pyfakefs Documentation

Release 3.7.2

John McGehee

Oct 08, 2022

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

pyfakefs implements a fake file system that mocks the Python file system modules. Using pyfakefs, your tests operate on a fake file system in memory without touching the real disk. The software under test requires no modification to work with pyfakefs.

pyfakefs works with CPython 2.7, 3.4 and above, on Linux, Windows and OSX (MacOS), and with PyPy2 and PyPy3. Note that this is the last major release that still supports Python 2.7/PyPy2 and Python 3.4.

pyfakefs works with PyTest version 2.8.6 or above.

1.1 Installation

pyfakefs is available on PyPi. The latest released version can be installed from pypi:

```
pip install pyfakefs
```

The latest master can be installed from the GitHub sources:

```
pip install git+https://github.com/jmcgeheeiv/pyfakefs
```

1.2 Limitations

pyfakefs will not work with Python libraries that use C libraries to access the file system, because it cannot patch the underlying C libraries' file access functions.

Depending on the kind of import statements used, pyfakefs may not patch the file system modules automatically. See *Customizing Patcher and TestCase* for more information and ways to work around this.

pyfakefs is only tested with CPython and newest PyPy versions, other Python implementations will probably not work.

Differences in the behavior in different Linux distributions or different MacOS or Windows versions may not be reflected in the implementation, as well as some OS-specific low-level file system behavior. The systems used for automatic tests in Travis.CI and AppVeyor are considered as reference systems.

1.3 History

pyfakefs was initially developed at Google by Mike Bland as a modest fake implementation of core Python modules. It was introduced to all of Google in September 2006. Since then, it has been enhanced to extend its functionality and usefulness. At last count, pyfakefs was used in over 2,000 Python tests at Google.

Google released pyfakefs to the public in 2011 as Google Code project pyfakefs:

- Fork jmcgeheeiv-pyfakefs added direct support for unittest and doctest as described in *Automatically find and patch file functions and modules*
- Fork shiffdane-jmcgeheeiv-pyfakefs added further corrections

After the shutdown of Google Code was announced, John McGehee merged all three Google Code projects together on GitHub where an enthusiastic community actively maintains and extends pyfakefs.

CHAPTER

TWO

USAGE

2.1 Test Scenarios

There are several approaches to implementing tests using `pyfakefs`.

2.1.1 Patch using fake_filesystem_unittest

If you are using the Python `unittest` package, the easiest approach is to use test classes derived from `fake_filesystem_unittest.TestCase`.

If you call `setUpPyfakefs()` in your `setUp()`, `pyfakefs` will automatically find all real file functions and modules, and stub these out with the fake file system functions and modules:

```
from pyfakefs.fake_filesystem_unittest import TestCase

class ExampleTestCase(TestCase):
    def setUp(self):
        self.setUpPyfakefs()

    def test_create_file(self):
        file_path = '/test/file.txt'
        self.assertFalse(os.path.exists(file_path))
        self.fs.create_file(file_path)
        self.assertTrue(os.path.exists(file_path))
```

The usage is explained in more detail in *Automatically find and patch file functions and modules* and demonstrated in the files `example.py` and `example_test.py`.

2.1.2 Patch using the PyTest plugin

If you use PyTest, you will be interested in the PyTest plugin in `pyfakefs`. This automatically patches all file system functions and modules in a similar manner as described above.

The PyTest plugin provides the `fs` fixture for use in your test. For example:

```
def my_fakefs_test(fs):
    # "fs" is the reference to the fake file system
    fs.create_file('/var/data/xx1.txt')
    assert os.path.exists('/var/data/xx1.txt')
```

2.1.3 Patch using fake_filesystem_unittest.Patcher

If you are using other means of testing like nose, you can do the patching using `fake_filesystem_unittest.Patcher` - the class doing the actual work of replacing the filesystem modules with the fake modules in the first two approaches.

The easiest way is to just use `Patcher` as a context manager:

```
from pyfakefs.fake_filesystem_unittest import Patcher

with Patcher() as patcher:
    # access the fake_filesystem object via patcher.fs
    patcher.fs.create_file('/foo/bar', contents='test')

    # the following code works on the fake filesystem
    with open('/foo/bar') as f:
        contents = f.read()
```

You can also initialize `Patcher` manually:

```
from pyfakefs.fake_filesystem_unittest import Patcher

patcher = Patcher()
patcher.setUp()     # called in the initialization code
...
patcher.tearDown()  # somewhere in the cleanup code
```

2.1.4 Patch using unittest.mock (deprecated)

You can also use `mock.patch()` to patch the modules manually. This approach will only work for the directly imported modules, therefore it is not suited for testing larger code bases. As the other approaches are more convenient, this one is considered deprecated and will not be described in detail.

2.2 Customizing Patcher and TestCase

Both `fake_filesystem_unittest.Patcher` and `fake_filesystem_unittest.TestCase` provide a few arguments to handle cases where patching does not work out of the box. In case of `fake_filesystem_unittest.TestCase`, these arguments can either be set in the TestCase instance initialization, or passed to `setUpPyfakefs()`.

Note: If you need these arguments in `PyTest`, you must use `Patcher` directly instead of the `fs` fixture. Alternatively, you can add your own fixture with the needed parameters.

An example for both approaches can be found in pytest_fixture_test.py with the example fixture in conftest.py. We advice to use this example fixture code as a template for your customized pytest plugins.

2.2.1 modules_to_reload

Pyfakefs patches modules that are imported before starting the test by finding and replacing file system modules in all loaded modules at test initialization time. This allows to automatically patch file system related modules that are:

- imported directly, for example:

```
import os
import pathlib.Path
```

- imported as another name:

```
import os as my_os
```

- imported using one of these two specially handled statements:

```
from os import path
from pathlib import Path
```

Additionally, functions from file system related modules are patched automatically if imported like:

```
from os.path import exists
from os import stat
```

This also works if importing the functions as another name:

```
from os.path import exists as my_exists
from io import open as io_open
from builtins import open as bltn_open
```

There are a few cases where automatic patching does not work. We know of two specific cases where this is the case:

- initializing global variables:

```
from pathlib import Path

path = Path("/example_home")
```

In this case, `path` will hold the real file system path inside the test.

- initializing a default argument:

```
import os

def check_if_exists(filepath, file_exists=os.path.exists):
    return file_exists(filepath)
```

Here, `file_exists` will not be patched in the test.

To get these cases to work as expected under test, the respective modules containing the code shall be added to the `modules_to_reload` argument (a module list). The passed modules will be reloaded, thus allowing pyfakefs to patch them dynamically. All modules loaded after the initial patching described above will be patched using this second mechanism.

Given that the example code shown above is located in the file `example/sut.py`, the following code will work:

```
# example using unittest
class ReloadModuleTest(fake_filesystem_unittest.TestCase):
    def setUp(self):
        self.setUpPyfakefs(modules_to_reload=[example.sut])

    def test_path_exists(self):
        file_path = '/foo/bar'
        self.fs.create_dir(file_path)
        self.assertTrue(example.sut.check_if_exists(file_path))

# example using Patcher
def test_path_exists():
    with Patcher() as patcher:
      file_path = '/foo/bar'
      patcher.fs.create_dir(file_path)
      assert example.sut.check_if_exists(file_path)
```

Example using pytest:

```
# conftest.py
...
from example import sut

@pytest.fixture
def fs_reload_sut():
    patcher = Patcher(modules_to_reload=[sut])
    patcher.setUp()
    linecache.open = patcher.original_open
    tokenize._builtin_open = patcher.original_open
    yield patcher.fs
    patcher.tearDown()

# test_code.py
...
def test_path_exists(fs_reload_sut):
    file_path = '/foo/bar'
    fs_reload_sut.create_dir(file_path)
    assert example.sut.check_if_exists(file_path)
```

2.2.2 modules_to_patch

Sometimes there are file system modules in other packages that are not patched in standard pyfakefs. To allow patching such modules, `modules_to_patch` can be used by adding a fake module implementation for a module name. The argument is a dictionary of fake modules mapped to the names to be faked.

This mechanism is used in pyfakefs itself to patch the external modules *pathlib2* and *scandir* if present, and the following example shows how to fake a module in Django that uses OS file system functions:

```
class FakeLocks(object):
    """django.core.files.locks uses low level OS functions, fake it."""
    _locks_module = django.core.files.locks
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, fs):
        """Each fake module expects the fake file system as an __init__
        parameter."""
        # fs represents the fake filesystem; for a real example, it can be
        # saved here and used in the implementation
        pass

    @staticmethod
    def lock(f, flags):
        return True

    @staticmethod
    def unlock(f):
        return True

    def __getattr__(self, name):
        return getattr(self._locks_module, name)

...
# test code using Patcher
with Patcher(modules_to_patch={'django.core.files.locks': FakeLocks}):
    test_django_stuff()

# test code using unittest
class TestUsingDjango(fake_filesystem_unittest.TestCase):
    def setUp(self):
        self.setUpPyfakefs(modules_to_patch={'django.core.files.locks': FakeLocks})

    def test_django_stuff()
        ...
```

2.2.3 additional_skip_names

This may be used to add modules that shall not be patched. This is mostly used to avoid patching the Python file system modules themselves, but may be helpful in some special situations, for example if a testrunner is accessing the file system after test setup. A known case is erratic behavior if running a debug session in PyCharm with Python 2.7, which can be avoided by adding the offending module to `additional_skip_names`:

```
with Patcher(additional_skip_names=['pydevd']) as patcher:
    patcher.fs.create_file('foo')
```

Alternatively to the module names, the modules themselves may be used:

```
import pydevd

with Patcher(additional_skip_names=[pydevd]) as patcher:
    patcher.fs.create_file('foo')
```

There is also the global variable `Patcher.SKIPNAMES` that can be extended for that purpose, though this seldom shall be needed (except for own pytest plugins, as shown in the example mentioned above).

2.2.4 allow_root_user

This is `True` by default, meaning that the user is considered a root user if the real user is a root user (e.g. has the user ID 0). If you want to run your tests as a non-root user regardless of the actual user rights, you may want to set this to `False`.

2.3 Using convenience methods

While `pyfakefs` can be used just with the standard Python file system functions, there are few convenience methods in `fake_filesystem` that can help you setting up your tests. The methods can be accessed via the `fake_filesystem` instance in your tests: `Patcher.fs`, the `fs` fixture in PyTest, or `TestCase.fs`.

2.3.1 File creation helpers

To create files, directories or symlinks together with all the directories in the path, you may use `create_file()`, `create_dir()` and `create_symlink()`, respectively.

`create_file()` also allows you to set the file mode and the file contents together with the encoding if needed. Alternatively, you can define a file size without contents - in this case, you will not be able to perform standard IO operations on the file (may be used to "fill up" the file system with large files).

```
from pyfakefs.fake_filesystem_unittest import TestCase

class ExampleTestCase(TestCase):
    def setUp(self):
        self.setUpPyfakefs()

    def test_create_file(self):
        file_path = '/foo/bar/test.txt'
        self.fs.create_file(file_path, contents = 'test')
        with open(file_path) as f:
            self.assertEqual('test', f.read())
```

`create_dir()` behaves like `os.makedirs()`, but can also be used in Python 2.

2.3.2 Access to files in the real file system

If you want to have read access to real files or directories, you can map them into the fake file system using `add_real_file()`, `add_real_directory()`, `add_real_symlink()` and `add_real_paths()`. They take a file path, a directory path, a symlink path, or a list of paths, respectively, and make them accessible from the fake file system. By default, the contents of the mapped files and directories are read only on demand, so that mapping them is relatively cheap. The access to the files is by default read-only, but even if you add them using `read_only=False`, the files are written only in the fake system (e.g. in memory). The real files are never changed.

`add_real_file()`, `add_real_directory()` and `add_real_symlink()` also allow you to map a file or a directory tree into another location in the fake filesystem via the argument `target_path`.

```
from pyfakefs.fake_filesystem_unittest import TestCase

class ExampleTestCase(TestCase):
```

(continues on next page)

(continued from previous page)

```
    fixture_path = os.path.join(os.path.dirname(__file__), 'fixtures')
    def setUp(self):
        self.setUpPyfakefs()
        # make the file accessible in the fake file system
        self.fs.add_real_directory(self.fixture_path)

    def test_using_fixture1(self):
        with open(os.path.join(self.fixture_path, 'fixture1.txt') as f:
            # file contents are copied to the fake file system
            # only at this point
            contents = f.read()
```

2.3.3 Handling mount points

Under Linux and MacOS, the root path (/) is the only mount point created in the fake file system. If you need support for more mount points, you can add them using `add_mount_point()`.

Under Windows, drives and UNC paths are internally handled as mount points. Adding a file or directory on another drive or UNC path automatically adds a mount point for that drive or UNC path root if needed. Explicitly adding mount points shall not be needed under Windows.

A mount point has a separate device ID (`st_dev`) under all systems, and some operations (like `rename`) are not possible for files located on different mount points. The fake file system size (if used) is also set per mount point.

2.3.4 Setting the file system size

If you need to know the file system size in your tests (for example for testing cleanup scripts), you can set the fake file system size using `set_disk_usage()`. By default, this sets the total size in bytes of the root partition; if you add a path as parameter, the size will be related to the mount point (see above) the path is related to.

By default, the size of the fake file system is considered infinite. As soon as you set a size, all files will occupy the space according to their size, and you may fail to create new files if the fake file system is full.

```
from pyfakefs.fake_filesystem_unittest import TestCase

class ExampleTestCase(TestCase):

    def setUp(self):
        self.setUpPyfakefs()
        self.fs.set_disk_usage(100)

    def test_disk_full(self):
        with open('/foo/bar.txt', 'w') as f:
            self.assertRaises(OSError, f.write, 'a' * 200)
```

To get the file system size, you may use `get_disk_usage()`, which is modeled after `shutil.disk_usage()`.

2.3.5 Pausing patching

Sometimes, you may want to access the real filesystem inside the test with no patching applied. This can be achieved by using the `pause/resume` functions, which exist in `fake_filesystem_unittest.Patcher`, `fake_filesystem_unittest.TestCase` and `fake_filesystem.FakeFilesystem`. There is also a context manager class `fake_filesystem_unittest.Pause` which encapsulates the calls to `pause()` and `resume()`.

Here is an example that tests the usage with the pyfakefs pytest fixture:

```
from pyfakefs.fake_filesystem_unittest import Pause

def test_pause_resume_contextmanager(fs):
    fake_temp_file = tempfile.NamedTemporaryFile()
    assert os.path.exists(fake_temp_file.name)
    fs.pause()
    assert not os.path.exists(fake_temp_file.name)
    real_temp_file = tempfile.NamedTemporaryFile()
    assert os.path.exists(real_temp_file.name)
    fs.resume()
    assert not os.path.exists(real_temp_file.name)
    assert os.path.exists(fake_temp_file.name)
```

Here is the same code using a context manager:

```
from pyfakefs.fake_filesystem_unittest import Pause

def test_pause_resume_contextmanager(fs):
    fake_temp_file = tempfile.NamedTemporaryFile()
    assert os.path.exists(fake_temp_file.name)
    with Pause(fs):
        assert not os.path.exists(fake_temp_file.name)
        real_temp_file = tempfile.NamedTemporaryFile()
        assert os.path.exists(real_temp_file.name)
    assert not os.path.exists(real_temp_file.name)
    assert os.path.exists(fake_temp_file.name)
```

2.4 Troubleshooting

2.4.1 Modules not working with pyfakefs

Modules may not work with `pyfakefs` for several reasons. `pyfakefs` works by patching some file system related modules and functions, specifically:

- most file system related functions in the `os` and `os.path` modules
- the `pathlib` module
- the build-in `open` function and `io.open`
- `shutil.disk_usage`

Other file system related modules work with `pyfakefs`, because they use exclusively these patched functions, specifically `shutil` (except for `disk_usage`), `tempfile`, `glob` and `zipfile`.

A module may not work with `pyfakefs` because of one of the following reasons:

- It uses a file system related function of the mentioned modules that is not or not correctly patched. Mostly these are functions that are seldom used, but may be used in Python libraries (this has happened for example with a changed implementation of `shutil` in Python 3.7). Generally, these shall be handled in issues and we are happy to fix them.
- It uses file system related functions in a way that will not be patched automatically. This is the case for functions that are executed while reading a module. This case and a possibility to make them work is documented above under `modules_to_reload`.
- It uses OS specific file system functions not contained in the Python libraries. These will not work out of the box, and we generally will not support them in `pyfakefs`. If these functions are used in isolated functions or classes, they may be patched by using the `modules_to_patch` parameter (see the example for file locks in Django above), and if there are more examples for patches that may be useful, we may add them in the documentation.
- It uses C libraries to access the file system. There is no way no make such a module work with `pyfakefs` - if you want to use it, you have to patch the whole module. In some cases, a library implemented in Python with a similar interface already exists. An example is `lxml`, which can be substituted with `ElementTree` in most cases for testing.

A list of Python modules that are known to not work correctly with `pyfakefs` will be collected here:

- `multiprocessing` has several issues (related to points 1 and 3 above). Currently there are no plans to fix this, but this may change in case of sufficient demand.

If you are not sure if a module can be handled, or how to do it, you can always write a new issue, of course!

2.4.2 OS temporary directories

Tests relying on a completely empty file system on test start will fail. As `pyfakefs` does not fake the `tempfile` module (as described above), a temporary directory is required to ensure `tempfile` works correctly, e.g., that `tempfile.gettempdir()` will return a valid value. This means that any newly created fake file system will always have either a directory named `/tmp` when running on Linux or Unix systems, `/var/folders/<hash>/T` when running on MacOs and `C:\Users\<user>\AppData\Local\Temp` on Windows.

2.4.3 User rights

If you run pyfakefs tests as root (this happens by default if run in a docker container), pyfakefs also behaves as a root user, for example can write to write-protected files. This may not be the expected behavior, and can be changed. Pyfakefs has a rudimentary concept of user rights, which differentiates between root user (with the user id 0) and any other user. By default, pyfakefs assumes the user id of the current user, but you can change that using `fake_filesystem.set_uid()` in your setup. This allows to run tests as non-root user in a root user environment and vice verse. Another possibility is the convenience argument `allow_root_user` described above.

CHAPTER

THREE

AUTOMATICALLY FIND AND PATCH FILE FUNCTIONS AND MODULES

The `fake_filesystem_unittest` module automatically finds all real file functions and modules, and stubs them out with the fake file system functions and modules. The pyfakefs source code contains files that demonstrate this usage model:

- `example.py` is the software under test. In production, it uses the real file system.
- `example_test.py` tests `example.py`. During testing, the pyfakefs fake file system is used by `example_test.py` and `example.py` alike.

3.1 Software Under Test

`example.py` contains a few functions that manipulate files. For instance:

```
def create_file(path):
    '''Create the specified file and add some content to it.  Use the open()
    built in function.

    For example, the following file operations occur in the fake file system.
    In the real file system, we would not even have permission to write /test:

    >>> os.path.isdir('/test')
    False
    >>> os.mkdir('/test')
    >>> os.path.isdir('/test')
    True
    >>> os.path.exists('/test/file.txt')
    False
    >>> create_file('/test/file.txt')
    >>> os.path.exists('/test/file.txt')
    True
    >>> with open('/test/file.txt') as f:
    ...     f.readlines()
    ["This is test file '/test/file.txt'.\\n", 'It was created using the open() function.\\
↪n']
    '''
    with open(path, 'w') as f:
        f.write("This is test file '{}'.\n".format(path))
        f.write("It was created using the open() function.\n")
```

No functional code in `example.py` even hints at a fake file system. In production, `create_file()` invokes the real file functions `open()` and `write()`.

3.2 Unit Tests and Doctests

`example_test.py` contains unit tests for `example.py`. `example.py` contains the doctests, as you can see above.

The module `fake_filesystem_unittest` contains code that finds all real file functions and modules, and stubs these out with the fake file system functions and modules:

```
import os
import unittest
from pyfakefs import fake_filesystem_unittest
# The module under test is example:
import example
```

3.2.1 Doctests

`example_test.py` defines `load_tests()`, which runs the doctests in `example.py`:

```
def load_tests(loader, tests, ignore):
    '''Load the pyfakefs/example.py doctest tests into unittest.'''
    return fake_filesystem_unittest.load_doctests(loader, tests, ignore, example)
```

Everything, including all imported modules and the test, is stubbed out with the fake filesystem. Thus you can use familiar file functions like `os.mkdir()` as part of your test fixture and they too will operate on the fake file system.

3.2.2 Unit Test Class

Next comes the `unittest` test class. This class is derived from `fake_filesystem_unittest.TestCase`, which is in turn derived from `unittest.TestClass`:

```
class TestExample(fake_filesystem_unittest.TestCase):

    def setUp(self):
        self.setUpPyfakefs()

    def tearDown(self):
        # It is no longer necessary to add self.tearDownPyfakefs()
        pass

    def test_create_file(self):
        '''Test example.create_file()'''
        # The os module has been replaced with the fake os module so all of the
        # following occurs in the fake filesystem.
        self.assertFalse(os.path.isdir('/test'))
        os.mkdir('/test')
        self.assertTrue(os.path.isdir('/test'))

        self.assertFalse(os.path.exists('/test/file.txt'))
```

(continues on next page)

(continued from previous page)

```
        example.create_file('/test/file.txt')
        self.assertTrue(os.path.exists('/test/file.txt'))

    ...
```

Just add `self.setUpPyfakefs()` in `setUp()`. You need add nothing to `tearDown()`. Write your tests as usual. From `self.setUpPyfakefs()` to the end of your `tearDown()` method, all file operations will use the fake file system.

CHAPTER

FOUR

PUBLIC MODULES AND CLASSES

Note: Only public classes and methods interesting to `pyfakefs` users are shown. Methods that mimic the behavior of standard Python functions and classes that are only needed internally are not listed.

4.1 Fake filesystem module

A fake filesystem implementation for unit testing.

Includes

- *FakeFile*: Provides the appearance of a real file.
- *FakeDirectory*: Provides the appearance of a real directory.
- *FakeFilesystem*: Provides the appearance of a real directory hierarchy.
- *FakeOsModule*: Uses *FakeFilesystem* to provide a fake `os` module replacement.
- *FakeIoModule*: Uses *FakeFilesystem* to provide a fake `io` module replacement.
- *FakePathModule*: Faked `os.path` module replacement.
- *FakeFileOpen*: Faked `file()` and `open()` function replacements.

Usage

```
>>> from pyfakefs import fake_filesystem
>>> filesystem = fake_filesystem.FakeFilesystem()
>>> os_module = fake_filesystem.FakeOsModule(filesystem)
>>> pathname = '/a/new/dir/new-file'
```

Create a new file object, creating parent directory objects as needed:

```
>>> os_module.path.exists(pathname)
False
>>> new_file = filesystem.create_file(pathname)
```

File objects can't be overwritten:

```
>>> os_module.path.exists(pathname)
True
>>> try:
...   filesystem.create_file(pathname)
```

(continues on next page)

(continued from previous page)

```
... except IOError as e:
...   assert e.errno == errno.EEXIST, 'unexpected errno: %d' % e.errno
...   assert e.strerror == 'File exists in the fake filesystem'
```

Remove a file object:

```
>>> filesystem.remove_object(pathname)
>>> os_module.path.exists(pathname)
False
```

Create a new file object at the previous path:

```
>>> beatles_file = filesystem.create_file(pathname,
...     contents='Dear Prudence\nWon\'t you come out to play?\n')
>>> os_module.path.exists(pathname)
True
```

Use the FakeFileOpen class to read fake file objects:

```
>>> file_module = fake_filesystem.FakeFileOpen(filesystem)
>>> for line in file_module(pathname):
...     print(line.rstrip())
...
Dear Prudence
Won't you come out to play?
```

File objects cannot be treated like directory objects:

```
>>> try:
...   os_module.listdir(pathname)
... except OSError as e:
...   assert e.errno == errno.ENOTDIR, 'unexpected errno: %d' % e.errno
...   assert e.strerror == 'Not a directory in the fake filesystem'
```

The FakeOsModule can list fake directory objects:

```
>>> os_module.listdir(os_module.path.dirname(pathname))
['new-file']
```

The FakeOsModule also supports stat operations:

```
>>> import stat
>>> stat.S_ISREG(os_module.stat(pathname).st_mode)
True
>>> stat.S_ISDIR(os_module.stat(os_module.path.dirname(pathname)).st_mode)
True
```

pyfakefs.fake_filesystem.**set_uid**(*uid*)

Set the global user id. This is used as st_uid for new files and to differentiate between a normal user and the root user (uid 0). For the root user, some permission restrictions are ignored.

Parameters

uid – (int) the user ID of the user calling the file system functions.

pyfakefs.fake_filesystem.**set_gid**(*gid*)

Set the global group id. This is only used to set st_gid for new files, no permision checks are performed.

Parameters

gid – (int) the group ID of the user calling the file system functions.

4.2 Fake filesystem classes

class pyfakefs.fake_filesystem.**FakeFilesystem**(*path_separator='/'*, *total_size=None*, *patcher=None*)

Provides the appearance of a real directory tree for unit testing.

path_separator

The path separator, corresponds to *os.path.sep*.

alternative_path_separator

Corresponds to *os.path.altsep*.

is_windows_fs

True in a real or faked Windows file system.

is_macos

True under MacOS, or if we are faking it.

is_case_sensitive

True if a case-sensitive file system is assumed.

root

The root *FakeDirectory* entry of the file system.

cwd

The current working directory path.

umask

The umask used for newly created files, see *os.umask*.

patcher

Holds the Patcher object if created from it. Allows access to the patcher object if using the pytest fs fixture.

Parameters

- **path_separator** – optional substitute for os.path.sep
- **total_size** – if not None, the total size in bytes of the root filesystem.

Example usage to emulate real file systems:

```
>>> filesystem = FakeFilesystem(
...     alt_path_separator='/' if _is_windows else None)
```

pause()

Pause the patching of the file system modules until *resume* is called. After that call, all file system calls are executed in the real file system. Calling pause() twice is silently ignored. Only allowed if the file system object was created by a Patcher object. This is also the case for the pytest *fs* fixture.

Raises

RuntimeError – if the file system was not created by a Patcher.

`resume()`

Resume the patching of the file system modules if *pause* has been called before. After that call, all file system calls are executed in the fake file system. Does nothing if patching is not paused. :raises RuntimeError: if the file system has not been created by *Patcher*.

`add_mount_point(path, total_size=None)`

Add a new mount point for a filesystem device. The mount point gets a new unique device number.

Parameters

- **path** – The root path for the new mount path.
- **total_size** – The new total size of the added filesystem device in bytes. Defaults to infinite size.

Returns
The newly created mount point dict.

Raises
OSError – if trying to mount an existing mount point again.

`get_disk_usage(path=None)`

Return the total, used and free disk space in bytes as named tuple, or placeholder values simulating unlimited space if not set.

Note: This matches the return value of shutil.disk_usage().

Parameters
path – The disk space is returned for the file system device where *path* resides. Defaults to the root path (e.g. '/' on Unix systems).

`set_disk_usage(total_size, path=None)`

Changes the total size of the file system, preserving the used space. Example usage: set the size of an auto-mounted Windows drive.

Parameters

- **total_size** – The new total size of the filesystem in bytes.
- **path** – The disk space is changed for the file system device where *path* resides. Defaults to the root path (e.g. '/' on Unix systems).

Raises
IOError – if the new space is smaller than the used size.

`get_object(file_path, check_read_perm=True)`

Search for the specified filesystem object within the fake filesystem.

Parameters

- **file_path** – Specifies the target FakeFile object to retrieve.
- **check_read_perm** – If True, raises OSError if a parent directory does not have read permission

Returns
The FakeFile object corresponding to *file_path*.

Raises
IOError – if the object is not found.

`create_dir`(*directory_path*, *perm_bits=511*)

Create *directory_path*, and all the parent directories.

Helper method to set up your test faster.

Parameters

- **directory_path** – The full directory path to create.
- **perm_bits** – The permission bits as set by *chmod*.

Returns
The newly created FakeDirectory object.

Raises
OSError – if the directory already exists.

`create_file`(*file_path*, *st_mode=33206*, *contents=''*, *st_size=None*, *create_missing_dirs=True*, *apply_umask=False*, *encoding=None*, *errors=None*, *side_effect=None*)

Create *file_path*, including all the parent directories along the way.

This helper method can be used to set up tests more easily.

Parameters

- **file_path** – The path to the file to create.
- **st_mode** – The stat constant representing the file type.
- **contents** – the contents of the file. If not given and st_size is None, an empty file is assumed.
- **st_size** – file size; only valid if contents not given. If given, the file is considered to be in "large file mode" and trying to read from or write to the file will result in an exception.
- **create_missing_dirs** – If *True*, auto create missing directories.
- **apply_umask** – *True* if the current umask must be applied on *st_mode*.
- **encoding** – If *contents* is a unicode string, the encoding used for serialization.
- **errors** – The error mode used for encoding/decoding errors.
- **side_effect** – function handle that is executed when file is written, must accept the file object as an argument.

Returns
The newly created FakeFile object.

Raises

- **IOError** – if the file already exists.
- **IOError** – if the containing directory is required and missing.

`add_real_file`(*source_path*, *read_only=True*, *target_path=None*)

Create *file_path*, including all the parent directories along the way, for an existing real file. The contents of the real file are read only on demand.

Parameters

- **source_path** – Path to an existing file in the real file system
- **read_only** – If *True* (the default), writing to the fake file raises an exception. Otherwise, writing to the file changes the fake file only.

- **target_path** – If given, the path of the target direction, otherwise it is equal to *source_path*.

Returns
the newly created FakeFile object.

Raises

- **OSError** – if the file does not exist in the real file system.
- **IOError** – if the file already exists in the fake file system.

Note: On most systems, accessing the fake file's contents may update both the real and fake files' *atime* (access time). In this particular case, *add_real_file()* violates the rule that *pyfakefs* must not modify the real file system.

---

add_real_symlink(*source_path*, *target_path=None*)

Create a symlink at source_path (or target_path, if given). It will point to the same path as the symlink on the real filesystem. Relative symlinks will point relative to their new location. Absolute symlinks will point to the same, absolute path as on the real filesystem.

Parameters

- **source_path** – The path to the existing symlink.
- **target_path** – If given, the name of the symlink in the fake fileystem, otherwise, the same as *source_path*.

Returns
the newly created FakeDirectory object.

Raises

- **OSError** – if the directory does not exist in the real file system.
- **OSError** – if the symlink could not be created (see *create_file()*).
- **OSError** – if on Windows before Python 3.2.
- **IOError** – if the directory already exists in the fake file system.

add_real_directory(*source_path*, *read_only=True*, *lazy_read=True*, *target_path=None*)

Create a fake directory corresponding to the real directory at the specified path. Add entries in the fake directory corresponding to the entries in the real directory. Symlinks are supported.

Parameters

- **source_path** – The path to the existing directory.
- **read_only** – If set, all files under the directory are treated as read-only, e.g. a write access raises an exception; otherwise, writing to the files changes the fake files only as usually.
- **lazy_read** – If set (default), directory contents are only read when accessed, and only until the needed subdirectory level.

  Note: This means that the file system size is only updated at the time the directory contents are read; set this to *False* only if you are dependent on accurate file system size in your test

  ---

- **target_path** – If given, the target directory, otherwise, the target directory is the same as *source_path*.

Returns
the newly created FakeDirectory object.

Raises

- **OSError** – if the directory does not exist in the real file system.
- **IOError** – if the directory already exists in the fake file system.

add_real_paths(*path_list*, *read_only=True*, *lazy_dir_read=True*)

This convenience method adds multiple files and/or directories from the real file system to the fake file system. See *add_real_file()* and *add_real_directory()*.

Parameters

- **path_list** – List of file and directory paths in the real file system.
- **read_only** – If set, all files and files under under the directories are treated as read-only, e.g. a write access raises an exception; otherwise, writing to the files changes the fake files only as usually.
- **lazy_dir_read** – Uses lazy reading of directory contents if set (see *add_real_directory*)

Raises

- **OSError** – if any of the files and directories in the list does not exist in the real file system.
- **OSError** – if any of the files and directories in the list already exists in the fake file system.

create_symlink(*file_path*, *link_target*, *create_missing_dirs=True*)

Create the specified symlink, pointed at the specified link target.

Parameters

- **file_path** – path to the symlink to create
- **link_target** – the target of the symlink
- **create_missing_dirs** – If *True*, any missing parent directories of file_path will be created

Returns
The newly created FakeFile object.

Raises

- **OSError** – if the symlink could not be created (see *create_file()*).
- **OSError** – if on Windows before Python 3.2.

class pyfakefs.fake_filesystem.**FakeFile**(*name*, *st_mode=33206*, *contents=None*, *filesystem=None*, *encoding=None*, *errors=None*, *side_effect=None*)

Provides the appearance of a real file.

Attributes currently faked out:

- *st_mode*: user-specified, otherwise S_IFREG
- *st_ctime*: the time.time() timestamp of the file change time (updated each time a file's attributes is modified).
- *st_atime*: the time.time() timestamp when the file was last accessed.
- *st_mtime*: the time.time() timestamp when the file was last modified.
- *st_size*: the size of the file

- *st_nlink*: the number of hard links to the file
- *st_ino*: the inode number - a unique number identifying the file
- *st_dev*: a unique number identifying the (fake) file system device the file belongs to
- ***st_uid*: always set to USER_ID, which can be changed globally using** *set_uid*
- ***st_gid*: always set to GROUP_ID, which can be changed globally using** *set_gid*

Note: The resolution for *st_ctime*, *st_mtime* and *st_atime* in the real file system depends on the used file system (for example it is only 1s for HFS+ and older Linux file systems, but much higher for ext4 and NTFS). This is currently ignored by pyfakefs, which uses the resolution of *time.time()*.

Under Windows, *st_atime* is not updated for performance reasons by default. pyfakefs never updates *st_atime* under Windows, assuming the default setting.

Parameters

- **name** – Name of the file/directory, without parent path information
- **st_mode** – The stat.S_IF* constant representing the file type (i.e. stat.S_IFREG, stat.S_IFDIR)
- **contents** – The contents of the filesystem object; should be a string or byte object for regular files, and a list of other FakeFile or FakeDirectory objects for FakeDirectory objects
- **filesystem** – The fake filesystem where the file is created.
- **encoding** – If contents is a unicode string, the encoding used for serialization.
- **errors** – The error mode used for encoding/decoding errors.
- **side_effect** – function handle that is executed when file is written, must accept the file object as an argument.

property byte_contents

Return the contents as raw byte array.

property contents

Return the contents as string with the original encoding.

is_large_file()

Return *True* if this file was initialized with size but no contents.

set_contents(*contents*, *encoding=None*)

Sets the file contents and size and increases the modification time. Also executes the side_effects if available.

Parameters

- **contents** – (str, bytes, unicode) new content of file.
- **encoding** – (str) the encoding to be used for writing the contents if they are a unicode string. If not given, the locale preferred encoding is used.

Raises

IOError – if *st_size* is not a non-negative integer, or if it exceeds the available file system space.

property path

Return the full path of the current object.

property size

Return the size in bytes of the file contents.

class pyfakefs.fake_filesystem.FakeDirectory(*name*, *perm_bits=511*, *filesystem=None*)

Provides the appearance of a real directory.

Parameters

- **name** – name of the file/directory, without parent path information
- **perm_bits** – permission bits. defaults to 0o777.
- **filesystem** – if set, the fake filesystem where the directory is created

property contents

Return the list of contained directory entries.

property ordered_dirs

Return the list of contained directory entry names ordered by creation order.

get_entry(*pathname_name*)

Retrieves the specified child file or directory entry.

Parameters
pathname_name – The basename of the child object to retrieve.

Returns
The fake file or directory object.

Raises
KeyError – if no child exists by the specified name.

remove_entry(*pathname_name*, *recursive=True*)

Removes the specified child file or directory.

Parameters

- **pathname_name** – Basename of the child object to remove.
- **recursive** – If True (default), the entries in contained directories are deleted first. Used to propagate removal errors (e.g. permission problems) from contained entries.

Raises

- **KeyError** – if no child exists by the specified name.
- **OSError** – if user lacks permission to delete the file, or (Windows only) the file is open.

property size

Return the total size of all files contained in this directory tree.

4.3 Unittest module classes

class pyfakefs.fake_filesystem_unittest.**TestCaseMixin**

Test case mixin that automatically replaces file-system related modules by fake implementations.

additional_skip_names

names of modules inside of which no module replacement shall be performed, in addition to the names in `fake_filesystem_unittest.Patcher.SKIPNAMES`. Instead of the module names, the modules themselves may be used.

modules_to_reload

A list of modules that need to be reloaded to be patched dynamically; may be needed if the module imports file system modules under an alias

> **Caution:** Reloading modules may have unwanted side effects.

modules_to_patch

A dictionary of fake modules mapped to the fully qualified patched module names. Can be used to add patching of modules not provided by *pyfakefs*.

If you specify some of these attributes here and you have DocTests, consider also specifying the same arguments to `load_doctests()`.

Example usage in derived test classes:

```
from unittest import TestCase
from fake_filesystem_unittest import TestCaseMixin

class MyTestCase(TestCase, TestCaseMixin):
    def __init__(self, methodName='runTest'):
        super(MyTestCase, self).__init__(
            methodName=methodName,
            additional_skip_names=['posixpath'])

import sut

class AnotherTestCase(TestCase, TestCaseMixin):
    def __init__(self, methodName='runTest'):
        super(MyTestCase, self).__init__(
            methodName=methodName, modules_to_reload=[sut])
```

setUpPyfakefs(*additional_skip_names=None*, *modules_to_reload=None*, *modules_to_patch=None*, *allow_root_user=True*)

Bind the file-related modules to the `pyfakefs` fake file system instead of the real file system. Also bind the fake *open()* function, and on Python 2, the *file()* function.

Invoke this at the beginning of the *setUp()* method in your unit test class. For the arguments, see the *TestCaseMixin* attribute description. If any of the arguments is not None, it overwrites the settings for the current test case. Settings the arguments here may be a more convenient way to adapt the setting than overwriting *__init__()*.

pause()

Pause the patching of the file system modules until *resume* is called. After that call, all file system calls are executed in the real file system. Calling pause() twice is silently ignored.

resume()

Resume the patching of the file system modules if *pause* has been called before. After that call, all file system calls are executed in the fake file system. Does nothing if patching is not paused.

class pyfakefs.fake_filesystem_unittest.**TestCase**(*methodName='runTest'*, *additional_skip_names=None*, *modules_to_reload=None*, *modules_to_patch=None*, *allow_root_user=True*)

Test case class that automatically replaces file-system related modules by fake implementations. Inherits *TestCaseMixin*.

The arguments are explained in *TestCaseMixin*.

Creates the test class instance and the patcher used to stub out file system related modules.

Parameters
methodName – The name of the test method (same as in unittest.TestCase)

class pyfakefs.fake_filesystem_unittest.**Patcher**(*additional_skip_names=None*, *modules_to_reload=None*, *modules_to_patch=None*, *allow_root_user=True*)

Instantiate a stub creator to bind and un-bind the file-related modules to the `pyfakefs` fake modules.

The arguments are explained in *TestCaseMixin*.

Patcher is used in *TestCaseMixin*. *Patcher* also works as a context manager for other tests:

```
with Patcher():
    doStuff()
```

For a description of the arguments, see TestCase.__init__

setUp(*doctester=None*)

Bind the file-related modules to the `pyfakefs` fake modules real ones. Also bind the fake *file()* and *open()* functions.

tearDown(*doctester=None*)

Clear the fake filesystem bindings created by *setUp()*.

pause()

Pause the patching of the file system modules until *resume* is called. After that call, all file system calls are executed in the real file system. Calling pause() twice is silently ignored.

resume()

Resume the patching of the file system modules if *pause* has been called before. After that call, all file system calls are executed in the fake file system. Does nothing if patching is not paused.

4.4 Faked module classes

class pyfakefs.fake_filesystem.**FakeOsModule**(*filesystem*, *os_path_module=None*)

Uses FakeFilesystem to provide a fake os module replacement.

Do not create os.path separately from os, as there is a necessary circular dependency between os and os.path to replicate the behavior of the standard Python modules. What you want to do is to just let FakeOsModule take care of *os.path* setup itself.

You always want to do this. filesystem = fake_filesystem.FakeFilesystem() my_os_module = fake_filesystem.FakeOsModule(filesystem)

Also exposes self.path (to fake os.path).

Parameters

- **filesystem** – FakeFilesystem used to provide file system information
- **os_path_module** – (deprecated) Optional FakePathModule instance

class pyfakefs.fake_filesystem.**FakePathModule**(*filesystem*, *os_module=None*)

Faked os.path module replacement.

FakePathModule should *only* be instantiated by FakeOsModule. See the FakeOsModule docstring for details.

Init.

Parameters

- **filesystem** – FakeFilesystem used to provide file system information
- **os_module** – (deprecated) FakeOsModule to assign to self.os

class pyfakefs.fake_filesystem.**FakeFileOpen**(*filesystem*, *delete_on_close=False*, *use_io=False*, *raw_io=False*)

Faked *file()* and *open()* function replacements.

Returns FakeFile objects in a FakeFilesystem in place of the *file()* or *open()* function.

Parameters

- **filesystem** – FakeFilesystem used to provide file system information
- **delete_on_close** – optional boolean, deletes file on close()
- **use_io** – if True, the io.open() version is used (ignored for Python 3, where io.open() is an alias to open())

class pyfakefs.fake_filesystem.**FakeIoModule**(*filesystem*)

Uses FakeFilesystem to provide a fake io module replacement.

Currently only used to wrap *io.open()* which is an alias to *open()*.

You need a fake_filesystem to use this: filesystem = fake_filesystem.FakeFilesystem() my_io_module = fake_filesystem.FakeIoModule(filesystem)

Parameters
filesystem – FakeFilesystem used to provide file system information.

class pyfakefs.fake_filesystem_shutil.**FakeShutilModule**(*filesystem*)

Uses a FakeFilesystem to provide a fake replacement for shutil module.

Construct fake shutil module using the fake filesystem.

Parameters
filesystem – FakeFilesystem used to provide file system information

class pyfakefs.fake_pathlib.**FakePathlibModule**(*filesystem*)

Uses FakeFilesystem to provide a fake pathlib module replacement. Can be used to replace both the standard *pathlib* module and the *pathlib2* package available on PyPi.

You need a fake_filesystem to use this: *filesystem = fake_filesystem.FakeFilesystem() fake_pathlib_module = fake_filesystem.FakePathlibModule(filesystem)*

Initializes the module with the given filesystem.

Parameters
filesystem – FakeFilesystem used to provide file system information

class pyfakefs.fake_scandir.**FakeScanDirModule**(*filesystem*)

Uses FakeFilesystem to provide a fake *scandir* module replacement.

Note: The `scandir` function is a part of the standard `os` module since Python 3.5. This class handles the separate `scandir` module that is available on pypi.

You need a fake_filesystem to use this: *filesystem = fake_filesystem.FakeFilesystem() fake_scandir_module = fake_filesystem.FakeScanDirModule(filesystem)*

CHAPTER

FIVE

API NOTES

With `pyfakefs 3.4`, the public API has changed to be PEP-8 conform. The old API is deprecated, and will be removed in some future version of pyfakefs. You can suppress the deprecation warnings for legacy code with the following code:

```
from pyfakefs.deprecator import Deprecator

Deprecator.show_warnings = False
```

Here is a list of selected changes:

pyfakefs.fake_filesystem.FakeFileSystem

CreateFile() -> create_file()

CreateDirectory() -> create_dir()

CreateLink() -> create_symlink()

GetDiskUsage() -> get_disk_usage()

SetDiskUsage() -> set_disk_usage()

pyfakefs.fake_filesystem.FakeFile

GetSize(), SetSize() -> size (property)

SetContents() -> set_contents()

SetATime() -> st_atime (property)

SetMTime() -> st_mtime (property)

SetCTime() -> st_ctime (property)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

p

INDEX

O

P

R

S

T

U