
Testpath Documentation

Release 0.4.2

Jupyter Development Team

Oct 02, 2018

Contents

1	Assertion functions for the filesystem	3
1.1	Unix specific	4
2	Mocking system commands	5
3	Modifying environment variables	7
4	Utilities for temporary directories	9
5	Indices and tables	11
	Python Module Index	13

Testpath is a collection of utilities for testing code which uses and manipulates the filesystem and system commands.

Install it with:

```
pip install testpath
```

Contents:

Assertion functions for the filesystem

These functions make it easy to check the state of files and directories. When the assertion is not true, they provide informative error messages.

`testpath.assert_path_exists` (*path*, *msg=None*)

Assert that something exists at the given path.

`testpath.assert_not_path_exists` (*path*, *msg=None*)

Assert that nothing exists at the given path.

`testpath.assert_isfile` (*path*, *follow_symlinks=True*, *msg=None*)

Assert that path exists and is a regular file.

With `follow_symlinks=True`, the default, this will pass if path is a symlink to a regular file. With `follow_symlinks=False`, it will fail in that case.

`testpath.assert_not_isfile` (*path*, *follow_symlinks=True*, *msg=None*)

Assert that path exists but is not a regular file.

With `follow_symlinks=True`, the default, this will fail if path is a symlink to a regular file. With `follow_symlinks=False`, it will pass in that case.

`testpath.assert_isdir` (*path*, *follow_symlinks=True*, *msg=None*)

Assert that path exists and is a directory.

With `follow_symlinks=True`, the default, this will pass if path is a symlink to a directory. With `follow_symlinks=False`, it will fail in that case.

`testpath.assert_not_isdir` (*path*, *follow_symlinks=True*, *msg=None*)

Assert that path exists but is not a directory.

With `follow_symlinks=True`, the default, this will fail if path is a symlink to a directory. With `follow_symlinks=False`, it will pass in that case.

`testpath.assert_islink` (*path*, *to=None*, *msg=None*)

Assert that path exists and is a symlink.

If `to` is specified, also check that it is the target of the symlink.

`testpath.assert_not_islink` (*path*, *msg=None*)
Assert that path exists but is not a symlink.

1.1 Unix specific

New in version 0.4.

These additional functions test for special Unix filesystem objects: named pipes and Unix domain sockets. The functions can be used on all platforms, but these types of objects do not exist on Windows.

`testpath.assert_ispipe` (*path*, *follow_symlinks=True*, *msg=None*)
Assert that path exists and is a named pipe (FIFO).

With `follow_symlinks=True`, the default, this will pass if path is a symlink to a named pipe. With `follow_symlinks=False`, it will fail in that case.

`testpath.assert_not_ispipe` (*path*, *follow_symlinks=True*, *msg=None*)
Assert that path exists but is not a named pipe (FIFO).

With `follow_symlinks=True`, the default, this will fail if path is a symlink to a named pipe. With `follow_symlinks=False`, it will pass in that case.

`testpath.assert_issocket` (*path*, *follow_symlinks=True*, *msg=None*)
Assert that path exists and is a Unix domain socket.

With `follow_symlinks=True`, the default, this will pass if path is a symlink to a Unix domain socket. With `follow_symlinks=False`, it will fail in that case.

`testpath.assert_not_issocket` (*path*, *follow_symlinks=True*, *msg=None*)
Assert that path exists but is not a Unix domain socket.

With `follow_symlinks=True`, the default, this will fail if path is a symlink to a Unix domain socket. With `follow_symlinks=False`, it will pass in that case.

Mocking system commands

Mocking is a technique to replace parts of a system with interfaces that don't do anything, but which your tests can check whether and how they were called. The `unittest.mock` module in Python 3 lets you mock Python functions and classes. These tools let you mock external commands.

Commands are mocked by creating a real file in a temporary directory which is added to the `PATH` environment variable, not by replacing Python functions. So if you mock `foo`, and your Python code runs a shell script which calls `foo`, it will be the mocked command that it runs.

By default, mocked commands record each call made to them, so that your test can check these. Using the `MockCommand` API, you can mock a command to do something else.

Note: These tools work by changing global state. They're not safe to use if commands may be called from multiple threads or coroutines.

`testpath.assert_calls` (*cmd*, *args=None*)

Assert that a block of code runs the given command.

If *args* is passed, also check that it was called at least once with the given arguments (not including the command name).

Use as a context manager, e.g.:

```
with assert_calls('git'):
    some_function_wrapping_git()

with assert_calls('git', ['add', myfile]):
    some_other_function()
```

`class testpath.MockCommand` (*name*, *content=None*)

Context manager to mock a system command.

The mock command will be written to a directory at the front of `$PATH`, taking precedence over any existing command with the same name.

By specifying content as a string, you can determine what running the command will do. The default content records each time the command is called and exits: you can access these records with `mockcmd.get_calls()`.

On Windows, the specified content will be run by the Python interpreter in use. On Unix, it should start with a shebang (`#!/path/to/interpreter`).

get_calls()

Get a list of calls made to this mocked command.

This relies on the default script content, so it will return an empty list if you specified a different content parameter.

For each time the command was run, the list will contain a dictionary with keys `argv`, `env` and `cwd`.

Modifying environment variables

These functions allow you to temporarily modify the environment variables, which is often useful for testing code that calls other processes.

`testpath.modified_env` (*changes*, *snapshot=True*)

Temporarily modify environment variables.

Specify the changes as a dictionary mapping names to new values, using `None` as the value for names that should be deleted.

Example use:

```
with modified_env({'SHELL': 'bash', 'PYTHONPATH': None}):  
    ...
```

When the context exits, there are two possible ways to restore the environment. If *snapshot* is `True`, the default, it will reset the whole environment to its state when the context was entered. If *snapshot* is `False`, it will restore only the specific variables it modified, leaving any changes made to other environment variables in the context.

`testpath.temporary_env` (*newenv*)

Completely replace the environment variables with the specified dict.

Use as a context manager:

```
with temporary_env({'PATH': my_path}):  
    ...
```

`testpath.make_env_restorer` ()

Snapshot the current environment, return a function to restore that.

This is intended to produce cleanup functions for tests. For example, using the `unittest.TestCase` API:

```
def setUp(self):  
    self.addCleanup(testpath.make_env_restorer())
```

Any changes a test makes to the environment variables will be wiped out before the next test is run.

Utilities for temporary directories

This module exposes `tempfile.TemporaryDirectory()`, with a backported copy so that it can be used on Python 2. In addition, it contains:

```
class testpath.tempdir.NamedFileInTemporaryDirectory (filename, mode='w+b',  
                                                    bufsize=-1, **kwds)
```

Open a file named *filename* in a temporary directory.

This context manager is preferred over `tempfile.NamedTemporaryFile` when one needs to reopen the file, because on Windows only one handle on a file can be open at a time. You can close the returned handle explicitly inside the context without deleting the file, and the context manager will delete the whole directory when it exits.

Arguments *mode* and *bufsize* are passed to *open*. Rest of the arguments are passed to *TemporaryDirectory*.

Usage example:

```
with NamedFileInTemporaryDirectory('myfile', 'wb') as f:  
    f.write('stuff')  
    f.close()  
    # You can now pass f.name to things that will re-open the file
```

```
class testpath.tempdir.TemporaryWorkingDirectory (suffix=None, prefix=None,  
                                                dir=None)
```

Creates a temporary directory and sets the cwd to that directory. Automatically reverts to previous cwd upon cleanup.

Usage example:

```
with TemporaryWorkingDirectory() as tmpdir:  
    ...
```


CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`

t

testpath, 3

testpath.tempdir, 9

A

`assert_calls()` (in module `testpath`), 5
`assert_isdir()` (in module `testpath`), 3
`assert_isfile()` (in module `testpath`), 3
`assert_islink()` (in module `testpath`), 3
`assert_ispipe()` (in module `testpath`), 4
`assert_issocket()` (in module `testpath`), 4
`assert_not_isdir()` (in module `testpath`), 3
`assert_not_isfile()` (in module `testpath`), 3
`assert_not_islink()` (in module `testpath`), 3
`assert_not_ispipe()` (in module `testpath`), 4
`assert_not_issocket()` (in module `testpath`), 4
`assert_not_path_exists()` (in module `testpath`), 3
`assert_path_exists()` (in module `testpath`), 3

E

environment variable
 `PATH`, 5

G

`get_calls()` (`testpath.MockCommand` method), 6

M

`make_env_restorer()` (in module `testpath`), 7
`MockCommand` (class in `testpath`), 5
`modified_env()` (in module `testpath`), 7

N

`NamedFileInTemporaryDirectory` (class in `testpath.tempdir`), 9

P

`PATH`, 5

T

`temporary_env()` (in module `testpath`), 7
`TemporaryWorkingDirectory` (class in `testpath.tempdir`),
 9
`testpath` (module), 3
`testpath.tempdir` (module), 9