Testpath Documentation

Release 0.4.4

Jupyter Development Team

Feb 23, 2022

Contents

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:

CHAPTER 1

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.

CHAPTER 2

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. The tools described here 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 `git`, and your Python code runs a shell script which calls `git`, 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 change what a mocked command does.

Note: Mocking a command affects all running threads or coroutines in a program. There's no way to mock a command for only the current thread/coroutine, because it uses environment variables, which are global.

`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*, *python=''*)

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.

The *python* parameter accepts a string of code for the command to run, in addition to the default behaviour of recording calls to the command. This will run with the same Python interpreter as the calling code, but in a new process.

The *content* parameter gives extra control, by providing a script which will run with no additions. On Unix, it should start with a shebang (e.g. `#!/usr/bin/env python`) specifying the interpreter. On Windows, it will always be run by the same Python interpreter as the calling code. Calls to the command will not be recorded when content is specified.

classmethod fixed_output(*name*, *stdout=''*, *stderr=''*, *exit_status=0*)

Make a mock command, producing fixed output when it is run:

```
t = 'Sat 24 Apr 17:11:58 BST 2021\n'
with MockCommand.fixed_output('date', t) as mock_date:
    ...
```

The stdout & stderr strings will be written to the respective streams, and the process will exit with the specified numeric status (the default of 0 indicates success).

This works with the recording mechanism, so you can check what arguments this command was called with.

get_calls()

Get a list of calls made to this mocked command.

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

This won't work if you used the *content* parameter to alter what the mocked command does.

assert_called(*args=None*)

Assert that the mock command has been called at least once.

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

```
with MockCommand('rsync') as mock_rsync:
    function_to_test()

mock_rsync.assert_called(['/var/log', 'backup-server:logs'])
```

This won't work if you used the *content* parameter to alter what the mocked command does.

CHAPTER 3

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.

CHAPTER 4

Utilities for temporary directories

The *testpath.tempdir* module contains a couple of utilities for working with temporary directories:

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

Release notes

5.1 0.6

February 2022

- Removed some code that's unused since dropping Python 2 support.
- Relax the version constraint for the `flit_core` build requirement.

5.2 0.5

May 2021

- Easier ways to use *`MockCommand`* to customise mocked commands, including `python=` to specify extra code to run, *`fixed_output()`*, and *`assert_called()`*.
- Command mocking will use `os.defpath` as the initial PATH if the PATH environment variable is not set.

CHAPTER 6

Indices and tables

- genindex
- search

Python Module Index

t

Index