# Granite Documentation

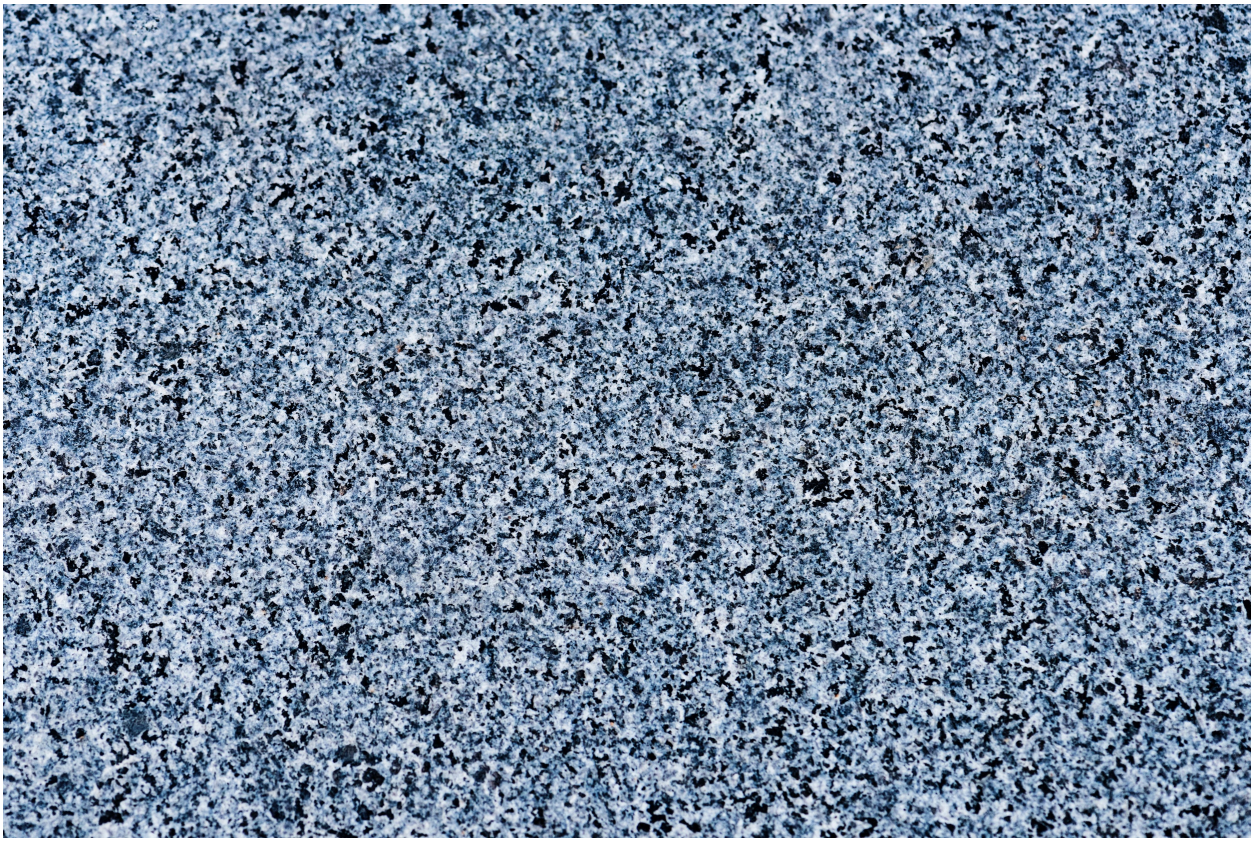
***Release 0.0.5***

**Aaron Boman**

**Jul 05, 2018**

# Contents

Make your Python applications as solid as granite.



Supports Python 2.7, 3.4+

This package provides helpers for testing, documentation, and reporting for Python packages. To quickly get started, have a look at one of the guides below or have a look at the *Granite library API*.

CHAPTER 1

# Using Granite

## 1.1 Asset Management

There are two times when tests need access to files on disk

- the function under test requires a file on disk (usually by filename)
- a function requires a large amount of data, specially formatted data, or the data is more easily stored as a file on disk rather than being embedded in the test itself. E.g.:
  - XML
  - JSON
  - Images
  - INI/Configuration files
  - A Python script
  - etc.

Granite supplies the *AssetMixin* to provide support for easily accessing test asset files.

**Note:** If you need to also write files to disk, check out *Temporary Projects*.

### 1.1.1 Setup

To setup, add the `AssetMixin` to the list of base classes on your TestCase class and create a class level attribute `ASSET_DIR` that points to the directory containing your asset files:

```
# tests/some_test.py
import os
```

(continues on next page)

(continued from previous page)

```
from granite.testcase import TestCase, AssetMixin

THIS_DIR = os.path.dirname(os.path.abspath(__file__))

class MyTestCase(AssetMixin, TestCase):
    # assume that the asset directory exists at `tests/assets`
    ASSET_DIR = os.path.join(THIS_DIR, 'assets')
```

**Note:** The `AssetMixin` comes *before* the concrete `TestCase` class.

### 1.1.2 Getting Asset Filenames

Use *get_asset_filename* to get the absolute path to a filename within the `ASSET_DIR`. For example:

```
# tests/some_test.py

def test_that_foo_can_read(self):
    # assume that `tests/assets/some_file.txt` exists
    filename = self.get_asset_filename('some_file.txt')
    # pass the absolute path to the foo() function
    foo(filename)
```

The path parameter acts just like `os.path.join()` and can accept multiple parameters to be joined. For example:

```
>>> self.get_asset_filename('path', 'to', 'my', 'file.txt')
'/absolute/path/to/my/file.txt'
```

If the given path does not exist, an *AssetNotFound* error will be raised:

```
>>> self.get_asset_filename('path/to/some/nonexistent/file.txt')
Traceback
    ...
AssetNotFound: self.get_asset_filename() was called with ...
```

### 1.1.3 Reading from an Asset File

Use *read_asset_file* to open the asset file and return its contents:

```
# tests/some_test.py

def test_that_xml_can_be_parsed(self):
    xml = self.read_asset_file('my.xml')
    root = foo(xml)
    self.assertEqual(etree.tostring(root), xml)
```

Under the hood, `read_asset_file()` uses `get_asset_filename()` so path also accepts multiple arguments and will `os.path.join()` all of them together to form a single path.

Additionally, use the `mode` keyword argument to specify how the file should be opened. For example, your function under test requires an image file that needs to be opened in binary mode:

```
# tests/some_test.py

def test_that_image_size_is_returned(self):
    img = self.read_asset_file('1920x1080.jpg', mode='rb')
    size = foo(img)
    self.assertEqual((1920, 1080), size)
```

### 1.1.4 Advanced setup

If you find that all (or most) of your tests require access to the asset directory, add the `AssetMixin` to your test's `BaseTestCase` class:

```
# tests/__init__.py
from granite.testcase import TestCase, AssetMixin

class BaseTestCase(AssetMixin, TestCase):
    # assume that `tests/assets` exists
    ASSET_DIR = os.path.join(THIS_DIR, 'assets')
```

Then, simply inherit from your `BaseTestCase` in your child `TestCase` classes to get the asset functionality:

```
# tests/some_test.py
from tests import BaseTestCase

class TestSomething(BaseTestCase):
    # self.get_asset_filename() and self.read_asset_file() exist!
```

#### Using different directories per TestCase

One test may require one directory, and another test may use another. Simply change the `ASSET_DIR` to use a different directory for the specific TestCase instance:

```
# tests/some_test.py
from tests import BaseTestCase

class TestSomething(BaseTestCase):
    ASSET_DIR = os.path.join(THIS_DIR, 'other', 'assets')
```

#### Better asset organization

If your tests require a lot of asset files it's a good idea to try and organize files that are specific to some tests into their own directory. For example, `test_foo.py` requires files `a.txt` and `b.txt` while `test_bar.py` requires `c.txt` and `d.txt`. The resulting file structure is suggested:

```
tests/
  \ assets
      \ foo
      | | - a.txt
      | | - b.txt
      \ bar
        | - c.txt
        | - d.txt
```

However, this requires every use of `get_asset_filename()` or `read_asset_file()` to require the directory prefix (either `foo` or `bar`). Instead, the `BaseTestCase.ASSET_DIR` attribute can be extended:

```
# tests/test_foo.py
import os

from tests import BaseTestCase

class TestFoo(BaseTestCase):
    ASSET_DIR = os.path.join(BaseTestCase.ASSET_DIR, 'foo')
```

This way, if the asset directory is ever moved, the BaseTestCase class will be the only place that needs to be updated.

## 1.2 Temporary Projects

When you need to be able to create a small environment to read files from or to write files to, you need to create a Temporary Project that only lasts as long as the test needs it. The *TemporaryProjectMixin* does just this.

### 1.2.1 Basic Setup

To setup, add the `TemporaryProjectMixin` to the list of base classes on your TestCase class:

```
# test/test_foo.py
from granite.testcase import TestCase, TemporaryProjectMixin

class TestFoo(TemporaryProjectMixin, TestCase):
    # ...
```

On test `setUp()`, the `TemporaryProjectMixin` will create a temporary directory and on test `tearDown()` that temporary directory will be destroyed.

### 1.2.2 Interacting with the Temporary Project

The `TemporaryProjectMixin` adds a new attribute to the TestCase instance: `temp_project`. This attribute is an instance of the *TemporaryProject* class and provides accessor methods for interacting with the temporary directory.

`TemporaryProject.`**`abspath`**(*filename*)

Get the absolute path to the filename found in the temp dir.

#### Notes

- Always use forward slashes in paths.
- This method does not check if the path is valid. If the filename given doesn't exist, an exception is not raised.

**Parameters** **`filename`** (*str*) – the relative path to the file within this temp directory

**Returns** the absolute path to the file within the temp directory.

**Return type** str

TemporaryProject.**read**(*filename*, *mode='r'*)
Read the contents of the file found in the temp directory.

**Parameters**

- **filename** (*str*) – the path to the file in the temp dir.
- **mode** (*str*) – a valid mode to open(). defaults to *'r'*

**Returns** The contents of the file.

TemporaryProject.**write**(*filename*, *contents=''*, *mode='w'*, *dedent=True*)
Write the given contents to the file in the temp dir.

If the file or the directories to the file do not exist, they will be created.

If the file already exists, its contents will be overwritten with the new contents unless mode is set to some variant of append: (`a`, `ab`).

Specify the dedent flag to automatically call `textwrap.dedent` on the contents before writing. This is especially useful when writing contents that depend on whitespace being exact (e.g. writing a Python script). This defaults to `True` except when the mode contains `'b'`

**Parameters**

- **filename** (*str*) – the relative path to a file in the temp dir
- **contents** (*any*) – any data to write to the file
- **mode** (*str*) – a valid open() mode
- **dedent** (*bool*) – automatically dedent the contents (default: `True`)

TemporaryProject.**remove**(*filename*)
Removes the filename found in the temp dir.

**Parameters** **filename** (*str*) – the relative path to the file

TemporaryProject.**touch**(*filename*)
Creates or updates timestamp on file given by filename.

**Parameters** **filename** (*str*) – the filename to touch

TemporaryProject.**glob**(*pattern*, *start=''*, *absolute=False*)
Recursively searches through the temp dir for a filename that matches the given pattern and returns the first one that matches.

**Parameters**

- **pattern** (*str*) – the glob pattern to match the filenames against. Uses the fnmatch module's fnmatch() function to determine matches.
- **start** (*str*) – a directory relative to the root of the temp dir to start searching from.
- **absolute** (*bool*) – whether the returned path should be an absolute path; defaults to being relative to the temp project.

**Returns**

**the relative path to the first filename that matches pattern** unless the absolute flag is given. If a match is not found None is returned.

**Return type** str

TemporaryProject.**snapshot**()
Creates a snapshot of the current state of this temp dir.

**Returns** the snapshot.

**Return type** *Snapshot*

`TemporaryProject.`**`copy_project`**(*dest*, *overwrite=False*, *symlinks=False*, *ignore=None*)

Allows for a copying the temp project to the destination given.

This provides test authors with the ability to preserve a tes environment at any point during a test.

By default, if the given destination is a directory that already exists, an error will be raised (shutil.copytree's error). Set the `overwrite` flag to `True` to overwrite an existing directory by first removing it and then copying the temp project.

**Parameters**

- **`dest`** (*`str`*) – the destination directory
- **`overwrite`** (*`bool`*) – if the directory exists, this will remove it first before copying
- **`symlinks`** (*`bool`*) – passed to shutil.copytree: should symlinks be traversed?
- **`ignore`** (*`bool`*) – ignore errors during copy?

`TemporaryProject.`**`teardown`**()

Provides a public way to delete the directory that this temp project manages.

This allows for the temporary directory to be cleaned up on demand.

Ignores all errors.

---

**Note:** The `temp_project` attribute has a public `.path` attribute which holds the absolute path to the temporary directory.

---

### Example of writing a file

When a file needs to be written to disk (a templated file, etc.) use the *`write`* method of the `TemporaryProject` to write that file to the temporary project:

```
# tests/test_foo.py

def test_that_file_is_written(self):
    # note that the path to the file should use forward
    # slashes (even on Windows!). The directories will be
    # created automatically.
    self.temp_project.write('path/to/some_file.txt', 'contents to write')

    # assert that the new file exists.
    # note: this uses the .path attribute of the `temp_project` in order
    #       to get the absolute path to the temporary directory
    self.assertTrue(
        os.path.exist(
            os.path.join(self.temp_project.path, 'path', 'to', 'some_file.txt')))

    # we can also assert that the new file exists by using the
    # TemporaryProjectMixin.assert_temp_path_exists() assert method.
    self.assert_temp_path_exists('path/to/some_file.txt')
```

### 1.2.3 TemporaryProjectMixin Assert Methods

The `TemporaryProjectMixin` provides additional assert methods useful for asserting conditions on the temporary project.

`TemporaryProjectMixin.`**`assert_in_temp_file`**(*substring*, *filename*, *msg=''*, *mode='r'*, *not_in=False*)

Asserts that the given contents are found in the file in the temp project.

**Parameters**

- **substring** (*str*) – the substring to look for in the file's contents
- **filename** (*str*) – the name of the file relative to the temp project
- **msg** (*str*) – the message to output in the event of a failure.
- **mode** (*str*) – the mode to open the file with. defaults to 'r'
- **not_in** (*bool*) – asserts that the contents are not in the file

`TemporaryProjectMixin.`**`assert_not_in_temp_file`**(*substring*, *filename*, *msg=''*, *mode='r'*)

Asserts that the given contents are not found in the file in the temp project.

**Parameters**

- **substring** (*str*) – the substring to look for in the file's contents
- **filename** (*str*) – the name of the file relative to the temp project
- **msg** (*str*) – the message to output in the event of a failure.
- **mode** (*str*) – the mode to open the file with. defaults to 'r'

`TemporaryProjectMixin.`**`assert_temp_path_exists`**(*path='.'*, *msg=''*)

Asserts that the path given exists relative to the root of the temp project.

**Parameters**

- **path** (*str*) – the string of the path relative to the root of the temp directory.
- **msg** (*str*) – a custom string to show in the event that this assertion fails.

### 1.2.4 Taking A Snapshot of the Temporary Project

Sometimes it's necessary to know what changed within a temporary project. Use the *snapshot* method on the `self.temp_project` in order to record a *Snapshot* of the complete state of all files and directories within the temp project. A snapshot by itself is somewhat useless, but with two snapshots, you can create a diff of the state of the temp project. A *SnapshotDiff* contains lists of `added`, `removed`, `modified`, and `touched` files. Note that a `touched` file is one whose timestamp has changed, but its contents have not. A `modified` file has had its contents change.

Example:

```
# tests/test_change_in_dir.py

class TestChangeInDir(TemporaryProjectMixin, BaseTestCase):
    def test_that_dir_changed(self):
        start = self.temp_project.snapshot()
        self.temp_project.write('hello.txt')
        end = self.temp_project.snapshot()
        diff = end - start
        self.assertIn('hello.txt', diff.added)
```

### 1.2.5 Advanced Setup

The `TemporaryProjectMixin` class allows for supplying some class-level attributes in order to configured the TestCase class.

`TemporaryProjectMixin.`**`TMP_DIR = None`**
: Allows for setting the temp directory. Defaults to `None` which will use Python's `tempfile.mkdtemp` to make the temp directory.

`TemporaryProjectMixin.`**`PRESERVE_DIR = None`**
: Sets where the preserved path should be dumped too. This overrides the `TMP_DIR` when `ENABLE_PRESERVE` is set to True.

`TemporaryProjectMixin.`**`ENABLE_PRESERVE = False`**
: A flag indicating whether the temp project should be preserved after the temp project object is destroyed. If True, the directory will still exist allowing a user to view the state of the directory after a test has run. This works in tandem with the `PRESERVE_DIR` class attribute.

`TemporaryProjectMixin.`**`TemporaryProjectClass = <class 'granite.environment.TemporaryProject`**
: Set this attribute to a class that implements the interface of *`TemporaryProject`*. This allows for creating a custom temporary project manager. A typical use case would be to subclass `TempProject` and override specific functionality then specify that new class here.

#### Setting a custom temp directory

Sometimes it's desirable to be in control of the temp directory. In order to change the location of the temporary directory, set the `TMP_DIR` attribute at the class level:

```
# tests/test_foo.py
import os

from granite.testcase import TestCase, TemporaryProjectMixin

THIS_DIR = os.path.dirname(os.path.abspath(__file__))

class TestFoo(TemporaryProjectMixin, TestCase):
    # set to be a directory named '.tmp' at the root of the project
    TMP_DIR = os.path.join(THIS_DIR, '..', '.tmp')
```

**Note:** There probably isn't a good reason to change this. Hard-coding a single path will make running tests in parallel impossible, so it's probably best to stick to the default. The only reason that a deterministic temporary path may be desirable is to inspect the contents of the temporary directory before, during, or after a test run in order to assert that tests are running as expected or to debug a test. For this case see below for setting a preserve path.

#### Setting a preserve path for temporary project debugging

During testing with temporary projects, inevitably it becomes desirable to preserve the temporary directory in order to debug its contents. The `TemporaryProjectMixin` provides an option for enabling the preservation of the temp project and setting a known location in order to view that temporary directory:

```
# tests/test_foo.py
import os
```

(continues on next page)

(continued from previous page)

```
from granite.testcase import TestCase, TemporaryProjectMixin

THIS_DIR = os.path.dirname(os.path.abspath(__file__))

class TestFoo(TemporaryProjectMixin, TestCase):
    # set to be a directory named '.tmp' at the root of the project
    PRESERVE_DIR = os.path.join(THIS_DIR, '..', '.tmp')
    # without this, the TMP_DIR option will still be used
    ENABLE_PRESERVE = True

    def test_foo(self):
        # ...
```

Setting the `PRESERVE_DIR` sets the root of all preserved directories. All temporary projects will be created underneath this directory using the TestCase's class name for the first directory and the resultant temp project will be created under another directory named after the test method.

For example, when the above `test_foo` is run, a temporary project will be created at `.tmp/TestFoo/test_foo` (relative to the project root). This allows for inspection of the temp project contents after the test has run at a predetermined path.

Note that the `ENABLE_PRESERVE` attribute can be parameterized based on command line arguments. For this reason, it's a good idea to provide a concrete `TemporaryProjectTestCase` class or to make some base class inherit the `TemporaryProjectMixin` so that the logic of enabling or disabling the preservation of temp projects is all in one place:

```
# tests/__init__.py
import sys

from granite.testcase import TestCase, TemporaryProjectMixin

# either make *all* tests use the TemporaryProjectMixin
class BaseTestCase(TemporaryProjectMixin, TestCase):
    # enable or disable preserve based on a command line flag
    ENABLE_PRESERVE = '--preserve' in sys.argv
    # set to be a directory named '.tmp' at the root of the project
    PRESERVE_DIR = os.path.join(THIS_DIR, '..', '.tmp')


# or provide a concrete temp project test case class only for
# tests that require a temp project
class TempProjectTestCase(TemporaryProjectMixin, TestCase):
    # enable or disable preserve based on a command line flag
    ENABLE_PRESERVE = '--preserve' in sys.argv
    # set to be a directory named '.tmp' at the root of the project
    PRESERVE_DIR = os.path.join(THIS_DIR, '..', '.tmp')
```

Doing the above provides the preservation configuration in one spot and all later tests can benefit by inheriting.

## 1.3 Renderable Templates

When you need to be able to create specific types of files for your tool (e.g. ini/config files, scripts, etc) in a parameterized way, use renderable templates.

**Note:** The *Renderable*, *RenderedFile*, and *SimpleFile* classes all use the Jinja2 templating system. See its documentation for any specific questions pertaining to the templates themselves.

**class RenderedFile**(*filename=''*)

A file that will be rendered to disk.

This class represents a file that, upon rendering, will appear on disk. By default, any attribute (instance or class) will appear in the context of the template. E.g., having an attribute of `self.foo` will make the `foo` variable exist within the template.

This class provides a "content" variable and should serve as an area of the template that can be appended to on a per-test basis. Use the add_content() method in order to add more content to the template at the time of render. This allows for content to be added over a period of time either as subsequent calls in the same test, through the setUp, or setUp calls in multiple classes (via inheritance).

Subclasses can override the get_context() method in order to alter the context (variable scope) provided to the template on render. See the get_context() method's documentation for more details.

While this class can be used by itself (note that the instances attributes `template` and `template_dirs` must be set before rendering!) the intended use is to subclass this class and define class-level attributes for `template` and `template_dirs`. This makes it so that a base class can point to a common template directory (through the template_dirs attribute) and all subclasses of it can supply the `template` attribute in order to determine which template to choose.

**Parameters** **filename** (*str*) – the full path to where the file should exist on disk when rendered

Examples:

```
# tests/environment.py
# assume that the templates directory is:
# tests/templates
from granite.environment import RenderedFile
from granite.testcase import TestCase, TempProjectMixin


class MyRenderedFile(RenderedFile):
    template_dirs = [
        os.path.join(os.path.dirname(__file__), 'templates')]


# assume that tests/templates/template.py exists
# and looks something like this:
print('Hello!')
print(
    '''
     {{ content }}
    '''
)
print('This test is currently running: {{ id }}')

class PythonScript(MyRenderedFile):
    template = 'template.py'


class MyTestCase(TempProjectMixin, TestCase):
    def setUp(self):
        super(MyTestCase, self).setUp()
```

(continues on next page)

(continued from previous page)

```
        self.script = PythonScript(
            os.path.join(self.temp_project.path, 'my_file.py'))
        self.script.add_content('My name is Aaron')

    def test_the_thing(self):
        # setting the `id` attribute provides the `id`
        # context variable in the template
        self.script.id = self.id()
        self.script.render()
        with open(self.script.full_name) as f:
            self.assertEqual(
                f.read(),
                """
                print('Hello!')
                print(
                    '''
                    My name is Aaron
                    '''
                )
                print('This test is currently running: test_the_thing')
                """
            )
```

**ADD_NEWLINE = True**
Determines whether a newline should be added at the end of the file or not. When generating C code to be compiled by the ARM/GCC compiler, this makes the compiler happy. Defaults to True. Set to False to disable.

**DISABLE_ESCAPING = False**
By default, the value in the "content" template variable is escaped. This makes adding content for script/language files (via add_content()) much easier to read and maintain as script/language files usually interpret the special esacped characters differently. Set this attribute to True in order to disable this functionality.

**WRITE_MODE = 'w'**
When rendering the file, defaults to non-binary mode. Set this to 'wb' or something similar for different behavior when writing the rendered contents to a file.

**add_content**(*content*)
Adds a string to the "content" variable available to the template.

**Parameters** **content** (*str or List[str]*) – a single string or a list of strings to add to the content variable.

**filename = ''**
The name of the rendered file (without the path)

**full_name = ''**
the full path to the file

**get_context**()
Gets the context needed for rendering the file associated with this instance.

A context is simply the template's scope of variables and functions.

**Returns**

**the variable names and their values; the scope to appear in** the template

**Return type** dict

**path = ''**
The path to this rendered file (without the filename itself)

**render()**
Renders the template and writes the contents to disk to this instance's filename.

## 1.4 AutoMockMixin

It's been my experience that when you're writing a TestCase that tests a specific function and that function needs to mock one (or more) of the functions that it uses, that function needs to be mocked throughout the entirety of the TestCase. The *AutoMockMixin* helps alleviate the burden this can cause.

`mock.patch` is typically used for most cases as a decorator on the test function like this:

```
# tests/test_foo.py
from unittest import mock

from tests import BaseTestCase

from myproject import foo


class TestFoo(BaseTestCase):
    @mock.patch('myproject.bar')
    def test_foo(self, mocked_bar):
        mocked_bar.return_value = 'baz'
        self.assertTrue(foo())
```

This works nicely when their is only a single function to mock and is even better if only one test needs to mock. However, if every test needs to mock and if each test needs more than one function to mock writing all of those `mock.patch` decorators can become hairy:

```
# tests/test_foo.py

class TestFoo(BaseTestCase):
    @mock.patch('myproject.bar')
    @mock.patch('myproject.baz')
    @mock.patch('myproject.biz')
    @mock.patch('myproject.boom')
    def test_1(self, bar, baz, biz, boom):
        # ...

    @mock.patch('myproject.bar')
    @mock.patch('myproject.baz')
    @mock.patch('myproject.biz')
    @mock.patch('myproject.boom')
    def test_2(self, bar, baz, biz, boom):
        # ...

    # ...
```

It's for times like this that you can return value of `mock.patch` to set up mocking in `setUp()` and `tearDown()`:

```
# tests/test_foo.py

class TestFoo(BaseTestCase):
```

(continues on next page)

(continued from previous page)

```
    def setUp(self):
        super(TestFoo, self).setUp()
        self.bar_patcher = mock.patch('myproject.bar')
        self.bar = self.bar_patcher.start()

        self.baz_patcher = mock.patch('myproject.baz')
        self.baz = self.bar_patcher.start()

        self.biz_patcher = mock.patch('myproject.biz')
        self.biz = self.bar_patcher.start()

    def tearDown(self):
        super(TestFoo, self).tearDown()
        self.bar_patcher.stop()
        self.bar_patcher = None
        self.bar = None

        self.baz_patcher.stop()
        self.baz_patcher = None
        self.baz = None

        self.biz_patcher.stop()
        self.biz_patcher = None
        self.biz = None

    def test_1(self):
        self.bar.return_value = 'baz'
        self.baz.return_value = True
        self.biz.return_value = 'asdf'
        self.AssertTrue(foo())
```

This kind of a setup only requires you to declare the mocked function once, but even this is fairly tedious. This is where the *AutoMockMixin* comes in handy. Using it, the above to examples can be rewritten as:

```
# tests/test_foo.py
from granite.testcase import AutoMockMixin

from tests import BaseTestCase

class TestFoo(AutoMockMixin, BaseTestCase):
    bar = mock.patch('myproject.bar')
    baz = mock.patch('myproject.baz')
    biz = mock.patch('myproject.biz')
    boom = mock.patch('myproject.boom')

    def test_1(self):
        self.bar.return_value = 'baz'
        self.baz.return_value = True
        self.biz.return_value = 'asdf'
        self.AssertTrue(foo())
```

Ahhh, much nicer. We no longer have to worry about managing the state of mock patching setup and teardown as the mixin does all of that work for us!

---

**Note:** The `AutoMockMixin` not only works with `mock.patch`, but it also works with `mock.dict` and `mock.`

object.

## 1.5 granite package

### 1.5.1 Submodules

#### granite.environment module

Provides utilities for setting a proper environment for testing.

**class FileStat**(*st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*, *md5*)

Bases: tuple

Mimics the os.stat() stat result object except this also includes the md5 hash of the file.

**md5**
Alias for field number 10

**st_atime**
Alias for field number 7

**st_ctime**
Alias for field number 9

**st_dev**
Alias for field number 2

**st_gid**
Alias for field number 5

**st_ino**
Alias for field number 1

**st_mode**
Alias for field number 0

**st_mtime**
Alias for field number 8

**st_nlink**
Alias for field number 3

**st_size**
Alias for field number 6

**st_uid**
Alias for field number 4

**class Renderable**

Bases: object

An interface for objects that can be rendered.

**get_context**()
Gets the context (or scope) used when rendering this class's template.

By default, the context is set to all of the available attributes on this class. This means that if `self.foo` is set to `'bar'`, then the variable `{{ foo }}` in the template will be rendered as the string `bar`.

**Returns**

**the context used for rendering. A mapping of variable name** to variable value as to be used in the template.

**Return type** dict

**get_environment**(*template_directories*)
Gets the jinja2.Environment instance needed for loading the templates.

Subclasses should override this if they need more customized power.

**Parameters** **template_directories** (*list*) – a list of directories to search through

Returns:

**get_template**()
Gets the template used by this class for rendering.

Subclasses can override this method is updating the *template* and `template_dirs` attributes is not sufficient.

**Returns** the template used to render

**Return type** jinja2.Template

**render**()
Renders the template as found by get_template() using the context as found by get_context() and returns the rendered string.

**Returns** the result of rendering the template with the context

**Return type** str

**template = None**
The name of the template that instances of this class should use when rendering. This attribute is required and must be set by subclasses.

**template_dirs = None**
The template search path; a list of strings. When searching for a template by name, then the first file found on the path will be chosen. This attribute is required and must be set by subclasses.

**class RenderedFile**(*filename=''*)
Bases: *granite.environment.Renderable*

A file that will be rendered to disk.

This class represents a file that, upon rendering, will appear on disk. By default, any attribute (instance or class) will appear in the context of the template. E.g., having an attribute of `self.foo` will make the `foo` variable exist within the template.

This class provides a "content" variable and should serve as an area of the template that can be appended to on a per-test basis. Use the add_content() method in order to add more content to the template at the time of render. This allows for content to be added over a period of time either as subsequent calls in the same test, through the setUp, or setUp calls in multiple classes (via inheritance).

Subclasses can override the get_context() method in order to alter the context (variable scope) provided to the template on render. See the get_context() method's documentation for more details.

While this class can be used by itself (note that the instances attributes `template` and `template_dirs` must be set before rendering!) the intended use is to subclass this class and define class-level attributes for `template` and `template_dirs`. This makes it so that a base class can point to a common template directory (through the template_dirs attribute) and all subclasses of it can supply the `template` attribute in order to determine which template to choose.

**Parameters** **filename** (*str*) – the full path to where the file should exist on disk when rendered

Examples:

```
# tests/environment.py
# assume that the templates directory is:
# tests/templates
from granite.environment import RenderedFile
from granite.testcase import TestCase, TempProjectMixin


class MyRenderedFile(RenderedFile):
    template_dirs = [
        os.path.join(os.path.dirname(__file__), 'templates')]


# assume that tests/templates/template.py exists
# and looks something like this:
print('Hello!')
print(
    '''
     {{ content }}
    '''
)
print('This test is currently running: {{ id }}')

class PythonScript(MyRenderedFile):
    template = 'template.py'


class MyTestCase(TempProjectMixin, TestCase):
    def setUp(self):
        super(MyTestCase, self).setUp()
        self.script = PythonScript(
            os.path.join(self.temp_project.path, 'my_file.py'))
        self.script.add_content('My name is Aaron')

    def test_the_thing(self):
        # setting the `id` attribute provides the `id`
        # context variable in the template
        self.script.id = self.id()
        self.script.render()
        with open(self.script.full_name) as f:
            self.assertEqual(
                f.read(),
                """
                print('Hello!')
                print(
                    '''
                    My name is Aaron
                    '''
                )
                print('This test is currently running: test_the_thing')
                """
            )
```

**ADD_NEWLINE = True**

Determines whether a newline should be added at the end of the file or not. When generating C code to be compiled by the ARM/GCC compiler, this makes the compiler happy. Defaults to True. Set to False to

disable.

**DISABLE_ESCAPING = False**
By default, the value in the "content" template variable is escaped. This makes adding content for script/language files (via add_content()) much easier to read and maintain as script/language files usually interpret the special esacped characters differently. Set this attribute to True in order to disable this functionality.

**WRITE_MODE = 'w'**
When rendering the file, defaults to non-binary mode. Set this to 'wb' or something similar for different behavior when writing the rendered contents to a file.

**add_content**(*content*)
Adds a string to the "content" variable available to the template.

**Parameters** **content** (*str or List[str]*) – a single string or a list of strings to add to the content variable.

**filename = ''**
The name of the rendered file (without the path)

**full_name = ''**
the full path to the file

**get_context**()
Gets the context needed for rendering the file associated with this instance.

A context is simply the template's scope of variables and functions.

**Returns**

**the variable names and their values; the scope to appear in** the template

**Return type** dict

**path = ''**
The path to this rendered file (without the filename itself)

**render**()
Renders the template and writes the contents to disk to this instance's filename.

**class SimpleFile**(*filename=''*)
Bases: *granite.environment.RenderedFile*

Follows the renderable interface and allows for building up a file with add_content() then rendering and writing to disk at a later time.

This class doesn't provide any sort of templating functionality. It just makes it easier to incorporate simple file writing into a framework that expects a Renderable.

**render**()
Renders all of the contents to the filename given.

**class Snapshot**(*directory*)
Bases: object

A snapshot of the state of the given directory at the time called.

This will recursively traverse the given directory and note all of the files and directories within it. For the most part, a snapshot is useless by itself and is more useful when another snapshot is created and compared with the first.

For example:

```
some_dir = 'path/to/some/dir'
s1 = Snapshot(some_dir)
with open(os.path.join(some_dir, 'hello.txt'), 'w') as f:
    f.write('Hello, World!')
s2 = Snapshot(some_dir)

diff = s2 - s1

assert diff.added == ['hello.txt']
```

To see the difference between two snapshots, simply subtract one snapshot from the other. This creates a SnapshotDiff object with the attributes `added`, `removed`, `modified`, and `touched`. See the *SnapshotDiff* documentation for more information.

Note: this does not keep track of directory information, only files. Also Note: the paths stored in both the snapshot and in the diff are relative to the root of the snapshot directory.

**Parameters** **directory** (*str*) – the directory to take a snapshot of

**class SnapshotDiff**(*a*, *b*)

Bases: `object`

The difference between Snapshot `a` and Snapshot `b`.

A difference object will have four attributes.

**added**
a collection of files that are new in `a`, but not in `b`

**modified**
a collection of files whose contents have changed between `a` and `b`

**removed**
a collection of files that are in `b`, but no longer in `a`

**touched**
a collection of files whose timestamps have changed between `a` and `b`, but their contents have not changed.

**exception TemplateNotFoundError**

Bases: `Exception`

Raised when a template name was requested but not found in the available directories

**class TemporaryProject**(*path=''*, *preserve=False*)

Bases: `object`

An interface for interacting with a temporary directory.

A temp directory is created on instantiation and it is deleted (recursively) when this object is destroyed.

**Keyword Arguments**

- **path** (*str*) – path of a directory to use for the temporary directory if specified. If the directory already exists, it is recursively deleted and then created. Otherwise, if the directory doesn't exist, it (and any intermediate directories) are created.
- **preserve** (*bool*) – if set to `True`, this directory will not be destroyed. Useful for debugging tests.

**TEMP_PREFIX = 'gprj_'**

This is the prefix used for the new temp directory.

**abspath** (*filename*)
Get the absolute path to the filename found in the temp dir.

### Notes

- Always use forward slashes in paths.
- This method does not check if the path is valid. If the filename given doesn't exist, an exception is not raised.

**Parameters** **filename** (*str*) – the relative path to the file within this temp directory

**Returns** the absolute path to the file within the temp directory.

**Return type** str

**copy_project** (*dest*, *overwrite=False*, *symlinks=False*, *ignore=None*)
Allows for a copying the temp project to the destination given.

This provides test authors with the ability to preserve a tes environment at any point during a test.

By default, if the given destination is a directory that already exists, an error will be raised (shutil.copytree's error). Set the `overwrite` flag to `True` to overwrite an existing directory by first removing it and then copying the temp project.

**Parameters**

- **dest** (*str*) – the destination directory
- **overwrite** (*bool*) – if the directory exists, this will remove it first before copying
- **symlinks** (*bool*) – passed to shutil.copytree: should symlinks be traversed?
- **ignore** (*bool*) – ignore errors during copy?

**glob** (*pattern*, *start=''*, *absolute=False*)
Recursively searches through the temp dir for a filename that matches the given pattern and returns the first one that matches.

**Parameters**

- **pattern** (*str*) – the glob pattern to match the filenames against. Uses the fnmatch module's fnmatch() function to determine matches.
- **start** (*str*) – a directory relative to the root of the temp dir to start searching from.
- **absolute** (*bool*) – whether the returned path should be an absolute path; defaults to being relative to the temp project.

**Returns**

**the relative path to the first filename that matches pattern** unless the absolute flag is given. If a match is not found None is returned.

**Return type** str

**read** (*filename*, *mode='r'*)
Read the contents of the file found in the temp directory.

**Parameters**

- **filename** (*str*) – the path to the file in the temp dir.
- **mode** (*str*) – a valid mode to open(). defaults to *'r'*

**Returns** The contents of the file.

**remove** (*filename*)
Removes the filename found in the temp dir.

**Parameters** **filename** (*str*) – the relative path to the file

**snapshot** ()
Creates a snapshot of the current state of this temp dir.

**Returns** the snapshot.

**Return type** *Snapshot*

**teardown** ()
Provides a public way to delete the directory that this temp project manages.

This allows for the temporary directory to be cleaned up on demand.

Ignores all errors.

**touch** (*filename*)
Creates or updates timestamp on file given by filename.

**Parameters** **filename** (*str*) – the filename to touch

**write** (*filename*, *contents=''*, *mode='w'*, *dedent=True*)
Write the given contents to the file in the temp dir.

If the file or the directories to the file do not exist, they will be created.

If the file already exists, its contents will be overwritten with the new contents unless mode is set to some variant of append: (`a`, `ab`).

Specify the dedent flag to automatically call `textwrap.dedent` on the contents before writing. This is especially useful when writing contents that depend on whitespace being exact (e.g. writing a Python script). This defaults to `True` except when the mode contains `'b'`

**Parameters**

- **filename** (*str*) – the relative path to a file in the temp dir
- **contents** (*any*) – any data to write to the file
- **mode** (*str*) – a valid open() mode
- **dedent** (*bool*) – automatically dedent the contents (default: `True`)

### granite.exceptions module

Exceptions.

**exception GraniteException**
Bases: `Exception`

Base Exception class for all exceptions.

**exception MisconfiguredError**
Bases: *granite.exceptions.GraniteException*

A class or Mixin is misconfigured.

### granite.io module

Provides utilities for handling I/O during test excution.

**capture_output**()
Captures both stdout and stderr and stores into a string buffer.

Example:

```
import sys

with capture_output() as (stdout, stderr):
    stdout = 'This is stdout'
    stderr = 'This is stderr'

    print(stdout)
    assert stdout.getvalue() == stdout.strip()

    sys.stderr.write(stderr)
    assert stderr.getvalue() == stderr
```

**capture_stderr**()
Captures stderr and stores in a string buffer.

Example:

```
with capture_stderr() as stderr:
    stderr = 'Hello, World!
    print(stderr)
    assert stderr.getvalue() == stderr.strip()
```

The yielded value is StringIO buffer. See its documentation for more details.

**capture_stdout**()
Captures stdout and stores in a string buffer.

Example:

```
with capture_stdout() as stdout:
    stdout = 'Hello, World!
    print(stdout)
    assert stdout.getvalue() == stdout.strip()
```

The yielded value is a StringIO buffer. See its documentation for more details.

### granite.sphinx module

Provides the main DocBuilder class for easily interfacing with sphinx to build the project's documentation.

**class DocBuilder**
Bases: object

Provides a simple interface for consistently building documentation.

By default, this class will generate the API docs using sphinx's apidoc script, overwriting any existing API .rst files (provided the API_OUTPUT_DIR attribute has been set. Afterward, sphinx itself is run on the project.

Projects are expected to create a file named build_docs.py (or something similar) within the docs directory of your project that contains a subclass of this class, instantiate it, then run its build method. An example:

```
import os
import sys

THIS_DIR = os.path.dirname(__file__)
sys.path.insert(0, os.path.join(THIS_DIR, '..', '..', 'src'))
from granite.sphinx import DocBuilder


def mkpath(*parts):
    """Makes an abspath from THIS_DIR"""
    return os.path.normpath(os.path.join(THIS_DIR, *parts))


class Builder(DocBuilder):
    # input path
    SOURCE_DIR = mkpath('source')
    # output path
    BUILD_DIR = mkpath('build')
    # remove this line if auto-api generation is not desired
    API_OUTPUT_DIR = mkpath('source', 'api')
    # the path to the python package
    PROJECT_DIR = mkpath('..', 'src', 'granite')

    FILES_TO_CLEAN = [
        API_OUTPUT_DIR,
        BUILD_DIR,
    ]


if __name__ == '__main__':
    Builder().build()
```

Each of the following attributes should be defined in order to configure DcoBuilder. Each attribute that is a path should be an absolute path.

**SOURCE_DIR**
*str* – the path to the project's source directory

**BUILD_DIR**
*str* – the path to the documentation output

**API_OUTPUT_DIR**
*str* – defining this will automatically call sphinx-apidoc on the project directory. See the generate_api_docs() method for more information. *Optional*

**PROJECT_DIR**
*str* – path to the directory containing Python project.

**FILES_TO_CLEAN**
*List[str]* – a list of files to clean when `--clean` is passed on the command line. *Optional*

**API_EXCLUDE_DIRS**
*List[str]* – a list of paths relative to PROJECT_DIR to exclude from the api-doc generation.

**API_EXCLUDE_DIRS = []**

**API_OUTPUT_DIR = ''**

**BUILD_DIR = ''**

**FILES_TO_CLEAN = []**

**PROJECT_DIR = ''**

**SOURCE_DIR = ''**

**add_argument**(**args*, ***kwargs*)
Add an argument to the ArgumentParser in `self.parser`.

Takes in the same args and kwargs as the `ArgumentParser.add_argument()` method. Use this method in order to add custom flags to the argument parsing. The argv flags are parsed in the `build()` method. This sets the parsed args into `self.args` and the leftover unknown flags into `self.argv`.

**build**(*argv=None*)
Gathers all command line arguments and then builds the docs.

This performs command line parsing and stores the known flags (those added with `self.add_argument()`) into `self.args` and all leftover unknown args into `self.argv` (see `argparse.ArgumentParser.parse_known_args()` for more information on the types of each).

After argparsing the following three methods are called in this order:

- *pre_build_hook()*
- *generate_documentation()*
- *post_build_hook()*

Override the pre and post build hooks in order to add custom checks or other functionality.

**Parameters** **argv** (*List[str]*) – command line flags to parse; defaults to sys.argv

**generate_api_docs**()
Generates the API documentation for all of the packages/modules/classes/functions.

Sphinx doesn't automatically generate the documentation for the api. This calls sphinx-apidoc which will create the API .rst files and dump them in the source directory. It is expected that one of the TOC directives calls out to the created API directory.

*Note:* if the attribute `API_OUTPUT_DIR` is not set on this class, then this method does nothing.

**generate_documentation**()
Runs sphinx on the project using the default conf.py file in the source directory.

**post_build_hook**()
This is called immediately after all documentation has been generated.

Override this method for any custom functionality.

**pre_build_hook**()
This is called after all arguments have been collected, but before sphinx is called.

Override this method for any custom functionality.

**safe_delete**(*filename*)
Tries to delete `filename` and ignores any error that is raised.

**setup_default_arguments**()
This method adds some default command line parameters.

Current, the default flags are:

- **--clean: Cleans all files found in the FILES_TO_CLEAN** list.

**try_clean**()
Attempts to clean all of the files found in `self.FILES_TO_CLEAN`.

Ignores all errors.

**exception RequiredAttributeException**(*attribute*, *class_*, *description*)
Bases: *granite.exceptions.GraniteException*

Raised when an attribute on the DocBuilder class has not been defined

### granite.testcase module

Extends unittest.

**exception AssetDirectoryNotSet**
Bases: *granite.exceptions.GraniteException*

Raised when the ASSETS_DIR attribute is not set.

**class AssetMixin**(**args*, ***kwargs*)
Bases: *granite.testcase.TestCaseMixin*

Provides support for accessing assets needed by tests.

**ASSET_DIR = None**

**get_asset_filename**(**parts*)
Gets the absolute filename of the asset file given by `filename` and `*parts` relative to the asset directory.

Treat this input like that of `os.path.join`.

Assume the absolute path to the `ASSETS_DIR` is `/path/to/assets/` and the assets directory contains `some_file.txt`, then:

```
>>> self.get_asset_filename('some_file.txt')
'path/to/assets/some_file.txt'
```

**Raises** *AssetNotFound* – when the filename to search for is not found on disk.

**Returns** the absolute path to the asset file.

**Return type** str

**read_asset_file**(*filename*, **parts*, ***kwargs*)
Gets the contents of the given asset filename.

Internally this calls `get_asset_filename()`

Pass the optional keyword argument `mode` in oder to set the file mode. For example use `mode='rb'` to read in bytes.

**Parameters**

- **filename** –
- ***parts** –
- ****kwargs** –

**Returns** The contents of the file using the given read `mode`.

**Raises** *AssetNotFound* – when the filename to search for is not found on disk.

**exception AssetNotFound**
Bases: *granite.exceptions.GraniteException*

Raised when an asset file requested is not found.

**class AutoMockMixin**
Bases: *granite.testcase.TestCaseMixin*

Helps prevent the boilerplate of having mock patcher and object setup and teardown logic for every function needing to be mocked.

**setUp**()
Attaches all mock patchers to this instance as mocked attributes

**class TemporaryProjectMixin**(**args*, ***kwargs*)
Bases: *granite.testcase.TestCaseMixin*

Provides support for temporary project (directory) creation on a per-test basis.

In order to use this mixin, the base TestCase class should inherit this mixin. This provides a new attribute named `temp_project` which is an instance of TemporaryProject.

See the *TemporaryProject* class for all of its methods for how to manipulate the created temp project.

Example:

```
import os

from granite.testcase import TestCase, TemporaryProjectMixin

class TestSomeThing(TemporaryProjectMixin, TestCase):
    def test_some_thing(self):
        # a new temporary directory has already been created by
        # this point. let's create a new file and add some contents:
        self.temp_project.write('some_file', 'Ohai :)')
        # get the temp_project's path by its .path attribute.
        # This proves that the file was created and exists on disk:
        self.assertTrue(os.path.exists(
            os.path.join(self.temp_project.path, 'some_file')))
        # read the contents of a file relative to the temp project's
        # directory:
        contents = self.temp_project.read('some_file')
        self.assertEqual(contents, 'Ohai :)')
```

**ENABLE_PRESERVE = False**
A flag indicating whether the temp project should be preserved after the temp project object is destroyed. If True, the directory will still exist allowing a user to view the state of the directory after a test has run. This works in tandem with the `PRESERVE_DIR` class attribute.

**PRESERVE_DIR = None**
Sets where the preserved path should be dumped too. This overrides the `TMP_DIR` when `ENABLE_PRESERVE` is set to True.

**TMP_DIR = None**
Allows for setting the temp directory. Defaults to `None` which will use Python's `tempfile.mkdtemp` to make the temp directory.

**TemporaryProjectClass**
alias of *granite.environment.TemporaryProject*

**assert_in_temp_file**(*substring*, *filename*, *msg=''*, *mode='r'*, *not_in=False*)
Asserts that the given contents are found in the file in the temp project.

**Parameters**

- **substring** (*str*) – the substring to look for in the file's contents
- **filename** (*str*) – the name of the file relative to the temp project

- **msg** (*str*) – the message to output in the event of a failure.
- **mode** (*str*) – the mode to open the file with. defaults to 'r'
- **not_in** (*bool*) – asserts that the contents are not in the file

**assert_not_in_temp_file**(*substring*, *filename*, *msg=''*, *mode='r'*)
Asserts that the given contents are not found in the file in the temp project.

**Parameters**

- **substring** (*str*) – the substring to look for in the file's contents
- **filename** (*str*) – the name of the file relative to the temp project
- **msg** (*str*) – the message to output in the event of a failure.
- **mode** (*str*) – the mode to open the file with. defaults to 'r'

**assert_temp_path_exists**(*path='.'*, *msg=''*)
Asserts that the path given exists relative to the root of the temp project.

**Parameters**

- **path** (*str*) – the string of the path relative to the root of the temp directory.
- **msg** (*str*) – a custom string to show in the event that this assertion fails.

**setUp**()
Sets up the temporary project on test startup.

**tearDown**()
Deletes the temp project.

**class TestCase**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Extends the Standard Library's TestCase class.

**assert_exists**(*path*, *msg=''*)
Asserts that the given path exists on disk.

This function acts like os.path.join() in that it can accept multiple arguments all of which will be joined together before checking for existence.

**Parameters**

- **path** (*str*) – the root path to check for
- **msg** (*str*) – the message to show if the assertion fails.

**assert_iterable_of_type**(*iterable*, *types*, *msg=''*)
Assert that all items in the given iterable are of the given type(s).

**Parameters**

- **iterable** (*Iterable*) – the items to check
- **types** (*Union[object, tuple]*) – valid type input for isinstance.
- **msg** (*str*) – optional message if the assertion fails

**assert_length**(*sized*, *length*, *msg=''*)
Asserts that the *sized* object has *length* number of items.

**Parameters**

- **sized** (*Sized*) – any object that implements the __len__() method.

- **length** (*int*) – the number of items that *sized* should contain
- **msg** (*Optional[str]*) – a message to display in the event that this
- **fails.** (*assert*) –

Example:

```
from granite.testcase import TestCase

class MyTestCase(TestCase):
    def test_that_contents_are_correct_length(self):
        contents = [1, 2, 3]
        self.assert_length(
            contents, 3, msg='Some how, the length is not 3???')
```

**class TestCaseMixin**

Bases: object

Base TestCase Mixin class. All Mixins should inherit this class.

### granite.utils module

Utility for internal library use.

**cached_property**(*fn*)

Converts a class's method into property and cache's the getter value.

Simple decorator a method with this function and it will be converted into a property (attribute access) and the calculation to retrieve the value will be cached so that the initial call performs the calculation, but subsequent calls will use the cache.

**get_mock_patcher_types**()

Gets the Classes of the mock.patcher functions.

We're using this for the automatic mocking mixin in order to determine if a class-level attribute is a patcher instance.

**Returns** a unique list of the patcher types used by mock.

**Return type** tuple

**path_as_key**(*path*, *relative_to=None*)

Converts a path to a unique key.

Paths can take on several unique forms but all describe the same node on the file system. This function will take a path and produce a key that is unique such that several paths that point to the same node on the file system will all be converted to the same path.

This is useful for converting file paths to keys for a dictionary.

Note: all paths will be separated by forward slashes.

**Parameters**

- **path** (*str*) – the path to convert.
- **relative_to** (*str*) – make the key relative to this path

**Returns** the unique key from the path

**Return type** str

Examples:

```
>>> path_as_key('./file.txt') == path_as_key('file.txt')
True
```

### 1.5.2 Module contents

The Granite library

Want to see an example of each of the above items? Have a look at the tests!

# Python Module Index

## g

# Index

## R



## S



## T



## W