# torment Documentation

***Release 3.0.3***

**Alex Brandt**

January 15, 2016

Contents

Torment is scalable testing fixtures.

CHAPTER 1

# Getting Started

Torment has many options to generate fixtures to fit your testing needs.

Multiple fixtures with the different data:

```
register(globals(), ( RequestFixture, ), {...})
```

Multiple fixtures using the same data (the runtime changes the behavior of the test):

```
p = {...}

register(globals(), ( WriteFixture, ), p)
regsiter(globals(), ( ReadFixture, ), p)
```

Multiple fixtures using dynamic data:

```
for a in fixtures.of(( AccountModelFixture, )):
    register(globals(), ( RequestFixture, ), {
        'account': a,
    })
```

Automatic mocking via the fixture data:

```
   p = {
       'mocks': {
           'mymodule.myfunc': {
               'return_value': True,
           },
       },

* ``mocked_mymodule.myfunc`` is available in your tests and returns True
```

## 1.1 Torment Usage

In order to work as expected, Torment is based on a series of rules. The minimum requirements to get started are listed below.

1. A filename with the following format: **[descriptive-statement]_{UUID}.py**
    - Where are these files located?
        - These can be located anywhere you would like. In source, out of source, whatever is desired. Normally alongside other tests.

- How do I load these files?
  - `torment.helpers.import_directory` recursively loads python modules in a directory:

```
helpers.import_directory(__name__, os.path.dirname(__file__))
```

2. The newly created file must contain at least one register to build a testcase
   - `torment.fixtures.register` associates runtime with data, in other words it puts the data & class together
3. The register requires a FixtureClass (type is defined elsewhere)
   - What kind of class?
     - Must be a subclass of `torment.fixtures.Fixture`
   - Where do I define it?
     - There are no restrictions on where you define
4. A FixtureClass requires a TestContext
   - What goes into TestContext class, etc?
     - TestContext specifies which fixtures it should test:

```
class HelperUnitTest(TestContext, metaclass = contexts.MetaContext):
    fixture_classes = (
        ExtendFixture,
    )
```

   - Why do I have to set my metaclass to metacontext?
     - The metacontext turns fixtures into test methods

---

**Note:** A metaclass is the object that specifies how a class is created. `torment.contexts.MetaContext` is a metaclass we created to build TestContext classes.

If you are unfamiliar with metaclasses, it is highly recommended that you read the offical Python documentation here before getting started. For a quick primer refer to Jake Vanderplas' blog post from 2012.

---

### 1.1.1 `torment.contexts` — Testing Contexts

**class** `torment.contexts.`**`MetaContext`**(*name*, *bases*, *dct*) → None
`torment.TestContext` class creator.

Generates all testing methods that correspond with the fixtures associated with a `torment.TestContext`. Also updates the definitions of `mocks_mask` and `mocks` to include the union of all involved classes in the creation process (all parent classes and the class being created).

When creating a `torment.TestContext` subclass, ensure you specify this class as its metaclass to automatically generate test cases based on its `fixture_classes` property.

**`module`**
Actual module name corresponding to this context's testing module.

**class** `torment.contexts.`**`TestContext`**(*methodName='runTest'*)
Environment for Fixture execution.

Provides convenience methods indicating the environment a Fixture is executing in. This includes a references to the real module corresponding to the context's testing module as well as a housing for the assertion methods.

Inherits most of its functionality from `unittest.TestCase` with a couple of additions. TestContext does extend setUp.

When used in conjunction with `torment.MetaContext`, the `fixture_classes` property must be an iterable of subclasses of `torment.fixtures.Fixture`.

**Properties**

- `module`

**Public Methods**

- `patch`

**Class Variables**

**Mocks_mask** set of mocks to mask from being mocked

**Mocks** set of mocks this TestContext provides

**`module`**
Actual module name corresponding to this context's testing module.

**`patch`**(*name: str*, *relative: bool=True*) → None
Patch name with mock in actual module.

Sets up mock objects for the given symbol in the actual module corresponding to this context's testing module.

**Parameters**

**Name** the symbol to mock—must exist in the actual module under test

**Relative** prefix actual module corresponding to this context's testing module to the given symbol to patch

### 1.1.2 `torment.fixtures` — Torment Fixtures

#### Fixture

**class** `torment.fixtures.`**`Fixture`**(*context: 'torment.TestContext'*) → None
Collection of data and actions for a particular test case.

Intended as a base class for custom fixtures. Fixture provides an API that simplifies writing scalable test cases.

Creating Fixture objects is broken into two parts. This keeps the logic for a class of test cases separate from the data for particular cases while allowing re-use of the data provided by a fixture.

The first part of Fixture object creation is crafting a proper subclass that implements the necessary actions:

**__init__** pre-data population initialization

**Initialize** post-data population initialization

**Setup** pre-run setup

**Run** REQUIRED—run code under test

**Check** verify results of run

**Note:** `initialize` is run during `__init__` and setup is run after; otherwise, they serve the same function. The split allows different actions to occur in different areas of the class heirarchy and generally isn't necessary.

By default all actions are noops and simply do nothing but run is required. These actions allow complex class hierarchies to provide nuanced testing behavior. For example, Fixture provides the absolute bare minimum to test any Fixture and no more. By adding a set of subclasses, common initialization and checks can be performed at one layer while specific run decisions and checks can happen at a lower layer.

The second part of Fixture object creation is crafting the data. Tying data to a Fixture class should be done with `torment.fixtures.register`. It provides a declarative interface that binds a dictionary to a Fixture (keys of dictionary become Fixture properties). `torment.fixtures.register` creates a subclass that the rest of the torment knows how to transform into test cases that are compatible with nose.

**Examples**

Simplest Fixture subclass:

```
class MyFixture(Fixture):
    pass
```

Of course, to be useful the Fixture needs definitions of setup, run, and check that actually test the code we're interested in checking:

```
def add(x, y):
    return x + y

class AddFixture(Fixture):
    def run(self):
        self.result = add(self.parameters['x'], self.parameters['y'])

    def check(self):
        self.context.assertEqual(self.result, self.expected)
```

This fixture uses a couple of conventions (not requirements):

1. `self.parameters` as a dictionary of parameter names to values
2. `self.expected` as the value we expect as a result
3. `self.result` as the holder inside the fixture between `run` and `check`

This show-cases the ridiculity of using this testing framework for simple functions that have few cases that require testing. This framework is designed to allow many cases to be easily and declaritively defined.

The last component required to get these fixtures to actually run is hooking them together with a context:

```
from torment import contexts

class AddUnitTest(contexts.TestContext, metaclass = contexts.MetaContext):
    fixture_classes = (
        MyFixture,
        AddFixture,
    )
```

The context that wraps a Fixture subclass should eventually inherit from TestContext (which inherits from `unittest.TestCase` and provides its assert methods). In order for nose to find and execute this `TestContext`, it must have a name that contains Test.

**Properties**

- `category`

- `description` (override)
- `name` (do **not** override)

**Methods To Override**

- `__init__`
- `check`
- `initialize`
- `run (required)`
- `setup`

**Instance Variables**

**Context** the `torment.TestContext` this case is running in which provides the assertion methods of `unittest.TestCase`.

**`category`**
Fixture's category (the containing testing module name)

**Examples**

**Module** test_torment.test_unit.test_fixtures.fixture_a44bc6dda6654b1395a8c2cbd55d964d

**Category** fixtures

**`check`**`()` → None
Check that run ran as expected.

---

**Note:** Override as necessary. Default provided so re-defenition is not necessary.

---

Called after `run` and should be used to verify that run performed the expected actions.

**`description`**
Test name in nose output (intended to be overridden).

**`initialize`**`()` → None
Post-data population initialization hook.

---

**Note:** Override as necessary. Default provided so re-defenition is not necessary.

---

Called during `__init__` and after properties have been populated by `torment.fixtures.register`.

**`name`**
Method name in nose runtime.

**`setup`**`()` → None
Pre-run initialization hook.

---

**Note:** Override as necessary. Default provided so re-defenition is not necessary.

---

Called after properties have been populated by `torment.fixtures.register`.

### Registration

`torment.fixtures.`**`register`**(*namespace*, *base_classes: typing.Tuple*, *properties: typing.Dict*) → None

Register a Fixture class in namespace with the given properties.

Creates a Fixture class (not object) and inserts it into the provided namespace. The properties is a dict but allows functions to reference other properties and acts like a small DSL (domain specific language). This is really just a declarative way to compose data about a test fixture and make it repeatable.

Files calling this function are expected to house one or more Fixtures and have a name that ends with a UUID without its hyphens. For example: foo_38de9ceec5694c96ace90c9ca37e5bcb.py. This UUID is used to uniquely track the Fixture through the test suite and allow Fixtures to scale without concern.

**Parameters**

**Namespace** dictionary to insert the generated class into

**Base_classes** list of classes the new class should inherit

**Properties** dictionary of properties with their values

Properties can have the following forms:

**Functions** invoked with the Fixture as it's argument

**Classes** instantiated without any arguments (unless it subclasses `torment.fixtures.Fixture` in which case it's passed context)

**Literals** any standard python type (i.e. int, str, dict)

---

**Note:** function execution may error (this will be emitted as a logging event). functions will continually be tried until they resolve or the same set of functions is continually erroring. These functions that failed to resolve are left in tact for later processing.

---

Properties by the following names also have defined behavior:

**Description** added to the Fixture's description as an addendum

**Error** must be a dictionary with three keys: :class: class to instantiate (usually an exception) :args: arguments to pass to class initialization :kwargs: keyword arguments to pass to class initialization

**Mocks** dictionary mapping mock symbols to corresponding values

Properties by the following names are reserved and should not be used:

- name

- genindex
- modindex
- search

## t

## C



## D



## F



## I



## M



## N



## P



## R



## S



## T