sort the elements in ``i``.'''
    # Purposefully sort in the wrong order so our unit test will fail
    return sorted(i, reverse=~reverse)


class SortTestCase(marbles.core.TestCase, marbles.mixins.MonotonicMixins):

    def test_sort(self):
        i = [1, 3, 4, 5, 2, 0, 8]

        self.assertMonotonicIncreasing(my_sort(i))
        self.assertMonotonicDecreasing(my_sort(i, reverse=True))


if __name__ == '__main__':
    marbles.core.main()
```

These custom assertions are provided via mixin classes so that they can use other assertions as building blocks. Using mixins, instead of straight inheritance, means that you can compose multiple mixins to create a test case with all the assertions that you need.

Warning: *marbles.mixins* can be mixed into a `unittest.TestCase`, a *marbles.core.TestCase*, a *marbles.core.AnnotatedTestCase*, or any other class that implements a `unittest.TestCase` interface. To enforce this, mixins define abstract methods. This means that, when mixing them into your test case, they must come *after* the class(es) that implement those methods instead of appearing first in the inheritance list like normal mixins.

Writing your own mixins

You can write your own assertions and mix them in to your test cases, too. We recommend reading the *marbles.mixins* source code to see how to do this. Here is the *UniqueMixins* source as an example:

```
class UniqueMixins(abc.ABC):
    '''Built-in assertions about uniqueness.

    These assertions can handle containers that contain unhashable
    elements.
    '''

    @abc.abstractmethod
    def fail(self, msg):
        pass  # pragma: no cover

    @abc.abstractmethod
    def assertIsInstance(self, obj, cls, msg):
        pass  # pragma: no cover

    @abc.abstractmethod
    def _formatMessage(self, msg, standardMsg):
        pass  # pragma: no cover

    def assertUnique(self, container, msg=None):
        '''Fail if elements in ``container`` are not unique.

        Parameters
        ----------
        container : iterable
        msg : str
            If not provided, the :mod:`marbles.mixins` or
            :mod:`unittest` standard message will be used.

        Raises
        ------
        TypeError
            If ``container`` is not iterable.
        '''
        if not isinstance(container, collections.abc.Iterable):
            raise TypeError('First argument is not iterable')

        standardMsg = 'Elements in %s are not unique' % (container,)

        # We iterate over each element in the container instead of
        # comparing len(container) == len(set(container)) to allow
        # for containers that contain unhashable types
        for idx, elem in enumerate(container):
            # If elem appears at an earlier or later index position
```

(continues on next page)

(continued from previous page)

```
            # the elements are not unique
            if elem in container[:idx] or elem in container[idx+1:]:
                self.fail(self._formatMessage(msg, standardMsg))

    def assertNotUnique(self, container, msg=None):
        '''Fail if elements in ``container`` are unique.

        Parameters
        ----------
        container : iterable
        msg : str
            If not provided, the :mod:`marbles.mixins` or
            :mod:`unittest` standard message will be used.

        Raises
        ------
        TypeError
            If ``container`` is not iterable.
        '''
        if not isinstance(container, collections.abc.Iterable):
            raise TypeError('First argument is not iterable')

        standardMsg = 'Elements in %s are unique' % (container,)

        # We iterate over each element in the container instead of
        # comparing len(container) == len(set(container)) to allow
        # for containers that contain unhashable types
        for idx, elem in enumerate(container):
            # If elem appears at an earlier or later index position
            # the elements are not unique
            if elem in container[:idx] or elem in container[idx+1:]:
                return  # succeed fast

        self.fail(self._formatMessage(msg, standardMsg))
```

If you write assertions that you think would be useful for others, we would love to see a pull request from you!

Logging

You can configure *marbles.core* to log information about every assertion made during a test run as a JSON blob. This includes the test method name, the assertion, the result of the assertion, the arguments passed to the assertion, runtime variables, etc.

These logs can be transferred to another system for later analysis and reporting. For example, you could run **logstash** after a test run to upload your logs to Elasticsearch, and then use Kibana to analyze them, maybe creating dashboards that show how many assertion failures you get over time, grouped by whether or not assertions are annotated.

See *marbles.core.log* for information on configuring the logger.

2.1.4 Comparisons

How does marbles compare to other Python testing tools?

Marbles extends Python's built-in unittest library, so some of what distinguishes marbles from other testing tools isn't

about marbles as much as it's about unittest. That being said, marbles extends unittest —as opposed to another Python testing tool—for a reason.

In this section, we'll call out differentiating features of marbles specifically, as well as differentiating features of unittest that make unittest the right foundation for marbles.

For now, we focus on pytest, which is widely used and whose failure messages have the most in common with marbles failure messages.

pytest

As far as failure messages go, marbles has the most in common with pytest. However, because marbles is built on top of unittest, writing marbles tests is pretty different than writing pytest tests.

Marbles is all about the test consumer, while pytest is all about the test author. Pytest tries to make you more efficient while writing tests and marbles tries to make you more efficient while reading, reasoning about, and responding to test failures. You could say (and we sometimes do) that pytest is write optimized and marbles is read optimized.

If you're familiar with pytest, you'll probably find that writing marbles tests is more typing than you're used to, but we hope, no matter which tool you know well, responding to test failures will be faster and easier with marbles. Marbles achieves this in a couple of ways:

1. Marbles failures expose more information than pytest failures
2. Giving test authors the ability to curate what appears in the failure message encourages them to design their tests with the test consumer in mind
3. Unittest tests are more explicit than pytest tests, meaning it's easier to determine and reason about what tests are doing

Similarities

Assertion Source

Both marbles and pytest present the *source code* of the whole assertion statement that failed, which is more useful than a typical Python traceback.

We believe both tools provide an equivalent benefit here.

Local Variables

Both marbles and pytest present some of the local variables present at the time an assertion in your test failed.

Pytest exposes only the variables that are involved in the assertion (and shows each sub-expression involved in the assertion). Marbles exposes any public variables that are in scope at the time the assertion failed, whether or not they are directly involved in the assertion.

Advantages of marbles

Note annotations

Marbles allows test authors to *annotate assertion statements* with additional information about the test and the author's intent that will help the test consumer put the failed assertion in context.

Explicit assertion names

Pytest relies on the bare `assert` keyword, and encourages use of it directly. This puts the burden on the test consumer to derive the author's intent. As a consumer, you need to parse the logic of the assertion condition and read the rest of the test to understand what's going on.

Instead of the `assert` keyword, marbles tests use the assertion methods provided by unittest. Unittest's assertions methods have semantically-rich names that help convey the author's intent to the test consumer in almost-plain English. We believe that, because the assertion statement that failed will be exposed in the failure message, is is worthwhile to write assertion statements that are as descriptive and easy to understand as possible.

Furthermore, relying on the `assert` keyword makes it difficult to ensure that similar expectations are being asserted in comparable ways. Having a standard set of specific assertion methods helps ensure that similar assertions are made in the same way. For example, every test that uses the `assertRegex()` assertion will test for a regex match in the same way.

The *marbles.mixins* package provides even more and semantically-richer assertion methods on top of the standard set of unittest assertions. You are also free to write your own assertion methods. The *marbles.mixins* provide a good template for building out a set of custom assertions that may be unique to your business or use case.

Local variable control

Both marbles and pytest expose some of the test's local "state". Pytest failure messages include any variables included in the assertion statement, and will expand any complex expressions that are present in the assertion. Any variables that are not used in the assertion will not be displayed, meaning we don't see any variables that may have been defined leading up to the assertion.

For example, consider the following pytest code:

```
assert a * b < c * d
```

If this assertion fails, pytest will show you the values of the expressions `a * b` and `c * d`, as well as the individual values of each variable `a`, `b`, `c`, and `d`.

Marbles will display any public local variables defined within the test at the time it failed, regardless of whether or not they were used in the failing assertion.

Consider the same example as above written in marbles:

```
self.assertLess(a * b, c * d)
```

If this assertion fails, marbles will show you the values of `a`, `b`, `c`, and `d`, but not the values of the expressions `a * b` or `c * d`. If it's valuable for the test consumer to also see the values of these expressions, we can achieve that by assigning them to variables:

```
lhs = a * b
rhs = c * d
self.assertLess(lhs, rhs)
```

If we want to exclude any local variables from the failure message, all we need to do is give them "internal" or "private" names, i.e., prefix the variable names with an underscore.

This gives the test author natural—and pretty neutral—control over what local variables will be displayed in the failure message. In pytest, in order for locals to appear in the failure message they need to be used in the assertion. In marbles, they need only be public.

Pure extension of unittest

While pytest gives you lots of power to be clever with fixtures when writing tests, this is often at the expense of being able to easily understand what any given test is doing when you're trying to debug a failure.

It can be hard to piece together where fixtures come from: they might not even be in the same file, or any that are imported. Even if you can find the fixtures, it's unclear exactly what the control flow is. This gets particularly complicated if the author used `conftest.py` anywhere in the project.

Marbles works with unmodified `unittest` tests. We find unittest tests have a clearer structure than pytest tests, especially those with complicated fixtures. With unittest, control flow is explicit, as long as you understand basic Python semantics. There's no magic, it's just inheritance.

We have found that, at scale, unittest's boilerplate is a benefit rather than a burden. It makes for tests that are more explicit about what they're doing, and it encourages logical grouping of tests, both of which reduce the test consumer's time to understand a failure.

2.2 Reference

2.2.1 API Reference

marbles.core

marbles.core	Extends `unittest` to provide more information to the test consumer on test failure.
marbles.core.TestCase([methodName])	An extension of `unittest.TestCase`.
marbles.core.AnnotatedTestCase([methodName])	An extension of *marbles.core.TestCase*.
marbles.core.ContextualAssertionError(*args)	Extends `AssertionError` to accept and display additona information beyond the static `msg` parameter provided by `unittest` assertions.
marbles.core.main(**kwargs)	
marbles.core.log	`marbles` can log information about each assertion called.
marbles.core.log.AssertionLogger()	The `AssertionLogger` logs json about each assertion.

Extends `unittest` to provide more information to the test consumer on test failure. This additional information includes local variables defined within the test at the time it failed, the full assertion statement that failed, and a free-form annotation provided by the test author. This additional information helps test consumers respond to test failures without digging into the test code themselves.

To get this additional information in your tests, you can inherit from *marbles.core.TestCase* anywhere you would normally inherit from `unittest.TestCase`. By simply inheriting from *marbles.core.TestCase* instead, your test failures will include the full assertion statement that failed and any local variables in scope at the time the test failed.

Assertions on a *marbles.core.TestCase* also accept an optional `note` string that will be exposed to the test consumer on test failure. This annotation can contain whatever the test author feels is important, but it is especially useful for communicating their intent and any relevant context about the test. This annotation can be a format string that will be expanded with local variables if/when the test fails.

You can also inherit from *marbles.core.AnnotatedTestCase*. The only difference is that, if you inherit from *marbles.core.AnnotatedTestCase*, you must provide `note` annotations to all assertions. Calling an

assertion without the `note` parameter on a *marbles.core.AnnotatedTestCase* will produce a `marbles.core.AnnotationError`.

class marbles.core.**TestCase**(*methodName='runTest'*)
An extension of `unittest.TestCase`.

Failure messages from *marbles.core.TestCase* tests contain more information than `unittest.TestCase` tests, including local variables defined within the test at the time the test failed and the full assertion statement that failed.

All assert statements, e.g., `unittest.TestCase.assertEqual()`, in addition to accepting the optional final string parameter `msg`, also accept a free-form `note` annotation provided by the test author. This annotation can contain whatever the test author feels is important, but it is especially useful for communicating their intent and any relevant context about the test that will help the test consumer understand and debug the failure. For example, this annotation can be used to provide context on a specific edge case that a test exercises, without sacrificing the `msg` or trying to embed that context into the test method name.

The `note` string, if provided, is formatted with `str.format()` given the local variables defined within the test itself.

Example:

```
import requests
import marbles.core


    def put(cls, endpoint, data=None):
        return Response(status_code=409,
                        reason=('The request could not be completed '
                                'due to a conflict with the current '
                                'state of the target resource.'))


class ResponseTestCase(marbles.core.TestCase):

    def test_create_resource(self):
        endpoint = 'http://example.com/api/v1/resource'
        data = {'id': 1, 'name': 'Little Bobby Tables'}

        res = requests.put(endpoint, data=data)
        self.assertEqual(
            res.status_code,
            201,
            note=res.reason
```

failureException
alias of *ContextualAssertionError*

class marbles.core.**AnnotatedTestCase**(*methodName='runTest'*)
An extension of *marbles.core.TestCase*.

An *AnnotatedTestCase* is only different from a *marbles.core.TestCase* in that it enforces that `note` is provided for every assertion. Calling an assertion without the `note` parameter on a *marbles.core.AnnotatedTestCase* will produce a `marbles.core.AnnotationError`.

For other details, see *marbles.core.TestCase*.

exception marbles.core.**ContextualAssertionError**(**args*)
Extends `AssertionError` to accept and display additiona information beyond the static `msg` parameter provided by `unittest` assertions.

This additional information includes the full assertion statement that failed and any local variables in scope at the time the test failed.

This additional information may also include a `note` string that can explain the intent of the test, provide any relevant context, and/or describe what to do if/when the assertion fails. This string is formatted with the local context where the assertion error is raised.

locals
A dict containing the locals defined within the test.

public_test_locals
A dict containing the public (a.k.a., not internal or name-mangled) locals defined within the test.

assert_stmt
Returns a string displaying the whole statement that failed, with a '>' indicator on the line starting the expression.

marbles.core.main

Main method for marbles.

With `unittest`, you can run `python -m unittest` to discover and run your tests.

With `marbles`, you can run `python -m marbles` to discover and run them, but with `marbles`-style assertion failure messages, including local variables and additional source code scope.

class marbles.core.main.**MarblesTestResult**(**args*, ***kwargs*)
A TestResult which omits the traceback.

Because marbles failure messages contain all of the information included in a normal unit test traceback, we can hide the traceback to make failure messages easier to read without losing any information. Here, we remove the traceback from the failure message, unless the user has asked for verbose output.

class marbles.core.main.**MarblesTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, ***, *tb_locals=False*)
A TestRunner which uses marbles-style assertion failure messages.

resultclass
alias of *MarblesTestResult*

marbles.core.log

`marbles` can log information about each assertion called.

If configured, the `marbles.core.log.logger` will log a json object for each assertion and its success or failure, as well as any other attributes of interest specified by the test author.

The captured information includes the assertion's args and kwargs, msg, note, local variables (for failed assertions, and also for passing assertions if verbose logging is turned on), and the result of the assertion.

Configuration is handled via the environment variables `MARBLES_LOGFILE`, `MARBLES_TEST_CASE_ATTRS`, `MARBLES_TEST_CASE_ATTRS_VERBOSE`, `MARBLES_LOG_VERBOSE`, or via the *AssertionLogger.configure()* method. Environment variables override those set with the *configure()* method, so if a `marbles` program configures these programmatically, they can always be overridden without changing the program.

Note that *AssertionLogger* should not be instantiated directly; instead, test authors should import and configure the `marbles.core.log.logger` as needed.

class marbles.core.log.**AssertionLogger**

The *AssertionLogger* logs json about each assertion.

This module exposes a single *AssertionLogger*, `marbles.core.log.logger`, that is used during a marbles test run. It can be configured with *configure()* before running the tests or via environment variables.

Example:

```
import marbles.core
from marbles.core import log

if __name__ == '__main__':
    log.logger.configure(logfile='/path/to/marbles.log',
                         attrs=['filename', 'date'])
    marbles.core.main()
```

Note: If you configure logging within an `if __name__ == '__main__'`, block (as opposed to via environment variables), you must run your tests with `python /path/to/tests.py`. If you run your tests with `python -m marbles`, the `if __name__ == '__main__'` block won't get executed and the logger won't get configured.

configure(***kwargs*)

Configure what assertion logging is done.

Settings configured with this method are overridden by environment variables.

Parameters

- **logfile** (*str or bytes or file object*) – If a string or bytes object, we write to that filename. If an open file object, we just write to it. If None, disable logging. If we open the file, we open it in `'w'` mode, so any contents will be overwritten.
- **attrs** (*list of str*) – Capture these attributes on the TestCase being run when logging an assertion. For example, if you are testing multiple resources, make sure the resource name is a member of your TestCase, and configure marbles logging with that name. These are only captured on failure.
- **verbose_attrs** (*list of str*) – Similar to attrs, but these attrs are captured even on success.
- **verbose** (*bool or list of str*) – Fields (within the set {msg, note, locals}) to capture even when the test is successful. By default, those three fields are only captured on failure.

marbles.mixins

marbles.mixins	This module provides custom `unittest`-style assertions.
marbles.mixins.BetweenMixins	Built-in assertions about betweenness.
marbles.mixins.MonotonicMixins	Built-in assertions about monotonicity.
marbles.mixins.UniqueMixins	Built-in assertions about uniqueness.
marbles.mixins.FileMixins	Built-in assertions for files.
marbles.mixins.CategoricalMixins	Built-in assertions for categorical data.
marbles.mixins.DateTimeMixins	Built-in assertions for `date` s, `datetime` s, and `time` s.

This module provides custom unittest-style assertions. For the most part, *marbles.mixins* assertions trivially wrap unittest assertions. For example, a call to *CategoricalMixins.assertCategoricalLevelIn()* will simply pass the provided arguments to assertIn().

Custom assertions are provided via mixins so that they can use other assertions as building blocks. Using mixins, instead of straight inheritance, means that users can compose multiple mixins to create a test case with all the assertions that they need.

Warning: *marbles.mixins* can be mixed into a unittest.TestCase, a *marbles.core.TestCase*, a *marbles.core.AnnotatedTestCase*, or any other class that implements a unittest.TestCase interface. To enforce this, mixins define abstract methods. This means that, when mixing them into your test case, they must come *after* the class(es) that implement those methods instead of appearing first in the inheritance list like normal mixins.

Example:

```
import unittest

from marbles.core import marbles
from marbles.mixins import mixins


class MyTestCase(unittest.TestCase, mixins.BetweenMixins):

    def test_me(self):
        self.assertBetween(5, lower=0, upper=10)


class MyMarblesTestCase(marbles.TestCase, mixins.BetweenMixins):

    def test_me(self):
        self.assertBetween(5, lower=0, upper=10)
```

class marbles.mixins.**BetweenMixins**

Built-in assertions about betweenness.

assertBetween(*obj*, *lower*, *upper*, *strict=True*, *msg=None*)

Fail if obj is not between lower and upper.

If strict=True (default), fail unless lower < obj < upper. If strict=False, fail unless lower <= obj <= upper.

This is equivalent to self.assertTrue(lower < obj < upper) or self.assertTrue(lower <= obj <= upper), but with a nicer default message.

Parameters

- **obj** –
- **lower** –
- **upper** –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

assertNotBetween(*obj*, *lower*, *upper*, *strict=True*, *msg=None*)

Fail if obj is between lower and upper.

If `strict=True` (default), fail if `lower <= obj <= upper`. If `strict=False`, fail if `lower < obj < upper`.

This is equivalent to `self.assertFalse(lower < obj < upper)` or `self.assertFalse(lower <= obj <= upper)`, but with a nicer default message.

Raises `ValueError` – If `lower` equals `upper` and `strict=True` is specified.

Parameters

- **obj** –
- **lower** –
- **upper** –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

class marbles.mixins.**CategoricalMixins**

Built-in assertions for categorical data.

This mixin includes some common categorical variables (e.g., weekdays, months, U.S. states, etc.) that test authors can use test resources against. For instance, if a dataset is supposed to contain data for all states in the U.S., test authors can test the state column in their dataset against the *US_STATES* attribute.

```
import unittest
from marbles.mixins import mixins


class MyTestCase(unittest.TestCase, mixins.CategoricalMixins):

    def test_that_all_states_are_present(self):
        df = ...
        self.assertCategoricalLevelsEqual(df['STATE'], self.US_STATES)
```

These categorical variables are provided as a convenience; test authors can and should manipulate these variables, or create their own, as needed. The idea is, for expectations that may apply across datasets, to ensure that the same expectation is being tested in the same way across different datasets.

WEEKDAYS

Type list

WEEKDAYS_ABBR

Weekdays abbreviated to three characters

Type list

MONTHS

Type list

MONTHS_ABBR

Months abbreviated to three characters

Type list

US_STATES

Type list

US_STATES_ABBR
U.S. state names abbreviated to two uppercase characters

Type list

US_TERRITORIES

Type list

US_TERRITORIES_ABBR
U.S. territory names abbreviated to two uppercase characters

Type list

CONTINENTS
7-continent model names

Type list

assertCategoricalLevelsEqual(*levels1*, *levels2*, *msg=None*)
Fail if `levels1` and `levels2` do not have the same domain.

Parameters

- **levels1** (*iterable*) –
- **levels2** (*iterable*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises `TypeError` – If either `levels1` or `levels2` is not iterable.

assertCategoricalLevelsNotEqual(*levels1*, *levels2*, *msg=None*)
Fail if `levels1` and `levels2` have the same domain.

Parameters

- **levels1** (*iterable*) –
- **levels2** (*iterable*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises `TypeError` – If either `levels1` or `levels2` is not iterable.

assertCategoricalLevelIn(*level*, *levels*, *msg=None*)
Fail if `level` is not in `levels`.

This is equivalent to `self.assertIn(level, levels)`.

Parameters

- **level** –
- **levels** (*iterable*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises `TypeError` – If `levels` is not iterable.

assertCategoricalLevelNotIn(*level*, *levels*, *msg=None*)
Fail if `level` is in `levels`.

This is equivalent to `self.assertNotIn(level, levels)`.

Parameters

- **level** –
- **levels** (*iterable*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If levels is not iterable.

class marbles.mixins.**DateTimeMixins**

Built-in assertions for date s, datetime s, and time s.

assertDateTimesBefore(*sequence*, *target*, *strict=True*, *msg=None*)

Fail if any elements in sequence are not before target.

If target is iterable, it must have the same length as sequence

If strict=True, fail unless all elements in sequence are strictly less than target. If strict=False, fail unless all elements in sequence are less than or equal to target.

Parameters

- **sequence** (*iterable*) –
- **target** (*datetime, date, iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises

- TypeError – If sequence is not iterable.
- ValueError – If target is iterable but does not have the same length as sequence.
- TypeError – If target is not a datetime or date object and is not iterable.

assertDateTimesAfter(*sequence*, *target*, *strict=True*, *msg=None*)

Fail if any elements in sequence are not after target.

If target is iterable, it must have the same length as sequence

If strict=True, fail unless all elements in sequence are strictly greater than target. If strict=False, fail unless all elements in sequence are greater than or equal to target.

Parameters

- **sequence** (*iterable*) –
- **target** (*datetime, date, iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises

- TypeError – If sequence is not iterable.
- ValueError – If target is iterable but does not have the same length as sequence.
- TypeError – If target is not a datetime or date object and is not iterable.

assertDateTimesPast(*sequence*, *strict=True*, *msg=None*)
Fail if any elements in `sequence` are not in the past.

If the max element is a datetime, "past" is defined as anything prior to `datetime.now()`; if the max element is a date, "past" is defined as anything prior to `date.today()`.

If `strict=True`, fail unless all elements in `sequence` are strictly less than `date.today()` (or `datetime.now()`). If `strict=False`, fail unless all elements in `sequence` are less than or equal to `date.today()` (or `datetime.now()`).

Parameters

- **sequence** (*iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises

- `TypeError` – If `sequence` is not iterable.
- `TypeError` – If max element in `sequence` is not a datetime or date object.

assertDateTimesFuture(*sequence*, *strict=True*, *msg=None*)
Fail if any elements in `sequence` are not in the future.

If the min element is a datetime, "future" is defined as anything after `datetime.now()`; if the min element is a date, "future" is defined as anything after `date.today()`.

If `strict=True`, fail unless all elements in `sequence` are strictly greater than `date.today()` (or `datetime.now()`). If `strict=False`, fail all elements in `sequence` are greater than or equal to `date.today()` (or `datetime.now()`).

Parameters

- **sequence** (*iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises

- `TypeError` – If `sequence` is not iterable.
- `TypeError` – If min element in `sequence` is not a datetime or date object.

assertDateTimesFrequencyEqual(*sequence*, *frequency*, *msg=None*)
Fail if any elements in `sequence` aren't separated by the expected `fequency`.

Parameters

- **sequence** (*iterable*) –
- **frequency** (*timedelta*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises

- `TypeError` – If `sequence` is not iterable.
- `TypeError` – If `frequency` is not a timedelta object.

assertDateTimesLagEqual(*sequence*, *lag*, *msg=None*)
Fail unless max element in `sequence` is separated from the present by `lag` as determined by the '==' operator.

If the max element is a datetime, "present" is defined as `datetime.now()`; if the max element is a date, "present" is defined as `date.today()`.

This is equivalent to `self.assertEqual(present - max(sequence), lag)`.

Parameters

- **sequence** (*iterable*) –
- **lag** (*timedelta*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises

- `TypeError` – If `sequence` is not iterable.
- `TypeError` – If `lag` is not a timedelta object.
- `TypeError` – If max element in `sequence` is not a datetime or date object.

assertDateTimesLagLess(*sequence*, *lag*, *msg=None*)
Fail if max element in `sequence` is separated from the present by `lag` or more as determined by the '<' operator.

If the max element is a datetime, "present" is defined as `datetime.now()`; if the max element is a date, "present" is defined as `date.today()`.

This is equivalent to `self.assertLess(present - max(sequence), lag)`.

Parameters

- **sequence** (*iterable*) –
- **lag** (*timedelta*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises

- `TypeError` – If `sequence` is not iterable.
- `TypeError` – If `lag` is not a timedelta object.
- `TypeError` – If max element in `sequence` is not a datetime or date object.

assertDateTimesLagLessEqual(*sequence*, *lag*, *msg=None*)
Fail if max element in `sequence` is separated from the present by more than `lag` as determined by the '<=' operator.

If the max element is a datetime, "present" is defined as `datetime.now()`; if the max element is a date, "present" is defined as `date.today()`.

This is equivalent to `self.assertLessEqual(present - max(sequence), lag)`.

Parameters

- **sequence** (*iterable*) –
- **lag** (*timedelta*) –

- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises

- TypeError – If sequence is not iterable.
- TypeError – If lag is not a timedelta object.
- TypeError – If max element in sequence is not a datetime or date object.

assertTimeZoneIsNone(*dt*, *msg=None*)
Fail if dt has a non-null tzinfo attribute.

Parameters

- **dt** (*datetime*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If dt is not a datetime object.

assertTimeZoneIsNotNone(*dt*, *msg=None*)
Fail unless dt has a non-null tzinfo attribute.

Parameters

- **dt** (*datetime*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If dt is not a datetime object.

assertTimeZoneEqual(*dt*, *tz*, *msg=None*)
Fail unless dt's tzinfo attribute equals tz as determined by the '==' operator.

Parameters

- **dt** (*datetime*) –
- **tz** (*timezone*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises

- TypeError – If dt is not a datetime object.
- TypeError – If tz is not a timezone object.

assertTimeZoneNotEqual(*dt*, *tz*, *msg=None*)
Fail if dt's tzinfo attribute equals tz as determined by the '!=' operator.

Parameters

- **dt** (*datetime*) –
- **tz** (*timezone*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises

- TypeError – If dt is not a datetime object.

- TypeError – If tz is not a timezone object.

class marbles.mixins.**FileMixins**

Built-in assertions for files.

With the exception of *assertFileExists()* and *assertFileNotExists()*, all custom file assertions take a filename argument which can accept a file name as a str or bytes object, or a file-like object. Accepting a file-like object is useful for testing files that are not present locally, e.g., files in HDFS.

```
import unittest

import hdfs3
from marbles.mixins import mixins


class MyFileTest(unittest.TestCase, mixins.FileMixins):

    def test_file_encoding(self):
        fname = 'myfile.csv'

        # You can pass fname directly to the assertion (if the
        # file exists locally)
        self.assertFileEncodingEqual(fname, 'utf-8')

        # Or open the file and pass a file descriptor to the
        # assertion
        with open(fname) as f:
            self.assertFileEncodingEqual(f, 'utf-8')

    def test_hdfs_file_encoding(self):
        hdfspath = '/path/to/myfile.csv'

        client = hdfs3.HDFileSystem(host='host', port='port')
        with client.open(hdfspath) as f:
            self.assertFileEncodingEqual(f, 'utf-8')
```

Note that not all file-like objects implement the expected interface. These custom file assertions expect the following methods and attributes:

- read()
- write()
- seek()
- tell()
- name
- encoding

assertFileExists(*filename*, *msg=None*)

Fail if filename does not exist as determined by os.path.isfile(filename).

Parameters

- **filename** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

assertFileNotExists(*filename*, *msg=None*)

Fail if filename exists as determined by ~os.path.isfile(filename).

Parameters

- **filename** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

assertFileNameEqual(*filename*, *name*, *msg=None*)
Fail if filename does not have the given name as determined by the == operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **name** (*str, byes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileNameNotEqual(*filename*, *name*, *msg=None*)
Fail if filename has the given name as determined by the != operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **name** (*str, byes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileNameRegex(*filename*, *expected_regex*, *msg=None*)
Fail unless filename matches expected_regex.

Parameters

- **filename** (*str, bytes, file-like*) –
- **expected_regex** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileNameNotRegex(*filename*, *expected_regex*, *msg=None*)
Fail if filename matches expected_regex.

Parameters

- **filename** (*str, bytes, file-like*) –
- **expected_regex** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileTypeEqual(*filename*, *extension*, *msg=None*)
Fail if filename does not have the given extension as determined by the == operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **extension** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileTypeNotEqual(*filename*, *extension*, *msg=None*)
Fail if filename has the given extension as determined by the != operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **extension** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileEncodingEqual(*filename*, *encoding*, *msg=None*)
Fail if filename is not encoded with the given encoding as determined by the '==' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **encoding** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileEncodingNotEqual(*filename*, *encoding*, *msg=None*)
Fail if filename is encoded with the given encoding as determined by the '!=' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **encoding** (*str, bytes*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeEqual(*filename*, *size*, *msg=None*)
Fail if filename does not have the given size as determined by the '==' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeNotEqual(*filename*, *size*, *msg=None*)
Fail if filename has the given size as determined by the '!=' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeAlmostEqual(*filename*, *size*, *places=None*, *msg=None*, *delta=None*)

Fail if filename does not have the given size as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or if their difference is greater than a given delta.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **places** (*int*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.
- **delta** (*int, float*) –

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeNotAlmostEqual(*filename*, *size*, *places=None*, *msg=None*, *delta=None*)

Fail unless filename does not have the given size as determined by their difference rounded to the given number ofdecimal places (default 7) and comparing to zero, or if their difference is greater than a given delta.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **places** (*int*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.
- **delta** (*int, float*) –

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeGreater(*filename*, *size*, *msg=None*)

Fail if filename's size is not greater than size as determined by the '>' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeGreaterEqual(*filename*, *size*, *msg=None*)

Fail if filename's size is not greater than or equal to size as determined by the '>=' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeLess(*filename*, *size*, *msg=None*)

Fail if filename's size is not less than size as determined by the '<' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

assertFileSizeLessEqual(*filename*, *size*, *msg=None*)

Fail if filename's size is not less than or equal to size as determined by the '<=' operator.

Parameters

- **filename** (*str, bytes, file-like*) –
- **size** (*int, float*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If filename is not a str or bytes object and is not file-like.

class marbles.mixins.**MonotonicMixins**

Built-in assertions about monotonicity.

assertMonotonicIncreasing(*sequence*, *strict=True*, *msg=None*)

Fail if sequence is not monotonically increasing.

If strict=True (default), fail unless each element in sequence is less than the following element as determined by the < operator. If strict=False, fail unless each element in sequence is less than or equal to the following element as determined by the <= operator.

```
assert all((i < j) for i, j in zip(sequence, sequence[1:]))
assert all((i <= j) for i, j in zip(sequence, sequence[1:]))
```

Parameters

- **sequence** (*iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If sequence is not iterable.

assertNotMonotonicIncreasing(*sequence*, *strict=True*, *msg=None*)
Fail if `sequence` is monotonically increasing.

If `strict=True` (default), fail if each element in `sequence` is less than the following element as determined by the < operator. If `strict=False`, fail if each element in `sequence` is less than or equal to the following element as determined by the <= operator.

```
assert not all((i < j) for i, j in zip(sequence, sequence[1:]))
assert not all((i <= j) for i, j in zip(sequence, sequence[1:]))
```

Parameters

- **sequence** (*iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises `TypeError` – If `sequence` is not iterable.

assertMonotonicDecreasing(*sequence*, *strict=True*, *msg=None*)
Fail if `sequence` is not monotonically decreasing.

If `strict=True` (default), fail unless each element in `sequence` is greater than the following element as determined by the > operator. If `strict=False`, fail unless each element in `sequence` is greater than or equal to the following element as determined by the >= operator.

```
assert all((i > j) for i, j in zip(sequence, sequence[1:]))
assert all((i >= j) for i, j in zip(sequence, sequence[1:]))
```

Parameters

- **sequence** (*iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises `TypeError` – If `sequence` is not iterable.

assertNotMonotonicDecreasing(*sequence*, *strict=True*, *msg=None*)
Fail if `sequence` is monotonically decreasing.

If `strict=True` (default), fail if each element in `sequence` is greater than the following element as determined by the > operator. If `strict=False`, fail if each element in `sequence` is greater than or equal to the following element as determined by the >= operator.

```
assert not all((i > j) for i, j in zip(sequence, sequence[1:]))
assert not all((i >= j) for i, j in zip(sequence, sequence[1:]))
```

Parameters

- **sequence** (*iterable*) –
- **strict** (*bool*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or `unittest` standard message will be used.

Raises TypeError – If sequence is not iterable.

class marbles.mixins.**UniqueMixins**
Built-in assertions about uniqueness.

These assertions can handle containers that contain unhashable elements.

assertUnique(*container*, *msg=None*)
Fail if elements in container are not unique.

Parameters

- **container** (*iterable*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If container is not iterable.

assertNotUnique(*container*, *msg=None*)
Fail if elements in container are unique.

Parameters

- **container** (*iterable*) –
- **msg** (*str*) – If not provided, the *marbles.mixins* or unittest standard message will be used.

Raises TypeError – If container is not iterable.

2.3 Developer Guide

2.3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/twosigma/marbles/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

The issue template will tell you how to collect version information.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

marbles could always use more documentation, whether as part of the official marbles docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/twosigma/marbles/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.

Build a Plugin

We've developed marbles in a pluggable way, so you can contribute to the marbles ecosystem without committing code to the marbles repo!

Just as marbles provides some mixins you can add to a `unittest.TestCase` by adding a superclass to your test, you can develop more mixins outside the marbles repo and publish them yourself, they'll interoperate just fine. Of course, we'd like to include mixins which may be useful to many people in the *`marbles.mixins`* package, so if you think that's the case, please send us a pull request.

The marbles annotation logging mechanism right now just writes JSON structured data to a file. You can use `logstash`, `mongoimport`, `Spark`, or any other tool that understands JSON to store and analyze them after the fact. But, you can also implement the marbles logging interface to do something else with the assertion metadata, instead of logging it to disk. If you'd like to share your logging plugin, or discuss ideas for how to build one, just open an issue and we can discuss it with you there.

Get Started!

Ready to contribute? Here's how to set up marbles for local development.

1. Fork the marbles repo on GitHub here.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/marbles.git
```

3. Install your local copy into a virtualenv. If you have pipenv installed, you can run:

```
$ cd marbles/
$ pipenv install --dev
$ pipenv shell
```

This will install all of the `marbles` development dependencies, install `marbles` in development mode, so your changes to the files in your clone will take effect immediately, and put you in a shell where you can run the tests, build the docs, etc.

If you don't use pipenv, you can get the same effect inside any other virtualenv by running:

```
$ pip install -r requirements.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature-or-docs
```

Now you can make your changes locally. `marbles` is developed as separate packages in the namespace package `marbles`:

1. The `TestCase` customizations and the assertion logging infrastructure live in *`marbles.core`*, which you'll find inside the repo under `marbles/core`.
2. The mixins live in *`marbles.mixins`*, which you'll find inside the repo under `marbles/mixins`.

5. As you make changes, you can run the tests and lint with flake8. These should be run inside the package you've made changes to, so if you've made changes to *`marbles.core`*, you should run this:

```
$ cd marbles/core
$ python setup.py flake8
$ python setup.py test
```

Note: Don't worry about bumping version numbers yourself. We'll handle this in the release that includes your changes.

For more developer workflows (linting, testing, test coverage, docs), see *Maintainer's Guide*.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.
8. We'll review your changes, merge them, and include them in the next release.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Make sure your new functionality is documented with docstrings and appropriate additions to the Sphinx docs, and add the feature to the list in README.md.
3. The pull request should work for Python 3.5, 3.6, and 3.7. Check https://travis-ci.org/twosigma/marbles/pull_requests and make sure that the tests pass for all supported Python versions.
4. In order to accept your code contributions, please fill out the appropriate Contributor License Agreement in the cla folder and submit it to tsos@twosigma.com. We need this before we can accept your pull request.

2.3.2 Maintainer's Guide

This page documents some common workflows for maintaining marbles. Most of these are also useful when developing marbles.

Environment

Marbles comes with a `Pipfile` that pipenv can use to manage your virtualenv for developing. To get started, install pipenv once for your machine with your package manager of choice:

```
$ pip install --user pipenv
```

Creating your environment

In the marbles codebase, create a development virtualenv, and enter it:

```
$ pipenv install --dev
$ pipenv shell
```

Inside this environment, you should have everything you need to work on marbles. See the other sections in this doc for common activities.

Adding new packages

If you want to add a new package for development, install it with pipenv:

```
$ pipenv install --dev pylint
```

This will update the `Pipfile` with `pylint` as a new development dependency, and will update `Pipfile.lock` with the exact version you installed. If this dependency is needed for something you added to marbles, you should include the changes to both `Pipfile` and `Pipfile.lock` in your pull request. Otherwise, please don't include the changes to `Pipfile` and `Pipfile.lock` in your pull request.

Linting

You can lint the code with flake8:

```
$ python -m flake8
```

Tests

You can run the tests for either *`marbles.core`* or *`marbles.mixins`* separately:

```
$ python marbles/core/setup.py test
$ python marbles/mixins/setup.py test
```

Coverage

Since the marbles tests are split across the subpackages, checking coverage isn't very straightforward, but we've configured tox to do it for you (more on tox below):

```
$ tox -e coverage
```

If you want to look at the source code annotated with coverage metrics, this produces an HTML report you can view, by loading file:///path/to/marbles/build/coverage/html/index.html in your browser.

Documentation

You can build the docs and view them locally:

```
$ python setup.py build_sphinx
```

Then, load file:///path/to/marbles/build/sphinx/html/index.html in your browser. If you make changes to just docstrings, but not `.rst` files, Sphinx may not rebuild those docs, you can embolden it to do so with these options:

```
$ python setup.py build_sphinx -Ea
```

Automation with tox

We use tox to run continuous integration builds for multiple versions of Python, and to run each piece of our continuous integration in a separate virtualenv. You can do this locally too, to make sure your change will build cleanly on Travis CI.

We've configured tox to be able to:

1. Run all the tests with Python 3.5 and 3.6
2. Measure and report on code coverage
3. Lint the code with flake8
4. Build the documentation

If you just run **`tox`** by itself, it will do all of the above, each in its own virtualenv:

```
$ tox
```

You can also run a subset of these with `-e`:

```
$ tox -e docs
$ tox -e py36
$ tox -e flake8,coverage
```

Maintaining the Changelog

The marbles *Changelog* is maintained by the Sphinx plugin releases, and its source is in `docs/changelog.rst`.

Most pull requests should add an item to the changelog, at the top, either a bug, feature, or support note.

Note: releases is clear about the distinction between bugs and other release notes. Bugs are included in the next patch version that appears above them, while features aren't included until the next major or minor version above them. The

decision of whether to note a change as a bug, feature, or support item will affect where it appears in the log, though this can be controlled with the keywords `major` (put bugs in the next major or minor release), and `backported` (put features in the next bugfix release).

See Release organization for details.

Right before releasing a new version of marbles, add a release item to the top of the changelog noting the version string and release date, then follow the below instructions on *Releasing a new version*.

Releasing a new version

The marbles meta-package and subpackage version strings are stored in a few different locations, due to the namespace package setup:

1. `setup.py`
2. `setup.cfg`
3. `marbles/core/marbles/core/VERSION`
4. `marbles/mixins/marbles/mixins/VERSION`

In addition, when we bump the version, we do so in an isolated commit, and tag that commit with the version number as well.

Note: Make sure you've groomed the *Changelog* before tagging a new release. See *Maintaining the Changelog* for details.

We use bumpversion to automate this. To run bumpversion, you need to be in a clean git tree (don't worry, it will complain to you if that's not the case).

You can increase either the `major`, `minor`, or `patch` version:

```
$ bumpversion major
$ bumpversion minor
$ bumpversion patch
```

This will update the version strings in all the above files and commit that change, but won't tag it. You should create a pull request for the version update, merge it (without squashing it into other commits), and then tag it once it's on the `master` branch: https://github.com/twosigma/marbles/releases/new.

You can read a digression about why we bump all the versions at the same time below, in *Versioning philosophy*.

Uploading to PyPI

Once you've tagged the latest version of marbles, pull from GitHub to make sure your clone is up to date and clean, build both `sdist` and `wheel` packages for all three packages, and upload them with twine. We have a tox rule to automate building and uploading:

```
$ tox -e pypi
```

Versioning philosophy

Marbles publishes two subpackages, *marbles.core* and *marbles.mixins*, and a metapackage depending on both, marbles. This allows users to install or depend on only one of the subpackages, and also suggests that anyone can publish their own mixins package.

This raises the question of how to version each of these three packages.

1. Release new versions of *marbles.core* and *marbles.mixins* independently, and have the marbles package basically only ever have one release, 1.0.0, since it doesn't actually change over time.
2. Give the marbles package a new version each time either subpackage gets one, to make it feel like we're moving forward.
3. Release all three packages with the same version string each time any of them gets a new release.

Jupyter takes the first approach, but keep in mind that Jupyter is a much larger project with distinct teams working on each component, so allowing subpackages to have independent release schedules makes more sense for that community.

The second approach has the problem that if we release the subpackages independently, it's unclear how to version the metapackage when that happens. Taking the max of the subpackage version strings doesn't work if the subpackage with a lower one gets an update by itself. There are a couple other possiblities here, but none of them seemed right.

The third approach, updating everything in lock-step, is what we've chosen. This will create multiple versions of one or the other package that are identical, in some cases, which is a little odd. However, it has the benefit of documenting which versions of *marbles.core* and *marbles.mixins* were reviewed and tested together and therefore can be expected to work together. It still allows users to install (and update) them independently, but encourages users of both to update them together.

2.3.3 Changelog

- #125: Fix long_description for PyPI
- #50: Add a distutils command for marbles
- #119: Upgrade to pandas 1.0
- #120: Upgrade to python 3.8
- #107: Allow triple-quoted annotations to be indented in source
- #99: Fixed assertCategoricalLevel(Not)In docstring parameters
- #101: Added note about how to execute logging configured tests
- #105: Fixed UniqueMixins literalinclude line numbers in docs
- #92: Improve indentation of multiline locals
- #88: Document how to install with conda
- #90: Added support for python 3.7
- #58: Fixed test failure on OSX
- #80: Added support for pandas<0.24
- #65: Fixed sdist installation
- #41: Removed developer dependencies in setup_requires and tests_require
- #40: Fixed "Locals" section in failure output to be hidden when no locals will be displayed

- #39: Added issue templates
- #43: Added version bumping automation and maintenance documentation
- #1: Fixed tests to run in virtualenvs
- #5: Changed copyright headers to refer to TSOS and the MIT license
- #15: Added Creative Commons attribution for test content from Wikipedia
- #30: Removed TS internal conda recipe
- #21: Removed TS internal details from documentation and comments
- #18: Removed TS internal details from README
- #14: Removed DataFrame and Panel mixins
- #16: Added Contributor License Agreement forms
- #17: Changed to pipenv for development environment management
- #26: Added development automation and CI with tox
- #28: Added Travis CI integration
- #31: Added PyPI packaging
- :

  Note: First public release

- : Changed test case and test method to log separately, and added marbles version
- : Changed annotation to be optional with `marbles.core.TestCase`
- : Removed Traceback display for marbles assertion failures
- : Added main method to provide `python -m marbles`
- : Large refactor and doc rewrite to prepare for open source
- : Split package into `marbles.core` and `marbles.mixins`
- : Fixed source code extraction to find it inside eggs
- : Fixed mixins that expect a specific type to raise `TypeError` instead of `AssertionError`
- : Added conda recipe (internal only)
- :

  Nice

  Nice

- : Changed annotation wrapping to wrap paragraphs in annotations individually for better formatting
- : Added verbose logging option
- : Added documentation about authoring good marbles docs
- : Added mixins library
- : Added richer text formatting in annotations

- : Added assertion logging
- : Removed extra `message` annotation
- : Removed display of “private” locals
- : Fixed positional argument handling
- : Added ability to capture and display locals
- : Added source code for the whole statement that failed to failure messages
- : Added annotation support in `assert*` methods

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

m

Index

A