# PyFilesystem Documentation

*Release 2.4.16*

**Will McGugan**

**May 02, 2022**

# Contents

Contents:

CHAPTER 1

# Introduction

PyFilesystem is a Python module that provides a common interface to any filesystem.

Think of PyFilesystem `FS` objects as the next logical step to Python's `file` objects. In the same way that file objects abstract a single file, FS objects abstract an entire filesystem.

## 1.1 Installing

You can install PyFilesystem with `pip` as follows:

```
pip install fs
```

Or to upgrade to the most recent version:

```
pip install fs --upgrade
```

PyFilesystem is also available on conda:

```
conda install fs -c conda-forge
```

Alternatively, if you would like to install from source, you can check out the code from Github.

CHAPTER 2

# Guide

The PyFilesytem interface simplifies most aspects of working with files and directories. This guide covers what you need to know about working with FS objects.

## 2.1 Why use PyFilesystem?

If you are comfortable using the Python standard library, you may be wondering; *why learn another API for working with files?*

The *PyFilesystem API* is generally simpler than the `os` and `io` modules – there are fewer edge cases and less ways to shoot yourself in the foot. This may be reason alone to use it, but there are other compelling reasons you should use `import fs` for even straightforward filesystem code.

The abstraction offered by FS objects means that you can write code that is agnostic to where your files are physically located. For instance, if you wrote a function that searches a directory for duplicates files, it will work unaltered with a directory on your hard-drive, or in a zip file, on an FTP server, on Amazon S3, etc.

As long as an FS object exists for your chosen filesystem (or any data store that resembles a filesystem), you can use the same API. This means that you can defer the decision regarding where you store data to later. If you decide to store configuration in the *cloud*, it could be a single line change and not a major refactor.

PyFilesystem can also be beneficial for unit-testing; by swapping the OS filesystem with an in-memory filesystem, you can write tests without having to manage (or mock) file IO. And you can be sure that your code will work on Linux, MacOS, and Windows.

## 2.2 Opening Filesystems

There are two ways you can open a filesystem. The first and most natural way is to import the appropriate filesystem class and construct it.

Here's how you would open a *OSFS* (Operating System File System), which maps to the files and directories of your hard-drive:

```
>>> from fs.osfs import OSFS
>>> home_fs = OSFS("~/")
```

This constructs an FS object which manages the files and directories under a given system path. In this case, `'~/'`, which is a shortcut for your home directory.

Here's how you would list the files/directories in your home directory:

```
>>> home_fs.listdir('/')
['world domination.doc', 'paella-recipe.txt', 'jokes.txt', 'projects']
```

Notice that the parameter to `listdir` is a single forward slash, indicating that we want to list the *root* of the filesystem. This is because from the point of view of `home_fs`, the root is the directory we used to construct the `OSFS`.

Also note that it is a forward slash, even on Windows. This is because FS paths are in a consistent format regardless of the platform. Details such as the separator and encoding are abstracted away. See *Paths* for details.

Other filesystems interfaces may have other requirements for their constructor. For instance, here is how you would open a FTP filesystem:

```
>>> from ftpfs import FTPFS
>>> debian_fs = FTPFS('ftp.mirror.nl')
>>> debian_fs.listdir('/')
['debian-archive', 'debian-backports', 'debian', 'pub', 'robots.txt']
```

The second, and more general way of opening filesystems objects, is via an *opener* which opens a filesystem from a URL-like syntax. Here's an alternative way of opening your home directory:

```
>>> from fs import open_fs
>>> home_fs = open_fs('osfs://~/')
>>> home_fs.listdir('/')
['world domination.doc', 'paella-recipe.txt', 'jokes.txt', 'projects']
```

The opener system is particularly useful when you want to store the physical location of your application's files in a configuration file.

If you don't specify the protocol in the FS URL, then PyFilesystem will assume you want a OSFS relative from the current working directory. So the following would be an equivalent way of opening your home directory:

```
>>> from fs import open_fs
>>> home_fs = open_fs('.')
>>> home_fs.listdir('/')
['world domination.doc', 'paella-recipe.txt', 'jokes.txt', 'projects']
```

## 2.3 Tree Printing

Calling *`tree()`* on a FS object will print an ascii tree view of your filesystem. Here's an example:

```
>>> from fs import open_fs
>>> my_fs = open_fs('.')
>>> my_fs.tree()
├── locale
│   └── readme.txt
├── logic
│   ├── content.xml
```

(continues on next page)

(continued from previous page)

```
│   ├── data.xml
│   ├── mountpoints.xml
│   └── readme.txt
├── lib.ini
└── readme.txt
```

This can be a useful debugging aid!

## 2.4 Closing

FS objects have a *close()* methd which will perform any required clean-up actions. For many filesystems (notably *OSFS*), the `close` method does very little. Other filesystems may only finalize files or release resources once `close()` is called.

You can call `close` explicitly once you are finished using a filesystem. For example:

```
>>> home_fs = open_fs('osfs://~/')
>>> home_fs.writetext('reminder.txt', 'buy coffee')
>>> home_fs.close()
```

If you use FS objects as a context manager, `close` will be called automatically. The following is equivalent to the previous example:

```
>>> with open_fs('osfs://~/') as home_fs:
...     home_fs.writetext('reminder.txt', 'buy coffee')
```

Using FS objects as a context manager is recommended as it will ensure every FS is closed.

## 2.5 Directory Information

Filesystem objects have a *listdir()* method which is similar to `os.listdir`; it takes a path to a directory and returns a list of file names. Here's an example:

```
>>> home_fs.listdir('/projects')
['fs', 'moya', 'README.md']
```

An alternative method exists for listing directories; *scandir()* returns an *iterable* of *Resource Info* objects. Here's an example:

```
>>> directory = list(home_fs.scandir('/projects'))
>>> directory
[<dir 'fs'>, <dir 'moya'>, <file 'README.md'>]
```

Info objects have a number of advantages over just a filename. For instance you can tell if an info object references a file or a directory with the *is_dir* attribute, without an additional system call. Info objects may also contain information such as size, modified time, etc. if you request it in the `namespaces` parameter.

---

**Note:** The reason that `scandir` returns an iterable rather than a list, is that it can be more efficient to retrieve directory information in chunks if the directory is very large, or if the information must be retrieved over a network.

---

Additionally, FS objects have a *filterdir()* method which extends `scandir` with the ability to filter directory contents by wildcard(s). Here's how you might find all the Python files in a directory:

```
>>> code_fs = OSFS('~/projects/src')
>>> directory = list(code_fs.filterdir('/', files=['*.py']))
```

By default, the resource information objects returned by `scandir` and `listdir` will contain only the file name and the `is_dir` flag. You can request additional information with the `namespaces` parameter. Here's how you can request additional details (such as file size and file modified times):

```
>>> directory = code_fs.filterdir('/', files=['*.py'], namespaces=['details'])
```

This will add a `size` and `modified` property (and others) to the resource info objects. Which makes code such as this work:

```
>>> sum(info.size for info in directory)
```

See *Resource Info* for more information.

## 2.6 Sub Directories

PyFilesystem has no notion of a *current working directory*, so you won't find a `chdir` method on FS objects. Fortunately you won't miss it; working with sub-directories is a breeze with PyFilesystem.

You can always specify a directory with methods which accept a path. For instance, `home_fs.listdir('/projects')` would get the directory listing for the `projects` directory. Alternatively, you can call *opendir()* which returns a new FS object for the sub-directory.

For example, here's how you could list the directory contents of a `projects` folder in your home directory:

```
>>> home_fs = open_fs('~/')
>>> projects_fs = home_fs.opendir('/projects')
>>> projects_fs.listdir('/')
['fs', 'moya', 'README.md']
```

When you call `opendir`, the FS object returns an instance of a *SubFS*. If you call any of the methods on a `SubFS` object, it will be as though you called the same method on the parent filesystem with a path relative to the sub-directory.

The *makedir* and *makedirs* methods also return `SubFS` objects for the newly create directory. Here's how you might create a new directory in `~/projects` and initialize it with a couple of files:

```
>>> home_fs = open_fs('~/')
>>> game_fs = home_fs.makedirs('projects/game')
>>> game_fs.touch('__init__.py')
>>> game_fs.writetext('README.md', "Tetris clone")
>>> game_fs.listdir('/')
['__init__.py', 'README.md']
```

Working with `SubFS` objects means that you can generally avoid writing much path manipulation code, which tends to be error prone.

## 2.7 Working with Files

You can open a file from a FS object with *open()*, which is very similar to `io.open` in the standard library. Here's how you might open a file called "reminder.txt" in your home directory:

```
>>> with open_fs('~/') as home_fs:
...     with home_fs.open('reminder.txt') as reminder_file:
...         print(reminder_file.read())
buy coffee
```

In the case of a `OSFS`, a standard file-like object will be returned. Other filesystems may return a different object supporting the same methods. For instance, *MemoryFS* will return a `io.BytesIO` object.

PyFilesystem also offers a number of shortcuts for common file related operations. For instance, *readbytes()* will return the file contents as bytes, and *readtext()* will read unicode text. These methods are generally preferable to explicitly opening files, as the FS object may have an optimized implementation.

Other *shortcut* methods are *download()*, *upload()*, *writebytes()*, *writetext()*.

## 2.8 Walking

Often you will need to scan the files in a given directory, and any sub-directories. This is known as *walking* the filesystem.

Here's how you would print the paths to all your Python files in your home directory:

```
>>> from fs import open_fs
>>> home_fs = open_fs('~/')
>>> for path in home_fs.walk.files(filter=['*.py']):
...     print(path)
```

The `walk` attribute on FS objects is instance of a *BoundWalker*, which should be able to handle most directory walking requirements.

See *Walking* for more information on walking directories.

## 2.9 Globbing

Closely related to walking a filesystem is *globbing*, which is a slightly higher level way of scanning filesystems. Paths can be filtered by a *glob* pattern, which is similar to a wildcard (such as `*.py`), but can match multiple levels of a directory structure.

Here's an example of globbing, which removes all the `.pyc` files in your project directory:

```
>>> from fs import open_fs
>>> open_fs('~/project').glob('**/*.pyc').remove()
62
```

See *Globbing* for more information.

## 2.10 Moving and Copying

You can move and copy file contents with *move()* and *copy()* methods, and the equivalent *movedir()* and *copydir()* methods which operate on directories rather than files.

These move and copy methods are optimized where possible, and depending on the implementation, they may be more performant than reading and writing files.

To move and/or copy files *between* filesystems (as apposed to within the same filesystem), use the *move* and *copy* modules. The methods in these modules accept both FS objects and FS URLS. For instance, the following will compress the contents of your projects folder:

```
>>> from fs.copy import copy_fs
>>> copy_fs('~/projects', 'zip://projects.zip')
```

Which is the equivalent to this, more verbose, code:

```
>>> from fs.copy import copy_fs
>>> from fs.osfs import OSFS
>>> from fs.zipfs import ZipFS
>>> copy_fs(OSFS('~/projects'), ZipFS('projects.zip'))
```

The *copy_fs()* and *copy_dir()* functions also accept a *Walker* parameter, which can you use to filter the files that will be copied. For instance, if you only wanted back up your python files, you could use something like this:

```
>>> from fs.copy import copy_fs
>>> from fs.walk import Walker
>>> copy_fs('~/projects', 'zip://projects.zip', walker=Walker(filter=['*.py']))
```

An alternative to copying is *mirroring*, which will copy a filesystem them keep it up to date by copying only changed files / directories. See *mirror()*.

CHAPTER 3

# Concepts

The following describes some core concepts when working with PyFilesystem. If you are skimming this documentation, pay particular attention to the first section on paths.

## 3.1 Paths

With the possible exception of the constructor, all paths in a filesystem are *PyFilesystem paths*, which have the following properties:

- Paths are `str` type in Python3, and `unicode` in Python2
- Path components are separated by a forward slash (`/`)
- Paths beginning with a `/` are *absolute*
- Paths not beginning with a forward slash are *relative*
- A single dot (`.`) means 'current directory'
- A double dot (`..`) means 'previous directory'

Note that paths used by the FS interface will use this format, but the constructor may not. Notably the *OSFS* constructor which requires an OS path – the format of which is platform-dependent.

**Note:** There are many helpful functions for working with paths in the *path* module.

PyFilesystem paths are platform-independent, and will be automatically converted to the format expected by your operating system – so you won't need to make any modifications to your filesystem code to make it run on other platforms.

## 3.2 System Paths

Not all Python modules can use file-like objects, especially those which interface with C libraries. For these situations you will need to retrieve the *system path*. You can do this with the *getsyspath()* method which converts a valid path in the context of the FS object to an absolute path that would be understood by your OS.

For example:

```
>>> from fs.osfs import OSFS
>>> home_fs = OSFS('~/')
>>> home_fs.getsyspath('test.txt')
'/home/will/test.txt'
```

Not all filesystems map to a system path (for example, files in a *MemoryFS* will only ever exists in memory).

If you call `getsyspath` on a filesystem which doesn't map to a system path, it will raise a *NoSysPath* exception. If you prefer a *look before you leap* approach, you can check if a resource has a system path by calling *hassyspath()*

## 3.3 Sandboxing

FS objects are not permitted to work with any files outside of their *root*. If you attempt to open a file or directory outside the filesystem instance (with a backref such as `"../foo.txt"`), a *IllegalBackReference* exception will be thrown. This ensures that any code using a FS object won't be able to read or modify anything you didn't intend it to, thus limiting the scope of any bugs.

Unlike your OS, there is no concept of a current working directory in PyFilesystem. If you want to work with a sub-directory of an FS object, you can use the *opendir()* method which returns another FS object representing the contents of that sub-directory.

For example, consider the following directory structure. The directory `foo` contains two sub-directories; `bar` and `baz`:

```
--foo
  |--bar
  |  |--readme.txt
  |  `--photo.jpg
  `--baz
     |--private.txt
     `--dontopen.jpg
```

We can open the `foo` directory with the following code:

```
from fs.osfs import OSFS
foo_fs = OSFS('foo')
```

The `foo_fs` object can work with any of the contents of `bar` and `baz`, which may not be desirable if we are passing `foo_fs` to a function that has the potential to delete files. Fortunately we can isolate a single sub-directory with the *opendir()* method:

```
bar_fs = foo_fs.opendir('bar')
```

This creates a completely new FS object that represents everything in the `foo/bar` directory. The root directory of `bar_fs` has been re- position, so that from `bar_fs`'s point of view, the readme.txt and photo.jpg files are in the root:

```
--bar
  |--readme.txt
  `--photo.jpg
```

**Note:** This *sandboxing* only works if your code uses the filesystem interface exclusively. It won't prevent code using standard OS level file manipulation.

## 3.4 Errors

PyFilesystem converts errors in to a common exception hierarchy. This ensures that error handling code can be written once, regardless of the filesystem being used. See *errors* for details.

CHAPTER 4

# Resource Info

Resource information (or *info*) describes standard file details such as name, type, size, etc., and potentially other less-common information associated with a file or directory.

You can retrieve resource info for a single resource by calling *getinfo()*, or by calling *scandir()* which returns an iterator of resource information for the contents of a directory. Additionally, *filterdir()* can filter the resources in a directory by type and wildcard.

Here's an example of retrieving file information:

```
>>> from fs.osfs import OSFS
>>> fs = OSFS('.')
>>> fs.writetext('example.txt', 'Hello, World!')
>>> info = fs.getinfo('example.txt', namespaces=['details'])
>>> info.name
'example.txt'
>>> info.is_dir
False
>>> info.size
13
```

## 4.1 Info Objects

PyFilesystem exposes the resource information via properties of *Info* objects.

## 4.2 Namespaces

All resource information is contained within one of a number of potential *namespaces*, which are logical key/value groups.

You can specify which namespace(s) you are interested in with the `namespaces` argument to *getinfo()*. For example, the following retrieves the `details` and `access` namespaces for a file:

```
resource_info = fs.getinfo('myfile.txt', namespaces=['details', 'access'])
```

In addition to the specified namespaces, the fileystem will also return the `basic` namespace, which contains the name of the resource, and a flag which indicates if the resource is a directory.

### 4.2.1 Basic Namespace

The `basic` namespace is always returned. It contains the following keys:

| Name | Type | Description |
|---|---|---|
| name | str | Name of the resource. |
| is_dir | bool | A boolean that indicates if the resource is a directory. |

The keys in this namespace can generally be retrieved very quickly. In the case of *OSFS* the namespace can be retrieved without a potentially expensive system call.

### 4.2.2 Details Namespace

The `details` namespace contains the following keys.

| Name | type | Description |
|---|---|---|
| accessed | datetime | The time the file was last accessed. |
| created | datetime | The time the file was created. |
| metadata_changed | datetime | The time of the last *metadata* (e.g. owner, group) change. |
| modified | datetime | The time file data was last changed. |
| size | int | Number of bytes used to store the resource. In the case of files, this is the number of bytes in the file. For directories, the *size* is the overhead (in bytes) used to store the directory entry. |
| type | ResourceType | Resource type, one of the values defined in *ResourceType*. |

The time values (`accessed_time`, `created_time` etc.) may be `None` if the filesystem doesn't store that information. The `size` and `type` keys are guaranteed to be available, although `type` may be *unknown* if the filesystem is unable to retrieve the resource type.

### 4.2.3 Access Namespace

The `access` namespace reports permission and ownership information, and contains the following keys.

| Name | type | Description |
|---|---|---|
| gid | int | The group ID. |
| group | str | The group name. |
| permissions | Permissions | An instance of *Permissions*, which contains the permissions for the resource. |
| uid | int | The user ID. |
| user | str | The user name of the owner. |

This namespace is optional, as not all filesystems have a concept of ownership or permissions. It is supported by *OSFS*. Some values may be `None` if they aren't supported by the filesystem.

### 4.2.4 Stat Namespace

The `stat` namespace contains information reported by a call to os.stat. This namespace is supported by *OSFS* and potentially other filesystems which map directly to the OS filesystem. Most other filesystems will not support this namespace.

### 4.2.5 LStat Namespace

The `lstat` namespace contains information reported by a call to os.lstat. This namespace is supported by *OSFS* and potentially other filesystems which map directly to the OS filesystem. Most other filesystems will not support this namespace.

### 4.2.6 Link Namespace

The `link` namespace contains information about a symlink.

| Name | type | Description |
|---|---|---|
| target | str | A path to the symlink target, or `None` if this path is not a symlink. Note, the meaning of this target is somewhat filesystem dependent, and may not be a valid path on the FS object. |

### 4.2.7 Other Namespaces

Some filesystems may support other namespaces not covered here. See the documentation for the specific filesystem for information on what namespaces are supported.

You can retrieve such implementation specific resource information with the *get()* method.

---

**Note:** It is not an error to request a namespace (or namespaces) that the filesystem does *not* support. Any unknown namespaces will be ignored.

---

## 4.3 Missing Namespaces

Some attributes on the Info objects require that a given namespace be present. If you attempt to reference them without the namespace being present (because you didn't request it, or the filesystem doesn't support it) then a *MissingInfoNamespace* exception will be thrown. Here's how you might handle such exceptions:

```
try:
    print('user is {}'.format(info.user))
except errors.MissingInfoNamespace:
    # No 'access' namespace
    pass
```

If you prefer a *look before you leap* approach, you can use use the *has_namespace()* method. Here's an example:

```
if info.has_namespace('access'):
    print('user is {}'.format(info.user))
```

See *Info* for details regarding info attributes.

## 4.4 Raw Info

The *Info* class is a wrapper around a simple data structure containing the *raw* info. You can access this raw info with the `info.raw` property.

---

**Note:** The following is probably only of interest if you intend to implement a filesystem yourself.

---

Raw info data consists of a dictionary that maps the namespace name on to a dictionary of information. Here's an example:

```
{
    'access': {
        'group': 'staff',
        'permissions': ['g_r', 'o_r', 'u_r', 'u_w'],
        'user': 'will'
    },
    'basic': {
        'is_dir': False,
        'name': 'README.txt'
    },
    'details': {
        'accessed': 1474979730.0,
        'created': 1462266356.0,
        'metadata_changed': 1473071537.0,
        'modified': 1462266356.0,
        'size': 79,
        'type': 2
    }
}
```

Raw resource information contains basic types only (strings, numbers, lists, dict, None). This makes the resource information simple to send over a network as it can be trivially serialized as JSON or other data format.

Because of this requirement, times are stored as epoch times. The Info object will convert these to datetime objects from the standard library. Additionally, the Info object will convert permissions from a list of strings in to a *Permissions* objects.

# CHAPTER 5

# FS URLs

PyFilesystem can open a filesystem via an *FS URL*, which is similar to a URL you might enter in to a browser. FS URLs are useful if you want to specify a filesystem dynamically, such as in a conf file or from the command line.

## 5.1 Format

FS URLs are formatted in the following way:

```
<protocol>://<username>:<password>@<resource>
```

The components are as follows:

- `<protocol>` Identifies the type of filesystem to create. e.g. `osfs`, `ftp`.
- `<username>` Optional username.
- `<password>` Optional password.
- `<resource>` A *resource*, which may be a domain, path, or both.

Here are a few examples:

```
osfs://~/projects
osfs://c://system32
ftp://ftp.example.org/pub
mem://
ftp://will:daffodil@ftp.example.org/private
```

If `<type>` is not specified then it is assumed to be an *OSFS*, i.e. the following FS URLs are equivalent:

```
osfs://~/projects
~/projects
```

**Note:** The `username` and `passwords` fields may not contain a colon (`:`) or an `@` symbol. If you need these symbols they may be percent encoded.

## 5.2 URL Parameters

FS URLs may also be appended with a `?` symbol followed by a url-encoded query string. For example:

```
myprotocol://example.org?key1=value1&key2
```

The query string would be decoded as `{"key1": "value1", "key2": ""}`.

Query strings are used to provide additional filesystem-specific information used when opening. See the filesystem documentation for information on what query string parameters are supported.

## 5.3 Opening FS URLS

To open a filesysem with a FS URL, you can use *open_fs()*, which may be imported and used as follows:

```
from fs import open_fs
projects_fs = open_fs('osfs://~/projects')
```

## 5.4 Manually registering Openers

The `fs.opener` registry uses an entry point to install external openers (see *Creating an extension*), and it does so once, when you import `fs` for the first time. In some rare cases where entry points are not available (for instance, when running an embedded interpreter) or when extensions are installed *after* the interpreter has started (for instance in a notebook, see PyFilesystem2#485).

However, a new opener can be installed manually at any time with the `fs.opener.registry.install` method. For instance, here's how the opener for the s3fs extension can be added to the registry:

```
import fs.opener
from fs_s3fs.opener import S3FSOpener

fs.opener.registry.install(S3FSOpener)
# fs.open_fs("s3fs://...") should now work
```

CHAPTER 6

# Walking

*Walking* a filesystem means recursively visiting a directory and any sub-directories. It is a fairly common requirement for copying, searching etc.

To walk a filesystem (or directory) you can construct a *Walker* object and use its methods to do the walking. Here's an example that prints the path to every Python file in your projects directory:

```
>>> from fs import open_fs
>>> from fs.walk import Walker
>>> home_fs = open_fs('~/projects')
>>> walker = Walker(filter=['*.py'])
>>> for path in walker.files(home_fs):
...     print(path)
```

Generally speaking, however, you will only need to construct a Walker object if you want to customize some behavior of the walking algorithm. This is because you can access the functionality of a Walker object via the `walk` attribute on FS objects. Here's an example:

```
>>> from fs import open_fs
>>> home_fs = open_fs('~/projects')
>>> for path in home_fs.walk.files(filter=['*.py']):
...     print(path)
```

Note that the `files` method above doesn't require a `fs` parameter. This is because the `walk` attribute is a property which returns a *BoundWalker* object, which associates the filesystem with a walker.

## 6.1 Walk Methods

If you call the `walk` attribute on a *BoundWalker* it will return an iterable of *Step* named tuples with three values; a path to the directory, a list of *Info* objects for directories, and a list of *Info* objects for the files. Here's an example:

```
for step in home_fs.walk(filter=['*.py']):
    print('In dir {}'.format(step.path))
```

(continues on next page)

(continued from previous page)

```
    print('sub-directories: {!r}'.format(step.dirs))
    print('files: {!r}'.format(step.files))
```

**Note:** Methods of *BoundWalker* invoke a corresponding method on a *Walker* object, with the *bound* filesystem.

The `walk` attribute may appear to be a method, but is in fact a callable object. It supports other convenient methods that supply different information from the walk. For instance, *files()*, which returns an iterable of file paths. Here's an example:

```
for path in home_fs.walk.files(filter=['*.py']):
    print('Python file: {}'.format(path))
```

The complement to `files` is *dirs()* which returns paths to just the directories (and ignoring the files). Here's an example:

```
for dir_path in home_fs.walk.dirs():
    print("{!r} contains sub-directory {}".format(home_fs, dir_path))
```

The *info()* method returns a generator of tuples containing a path and an *Info* object. You can use the `is_dir` attribute to know if the path refers to a directory or file. Here's an example:

```
for path, info in home_fs.walk.info():
    if info.is_dir:
        print("[dir] {}".format(path))
    else:
        print("[file] {}".format(path))
```

Finally, here's a nice example that counts the number of bytes of Python code in your home directory:

```
bytes_of_python = sum(
    info.size
    for info in home_fs.walk.info(namespaces=['details'])
    if not info.is_dir
)
```

## 6.2 Search Algorithms

There are two general algorithms for searching a directory tree. The first method is `"breadth"`, which yields resources in the top of the directory tree first, before moving on to sub-directories. The second is `"depth"` which yields the most deeply nested resources, and works backwards to the top-most directory.

Generally speaking, you will only need the a *depth* search if you will be deleting resources as you walk through them. The default *breadth* search is a generally more efficient way of looking through a filesystem. You can specify which method you want with the `search` parameter on most `Walker` methods.

CHAPTER 7

# Globbing

Globbing is the process of matching paths according to the rules used by the Unix shell.

Generally speaking, you can think of a glob pattern as a path containing one or more wildcard patterns, separated by forward slashes.

## 7.1 Matching Files and Directories

In a glob pattern, A `*` means match anything text in a filename. A `?` matches any single character. A `**` matches any number of subdirectories, making the glob *recursive*. If the glob pattern ends in a `/`, it will only match directory paths, otherwise it will match files and directories.

---

**Note:** A recursive glob requires that PyFilesystem scan a lot of files, and can potentially be slow for large (or network based) filesystems.

---

Here's a summary of glob patterns:

**`*`** Matches all files in the current directory.

**`*.py`** Matches all .py file in the current directory.

**`*.py?`** Matches all .py files and .pyi, .pyc etc in the currenct directory.

**`project/*.py`** Matches all .py files in a directory called `project`.

**`*/*.py`** Matches all .py files in any sub directory.

**`**/*.py`** Recursively matches all .py files.

**`**/.git/`** Recursively matches all the git directories.

## 7.2 Interface

PyFilesystem supports globbing via the `glob` attribute on every FS instance, which is an instance of *BoundGlobber*. Here's how you might use it to find all the Python files in your filesystem:

```
for match in my_fs.glob("**/*.py"):
    print(f"{match.path} is {match.info.size} bytes long")
```

Calling `.glob` with a pattern will return an iterator of *GlobMatch* named tuples for each matching file or directory. A glob match contains two attributes; `path` which is the full path in the filesystem, and `info` which is an *fs.info.Info* info object for the matched resource.

## 7.3 Batch Methods

In addition to iterating over the results, you can also call methods on the *Globber* which apply to every matched path.

For instance, here is how you can use glob to remove all `.pyc` files from a project directory:

```
>>> import fs
>>> fs.open_fs('~/projects/my_project').glob('**/*.pyc').remove()
29
```

CHAPTER 8

# Builtin Filesystems

## 8.1 App Filesystems

Manage filesystems in platform-specific application directories.

These classes abstract away the different requirements for user data across platforms, which vary in their conventions. They are all subclasses of *OSFS*.

**class** fs.appfs.**UserDataFS**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

A filesystem for per-user application data.

May also be opened with `open_fs('userdata://appname:author:version')`.

**__init__**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

Create a new application-specific filesystem.

**Parameters**

- **appname** (*str*) – The name of the application.
- **author** (*str*) – The name of the author (used on Windows).
- **version** (*str*) – Optional version string, if a unique location per version of the application is required.
- **roaming** (*bool*) – If `True`, use a *roaming* profile on Windows.
- **create** (*bool*) – If `True` (the default) the directory will be created if it does not exist.

**class** fs.appfs.**UserConfigFS**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

A filesystem for per-user config data.

May also be opened with `open_fs('userconf://appname:author:version')`.

**__init__**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

Create a new application-specific filesystem.

**Parameters**

- **appname** (*str*) – The name of the application.
- **author** (*str*) – The name of the author (used on Windows).
- **version** (*str*) – Optional version string, if a unique location per version of the application is required.
- **roaming** (*bool*) – If `True`, use a *roaming* profile on Windows.
- **create** (*bool*) – If `True` (the default) the directory will be created if it does not exist.

**class** fs.appfs.**SiteDataFS**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

A filesystem for application site data.

May also be opened with `open_fs('sitedata://appname:author:version')`.

**__init__**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

Create a new application-specific filesystem.

**Parameters**

- **appname** (*str*) – The name of the application.
- **author** (*str*) – The name of the author (used on Windows).
- **version** (*str*) – Optional version string, if a unique location per version of the application is required.
- **roaming** (*bool*) – If `True`, use a *roaming* profile on Windows.
- **create** (*bool*) – If `True` (the default) the directory will be created if it does not exist.

**class** fs.appfs.**SiteConfigFS**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

A filesystem for application config data.

May also be opened with `open_fs('siteconf://appname:author:version')`.

**__init__**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

Create a new application-specific filesystem.

**Parameters**

- **appname** (*str*) – The name of the application.
- **author** (*str*) – The name of the author (used on Windows).
- **version** (*str*) – Optional version string, if a unique location per version of the application is required.
- **roaming** (*bool*) – If `True`, use a *roaming* profile on Windows.
- **create** (*bool*) – If `True` (the default) the directory will be created if it does not exist.

**class** fs.appfs.**UserCacheFS**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

A filesystem for per-user application cache data.

May also be opened with `open_fs('usercache://appname:author:version')`.

**__init__**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

Create a new application-specific filesystem.

**Parameters**

- **appname** (*str*) – The name of the application.
- **author** (*str*) – The name of the author (used on Windows).

- **version** (*str*) – Optional version string, if a unique location per version of the application is required.
- **roaming** (*bool*) – If `True`, use a *roaming* profile on Windows.
- **create** (*bool*) – If `True` (the default) the directory will be created if it does not exist.

**class** fs.appfs.**UserLogFS**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

A filesystem for per-user application log data.

May also be opened with `open_fs('userlog://appname:author:version')`.

**__init__**(*appname*, *author=None*, *version=None*, *roaming=False*, *create=True*)

Create a new application-specific filesystem.

**Parameters**

- **appname** (*str*) – The name of the application.
- **author** (*str*) – The name of the author (used on Windows).
- **version** (*str*) – Optional version string, if a unique location per version of the application is required.
- **roaming** (*bool*) – If `True`, use a *roaming* profile on Windows.
- **create** (*bool*) – If `True` (the default) the directory will be created if it does not exist.

## 8.2 FTP Filesystem

Manage filesystems on remote FTP servers.

**class** fs.ftpfs.**FTPFS**(*host*, *user='anonymous'*, *passwd=''*, *acct=''*, *timeout=10*, *port=21*, *proxy=None*, *tls=False*)

A FTP (File Transport Protocol) Filesystem.

Optionally, the connection can be made securely via TLS. This is known as FTPS, or FTP Secure. TLS will be enabled when using the ftps:// protocol, or when setting the `tls` argument to True in the constructor.

### Examples

Create with the constructor:

```
>>> from fs.ftpfs import FTPFS
>>> ftp_fs = FTPFS("demo.wftpserver.com")
```

Or via an FS URL:

```
>>> ftp_fs = fs.open_fs('ftp://test.rebex.net')
```

Or via an FS URL, using TLS:

```
>>> ftp_fs = fs.open_fs('ftps://demo.wftpserver.com')
```

You can also use a non-anonymous username, and optionally a password, even within a FS URL:

```
>>> ftp_fs = FTPFS("test.rebex.net", user="demo", passwd="password")
>>> ftp_fs = fs.open_fs('ftp://demo:password@test.rebex.net')
```

Connecting via a proxy is supported. If using a FS URL, the proxy URL will need to be added as a URL parameter:

```
>>> ftp_fs = FTPFS("ftp.ebi.ac.uk", proxy="test.rebex.net")
>>> ftp_fs = fs.open_fs('ftp://ftp.ebi.ac.uk/?proxy=test.rebex.net')
```

**__init__**(*host*, *user='anonymous'*, *passwd=''*, *acct=''*, *timeout=10*, *port=21*, *proxy=None*, *tls=False*)

Create a new *FTPFS* instance.

**Parameters**

- **host** (*str*) – A FTP host, e.g. `'ftp.mirror.nl'`.
- **user** (*str*) – A username (default is `'anonymous'`).
- **passwd** (*str*) – Password for the server, or `None` for anon.
- **acct** (*str*) – FTP account.
- **timeout** (*int*) – Timeout for contacting server (in seconds, defaults to 10).
- **port** (*int*) – FTP port number (default 21).
- **proxy** (*str, optional*) – An FTP proxy, or `None` (default) for no proxy.
- **tls** (*bool*) – Attempt to use FTP over TLS (FTPS) (default: False)

**close**()

Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**create**(*path*, *wipe=False*)

Create an empty file.

The default behavior is to create a new file if one doesn't already exist. If `wipe` is `True`, any existing file will be truncated.

**Parameters**

- **path** (*str*) – Path to a new file in the filesystem.
- **wipe** (*bool*) – If `True`, truncate any existing file to 0 bytes (defaults to `False`).

**Returns** `True` if a new file had to be created.

**Return type** bool

**features**

Features of the remote FTP server.

**Type** dict

**ftp**
the underlying FTP client.

**Type** FTP

**ftp_url**
Get the FTP url this filesystem will open.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The "basic" namespace is alway included in the returned info, whatever the value of namespaces may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If path does not exist.

For more information regarding resource information, see *Resource Info*.

**getmeta**(*namespace='standard'*)
Get meta information regarding a filesystem.

**Parameters** **namespace** (*str*) – The meta namespace (defaults to "standard").

**Returns** the meta information.

**Return type** dict

Meta information is associated with a *namespace* which may be specified with the namespace parameter. The default namespace, "standard", contains common information regarding the filesystem's capabilities. Some filesystems may provide other namespaces which expose less common or implementation specific information. If a requested namespace is not supported by a filesystem, then an empty dictionary will be returned.

The "standard" namespace supports the following keys:

| key | Description |
|---|---|
| case_insensitive | True if this filesystem is case insensitive. |
| invalid_path_chars | A string containing the characters that may not be used on this filesystem. |
| max_path_length | Maximum number of characters permitted in a path, or None for no limit. |
| max_sys_path_length | Maximum number of characters permitted in a sys path, or None for no limit. |
| network | True if this filesystem requires a network. |
| read_only | True if this filesystem is read only. |
| supports_rename | True if this filesystem supports an os.rename operation. |

Most builtin filesystems will provide all these keys, and third- party filesystems should do so whenever possible, but a key may not be present if there is no way to know the value.

**Note:** Meta information is constant for the lifetime of the filesystem, and may be cached.

**getmodified**(*path*)
Get the timestamp of the last modifying access of a resource.

**Parameters** **path** (*str*) – A path to a resource.

**Returns** The timestamp of the last modification.

**Return type** datetime

The *modified timestamp* of a file is the point in time that the file was last changed. Depending on the file system, it might only have limited accuracy.

**geturl**(*path*, *purpose='download'*)
Get FTP url for resource.

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir**(*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**readbytes**(*path*)

Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.
- *fs.errors.ResourceNotFound* – if `path` does not exist.

**remove**(*path*)

Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)

Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**scandir**(*path*, *namespaces=None*, *page=None*)

Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.

- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setinfo**(*path*, *info*)

Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**supports_mdtm**

whether the server supports the MDTM feature.

**Type** bool

**supports_mlst**

whether the server supports MLST feature.

**Type** bool

**upload**(*path*, *file*, *chunk_size=None*, ***options*)

Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

#### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**writebytes**(*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** `TypeError` – if contents is not bytes.

## 8.3 Memory Filesystem

Manage a volatile in-memory filesystem.

**class** fs.memoryfs.**MemoryFS**
A filesystem that stored in memory.

Memory filesystems are useful for caches, temporary data stores, unit testing, etc. Since all the data is in memory, they are very fast, but non-permanent. The *MemoryFS* constructor takes no arguments.

#### Examples

Create with the constructor:

```
>>> from fs.memoryfs import MemoryFS
>>> mem_fs = MemoryFS()
```

Or via an FS URL:

```
>>> import fs
>>> mem_fs = fs.open_fs('mem://')
```

**__init__**()
Create an in-memory filesystem.

**close**()
Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**getinfo**(*path*, *namespaces=None*)

Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**listdir**(*path*)

Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir**(*path*, *permissions=None*, *recreate=False*)

Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.

- *fs.errors.ResourceNotFound* – If the path is not found.

**move**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Move a file from `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – A path on the filesystem to move.
- **dst_path** (*str*) – A path on the filesystem where the source file will be written to.
- **overwrite** (*bool*) – If `True`, destination path will be overwritten if it exists.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**Raises**

- *fs.errors.FileExpected* – If `src_path` maps to a directory instead of a file.
- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.
- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.

**movedir**(*src_path*, *dst_path*, *create=False*, *preserve_time=False*)
Move directory `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source directory on the filesystem.
- **dst_path** (*str*) – Path to destination directory.
- **create** (*bool*) – If `True`, then `dst_path` will be created if it doesn't exist already (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**Raises**

- *fs.errors.ResourceNotFound* – if `dst_path` does not exist, and `create` is `False`.
- *fs.errors.DirectoryExpected* – if `src_path` or one of its ancestors is not a directory.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**remove**(*path*)

Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)

Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**removetree**(*path*)

Recursively remove a directory and all its contents.

This method is similar to *removedir*, but will remove the contents of the directory if it is not empty.

**Parameters** **dir_path** (*str*) – Path to a directory on the filesystem.

**Raises**

- *fs.errors.ResourceNotFound* – If `dir_path` does not exist.
- *fs.errors.DirectoryExpected* – If `dir_path` is not a directory.

**Caution:** A filesystem should never delete its root folder, so `FS.removetree("/")` has different semantics: the contents of the root folder will be deleted, but the root will be untouched:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.removetree("/")
>>> home_fs.exists("/")
True
>>> home_fs.isempty("/")
True
```

Combined with *opendir*, this can be used to clear a directory without removing the directory itself:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.opendir("/Videos").removetree("/")
>>> home_fs.exists("/Videos")
True
>>> home_fs.isempty("/Videos")
True
```

**scandir** (*path*, *namespaces=None*, *page=None*)
Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setinfo** (*path*, *info*)
Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

## 8.4 Mount Filesystem

A Mount FS is a *virtual* filesystem which can seamlessly map sub-directories on to other filesystems.

For example, lets say we have two filesystems containing config files and resources respectively:

```
[config_fs]
|-- config.cfg
`-- defaults.cfg

[resources_fs]
|-- images
|   |-- logo.jpg
|   `-- photo.jpg
`-- data.dat
```

We can combine these filesystems in to a single filesystem with the following code:

```
from fs.mountfs import MountFS
combined_fs = MountFS()
combined_fs.mount('config', config_fs)
combined_fs.mount('resources', resources_fs)
```

This will create a filesystem where paths under `config/` map to `config_fs`, and paths under `resources/` map to `resources_fs`:

```
[combined_fs]
|-- config
|   |-- config.cfg
|   `-- defaults.cfg
`-- resources
    |-- images
    |   |-- logo.jpg
    |   `-- photo.jpg
    `-- data.dat
```

Now both filesystems may be accessed with the same path structure:

```
print(combined_fs.gettext('/config/defaults.cfg'))
read_jpg(combined_fs.open('/resources/images/logo.jpg', 'rb')
```

**class** fs.mountfs.**MountFS**(*auto_close=True*)

A virtual filesystem that maps directories on to other file-systems.

**__init__**(*auto_close=True*)

Create a new *MountFS* instance.

**Parameters** **auto_close** (*bool*) – If `True` (the default), the child filesystems will be closed when *MountFS* is closed.

**close**()

Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
```

(continues on next page)

(continued from previous page)

```
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**desc**(*path*)
Return a short descriptive text regarding a path.

**Parameters** **path** (*str*) – A path to a resource on the filesystem.

**Returns** a short description of the path.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**download**(*path*, *file*, *chunk_size=None*, ***options*)
Copy a file from the filesystem to a file-like object.

This may be more efficient that opening and copying files manually if the filesystem supplies an optimized method.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

**Parameters**

- **path** (*str*) – Path to a resource.
- **file** (*file-like*) – A file-like object open for writing in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

### Example

```
>>> with open('starwars.mov', 'wb') as write_file:
...     my_fs.download('/Videos/starwars.mov', write_file)
```

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**getsize**(*path*)
Get the size (in bytes) of a resource.

**Parameters** **path** (*str*) – A path to a resource.

**Returns** the *size* of the resource.

**Return type** int

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

The *size* of a file is the total number of readable bytes, which may not reflect the exact number of bytes of reserved disk space (or other storage medium).

The size of a directory is the number of bytes of overhead use to store the directory entry.

**getsyspath**(*path*)
Get the *system path* of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the *system path* of the resource, if any.

**Return type** str

**Raises** *fs.errors.NoSysPath* – If there is no corresponding system path.

A system path is one recognized by the OS, that may be used outside of PyFilesystem (in an application or a shell for example). This method will get the corresponding system path that would be referenced by `path`.

Not all filesystems have associated system paths. Network and memory based filesystems, for example, may not physically store data anywhere the OS knows about. It is also possible for some paths to have a system path, whereas others don't.

This method will always return a str on Py3.* and unicode on Py2.7. See `getospath` if you need to encode the path as bytes.

If `path` doesn't have a system path, a *NoSysPath* exception will be thrown.

---

**Note:** A filesystem may return a system path even if no resource is referenced by that path – as long as it can be certain what that system path would be.

---

**gettype**(*path*)
Get the type of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the type of the resource.

**Return type** *ResourceType*

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

A type of a resource is an integer that identifies the what the resource references. The standard type integers may be one of the values in the *ResourceType* enumerations.

The most common resource types, supported by virtually all filesystems are `directory` (1) and `file` (2), but the following types are also possible:

| ResourceType | value |
|---|---|
| unknown | 0 |
| directory | 1 |
| file | 2 |
| character | 3 |
| block_special_file | 4 |
| fifo | 5 |
| socket | 6 |
| symlink | 7 |

Standard resource types are positive integers, negative values are reserved for implementation specific resource types.

**geturl** (*path*, *purpose='download'*)
Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, `OSFS` supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

**hasurl** (*path*, *purpose='download'*)
Check if a path has a corresponding URL.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **purpose** (*str*) – A purpose parameter, as given in *geturl*.

**Returns** `True` if an URL for the given purpose exists.

**Return type** bool

**isdir** (*path*)
Check if a path maps to an existing directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a directory.

**Return type** bool

**isfile** (*path*)
Check if a path maps to an existing file.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a file.

**Return type** bool

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in `ResourceType`.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir**(*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (`Permissions`, *optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**mount**(*path*, *fs*)
Mounts a host FS object on a given path.

**Parameters**

- **path** (*str*) – A path within the MountFS.
- **fs** (`FS or str`) – A filesystem (instance or URL) to mount.

**open**(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=''*, ***options*)
Open a file.

**Parameters**

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open the file object with (defaults to *r*).
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, 1 to select line buffering, of any positive integer to indicate a buffer size).
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`)
- **errors** (*str*, *optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.

- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If the path is not a file.
- *fs.errors.FileExists* – If the file exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If the path does not exist.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***kwargs*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**readbytes**(*path*)
Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.
- *fs.errors.ResourceNotFound* – if `path` does not exist.

**readtext**(*path*, *encoding=None*, *errors=None*, *newline=''*)
Get the contents of a file as a string.

**Parameters**

- **path** (*str*) – A path to a readable file on the filesystem.

- **encoding** (*str, optional*) – Encoding to use when reading contents in text mode (defaults to None, reading in binary mode).
- **errors** (*str, optional*) – Unicode errors parameter.
- **newline** (*str*) – Newlines parameter.

**Returns** file contents.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**remove**(*path*)
Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)
Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**scandir**(*path*, *namespaces=None*, *page=None*)
Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or None to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setinfo**(*path*, *info*)
Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**upload**(*path*, *file*, *chunk_size=None*, ***options*)
Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**validatepath**(*path*)
Validate a path, returning a normalized absolute path on sucess.

Many filesystems have restrictions on the format of paths they support. This method will check that `path` is valid on the underlaying storage mechanism and throw a *InvalidPath* exception if it is not.

**Parameters** **path** (*str*) – A path.

**Returns** A normalized, absolute path.

**Return type** str

**Raises**

- *fs.errors.InvalidPath* – If the path is invalid.

- *fs.errors.FilesystemClosed* – if the filesystem is closed.
- *fs.errors.InvalidCharsInPath* – If the path contains invalid characters.

**writebytes**(*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** TypeError – if contents is not bytes.

**writetext**(*path*, *contents*, *encoding='utf-8'*, *errors=None*, *newline=''*)
Create or replace a file with text.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*str*) – Text to be written.
- **encoding** (*str, optional*) – Encoding of destination file (defaults to 'utf-8').
- **errors** (*str, optional*) – How encoding errors should be treated (same as io.open).
- **newline** (*str*) – Newline parameter (same as io.open).

**Raises** TypeError – if contents is not a unicode string.

**exception** fs.mountfs.**MountError**
Bases: Exception

Thrown when mounts conflict.

**__init__**
Initialize self. See help(type(self)) for accurate signature.

## 8.5 Multi Filesystem

A MultiFS is a filesystem composed of a sequence of other filesystems, where the directory structure of each overlays the previous filesystem in the sequence.

One use for such a filesystem would be to selectively override a set of files, to customize behavior. For example, to create a filesystem that could be used to *theme* a web application. We start with the following directories:

```
`-- templates
    |-- snippets
    |   `-- panel.html
    |-- index.html
    |-- profile.html
    `-- base.html

`-- theme
    |-- snippets
    |   |-- widget.html
    |   `-- extra.html
    |-- index.html
    `-- theme.html
```

And we want to create a single filesystem that will load a file from `templates/` only if it isn't found in `theme/`. Here's how we could do that:

```
from fs.osfs import OSFS
from fs.multifs import MultiFS

theme_fs = MultiFS()
theme_fs.add_fs('templates', OSFS('templates'))
theme_fs.add_fs('theme', OSFS('theme'))
```

Now we have a `theme_fs` filesystem that presents a single view of both directories:

```
|-- snippets
|   |-- panel.html
|   |-- widget.html
|   `-- extra.html
|-- index.html
|-- profile.html
|-- base.html
`-- theme.html
```

**class** fs.multifs.**MultiFS**(*auto_close=True*)

A filesystem that delegates to a sequence of other filesystems.

Operations on the MultiFS will try each 'child' filesystem in order, until it succeeds. In effect, creating a filesystem that combines the files and dirs of its children.

**__init__**(*auto_close=True*)

Create a new MultiFS.

**Parameters** **auto_close** (*bool*) – If `True` (the default), the child filesystems will be closed when *MultiFS* is closed.

**add_fs**(*name*, *fs*, *write=False*, *priority=0*)

Add a filesystem to the MultiFS.

**Parameters**

- **name** (*str*) – A unique name to refer to the filesystem being added.
- **fs** (*FS or str*) – The filesystem (instance or URL) to add.
- **write** (*bool*) – If this value is True, then the `fs` will be used as the writeable FS (defaults to False).
- **priority** (*int*) – An integer that denotes the priority of the filesystem being added. Filesystems will be searched in descending priority order and then by the reverse order they were added. So by default, the most recently added filesystem will be looked at first.

**close**()

Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**download**(*path*, *file*, *chunk_size=None*, ***options*)
Copy a file from the filesystem to a file-like object.

This may be more efficient that opening and copying files manually if the filesystem supplies an optimized method.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

**Parameters**

- **path** (*str*) – Path to a resource.
- **file** (*file-like*) – A file-like object open for writing in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

### Example

```
>>> with open('starwars.mov', 'wb') as write_file:
...     my_fs.download('/Videos/starwars.mov', write_file)
```

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

**get_fs**(*name*)
Get a filesystem from its name.

**Parameters** **name** (*str*) – The name of a filesystem previously added.

**Returns** the filesystem added as `name` previously.

**Return type** *FS*

**Raises** `KeyError` – If no filesystem with given `name` could be found.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**getsize**(*path*)
Get the size (in bytes) of a resource.

**Parameters** **path** (*str*) – A path to a resource.

**Returns** the *size* of the resource.

**Return type** int

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

The *size* of a file is the total number of readable bytes, which may not reflect the exact number of bytes of reserved disk space (or other storage medium).

The size of a directory is the number of bytes of overhead use to store the directory entry.

**getsyspath**(*path*)
Get the *system path* of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the *system path* of the resource, if any.

**Return type** str

**Raises** *fs.errors.NoSysPath* – If there is no corresponding system path.

A system path is one recognized by the OS, that may be used outside of PyFilesystem (in an application or a shell for example). This method will get the corresponding system path that would be referenced by `path`.

Not all filesystems have associated system paths. Network and memory based filesystems, for example, may not physically store data anywhere the OS knows about. It is also possible for some paths to have a system path, whereas others don't.

This method will always return a str on Py3.* and unicode on Py2.7. See `getospath` if you need to encode the path as bytes.

If `path` doesn't have a system path, a *NoSysPath* exception will be thrown.

---

**Note:** A filesystem may return a system path even if no resource is referenced by that path – as long as it can be certain what that system path would be.

---

**gettype**(*path*)
Get the type of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the type of the resource.

**Return type** *ResourceType*

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

A type of a resource is an integer that identifies the what the resource references. The standard type integers may be one of the values in the *ResourceType* enumerations.

The most common resource types, supported by virtually all filesystems are `directory` (1) and `file` (2), but the following types are also possible:

| ResourceType | value |
|---|---|
| unknown | 0 |
| directory | 1 |
| file | 2 |
| character | 3 |
| block_special_file | 4 |
| fifo | 5 |
| socket | 6 |
| symlink | 7 |

Standard resource types are positive integers, negative values are reserved for implementation specific resource types.

**geturl** (*path*, *purpose='download'*)
Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, `OSFS` supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

**hassyspath** (*path*)
Check if a path maps to a system path.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if the resource at `path` has a *syspath*.

**Return type** bool

**hasurl** (*path*, *purpose='download'*)
Check if a path has a corresponding URL.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **purpose** (*str*) – A purpose parameter, as given in *geturl*.

**Returns** `True` if an URL for the given purpose exists.

**Return type** bool

**isdir** (*path*)
Check if a path maps to an existing directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a directory.

**Return type** bool

**isfile** (*path*)
Check if a path maps to an existing file.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a file.

**Return type** bool

**iterate_fs**()
Get iterator that returns (name, fs) in priority order.

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir**(*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (`Permissions`, *optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**makedirs**(*path*, *permissions=None*, *recreate=False*)
Make a directory, and any missing intermediate directories.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (`Permissions`, *optional*) – Initial permissions, or `None` to use defaults.
- **recreate** (*bool*) – If `False` (the default), attempting to create an existing directory will raise an error. Set to `True` to ignore existing directories.

**Returns** A sub-directory filesystem.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – if the path is already a directory, and `recreate` is `False`.
- *fs.errors.DirectoryExpected* – if one of the ancestors in the path is not a directory.

**open**(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=''*, ***kwargs*)
Open a file.

**Parameters**

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open the file object with (defaults to *r*).
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, 1 to select line buffering, of any positive integer to indicate a buffer size).
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`)
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If the path is not a file.
- *fs.errors.FileExists* – If the file exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If the path does not exist.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).

- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**readbytes**(*path*)
Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.
- *fs.errors.ResourceNotFound* – if `path` does not exist.

**readtext**(*path*, *encoding=None*, *errors=None*, *newline=''*)
Get the contents of a file as a string.

**Parameters**

- **path** (*str*) – A path to a readable file on the filesystem.
- **encoding** (*str, optional*) – Encoding to use when reading contents in text mode (defaults to `None`, reading in binary mode).
- **errors** (*str, optional*) – Unicode errors parameter.
- **newline** (*str*) – Newlines parameter.

**Returns** file contents.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**remove**(*path*)
Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)
Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**scandir**(*path*, *namespaces=None*, *page=None*)
Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setinfo**(*path*, *info*)

Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**upload**(*path*, *file*, *chunk_size=None*, ***options*)

Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**validatepath** (*path*)
Validate a path, returning a normalized absolute path on sucess.

Many filesystems have restrictions on the format of paths they support. This method will check that `path` is valid on the underlaying storage mechanism and throw a *InvalidPath* exception if it is not.

**Parameters** **path** (*str*) – A path.

**Returns** A normalized, absolute path.

**Return type** str

**Raises**

- *fs.errors.InvalidPath* – If the path is invalid.
- *fs.errors.FilesystemClosed* – if the filesystem is closed.
- *fs.errors.InvalidCharsInPath* – If the path contains invalid characters.

**which** (*path*, *mode='r'*)
Get a tuple of (name, fs) that the given path would map to.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – An `io.open` mode.

**writebytes** (*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** `TypeError` – if contents is not bytes.

**writetext** (*path*, *contents*, *encoding='utf-8'*, *errors=None*, *newline=''*)
Create or replace a file with text.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*str*) – Text to be written.
- **encoding** (*str, optional*) – Encoding of destination file (defaults to `'utf-8'`).
- **errors** (*str, optional*) – How encoding errors should be treated (same as `io.open`).
- **newline** (*str*) – Newline parameter (same as `io.open`).

**Raises** `TypeError` – if `contents` is not a unicode string.

## 8.6 OS Filesystem

Manage the filesystem provided by your OS.

In essence, an *OSFS* is a thin layer over the `io` and `os` modules of the Python standard library.

**class** `fs.osfs.`**OSFS**(*root_path*, *create=False*, *create_mode=511*, *expand_vars=True*)
Create an OSFS.

### Examples

```
>>> current_directory_fs = OSFS('.')
>>> home_fs = OSFS('~/')
>>> windows_system32_fs = OSFS('c://system32')
```

**__init__**(*root_path*, *create=False*, *create_mode=511*, *expand_vars=True*)
Create an OSFS instance.

**Parameters**

- **root_path** (*str or PathLike*) – An OS path or path-like object to the location on your HD you wish to manage.
- **create** (*bool*) – Set to `True` to create the root directory if it does not already exist, otherwise the directory should exist prior to creating the `OSFS` instance (defaults to `False`).
- **create_mode** (*int*) – The permissions that will be used to create the directory if `create` is `True` and the path doesn't exist, defaults to `0o777`.
- **expand_vars** (*bool*) – If `True` (the default) environment variables of the form `~`, `$name` or `${name}` will be expanded.

**Raises** *fs.errors.CreateFailed* – If `root_path` does not exist, or could not be created.

**copy**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Copy file contents from `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source file.
- **dst_path** (*str*) – Path to destination file.
- **overwrite** (*bool*) – If `True`, overwrite the destination file if it exists (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Raises**

- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.
- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.
- *fs.errors.FileExpected* – If `src_path` is not a file.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**getsyspath**(*path*)

Get the *system path* of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the *system path* of the resource, if any.

**Return type** str

**Raises** *fs.errors.NoSysPath* – If there is no corresponding system path.

A system path is one recognized by the OS, that may be used outside of PyFilesystem (in an application or a shell for example). This method will get the corresponding system path that would be referenced by `path`.

Not all filesystems have associated system paths. Network and memory based filesystems, for example, may not physically store data anywhere the OS knows about. It is also possible for some paths to have a system path, whereas others don't.

This method will always return a str on Py3.* and unicode on Py2.7. See `getospath` if you need to encode the path as bytes.

If `path` doesn't have a system path, a *NoSysPath* exception will be thrown.

---

**Note:** A filesystem may return a system path even if no resource is referenced by that path – as long as it can be certain what that system path would be.

---

**gettype**(*path*)

Get the type of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the type of the resource.

**Return type** *ResourceType*

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

A type of a resource is an integer that identifies the what the resource references. The standard type integers may be one of the values in the *ResourceType* enumerations.

The most common resource types, supported by virtually all filesystems are `directory` (1) and `file` (2), but the following types are also possible:

| ResourceType | value |
|---|---|
| unknown | 0 |
| directory | 1 |
| file | 2 |
| character | 3 |
| block_special_file | 4 |
| fifo | 5 |
| socket | 6 |
| symlink | 7 |

Standard resource types are positive integers, negative values are reserved for implementation specific resource types.

**geturl** (*path*, *purpose='download'*)
Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, *OSFS* supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

**islink** (*path*)
Check if a path maps to a symlink.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a symlink.

**Return type** bool

**listdir** (*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir** (*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.

- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**open**(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=''*, *line_buffering=False*, ***options*)

Open a file.

**Parameters**

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open the file object with (defaults to *r*).
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, 1 to select line buffering, of any positive integer to indicate a buffer size).
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`)
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If the path is not a file.
- *fs.errors.FileExists* – If the file exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If the path does not exist.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)

Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**remove**(*path*)

Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)

Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**scandir**(*path*, *namespaces=None*, *page=None*)

Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setinfo** (*path*, *info*)
Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**validatepath** (*path*)
Check path may be encoded, in addition to usual checks.

## 8.7 Sub Filesystem

Manage a directory in a *parent* filesystem.

**class** fs.subfs.**SubFS** (*parent_fs*, *path*)
A sub-directory on a parent filesystem.

A SubFS is a filesystem object that maps to a sub-directory of another filesystem. This is the object that is returned by *opendir*.

**__init__** (*parent_fs*, *path*)
Create a filesystem. See help(type(self)) for accurate signature.

**delegate_fs** ()
Get the proxied filesystem.

This method should return a filesystem for methods not associated with a path, e.g. *getmeta*.

**delegate_path** (*path*)
Encode a path for proxied filesystem.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** a tuple of `(<filesystem>, <new_path>)`

**Return type** (*FS*, str)

**class** fs.subfs.**ClosingSubFS** (*parent_fs*, *path*)
A version of *SubFS* which closes its parent when closed.

**close** ()
Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly

(it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

#### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

## 8.8 Tar Filesystem

Manage the filesystem in a Tar archive.

**class** fs.tarfs.**TarFS**(*wrap_fs*)

Read and write tar files.

There are two ways to open a *TarFS* for the use cases of reading a tar file, and creating a new one.

If you open the *TarFS* with `write` set to `False` (the default), then the filesystem will be a read only filesystem which maps to the files and directories within the tar file. Files are decompressed on the fly when you open them.

Here's how you might extract and print a readme from a tar file:

```
with TarFS('foo.tar.gz') as tar_fs:
    readme = tar_fs.readtext('readme.txt')
```

If you open the TarFS with `write` set to `True`, then the *TarFS* will be a empty temporary filesystem. Any files / directories you create in the *TarFS* will be written in to a tar file when the *TarFS* is closed. The compression is set from the new file name but may be set manually with the `compression` argument.

Here's how you might write a new tar file containing a readme.txt file:

```
with TarFS('foo.tar.xz', write=True) as new_tar:
    new_tar.writetext(
        'readme.txt',
        'This tar file was written by PyFilesystem'
    )
```

**Parameters**

- **file** (*str or io.IOBase*) – An OS filename, or an open file handle.
- **write** (*bool*) – Set to `True` to write a new tar file, or use default (`False`) to read an existing tar file.
- **compression** (*str, optional*) – Compression to use (one of the formats supported by `tarfile`: `xz`, `gz`, `bz2`, or `None`).
- **temp_fs** (*str*) – An FS URL or an FS instance to use to store data prior to tarring. Defaults to creating a new *TempFS*.

**class** fs.tarfs.**WriteTarFS**(*file*, *compression=None*, *encoding='utf-8'*, *temp_fs='temp://__tartemp__'*)

A writable tar file.

**__init__**(*file*, *compression=None*, *encoding='utf-8'*, *temp_fs='temp://__tartemp__'*)

Create a filesystem. See help(type(self)) for accurate signature.

**delegate_path**(*path*)

Encode a path for proxied filesystem.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** a tuple of `(<filesystem>, <new_path>)`

**Return type** (*FS*, str)

**delegate_fs**()

Get the proxied filesystem.

This method should return a filesystem for methods not associated with a path, e.g. *getmeta*.

**close**()

Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**write_tar**(*file=None*, *compression=None*, *encoding=None*)

Write tar to a file.

**Parameters**

- **file** (*str or io.IOBase, optional*) – Destination file, may be a file name or an open file object.
- **compression** (*str, optional*) – Compression to use (one of the constants defined in `tarfile` in the stdlib).
- **encoding** (*str, optional*) – The character encoding to use (default uses the encoding defined in *__init__*).

---

**Note:** This is called automatically when the TarFS is closed.

---

**class** fs.tarfs.**ReadTarFS**(*file*, *encoding='utf-8'*)

A readable tar file.

**__init__**(*file*, *encoding='utf-8'*)

Create a filesystem. See help(type(self)) for accurate signature.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**isdir**(*path*)
Check if a path maps to an existing directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a directory.

**Return type** bool

**isfile**(*path*)
Check if a path maps to an existing file.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a file.

**Return type** bool

**setinfo**(*path*, *info*)
Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir**(*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**remove**(*path*)
Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)
Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**close**()
Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**isclosed**()
Check if the filesystem is closed.

**geturl**(*path*, *purpose='download'*)
Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, `OSFS` supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

## 8.9 Temporary Filesystem

Manage filesystems in temporary locations.

A temporary filesytem is stored in a location defined by your OS (`/tmp` on linux). The contents are deleted when the filesystem is closed.

A *TempFS* is a good way of preparing a directory structure in advance, that you can later copy. It can also be used as a temporary data store.

**class** fs.tempfs.**TempFS**(*identifier='__tempfs__'*, *temp_dir=None*, *auto_clean=True*, *ignore_clean_errors=True*)

A temporary filesystem on the OS.

Temporary filesystems are created using the `tempfile.mkdtemp` function to obtain a temporary folder in an OS-specific location. You can provide an alternative location with the `temp_dir` argument of the constructor.

### Examples

Create with the constructor:

```
>>> from fs.tempfs import TempFS
>>> tmp_fs = TempFS()
```

Or via an FS URL:

```
>>> import fs
>>> tmp_fs = fs.open_fs("temp://")
```

Use a specific identifier for the temporary folder to better illustrate its purpose:

```
>>> named_tmp_fs = fs.open_fs("temp://local_copy")
>>> named_tmp_fs = TempFS(identifier="local_copy")
```

**__init__**(*identifier='__tempfs__'*, *temp_dir=None*, *auto_clean=True*, *ignore_clean_errors=True*)

Create a new *TempFS* instance.

**Parameters**

- **identifier** (*str*) – A string to distinguish the directory within the OS temp location, used as part of the directory name.
- **temp_dir** (*str, optional*) – An OS path to your temp directory (leave as `None` to auto-detect).
- **auto_clean** (*bool*) – If `True` (the default), the directory contents will be wiped on close.
- **ignore_clean_errors** (*bool*) – If `True` (the default), any errors in the clean process will be suppressed. If `False`, they will be raised.

**clean**()

Clean (delete) temporary files created by this filesystem.

**close**()

Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly

(it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

---

**Hint:** Depending on the value of `auto_clean` passed when creating the *TempFS*, the underlying temporary folder may be removed or not.

---

#### Example

```
>>> tmp_fs = TempFS(auto_clean=False)
>>> syspath = tmp_fs.getsyspath("/")
>>> tmp_fs.close()
>>> os.path.exists(syspath)
True
```

## 8.10 Zip Filesystem

Manage the filesystem in a Zip archive.

**class** fs.zipfs.**ZipFS**(*wrap_fs*)

Read and write zip files.

There are two ways to open a *ZipFS* for the use cases of reading a zip file, and creating a new one.

If you open the *ZipFS* with `write` set to `False` (the default) then the filesystem will be a read-only filesystem which maps to the files and directories within the zip file. Files are decompressed on the fly when you open them.

Here's how you might extract and print a readme from a zip file:

```
with ZipFS('foo.zip') as zip_fs:
    readme = zip_fs.readtext('readme.txt')
```

If you open the *ZipFS* with `write` set to `True`, then the *ZipFS* will be an empty temporary filesystem. Any files / directories you create in the *ZipFS* will be written in to a zip file when the *ZipFS* is closed.

Here's how you might write a new zip file containing a `readme.txt` file:

```
with ZipFS('foo.zip', write=True) as new_zip:
    new_zip.writetext(
        'readme.txt',
        'This zip file was written by PyFilesystem'
    )
```

**Parameters**

- **file** (*str or io.IOBase*) – An OS filename, or an open file object.
- **write** (*bool*) – Set to `True` to write a new zip file, or `False` (default) to read an existing zip file.
- **compression** (*int*) – Compression to use (one of the constants defined in the `zipfile` module in the stdlib).
- **temp_fs** (*str or FS*) – An FS URL or an FS instance to use to store data prior to zipping. Defaults to creating a new *TempFS*.

**class** fs.zipfs.**WriteZipFS**(*file*, *compression=8*, *encoding='utf-8'*, *temp_fs='temp://__ziptemp__'*)
A writable zip file.

**__init__**(*file*, *compression=8*, *encoding='utf-8'*, *temp_fs='temp://__ziptemp__'*)
Create a filesystem. See help(type(self)) for accurate signature.

**delegate_path**(*path*)
Encode a path for proxied filesystem.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** a tuple of `(<filesystem>, <new_path>)`

**Return type** (*FS*, str)

**delegate_fs**()
Get the proxied filesystem.

This method should return a filesystem for methods not associated with a path, e.g. *getmeta*.

**close**()
Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**write_zip**(*file=None*, *compression=None*, *encoding=None*)
Write zip to a file.

**Parameters**

- **file** (*str or io.IOBase, optional*) – Destination file, may be a file name or an open file handle.
- **compression** (*int, optional*) – Compression to use (one of the constants defined in the zipfile module in the stdlib).
- **encoding** (*str, optional*) – The character encoding to use (default uses the encoding defined in *__init__*).

---

**Note:** This is called automatically when the ZipFS is closed.

---

**class** fs.zipfs.**ReadZipFS**(*file*, *encoding='utf-8'*)
A readable zip file.

**__init__**(*file*, *encoding='utf-8'*)
Create a filesystem. See help(type(self)) for accurate signature.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**setinfo**(*path*, *info*)
Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**makedir**(*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.

- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***kwargs*)

Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**remove**(*path*)

Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)

Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.

- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**close**()

Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**readbytes**(*path*)

Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.
- *fs.errors.ResourceNotFound* – if `path` does not exist.

**geturl**(*path*, *purpose='download'*)

Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, `OSFS` supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

CHAPTER 9

# Implementing Filesystems

With a little care, you can implement a PyFilesystem interface for any filesystem, which will allow it to work interchangeably with any of the built-in FS classes and tools.

To create a PyFilesystem interface, derive a class from *FS* and implement the *Essential Methods*. This should give you a working FS class.

Take care to copy the method signatures *exactly*, including default values. It is also essential that you follow the same logic with regards to exceptions, and only raise exceptions in *errors*.

## 9.1 Constructor

There are no particular requirements regarding how a PyFilesystem class is constructed, but be sure to call the base class `__init__` method with no parameters.

## 9.2 Thread Safety

All Filesystems should be *thread-safe*. The simplest way to achieve that is by using the `_lock` attribute supplied by the *FS* constructor. This is a `RLock` object from the standard library, which you can use as a context manager, so methods you implement will start something like this:

```
with self._lock:
    do_something()
```

You aren't *required* to use `_lock`. Just as long as calling methods on the FS object from multiple threads doesn't break anything.

## 9.3 Python Versions

PyFilesystem supports Python2.7 and Python3.X. The differences between the two major Python versions are largely managed by the `six` library.

You aren't obligated to support the same versions of Python that PyFilesystem itself supports, but it is recommended if your project is for general use.

## 9.4 Testing Filesystems

To test your implementation, you can borrow the test suite used to test the built in filesystems. If your code passes these tests, then you can be confident your implementation will work seamlessly.

Here's the simplest possible example to test a filesystem class called `MyFS`:

```
import unittest
from fs.test import FSTestCases

class TestMyFS(FSTestCases, unittest.TestCase):

    def make_fs(self):
        # Return an instance of your FS object here
        return MyFS()
```

You may also want to override some of the methods in the test suite for more targeted testing:

**class** fs.test.**FSTestCases**
Basic FS tests.

**assert_bytes**(*path*, *contents*)
Assert a file contains the given bytes.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **contents** (*bytes*) – Bytes to compare.

**assert_exists**(*path*)
Assert a path exists.

**Parameters** **path** (*str*) – A path on the filesystem.

**assert_isdir**(*path*)
Assert a path is a directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**assert_isempty**(*path*)
Assert a path is an empty directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**assert_isfile**(*path*)
Assert a path is a file.

**Parameters** **path** (*str*) – A path on the filesystem.

**assert_not_exists**(*path*)
Assert a path does not exist.

**Parameters** **path** (*str*) – A path on the filesystem.

**assert_text**(*path*, *contents*)
Assert a file contains the given text.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **contents** (*str*) – Text to compare.

**destroy_fs**(*fs*)
Destroy a FS instance.

**Parameters** **fs** (FS) – A filesystem instance previously opened by *make_fs*.

**make_fs**()
Return an FS instance.

**test_geturl_purpose**()
Check an unknown purpose raises a NoURL error.

**test_validatepath**()
Check validatepath returns an absolute path.

---

**Note:** As of version 2.4.11 this project uses pytest to run its tests. While it's completely compatible with `unittest`-style tests, it's much more powerful and feature-rich. We suggest you take advantage of it and its plugins in new tests you write, rather than sticking to strict `unittest` features. For benefits and limitations, see here.

---

## 9.5 Essential Methods

The following methods MUST be implemented in a PyFilesystem interface.

- *getinfo()* Get info regarding a file or directory.
- *listdir()* Get a list of resources in a directory.
- *makedir()* Make a directory.
- *openbin()* Open a binary file.
- *remove()* Remove a file.
- *removedir()* Remove a directory.
- *setinfo()* Set resource information.

## 9.6 Non - Essential Methods

The following methods MAY be implemented in a PyFilesystem interface.

These methods have a default implementation in the base class, but may be overridden if you can supply a more optimal version.

Exactly which methods you should implement depends on how and where the data is stored. For network filesystems, a good candidate to implement, is the `scandir` method which would otherwise call a combination of `listdir` and `getinfo` for each file.

In the general case, it is a good idea to look at how these methods are implemented in *FS*, and only write a custom version if it would be more efficient than the default.

- *appendbytes()*
- *appendtext()*
- *close()*
- *copy()*
- *copydir()*
- *create()*
- *desc()*
- *download()*
- *exists()*
- *filterdir()*
- *getmeta()*
- *getospath()*
- *getsize()*
- *getsyspath()*
- *gettype()*
- *geturl()*
- *hassyspath()*
- *hasurl()*
- *isclosed()*
- *isempty()*
- *isdir()*
- *isfile()*
- *islink()*
- *lock()*
- *makedirs()*
- *move()*
- *movedir()*
- *open()*
- *opendir()*
- *readbytes()*
- *readtext()*
- *removetree()*
- *scandir()*
- *settimes()*

- *touch()*
- *upload()*
- *validatepath()*
- *writebytes()*
- *writefile()*
- *writetext()*

## 9.7 Helper Methods

These methods SHOULD NOT be implemented.

Implementing these is highly unlikely to be worthwhile.

- *check()*
- *getbasic()*
- *getdetails()*
- *hash()*
- *match()*
- *tree()*

# CHAPTER 10

# Creating an extension

Once a filesystem has been implemented, it can be integrated with other applications and projects using PyFilesystem.

## 10.1 Naming Convention

For visibility in PyPi, we recommend that your package be prefixed with `fs-`. For instance if you have implemented an `AwesomeFS` PyFilesystem class, your packaged could be be named `fs-awesome` or `fs-awesomefs`.

## 10.2 Opener

In order for your filesystem to be opened with an *FS URL* you should define an *Opener* class.

Here's an example taken from an Amazon S3 Filesystem:

```
"""Defines the S3FSOpener."""

__all__ = ['S3FSOpener']

from fs.opener import Opener
from fs.opener.errors import OpenerError

from ._s3fs import S3FS


class S3FSOpener(Opener):
    protocols = ['s3']

    def open_fs(self, fs_url, parse_result, writeable, create, cwd):
        bucket_name, _, dir_path = parse_result.resource.partition('/')
        if not bucket_name:
            raise OpenerError(
```

(continues on next page)

(continued from previous page)

```
                "invalid bucket name in '{}'".format(fs_url)
            )
        s3fs = S3FS(
            bucket_name,
            dir_path=dir_path or '/',
            aws_access_key_id=parse_result.username or None,
            aws_secret_access_key=parse_result.password or None,
        )
        return s3fs
```

By convention this would be defined in `opener.py`.

To register the opener you will need to define an entry point in your setup.py. See below for an example.

## 10.3 The setup.py file

Refer to the setuptools documentation to see how to write a `setup.py` file. There are only a few things that should be kept in mind when creating a Pyfilesystem2 extension. Make sure that:

- `fs` is in the `install_requires` list. You should reference the version number with the ~= operator which ensures that the install will get any bugfix releases of PyFilesystem but not any potentially breaking changes.
- Ìf you created an opener, include it as an `fs.opener` entry point, using the name of the entry point as the protocol to be used.

Here is an minimal `setup.py` for our project:

```
from setuptools import setup
setup(
    name='fs-awesomefs',  # Name in PyPi
    author="You !",
    author_email="your.email@domain.ext",
    description="An awesome filesystem for pyfilesystem2 !",
    install_requires=[
        "fs~=2.0.5"
    ],
    entry_points = {
        'fs.opener': [
            'awe = awesomefs.opener:AwesomeFSOpener',
        ]
    },
    license="MY LICENSE",
    packages=['awesomefs'],
    version="X.Y.Z",
)
```

## 10.4 Good Practices

Keep track of your achievements! Add the following values to your `__init__.py`:

- `__version__` The version of the extension (we recommend following Semantic Versioning),
- `__author__` Your name(s).
- `__author_email__` Your email(s).

- `__license__` The module's license.

## 10.5 Let us Know

Contact us to add your filesystem to the PyFilesystem Wiki.

## 10.6 Live Example

See fs.sshfs for a functioning PyFilesystem2 extension implementing a Pyfilesystem2 filesystem over SSH.

CHAPTER 11

# External Filesystems

See the following wiki page for a list of filesystems not in the core library, and community contributed filesystems.

https://www.pyfilesystem.org/page/index-of-filesystems/

If you have developed a filesystem that you would like added to the above page, please let us know by opening a Github issue.

# CHAPTER 12

# PyFilesystem API

The following is a complete list of methods on PyFilesystem objects.

- *appendbytes()* Append bytes to a file.
- *appendtext()* Append text to a file.
- *check()* Check if a filesystem is open or raise error.
- *close()* Close the filesystem.
- *copy()* Copy a file to another location.
- *copydir()* Copy a directory to another location.
- *create()* Create or truncate a file.
- *desc()* Get a description of a resource.
- *download()* Copy a file on the filesystem to a file-like object.
- *exists()* Check if a path exists.
- *filterdir()* Iterate resources, filtering by wildcard(s).
- *getbasic()* Get basic info namespace for a resource.
- *getdetails()* Get details info namespace for a resource.
- *getinfo()* Get info regarding a file or directory.
- *getmeta()* Get meta information for a resource.
- *getmodified()* Get the last modified time of a resource.
- *getospath()* Get path with encoding expected by the OS.
- *getsize()* Get the size of a file.
- *getsyspath()* Get the system path of a resource, if one exists.
- *gettype()* Get the type of a resource.
- *geturl()* Get a URL to a resource, if one exists.

- *hassyspath()* Check if a resource maps to the OS filesystem.
- *hash()* Get the hash of a file's contents.
- *hasurl()* Check if a resource has a URL.
- *isclosed()* Check if the filesystem is closed.
- *isempty()* Check if a directory is empty.
- *isdir()* Check if path maps to a directory.
- *isfile()* Check if path maps to a file.
- *islink()* Check if path is a link.
- *listdir()* Get a list of resources in a directory.
- *lock()* Get a thread lock context manager.
- *makedir()* Make a directory.
- *makedirs()* Make a directory and intermediate directories.
- *match()* Match one or more wildcard patterns against a path.
- *move()* Move a file to another location.
- *movedir()* Move a directory to another location.
- *open()* Open a file on the filesystem.
- *openbin()* Open a binary file.
- *opendir()* Get a filesystem object for a directory.
- *readbytes()* Read file as bytes.
- *readtext()* Read file as text.
- *remove()* Remove a file.
- *removedir()* Remove a directory.
- *removetree()* Recursively remove file and directories.
- *scandir()* Scan files and directories.
- *setinfo()* Set resource information.
- *settimes()* Set modified times for a resource.
- *touch()* Create a file or update times.
- *tree()* Render a tree view of the filesystem.
- *upload()* Copy a binary file to the filesystem.
- *validatepath()* Check a path is valid and return normalized path.
- *writebytes()* Write a file as bytes.
- *writefile()* Write a file-like object to the filesystem.
- *writetext()* Write a file as text.

CHAPTER 13

# Reference

## 13.1 fs.base

PyFilesystem base class.

The filesystem base class is common to all filesystems. If you familiarize yourself with this (rather straightforward) API, you can work with any of the supported filesystems.

**class** fs.base.**FS**

Base class for FS objects.

**__del__**()

Auto-close the filesystem on exit.

**__enter__**()

Allow use of filesystem as a context manager.

**__exit__**(*exc_type*, *exc_value*, *traceback*)

Close filesystem on exit.

**__init__**()

Create a filesystem. See help(type(self)) for accurate signature.

**appendbytes**(*path*, *data*)

Append bytes to the end of a file, creating it if needed.

**Parameters**

- **path** (*str*) – Path to a file.
- **data** (*bytes*) – Bytes to append.

**Raises**

- TypeError – If data is not a bytes instance.
- *fs.errors.ResourceNotFound* – If a parent directory of path does not exist.

**appendtext**(*path*, *text*, *encoding='utf-8'*, *errors=None*, *newline=''*)
Append text to the end of a file, creating it if needed.

**Parameters**

- **path** (*str*) – Path to a file.
- **text** (*str*) – Text to append.
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`).
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.

**Raises**

- `TypeError` – if `text` is not an unicode string.
- *fs.errors.ResourceNotFound* – if a parent directory of `path` does not exist.

**check**()
Check if a filesystem may be used.

**Raises** *fs.errors.FilesystemClosed* – if the filesystem is closed.

**close**()
Close the filesystem and release any resources.

It is important to call this method when you have finished working with the filesystem. Some filesystems may not finalize changes until they are closed (archives for example). You may call this method explicitly (it is safe to call close multiple times), or you can use the filesystem as a context manager to automatically close.

### Example

```
>>> with OSFS('~/Desktop') as desktop_fs:
...     desktop_fs.writetext(
...         'note.txt',
...         "Don't forget to tape Game of Thrones"
...     )
```

If you attempt to use a filesystem that has been closed, a *FilesystemClosed* exception will be thrown.

**copy**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Copy file contents from `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source file.
- **dst_path** (*str*) – Path to destination file.
- **overwrite** (*bool*) – If `True`, overwrite the destination file if it exists (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Raises**

- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.

- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.
- *fs.errors.FileExpected* – If `src_path` is not a file.

**copydir**(*src_path*, *dst_path*, *create=False*, *preserve_time=False*)
Copy the contents of `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source directory.
- **dst_path** (*str*) – Path to destination directory.
- **create** (*bool*) – If `True`, then `dst_path` will be created if it doesn't exist already (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Raises**

- *fs.errors.ResourceNotFound* – If the `dst_path` does not exist, and `create` is not `True`.
- *fs.errors.DirectoryExpected* – If `src_path` is not a directory.

**create**(*path*, *wipe=False*)
Create an empty file.

The default behavior is to create a new file if one doesn't already exist. If `wipe` is `True`, any existing file will be truncated.

**Parameters**

- **path** (*str*) – Path to a new file in the filesystem.
- **wipe** (*bool*) – If `True`, truncate any existing file to 0 bytes (defaults to `False`).

**Returns** `True` if a new file had to be created.

**Return type** bool

**desc**(*path*)
Return a short descriptive text regarding a path.

**Parameters** **path** (*str*) – A path to a resource on the filesystem.

**Returns** a short description of the path.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**download**(*path*, *file*, *chunk_size=None*, ***options*)
Copy a file from the filesystem to a file-like object.

This may be more efficient that opening and copying files manually if the filesystem supplies an optimized method.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

**Parameters**

- **path** (*str*) – Path to a resource.
- **file** (*file-like*) – A file-like object open for writing in binary mode.

- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

#### Example

```
>>> with open('starwars.mov', 'wb') as write_file:
...     my_fs.download('/Videos/starwars.mov', write_file)
```

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

**exists**(*path*)

Check if a path maps to a resource.

**Parameters** **path** (*str*) – Path to a resource.

**Returns** `True` if a resource exists at the given path.

**Return type** bool

**filterdir**(*path*, *files=None*, *dirs=None*, *exclude_dirs=None*, *exclude_files=None*, *namespaces=None*, *page=None*)

Get an iterator of resource info, filtered by patterns.

This method enhances *scandir* with additional filtering functionality.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **files** (*list, optional*) – A list of UNIX shell-style patterns to filter file names, e.g. `['*.py']`.
- **dirs** (*list, optional*) – A list of UNIX shell-style patterns to filter directory names.
- **exclude_dirs** (*list, optional*) – A list of patterns used to exclude directories.
- **exclude_files** (*list, optional*) – A list of patterns used to exclude files.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**getbasic**(*path*)

Get the *basic* resource info.

This method is shorthand for the following:

```
fs.getinfo(path, namespaces=['basic'])
```

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** Resource information object for `path`.

**Return type** *Info*

---

**Note:** Deprecated since version 2.4.13: Please use `getinfo` directly, which is required to always return the *basic* namespace.

---

**getbytes**(*path*)

Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.
- *fs.errors.ResourceNotFound* – if `path` does not exist.

---

**Note:** Deprecated since version 2.2.0: Please use *readbytes*

---

**getdetails**(*path*)

Get the *details* resource info.

This method is shorthand for the following:

```
fs.getinfo(path, namespaces=['details'])
```

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** Resource information object for `path`.

**Return type** *Info*

**getfile**(*path*, *file*, *chunk_size=None*, ***options*)

Copy a file from the filesystem to a file-like object.

This may be more efficient that opening and copying files manually if the filesystem supplies an optimized method.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

**Parameters**

- **path** (*str*) – Path to a resource.
- **file** (*file-like*) – A file-like object open for writing in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

### Example

```
>>> with open('starwars.mov', 'wb') as write_file:
...     my_fs.download('/Videos/starwars.mov', write_file)
```

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

---

**Note:** Deprecated since version 2.2.0: Please use *download*

---

**getinfo**(*path*, *namespaces=None*)

Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**getmeta**(*namespace='standard'*)

Get meta information regarding a filesystem.

**Parameters** **namespace** (*str*) – The meta namespace (defaults to `"standard"`).

**Returns** the meta information.

**Return type** dict

Meta information is associated with a *namespace* which may be specified with the `namespace` parameter. The default namespace, `"standard"`, contains common information regarding the filesystem's capabilities. Some filesystems may provide other namespaces which expose less common or implementation specific information. If a requested namespace is not supported by a filesystem, then an empty dictionary will be returned.

The `"standard"` namespace supports the following keys:

| key | Description |
|---|---|
| case_insensitive | `True` if this filesystem is case insensitive. |
| invalid_path_chars | A string containing the characters that may not be used on this filesystem. |
| max_path_length | Maximum number of characters permitted in a path, or `None` for no limit. |
| max_sys_path_length | Maximum number of characters permitted in a sys path, or `None` for no limit. |
| network | `True` if this filesystem requires a network. |
| read_only | `True` if this filesystem is read only. |
| supports_rename | `True` if this filesystem supports an `os.rename` operation. |

Most builtin filesystems will provide all these keys, and third- party filesystems should do so whenever possible, but a key may not be present if there is no way to know the value.

**Note:** Meta information is constant for the lifetime of the filesystem, and may be cached.

**getmodified**(*path*)
Get the timestamp of the last modifying access of a resource.

**Parameters** **path** (*str*) – A path to a resource.

**Returns** The timestamp of the last modification.

**Return type** datetime

The *modified timestamp* of a file is the point in time that the file was last changed. Depending on the file system, it might only have limited accuracy.

**getospath**(*path*)
Get the *system path* to a resource, in the OS' prefered encoding.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the *system path* of the resource, if any.

**Return type** str

**Raises** *fs.errors.NoSysPath* – If there is no corresponding system path.

This method takes the output of *getsyspath* and encodes it to the filesystem's prefered encoding. In Python3 this step is not required, as the `os` module will do it automatically. In Python2.7, the encoding step is required to support filenames on the filesystem that don't encode correctly.

**Note:** If you want your code to work in Python2.7 and Python3 then use this method if you want to work with the OS filesystem outside of the OSFS interface.

**getsize**(*path*)
Get the size (in bytes) of a resource.

**Parameters** **path** (*str*) – A path to a resource.

**Returns** the *size* of the resource.

**Return type** int

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

The *size* of a file is the total number of readable bytes, which may not reflect the exact number of bytes of reserved disk space (or other storage medium).

The size of a directory is the number of bytes of overhead use to store the directory entry.

**getsyspath**(*path*)
Get the *system path* of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the *system path* of the resource, if any.

**Return type** str

**Raises** *fs.errors.NoSysPath* – If there is no corresponding system path.

A system path is one recognized by the OS, that may be used outside of PyFilesystem (in an application or a shell for example). This method will get the corresponding system path that would be referenced by `path`.

Not all filesystems have associated system paths. Network and memory based filesystems, for example, may not physically store data anywhere the OS knows about. It is also possible for some paths to have a system path, whereas others don't.

This method will always return a str on Py3.* and unicode on Py2.7. See *getospath* if you need to encode the path as bytes.

If `path` doesn't have a system path, a *NoSysPath* exception will be thrown.

---

**Note:** A filesystem may return a system path even if no resource is referenced by that path – as long as it can be certain what that system path would be.

---

**gettext** (*path*, *encoding=None*, *errors=None*, *newline=''*)

Get the contents of a file as a string.

**Parameters**

- **path** (*str*) – A path to a readable file on the filesystem.
- **encoding** (*str, optional*) – Encoding to use when reading contents in text mode (defaults to None, reading in binary mode).
- **errors** (*str, optional*) – Unicode errors parameter.
- **newline** (*str*) – Newlines parameter.

**Returns** file contents.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

---

**Note:** Deprecated since version 2.2.0: Please use *readtext*

---

**gettype** (*path*)

Get the type of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the type of the resource.

**Return type** *ResourceType*

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

A type of a resource is an integer that identifies the what the resource references. The standard type integers may be one of the values in the *ResourceType* enumerations.

The most common resource types, supported by virtually all filesystems are `directory` (1) and `file` (2), but the following types are also possible:

| ResourceType | value |
|---|---|
| unknown | 0 |
| directory | 1 |
| file | 2 |
| character | 3 |
| block_special_file | 4 |
| fifo | 5 |
| socket | 6 |
| symlink | 7 |

Standard resource types are positive integers, negative values are reserved for implementation specific resource types.

**geturl** (*path*, *purpose='download'*)
Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, `OSFS` supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

**glob**
a globber object..

**Type** *BoundGlobber*

**hash** (*path*, *name*)
Get the hash of a file's contents.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **name** (*str*) – One of the algorithms supported by the `hashlib` module, e.g. `"md5"` or `"sha256"`.

**Returns** The hex digest of the hash.

**Return type** str

**Raises**

- *fs.errors.UnsupportedHash* – If the requested hash is not supported.
- *fs.errors.ResourceNotFound* – If `path` does not exist.
- *fs.errors.FileExpected* – If `path` exists but is not a file.

**hassyspath** (*path*)
Check if a path maps to a system path.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if the resource at `path` has a *syspath*.

**Return type** bool

**hasurl**(*path*, *purpose='download'*)
Check if a path has a corresponding URL.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **purpose** (*str*) – A purpose parameter, as given in *geturl*.

**Returns** `True` if an URL for the given purpose exists.

**Return type** bool

**isclosed**()
Check if the filesystem is closed.

**isdir**(*path*)
Check if a path maps to an existing directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a directory.

**Return type** bool

**isempty**(*path*)
Check if a directory is empty.

A directory is considered empty when it does not contain any file or any directory.

**Parameters** **path** (*str*) – A path to a directory on the filesystem.

**Returns** `True` if the directory is empty.

**Return type** bool

**Raises**

- `errors.DirectoryExpected` – If `path` is not a directory.
- `errors.ResourceNotFound` – If `path` does not exist.

**isfile**(*path*)
Check if a path maps to an existing file.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a file.

**Return type** bool

**islink**(*path*)
Check if a path maps to a symlink.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a symlink.

**Return type** bool

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**lock**()

Get a context manager that *locks* the filesystem.

Locking a filesystem gives a thread exclusive access to it. Other threads will block until the threads with the lock has left the context manager.

**Returns** a lock specific to the filesystem instance.

**Return type** threading.RLock

### Example

```
>>> with my_fs.lock():  # May block
...     # code here has exclusive access to the filesystem
...     pass
```

It is a good idea to put a lock around any operations that you would like to be *atomic*. For instance if you are copying files, and you don't want another thread to delete or modify anything while the copy is in progress.

Locking with this method is only required for code that calls multiple filesystem methods. Individual methods are thread safe already, and don't need to be locked.

---

**Note:** This only locks at the Python level. There is nothing to prevent other processes from modifying the filesystem outside of the filesystem instance.

---

**makedir**(*path*, *permissions=None*, *recreate=False*)

Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**makedirs**(*path*, *permissions=None*, *recreate=False*)

Make a directory, and any missing intermediate directories.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (Permissions, *optional*) – Initial permissions, or None to use defaults.
- **recreate** (*bool*) – If False (the default), attempting to create an existing directory will raise an error. Set to True to ignore existing directories.

**Returns** A sub-directory filesystem.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – if the path is already a directory, and recreate is False.
- *fs.errors.DirectoryExpected* – if one of the ancestors in the path is not a directory.

**match**(*patterns*, *name*)

Check if a name matches any of a list of wildcards.

If a filesystem is case *insensitive* (such as Windows) then this method will perform a case insensitive match (i.e. *.py will match the same names as *.PY). Otherwise the match will be case sensitive (*.py and *.PY will match different names).

**Parameters**

- **patterns** (*list, optional*) – A list of patterns, e.g. ['*.py'], or None to match everything.
- **name** (*str*) – A file or directory name (not a path)

**Returns** True if name matches any of the patterns.

**Return type** bool

**Raises** TypeError – If patterns is a single string instead of a list (or None).

### Example

```
>>> my_fs.match(['*.py'], '__init__.py')
True
>>> my_fs.match(['*.jpg', '*.png'], 'foo.gif')
False
```

**Note:** If patterns is None (or ['*']), then this method will always return True.

**move**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)

Move a file from src_path to dst_path.

**Parameters**

- **src_path** (*str*) – A path on the filesystem to move.
- **dst_path** (*str*) – A path on the filesystem where the source file will be written to.
- **overwrite** (*bool*) – If True, destination path will be overwritten if it exists.

- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**Raises**

- *fs.errors.FileExpected* – If `src_path` maps to a directory instead of a file.
- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.
- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.

**movedir**(*src_path*, *dst_path*, *create=False*, *preserve_time=False*)
Move directory `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source directory on the filesystem.
- **dst_path** (*str*) – Path to destination directory.
- **create** (*bool*) – If `True`, then `dst_path` will be created if it doesn't exist already (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**Raises**

- *fs.errors.ResourceNotFound* – if `dst_path` does not exist, and `create` is `False`.
- *fs.errors.DirectoryExpected* – if `src_path` or one of its ancestors is not a directory.

**open**(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=''*, ***options*)
Open a file.

**Parameters**

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open the file object with (defaults to *r*).
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, 1 to select line buffering, of any positive integer to indicate a buffer size).
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`)
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If the path is not a file.

- *fs.errors.FileExists* – If the file exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If the path does not exist.

**openbin** (*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**opendir** (*path*, *factory=None*)
Get a filesystem object for a sub-directory.

**Parameters**

- **path** (*str*) – Path to a directory on the filesystem.
- **factory** (*callable, optional*) – A callable that when invoked with an FS instance and `path` will return a new FS object representing the sub-directory contents. If no `factory` is supplied then `subfs_class` will be used.

**Returns** A filesystem representing a sub-directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.ResourceNotFound* – If `path` does not exist.
- *fs.errors.DirectoryExpected* – If `path` is not a directory.

**readbytes** (*path*)
Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.

- *fs.errors.ResourceNotFound* – if `path` does not exist.

**readtext**(*path*, *encoding=None*, *errors=None*, *newline=''*)
Get the contents of a file as a string.

**Parameters**

- **path** (*str*) – A path to a readable file on the filesystem.
- **encoding** (*str, optional*) – Encoding to use when reading contents in text mode (defaults to `None`, reading in binary mode).
- **errors** (*str, optional*) – Unicode errors parameter.
- **newline** (*str*) – Newlines parameter.

**Returns** file contents.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**remove**(*path*)
Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)
Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**removetree**(*dir_path*)
Recursively remove a directory and all its contents.

This method is similar to *removedir*, but will remove the contents of the directory if it is not empty.

**Parameters** **dir_path** (*str*) – Path to a directory on the filesystem.

**Raises**

- *fs.errors.ResourceNotFound* – If `dir_path` does not exist.
- *fs.errors.DirectoryExpected* – If `dir_path` is not a directory.

**Caution:** A filesystem should never delete its root folder, so `FS.removetree("/")` has different semantics: the contents of the root folder will be deleted, but the root will be untouched:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.removetree("/")
>>> home_fs.exists("/")
True
>>> home_fs.isempty("/")
True
```

Combined with *opendir*, this can be used to clear a directory without removing the directory itself:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.opendir("/Videos").removetree("/")
>>> home_fs.exists("/Videos")
True
>>> home_fs.isempty("/Videos")
True
```

**scandir** (*path*, *namespaces=None*, *page=None*)

Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setbinfile** (*path*, *file*, *chunk_size=None*, ***options*)

Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**Note:** Deprecated since version 2.2.0: Please use *upload*

**setbytes** (*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** TypeError – if contents is not bytes.

**Note:** Deprecated since version 2.2.0: Please use *writebytes*

**setfile** (*path*, *file*, *encoding=None*, *errors=None*, *newline=''*)
Set a file to the contents of a file object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – A file object open for reading.
- **encoding** (*str, optional*) – Encoding of destination file, defaults to None for binary.
- **errors** (*str, optional*) – How encoding errors should be treated (same as io.open).
- **newline** (*str*) – Newline parameter (same as io.open).

This method is similar to *upload*, in that it copies data from a file-like object to a resource on the filesystem, but unlike upload, this method also supports creating files in text-mode (if the encoding argument is supplied).

Note that the file object file will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('myfile.txt') as read_file:
...     my_fs.writefile('myfile.txt', read_file)
```

**Note:** Deprecated since version 2.2.0: Please use *writefile*

**setinfo** (*path*, *info*)
Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**settext** (*path*, *contents*, *encoding='utf-8'*, *errors=None*, *newline=''*)

Create or replace a file with text.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*str*) – Text to be written.
- **encoding** (*str, optional*) – Encoding of destination file (defaults to `'utf-8'`).
- **errors** (*str, optional*) – How encoding errors should be treated (same as `io.open`).
- **newline** (*str*) – Newline parameter (same as `io.open`).

**Raises** `TypeError` – if `contents` is not a unicode string.

---

**Note:** Deprecated since version 2.2.0: Please use *writetext*

---

**settimes** (*path*, *accessed=None*, *modified=None*)

Set the accessed and modified time on a resource.

**Parameters**

- **path** – A path to a resource on the filesystem.
- **accessed** (*datetime, optional*) – The accessed time, or `None` (the default) to use the current time.
- **modified** (*datetime, optional*) – The modified time, or `None` (the default) to use the same time as the `accessed` parameter.

**touch** (*path*)

Touch a file on the filesystem.

Touching a file means creating a new file if `path` doesn't exist, or update accessed and modified times if the path does exist. This method is similar to the linux command of the same name.

**Parameters** **path** (*str*) – A path to a file on the filesystem.

**tree** (***kwargs*)

Render a tree view of the filesystem to stdout or a file.

The parameters are passed to *render()*.

**Keyword Arguments**

- **path** (*str*) – The path of the directory to start rendering from (defaults to root folder, i.e. '/').
- **file** (*io.IOBase*) – An open file-like object to render the tree, or None for stdout.
- **encoding** (*str*) – Unicode encoding, or None to auto-detect.
- **max_levels** (*int*) – Maximum number of levels to display, or None for no maximum.
- **with_color** (*bool*) – Enable terminal color output, or None to auto-detect terminal.
- **dirs_first** (*bool*) – Show directories first.
- **exclude** (*list*) – Option list of directory patterns to exclude from the tree render.
- **filter** (*list*) – Optional list of files patterns to match in the tree render.

**upload**(*path*, *file*, *chunk_size=None*, ***options*)

Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or None to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of path does not exist.

Note that the file object file will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**validatepath**(*path*)

Validate a path, returning a normalized absolute path on sucess.

Many filesystems have restrictions on the format of paths they support. This method will check that path is valid on the underlaying storage mechanism and throw a *InvalidPath* exception if it is not.

**Parameters** **path** (*str*) – A path.

**Returns** A normalized, absolute path.

**Return type** str

**Raises**

- *fs.errors.InvalidPath* – If the path is invalid.
- *fs.errors.FilesystemClosed* – if the filesystem is closed.
- *fs.errors.InvalidCharsInPath* – If the path contains invalid characters.

**walk**
a walker bound to this filesystem.

**Type** *BoundWalker*

**walker_class**
alias of *fs.walk.Walker*

**writebytes**(*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** TypeError – if contents is not bytes.

**writefile**(*path*, *file*, *encoding=None*, *errors=None*, *newline=''*)
Set a file to the contents of a file object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – A file object open for reading.
- **encoding** (*str, optional*) – Encoding of destination file, defaults to None for binary.
- **errors** (*str, optional*) – How encoding errors should be treated (same as io.open).
- **newline** (*str*) – Newline parameter (same as io.open).

This method is similar to *upload*, in that it copies data from a file-like object to a resource on the filesystem, but unlike upload, this method also supports creating files in text-mode (if the encoding argument is supplied).

Note that the file object file will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('myfile.txt') as read_file:
...     my_fs.writefile('myfile.txt', read_file)
```

**writetext**(*path*, *contents*, *encoding='utf-8'*, *errors=None*, *newline=''*)
Create or replace a file with text.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*str*) – Text to be written.
- **encoding** (*str, optional*) – Encoding of destination file (defaults to 'utf-8').
- **errors** (*str, optional*) – How encoding errors should be treated (same as io.open).
- **newline** (*str*) – Newline parameter (same as io.open).

**Raises** TypeError – if contents is not a unicode string.

## 13.2 fs.compress

Functions to compress the contents of a filesystem.

Currently zip and tar are supported, using the `zipfile` and `tarfile` modules from the standard library.

`fs.compress.`**`write_tar`**(*src_fs*, *file*, *compression=None*, *encoding='utf-8'*, *walker=None*)
Write the contents of a filesystem to a tar file.

**Parameters**

- **src_fs** (*FS*) – The source filesystem to compress.
- **file** (*str or io.IOBase*) – Destination file, may be a file name or an open file object.
- **compression** (*str, optional*) – Compression to use, or `None` for a plain Tar archive without compression.
- **encoding** (*str*) – The encoding to use for filenames. The default is `"utf-8"`.
- **walker** (*Walker, optional*) – A `Walker` instance, or `None` to use default walker. You can use this to specify which files you want to compress.

`fs.compress.`**`write_zip`**(*src_fs*, *file*, *compression=8*, *encoding='utf-8'*, *walker=None*)
Write the contents of a filesystem to a zip file.

**Parameters**

- **src_fs** (*FS*) – The source filesystem to compress.
- **file** (*str or io.IOBase*) – Destination file, may be a file name or an open file object.
- **compression** (*int*) – Compression to use (one of the constants defined in the `zipfile` module in the stdlib). Defaults to `zipfile.ZIP_DEFLATED`.
- **encoding** (*str*) – The encoding to use for filenames. The default is `"utf-8"`, use `"CP437"` if compatibility with WinZip is desired.
- **walker** (*Walker, optional*) – A `Walker` instance, or `None` to use default walker. You can use this to specify which files you want to compress.

## 13.3 fs.copy

Functions for copying resources *between* filesystem.

`fs.copy.`**`copy_dir`**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *walker=None*, *on_copy=None*, *workers=0*, *preserve_time=False*)
Copy a directory from one filesystem to another.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a directory on the source filesystem.
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a directory on the destination filesystem.
- **walker** (*Walker, optional*) – A walker object that will be used to scan for files in `src_fs`. Set this if you only want to consider a sub-set of the resources in `src_fs`.

- **on_copy** (*callable, optional*) – A function callback called after a single file copy is executed. Expected signature is `(src_fs, src_path, dst_fs, dst_path)`.
- **workers** (*int*) – Use `worker` threads to copy data, or `0` (default) for a single-threaded copy.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

`fs.copy.`**copy_dir_if**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *condition*, *walker=None*, *on_copy=None*, *workers=0*, *preserve_time=False*)

Copy a directory from one filesystem to another, depending on a condition.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a directory on the source filesystem.
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a directory on the destination filesystem.
- **condition** (*str*) – Name of the condition to check for each file.
- **walker** (*Walker, optional*) – A walker object that will be used to scan for files in `src_fs`. Set this if you only want to consider a sub-set of the resources in `src_fs`.
- **on_copy** (*callable*) – A function callback called after a single file copy is executed. Expected signature is `(src_fs, src_path, dst_fs, dst_path)`.
- **workers** (*int*) – Use `worker` threads to copy data, or `0` (default) for a single-threaded copy.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**See also:**

*copy_file_if* for the full list of supported values for the `condition` argument.

`fs.copy.`**copy_dir_if_newer**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *walker=None*, *on_copy=None*, *workers=0*, *preserve_time=False*)

Copy a directory from one filesystem to another, checking times.

Deprecated since version 2.5.0: Use *copy_dir_if* with `condition="newer"` instead.

`fs.copy.`**copy_file**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *preserve_time=False*)

Copy a file from one filesystem to another.

If the destination exists, and is a file, it will be first truncated.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a file on the source filesystem.
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a file on the destination filesystem.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

fs.copy.**copy_file_if**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *condition*, *preserve_time=False*)
Copy a file from one filesystem to another, depending on a condition.

Depending on the value of `condition`, certain requirements must be fulfilled for a file to be copied to `dst_fs`. The following values are supported:

**"always"** The source file is always copied.

**"newer"** The last modification time of the source file must be newer than that of the destination file. If either file has no modification time, the copy is performed always.

**"older"** The last modification time of the source file must be older than that of the destination file. If either file has no modification time, the copy is performed always.

**"exists"** The source file is only copied if a file of the same path already exists in `dst_fs`.

**"not_exists"** The source file is only copied if no file of the same path already exists in `dst_fs`.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a file on the source filesystem.
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a file on the destination filesystem.
- **condition** (*str*) – Name of the condition to check for each file.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Returns** `True` if the file copy was executed, `False` otherwise.

**Return type** bool

fs.copy.**copy_file_if_newer**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *preserve_time=False*)
Copy a file from one filesystem to another, checking times.

Deprecated since version 2.5.0: Use *copy_file_if* with `condition="newer"` instead.

fs.copy.**copy_file_internal**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *preserve_time=False*, *lock=False*)
Copy a file at low level, without calling `manage_fs` or locking.

If the destination exists, and is a file, it will be first truncated.

This method exists to optimize copying in loops. In general you should prefer *copy_file*.

**Parameters**

- **src_fs** (*FS*) – Source filesystem.
- **src_path** (*str*) – Path to a file on the source filesystem.
- **dst_fs** (*FS*) – Destination filesystem.
- **dst_path** (*str*) – Path to a file on the destination filesystem.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).
- **lock** (*bool*) – Lock both filesystems before copying.

fs.copy.**copy_fs**(*src_fs*, *dst_fs*, *walker=None*, *on_copy=None*, *workers=0*, *preserve_time=False*)
Copy the contents of one filesystem to another.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (URL or instance).
- **dst_fs** (*FS or str*) – Destination filesystem (URL or instance).
- **walker** (*Walker, optional*) – A walker object that will be used to scan for files in `src_fs`. Set this if you only want to consider a sub-set of the resources in `src_fs`.
- **on_copy** (*callable*) – A function callback called after a single file copy is executed. Expected signature is `(src_fs, src_path, dst_fs, dst_path)`.
- **workers** (*int*) – Use `worker` threads to copy data, or `0` (default) for a single-threaded copy.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

fs.copy.**copy_fs_if**(*src_fs*, *dst_fs*, *condition='always'*, *walker=None*, *on_copy=None*, *workers=0*, *preserve_time=False*)

Copy the contents of one filesystem to another, depending on a condition.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (URL or instance).
- **dst_fs** (*FS or str*) – Destination filesystem (URL or instance).
- **condition** (*str*) – Name of the condition to check for each file.
- **walker** (*Walker, optional*) – A walker object that will be used to scan for files in `src_fs`. Set this if you only want to consider a sub-set of the resources in `src_fs`.
- **on_copy** (*callable*) – A function callback called after a single file copy is executed. Expected signature is `(src_fs, src_path, dst_fs, dst_path)`.
- **workers** (*int*) – Use `worker` threads to copy data, or `0` (default) for a single-threaded copy.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**See also:**

*copy_file_if* for the full list of supported values for the `condition` argument.

fs.copy.**copy_fs_if_newer**(*src_fs*, *dst_fs*, *walker=None*, *on_copy=None*, *workers=0*, *preserve_time=False*)

Copy the contents of one filesystem to another, checking times.

Deprecated since version 2.5.0: Use *copy_fs_if* with `condition="newer"` instead.

fs.copy.**copy_modified_time**(*src_fs*, *src_path*, *dst_fs*, *dst_path*)

Copy modified time metadata from one file to another.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a directory on the source filesystem.
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a directory on the destination filesystem.

fs.copy.**copy_structure**(*src_fs*, *dst_fs*, *walker=None*, *src_root='/'*, *dst_root='/'*)

Copy directories (but not files) from `src_fs` to `dst_fs`.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **walker** (*Walker, optional*) – A walker object that will be used to scan for files in `src_fs`. Set this if you only want to consider a sub-set of the resources in `src_fs`.
- **src_root** (*str*) – Path of the base directory to consider as the root of the tree structure to copy.
- **dst_root** (*str*) – Path to the target root of the tree structure.

## 13.4 fs.enums

Enums used by PyFilesystem.

**class** fs.enums.**ResourceType**

Resource Types.

Positive values are reserved, negative values are implementation dependent.

Most filesystems will support only directory(1) and file(2). Other types exist to identify more exotic resource types supported by Linux filesystems.

**unknown = 0**
Unknown resource type, used if the filesystem is unable to tell what the resource is.

**directory = 1**
A directory.

**file = 2**
A simple file.

**character = 3**
A character file.

**block_special_file = 4**
A block special file.

**fifo = 5**
A first in first out file.

**socket = 6**
A socket.

**symlink = 7**
A symlink.

**class** fs.enums.**Seek**

Constants used by `io.IOBase.seek`.

These match `os.SEEK_CUR`, `os.SEEK_END`, and `os.SEEK_SET` from the standard library.

**current = 1**
Seek from the current file position.

**end = 2**
Seek from the end of the file.

**set = 0**
Seek from the start of the file.

## 13.5 fs.errors

Exception classes thrown by filesystem operations.

Errors relating to the underlying filesystem are translated in to one of the following exceptions.

All Exception classes are derived from *FSError* which may be used as a catch-all filesystem exception.

**exception** fs.errors.**BulkCopyFailed**(*errors*)
Bases: *fs.errors.FSError*

A copy operation failed in worker threads.

**__init__**(*errors*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**CreateFailed**(*msg=None*, *exc=None*)
Bases: *fs.errors.FSError*

Filesystem could not be created.

**__init__**(*msg=None*, *exc=None*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**DestinationExists**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Target destination already exists.

**exception** fs.errors.**DirectoryExists**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Directory already exists.

**exception** fs.errors.**DirectoryExpected**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceInvalid*

Operation only works on directories.

**exception** fs.errors.**DirectoryNotEmpty**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Attempt to remove a non-empty directory.

**exception** fs.errors.**FileExists**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

File already exists.

**exception** fs.errors.**FileExpected**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceInvalid*

Operation only works on files.

**exception** fs.errors.**FilesystemClosed**(*msg=None*)
Bases: *fs.errors.FSError*

Attempt to use a closed filesystem.

**exception** fs.errors.**FSError**(*msg=None*)
Bases: Exception

Base exception for the fs module.

**__init__**(*msg=None*)
Initialize self. See help(type(self)) for accurate signature.

**__str__**()
Return the error message.

**exception** fs.errors.**IllegalBackReference**(*path*)
Bases: ValueError

Too many backrefs exist in a path.

This error will occur if the back references in a path would be outside of the root. For example, "/foo/../../", contains two back references which would reference a directory above the root.

---

**Note:** This exception is a subclass of ValueError as it is not strictly speaking an issue with a filesystem or resource.

---

**__init__**(*path*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**InsufficientStorage**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.OperationFailed*

Storage is insufficient for requested operation.

**exception** fs.errors.**InvalidCharsInPath**(*path*, *msg=None*, *exc=None*)
Bases: *fs.errors.InvalidPath*

Path contains characters that are invalid on this filesystem.

**exception** fs.errors.**InvalidPath**(*path*, *msg=None*, *exc=None*)
Bases: *fs.errors.PathError*

Path can't be mapped on to the underlaying filesystem.

**exception** fs.errors.**MissingInfoNamespace**(*namespace*)
Bases: AttributeError

An expected namespace is missing.

**__init__**(*namespace*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**NoSysPath**(*path*, *msg=None*, *exc=None*)
Bases: *fs.errors.PathError*

The filesystem does not provide *sys paths* to the resource.

**exception** fs.errors.**NoURL**(*path*, *purpose*, *msg=None*)
Bases: *fs.errors.PathError*

The filesystem does not provide an URL for the resource.

**__init__**(*path*, *purpose*, *msg=None*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**OperationFailed**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.FSError*

A specific operation failed.

**__init__**(*path=None*, *exc=None*, *msg=None*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**OperationTimeout**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.OperationFailed*

Filesystem took too long.

**exception** fs.errors.**PathError**(*path*, *msg=None*, *exc=None*)
Bases: *fs.errors.FSError*

Base exception for errors to do with a path string.

**__init__**(*path*, *msg=None*, *exc=None*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**PermissionDenied**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.OperationFailed*

Not enough permissions.

**exception** fs.errors.**RemoteConnectionError**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.OperationFailed*

Operations encountered remote connection trouble.

**exception** fs.errors.**RemoveRootError**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.OperationFailed*

Attempt to remove the root directory.

**exception** fs.errors.**ResourceError**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.FSError*

Base exception class for error associated with a specific resource.

**__init__**(*path*, *exc=None*, *msg=None*)
Initialize self. See help(type(self)) for accurate signature.

**exception** fs.errors.**ResourceInvalid**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Resource has the wrong type.

**exception** fs.errors.**ResourceLocked**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Attempt to use a locked resource.

**exception** fs.errors.**ResourceNotFound**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Required resource not found.

**exception** fs.errors.**ResourceReadOnly**(*path*, *exc=None*, *msg=None*)
Bases: *fs.errors.ResourceError*

Attempting to modify a read-only resource.

**exception** fs.errors.**Unsupported**(*path=None*, *exc=None*, *msg=None*)
Bases: *fs.errors.OperationFailed*

Operation not supported by the filesystem.

**exception** fs.errors.**UnsupportedHash**
Bases: ValueError

The requested hash algorithm is not supported.

This exception will be thrown if a hash algorithm is requested that is not supported by hashlib.

## 13.6 fs.glob

Useful functions for working with glob patterns.

**class** fs.glob.**BoundGlobber**(*fs*)
A *Globber* object bound to a filesystem.

An instance of this object is available on every Filesystem object as the *glob* property.

**__call__**(*pattern*, *path='/'*, *namespaces=None*, *case_sensitive=True*, *exclude_dirs=None*)
Match resources on the bound filesystem againsts a glob pattern.

**Parameters**

- **pattern** (*str*) – A glob pattern, e.g. `"**/*.py"`
- **namespaces** (*list*) – A list of additional info namespaces.
- **case_sensitive** (*bool*) – If `True`, the path matching will be case *sensitive* i.e. `"FOO.py"` and `"foo.py"` will be different, otherwise path matching will be case **insensitive**.
- **exclude_dirs** (*list*) – A list of patterns to exclude when searching, e.g. `["*.git"]`.

**Returns** An object that may be queried for the glob matches.

**Return type** *Globber*

**__init__**(*fs*)
Create a new bound Globber.

**Parameters** **fs** (FS) – A filesystem object to bind to.

**class** fs.glob.**Counts**(*files*, *directories*, *data*)

**data**
Alias for field number 2

**directories**
Alias for field number 1

**files**
Alias for field number 0

**class** fs.glob.**GlobMatch**(*path*, *info*)

**info**
Alias for field number 1

**path**
Alias for field number 0

**class** fs.glob.**Globber**(*fs*, *pattern*, *path='/'*, *namespaces=None*, *case_sensitive=True*, *exclude_dirs=None*)
A generator of glob results.

**__init__**(*fs*, *pattern*, *path='/'*, *namespaces=None*, *case_sensitive=True*, *exclude_dirs=None*)
Create a new Globber instance.

**Parameters**

- **fs** (FS) – A filesystem object

- **pattern** (*str*) – A glob pattern, e.g. `"**/*.py"`
- **path** (*str*) – A path to a directory in the filesystem.
- **namespaces** (*list*) – A list of additional info namespaces.
- **case_sensitive** (*bool*) – If `True`, the path matching will be case *sensitive* i.e. `"FOO.py"` and `"foo.py"` will be different, otherwise path matching will be case *insensitive*.
- **exclude_dirs** (*list*) – A list of patterns to exclude when searching, e.g. `["*.git"]`.

**__iter__**()
Get an iterator of *fs.glob.GlobMatch* objects.

**count**()
Count files / directories / data in matched paths.

#### Example

```
>>> my_fs.glob('**/*.py').count()
Counts(files=2, directories=0, data=55)
```

**Returns** A named tuple containing results.

**Return type** *Counts*

**count_lines**()
Count the lines in the matched files.

**Returns** A named tuple containing line counts.

**Return type** *LineCounts*

#### Example

```
>>> my_fs.glob('**/*.py').count_lines()
LineCounts(lines=4, non_blank=3)
```

**remove**()
Remove all matched paths.

**Returns** Number of file and directories removed.

**Return type** int

#### Example

```
>>> my_fs.glob('**/*.pyc').remove()
2
```

**class** fs.glob.**LineCounts**(*lines*, *non_blank*)

**lines**
Alias for field number 0

**non_blank**
Alias for field number 1

fs.glob.**imatch**(*pattern*, *path*)
Compare a glob pattern with a path (case insensitive).

**Parameters**

- **pattern** (*str*) – A glob pattern.
- **path** (*str*) – A path.

**Returns** `True` if the path matches the pattern.

**Return type** bool

fs.glob.**match**(*pattern*, *path*)
Compare a glob pattern with a path (case sensitive).

**Parameters**

- **pattern** (*str*) – A glob pattern.
- **path** (*str*) – A path.

**Returns** `True` if the path matches the pattern.

**Return type** bool

### Example

```
>>> from fs.glob import match
>>> match("**/*.py", "/fs/glob.py")
True
```

## 13.7 fs.info

Container for filesystem resource informations.

**class** fs.info.**Info**(*raw_info*, *to_datetime=<function epoch_to_datetime>*)
Container for *Resource Info*.

Resource information is returned by the following methods:

- *getinfo*
- *scandir*
- *filterdir*

**Parameters**

- **raw_info** (*dict*) – A dict containing resource info.
- **to_datetime** (*callable*) – A callable that converts an epoch time to a datetime object. The default uses `epoch_to_datetime`.

**__init__**(*raw_info*, *to_datetime=<function epoch_to_datetime>*)
Create a resource info object from a raw info dict.

**accessed**
the resource last access time, or None.

Requires the "details" namespace.

**Raises** *MissingInfoNamespace* – if the "details" namespace is not in the Info.

**Type** datetime

**copy** (*to_datetime=None*)
Create a copy of this resource info object.

**created**
the resource creation time, or None.

Requires the "details" namespace.

**Raises** *MissingInfoNamespace* – if the "details" namespace is not in the Info.

**Type** datetime

**get** (*namespace*, *key*, *default=None*)
Get a raw info value.

**Parameters**

- **namespace** (*str*) – A namespace identifier.
- **key** (*str*) – A key within the namespace.
- **default** (*object, optional*) – A default value to return if either the namespace or the key within the namespace is not found.

### Example

```
>>> info = my_fs.getinfo("foo.py", namespaces=["details"])
>>> info.get('details', 'type')
2
```

**gid**
the group id of the resource, or None.

Requires the "access" namespace.

**Raises** *MissingInfoNamespace* – if the "access" namespace is not in the Info.

**Type** int

**group**
the group of the resource owner, or None.

Requires the "access" namespace.

**Raises** *MissingInfoNamespace* – if the "access" namespace is not in the Info.

**Type** str

**has_namespace** (*namespace*)
Check if the resource info contains a given namespace.

**Parameters** **namespace** (*str*) – A namespace identifier.

**Returns** True if the namespace was found, False otherwise.

**Return type** bool

**is_dir**
`True` if the resource references a directory.

**Type** `bool`

**is_file**
`True` if the resource references a file.

**Type** `bool`

**is_link**
`True` if the resource is a symlink.

**Type** `bool`

**is_writeable**(*namespace*, *key*)
Check if a given key in a namespace is writable.

When creating an *Info* object, you can add a `_write` key to each raw namespace that lists which keys are writable or not.

In general, this means they are compatible with the `setinfo` function of filesystem objects.

**Parameters**

- **namespace** (*str*) – A namespace identifier.
- **key** (*str*) – A key within the namespace.

**Returns** `True` if the key can be modified, `False` otherwise.

**Return type** bool

### Example

Create an *Info* object that marks only the `modified` key as writable in the `details` namespace:

```
>>> now = time.time()
>>> info = Info({
...     "basic": {"name": "foo", "is_dir": False},
...     "details": {
...         "modified": now,
...         "created": now,
...         "_write": ["modified"],
...     }
... })
>>> info.is_writeable("details", "created")
False
>>> info.is_writeable("details", "modified")
True
```

**make_path**(*dir_path*)
Make a path by joining `dir_path` with the resource name.

**Parameters** **dir_path** (*str*) – A path to a directory.

**Returns** A path to the resource.

**Return type** str

**metadata_changed**
the resource metadata change time, or `None`.

Requires the `"details"` namespace.

**Raises** *MissingInfoNamespace* – if the `"details"` namespace is not in the Info.

**Type** `datetime`

**modified**
the resource last modification time, or `None`.

Requires the `"details"` namespace.

**Raises** *MissingInfoNamespace* – if the `"details"` namespace is not in the Info.

**Type** `datetime`

**name**
the resource name.

**Type** `str`

**permissions**
the permissions of the resource, or `None`.

Requires the `"access"` namespace.

**Raises** *MissingInfoNamespace* – if the `"access"` namespace is not in the Info.

**Type** `Permissions`

**size**
the size of the resource, in bytes.

Requires the `"details"` namespace.

**Raises** *MissingInfoNamespace* – if the `"details"` namespace is not in the Info.

**Type** `int`

**stem**
the name minus any suffixes.

### Example

```
>>> info = my_fs.getinfo("foo.tar.gz")
>>> info.stem
'foo'
```

**Type** `str`

**suffix**
the last component of the name (with dot).

In case there is no suffix, an empty string is returned.

### Example

```
>>> info = my_fs.getinfo("foo.py")
>>> info.suffix
'.py'
>>> info2 = my_fs.getinfo("bar")
>>> info2.suffix
''
```

**Type** str

**suffixes**
a list of any suffixes in the name.

#### Example

```
>>> info = my_fs.getinfo("foo.tar.gz")
>>> info.suffixes
['.tar', '.gz']
```

**Type** List

**target**
the link target (if resource is a symlink), or None.

Requires the "link" namespace.

**Raises** *MissingInfoNamespace* – if the "link" namespace is not in the Info.

**Type** str

**type**
the type of the resource.

Requires the "details" namespace.

**Raises** *MissingInfoNamespace* – if the 'details' namespace is not in the Info.

**Type** *ResourceType*

**uid**
the user id of the resource, or None.

Requires the "access" namespace.

**Raises** *MissingInfoNamespace* – if the "access" namespace is not in the Info.

**Type** int

**user**
the owner of the resource, or None.

Requires the "access" namespace.

**Raises** *MissingInfoNamespace* – if the "access" namespace is not in the Info.

**Type** str

## 13.8 fs.filesize

Functions for reporting filesizes.

The functions declared in this module should cover the different usecases needed to generate a string representation of a file size using several different units. Since there are many standards regarding file size units, three different functions have been implemented.

**See also:**

- Wikipedia: Binary prefix

fs.filesize.**traditional**(*size*)

Convert a filesize in to a string (powers of 1024, JDEC prefixes).

In this convention, `1024 B = 1 KB`.

This is the format that was used to display the size of DVDs (*700 MB* meaning actually about *734 003 200 bytes*) before standardisation of IEC units among manufacturers, and still used by **Windows** to report the storage capacity of hard drives (*279.4 GB* meaning *279.4 × $1024^3$ bytes*).

**Parameters** **size** (*int*) – A file size.

**Returns** A string containing an abbreviated file size and units.

**Return type** str

### Example

```
>>> fs.filesize.traditional(30000)
'29.3 KB'
```

fs.filesize.**decimal**(*size*)

Convert a filesize in to a string (powers of 1000, SI prefixes).

In this convention, `1000 B = 1 kB`.

This is typically the format used to advertise the storage capacity of USB flash drives and the like (*256 MB* meaning actually a storage capacity of more than *256 000 000 B*), or used by **Mac OS X** since v10.6 to report file sizes.

**Parameters** **int** (*size*) – A file size.

**Returns** A string containing a abbreviated file size and units.

**Return type** str

### Example

```
>>> fs.filesize.decimal(30000)
'30.0 kB'
```

fs.filesize.**binary**(*size*)

Convert a filesize in to a string (powers of 1024, IEC prefixes).

In this convention, `1024 B = 1 KiB`.

This is the format that has gained adoption among manufacturers to avoid ambiguity regarding size units, since it explicitly states using a binary base (*KiB = kibi bytes = kilo binary bytes*). This format is notably being used by the **Linux** kernel (see `man 7 units`).

**Parameters** **int** (*size*) – A file size.

**Returns** A string containing a abbreviated file size and units.

**Return type** str

### Example

```
>>> fs.filesize.binary(30000)
'29.3 KiB'
```

## 13.9 fs.mirror

Function for *mirroring* a filesystem.

Mirroring will create a copy of a source filesystem on a destination filesystem. If there are no files on the destination, then mirroring is simply a straight copy. If there are any files or directories on the destination they may be deleted or modified to match the source.

In order to avoid redundant copying of files, *mirror* can compare timestamps, and only copy files with a newer modified date. This timestamp comparison is only done if the file sizes are different.

This scheme will work if you have mirrored a directory previously, and you would like to copy any changes. Otherwise you should set the `copy_if_newer` parameter to `False` to guarantee an exact copy, at the expense of potentially copying extra files.

`fs.mirror.`**`mirror`**(*src_fs*, *dst_fs*, *walker=None*, *copy_if_newer=True*, *workers=0*, *preserve_time=False*)

Mirror files / directories from one filesystem to another.

Mirroring a filesystem will create an exact copy of `src_fs` on `dst_fs`, by removing any files / directories on the destination that aren't on the source, and copying files that aren't.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (URL or instance).
- **dst_fs** (*FS or str*) – Destination filesystem (URL or instance).
- **walker** (*Walker, optional*) – An optional walker instance.
- **copy_if_newer** (*bool*) – Only copy newer files (the default).
- **workers** (*int*) – Number of worker threads used (0 for single threaded). Set to a relatively low number for network filesystems, 4 would be a good start.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

## 13.10 fs.move

Functions for moving files between filesystems.

`fs.move.`**`move_dir`**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *workers=0*, *preserve_time=False*)

Move a directory from one filesystem to another.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a directory on `src_fs`
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a directory on `dst_fs`.
- **workers** (*int*) – Use `worker` threads to copy data, or `0` (default) for a single-threaded copy.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

fs.move.**move_file**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *preserve_time=False*, *cleanup_dst_on_error=True*)

Move a file from one filesystem to another.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **src_path** (*str*) – Path to a file on `src_fs`.
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **dst_path** (*str*) – Path to a file on `dst_fs`.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).
- **cleanup_dst_on_error** (*bool*) – If `True`, tries to delete the file copied to `dst_fs` if deleting the file from `src_fs` fails (defaults to `True`).

fs.move.**move_fs**(*src_fs*, *dst_fs*, *workers=0*, *preserve_time=False*)

Move the contents of a filesystem to another filesystem.

**Parameters**

- **src_fs** (*FS or str*) – Source filesystem (instance or URL).
- **dst_fs** (*FS or str*) – Destination filesystem (instance or URL).
- **workers** (*int*) – Use `worker` threads to copy data, or `0` (default) for a single-threaded copy.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

## 13.11 fs.mode

Abstract I/O mode container.

Mode strings are used in in *open* and *openbin*.

**class** fs.mode.**Mode**(*mode*)

An abstraction for I/O modes.

A mode object provides properties that can be used to interrogate the mode strings used when opening files.

### Example

```
>>> mode = Mode('rb')
>>> mode.reading
True
>>> mode.writing
False
>>> mode.binary
True
>>> mode.text
False
```

**__contains__**(*character*)

Check if a mode contains a given character.

**__init__**(*mode*)
Create a new *Mode* instance.

**Parameters** **mode** (*str*) – A *mode* string, as used by `io.open`.

**Raises** `ValueError` – If the mode string is invalid.

**appending**
`True` if the mode permits appending.

**Type** `bool`

**binary**
`True` if a mode specifies binary.

**Type** `bool`

**create**
`True` if the mode would create a file.

**Type** `bool`

**exclusive**
`True` if the mode require exclusive creation.

**Type** `bool`

**reading**
`True` if the mode permits reading.

**Type** `bool`

**text**
`True` if a mode specifies text.

**Type** `bool`

**to_platform**()
Get a mode string for the current platform.

Currently, this just removes the 'x' on PY2 because PY2 doesn't support exclusive mode.

**to_platform_bin**()
Get a *binary* mode string for the current platform.

This removes the 't' and adds a 'b' if needed.

**truncate**
`True` if the mode would truncate an existing file.

**Type** `bool`

**updating**
`True` if the mode permits both reading and writing.

**Type** `bool`

**validate**(*_valid_chars=frozenset({'a', 'x', '+', 't', 'r', 'b', 'w'})*)
Validate the mode string.

**Raises** `ValueError` – if the mode contains invalid chars.

**validate_bin**()
Validate a mode for opening a binary file.

**Raises** `ValueError` – if the mode contains invalid chars.

**writing**
`True` if the mode permits writing.

**Type** `bool`

`fs.mode.`**check_readable**(*mode*)
Check a mode string allows reading.

**Parameters** **mode** (*str*) – A mode string, e.g. `"rt"`

**Returns** `True` if the mode allows reading.

**Return type** bool

`fs.mode.`**check_writable**(*mode*)
Check a mode string allows writing.

**Parameters** **mode** (*str*) – A mode string, e.g. `"wt"`

**Returns** `True` if the mode allows writing.

**Return type** bool

`fs.mode.`**validate_openbin_mode**(*mode*, *_valid_chars=frozenset({'a', 'x', '+', 'r', 'b', 'w'})*)
Check `mode` parameter of *openbin* is valid.

**Parameters** **mode** (*str*) – Mode parameter.

**Raises** `ValueError` if mode is not valid.

## 13.12 fs.opener

Open filesystems from a URL.

### 13.12.1 fs.opener.base

*Opener* abstract base class.

**class** `fs.opener.base.`**Opener**
The base class for filesystem openers.

An opener is responsible for opening a filesystem for a given protocol.

**open_fs**(*fs_url*, *parse_result*, *writeable*, *create*, *cwd*)
Open a filesystem object from a FS URL.

**Parameters**

- **fs_url** (*str*) – A filesystem URL.
- **parse_result** (`ParseResult`) – A parsed filesystem URL.
- **writeable** (*bool*) – `True` if the filesystem must be writable.
- **create** (*bool*) – `True` if the filesystem should be created if it does not exist.
- **cwd** (*str*) – The current working directory (generally only relevant for OS filesystems).

**Raises** *fs.opener.errors.OpenerError* – If a filesystem could not be opened for any reason.

**Returns** A filesystem instance.

**Return type** *FS*

### 13.12.2 fs.opener.parse

Function to parse FS URLs in to their constituent parts.

**class** fs.opener.parse.**ParseResult**
A named tuple containing fields of a parsed FS URL.

**protocol**
The protocol part of the url, e.g. `osfs` or `ftp`.

**Type** str

**username**
A username, or `None`.

**Type** str, optional

**password**
A password, or `None`.

**Type** str, optional

**resource**
A *resource*, typically a domain and path, e.g. `ftp.example.org/dir`.

**Type** str

**params**
A dictionary of parameters extracted from the query string.

**Type** dict

**path**
A path within the filesystem, or `None`.

**Type** str, optional

fs.opener.parse.**parse_fs_url**(*fs_url*)
Parse a Filesystem URL and return a *ParseResult*.

**Parameters** **fs_url** (*str*) – A filesystem URL.

**Returns** a parse result instance.

**Return type** *ParseResult*

**Raises** `ParseError` – if the FS URL is not valid.

### 13.12.3 fs.opener.registry

*Registry* class mapping protocols and FS URLs to their `Opener`.

**class** fs.opener.registry.**Registry**(*default_opener='osfs'*, *load_extern=False*)
A registry for `Opener` instances.

**__init__**(*default_opener='osfs'*, *load_extern=False*)
Create a registry object.

**Parameters**

- **default_opener** (*str, optional*) – The protocol to use, if one is not supplied. The default is to use 'osfs', so that the FS URL is treated as a system path if no protocol is given.
- **load_extern** (*bool, optional*) – Set to `True` to load openers from PyFilesystem2 extensions. Defaults to `False`.

**get_opener**(*protocol*)

Get the opener class associated to a given protocol.

**Parameters** **protocol** (*str*) – A filesystem protocol.

**Returns** an opener instance.

**Return type** *Opener*

**Raises**

- *UnsupportedProtocol* – If no opener could be found for the given protocol.
- `EntryPointLoadingError` – If the returned entry point is not an `Opener` subclass or could not be loaded successfully.

**install**(*opener*)

Install an opener.

**Parameters** **opener** (`Opener`) – an `Opener` instance, or a callable that returns an opener instance.

---

**Note:** May be used as a class decorator. For example:

```
registry = Registry()
@registry.install
class ArchiveOpener(Opener):
    protocols = ['zip', 'tar']
```

---

**manage_fs**(*fs_url*, *create=False*, *writeable=False*, *cwd='.'*)

Get a context manager to open and close a filesystem.

**Parameters**

- **fs_url** (*FS or str*) – A filesystem instance or a FS URL.
- **create** (*bool, optional*) – If `True`, then create the filesystem if it doesn't already exist.
- **writeable** (*bool, optional*) – If `True`, then the filesystem must be writeable.
- **cwd** (*str*) – The current working directory, if opening a *OSFS*.

Sometimes it is convenient to be able to pass either a FS object *or* an FS URL to a function. This context manager handles the required logic for that.

### Example

The *manage_fs* method can be used to define a small utility function:

```
>>> def print_ls(list_fs):
...     '''List a directory.'''
```

(continues on next page)

(continued from previous page)

```
...     with manage_fs(list_fs) as fs:
...         print(' '.join(fs.listdir()))
```

This function may be used in two ways. You may either pass a `str`, as follows:

```
>>> print_list('zip://projects.zip')
```

Or, an filesystem instance:

```
>>> from fs.osfs import OSFS
>>> projects_fs = OSFS('~/')
>>> print_list(projects_fs)
```

**open** (*fs_url*, *writeable=True*, *create=False*, *cwd='.'*, *default_protocol='osfs'*)
Open a filesystem from a FS URL.

Returns a tuple of a filesystem object and a path. If there is no path in the FS URL, the path value will be `None`.

**Parameters**

- **fs_url** (*str*) – A filesystem URL.
- **writeable** (*bool, optional*) – `True` if the filesystem must be writeable.
- **create** (*bool, optional*) – `True` if the filesystem should be created if it does not exist.
- **cwd** (*str*) – The current working directory.

**Returns** a tuple of `(<filesystem>, <path from url>)`

**Return type** (*FS*, str)

**open_fs** (*fs_url*, *writeable=False*, *create=False*, *cwd='.'*, *default_protocol='osfs'*)
Open a filesystem from a FS URL (ignoring the path component).

**Parameters**

- **fs_url** (*str*) – A filesystem URL. If a filesystem instance is given instead, it will be returned transparently.
- **writeable** (*bool, optional*) – `True` if the filesystem must be writeable.
- **create** (*bool, optional*) – `True` if the filesystem should be created if it does not exist.
- **cwd** (*str*) – The current working directory (generally only relevant for OS filesystems).
- **default_protocol** (*str*) – The protocol to use if one is not supplied in the FS URL (defaults to `"osfs"`).

**Returns** A filesystem instance.

**Return type** *FS*

**Caution:** The `writeable` parameter only controls whether the filesystem *needs* to be writable, which is relevant for some archive filesystems. Passing `writeable=False` will **not** make the return filesystem read-only. For this, consider using *`fs.wrap.read_only`* to wrap the returned instance.

**protocols**
the list of supported protocols.

**Type** list

### 13.12.4 fs.opener.errors

Errors raised when attempting to open a filesystem.

**exception** fs.opener.errors.**EntryPointError**
Bases: *fs.opener.errors.OpenerError*

An entry point could not be loaded.

**exception** fs.opener.errors.**NotWriteable**
Bases: *fs.opener.errors.OpenerError*

A writable FS could not be created.

**exception** fs.opener.errors.**OpenerError**
Bases: Exception

Base exception for opener related errors.

**exception** fs.opener.errors.**ParseError**
Bases: ValueError

Attempt to parse an invalid FS URL.

**exception** fs.opener.errors.**UnsupportedProtocol**
Bases: *fs.opener.errors.OpenerError*

No opener found for the given protocol.

## 13.13 fs.path

Useful functions for working with PyFilesystem paths.

This is broadly similar to the standard os.path module but works with paths in the canonical format expected by all FS objects (that is, separated by forward slashes and with an optional leading slash).

See *Paths* for an explanation of PyFilesystem paths.

fs.path.**abspath**(*path*)
Convert the given path to an absolute path.

Since FS objects have no concept of a *current directory*, this simply adds a leading / character if the path doesn't already have one.

**Parameters** **path** (*str*) – A PyFilesytem path.

**Returns** An absolute path.

**Return type** str

fs.path.**basename**(*path*)
Return the basename of the resource referenced by a path.

This is always equivalent to the 'tail' component of the value returned by split(path).

**Parameters** **path** (*str*) – A PyFilesytem path.

**Returns** the name of the resource at the given path.

**Return type** str

### Example

```
>>> basename('foo/bar/baz')
'baz'
>>> basename('foo/bar')
'bar'
>>> basename('foo/bar/')
''
```

fs.path.**combine**(*path1*, *path2*)

Join two paths together.

This is faster than *join()*, but only works when the second path is relative, and there are no back references in either path.

**Parameters**

- **path1** (*str*) – A PyFilesytem path.
- **path2** (*str*) – A PyFilesytem path.

**Returns** The joint path.

**Return type** str

### Example

```
>>> combine("foo/bar", "baz")
'foo/bar/baz'
```

fs.path.**dirname**(*path*)

Return the parent directory of a path.

This is always equivalent to the 'head' component of the value returned by `split(path)`.

**Parameters** **path** (*str*) – A PyFilesytem path.

**Returns** the parent directory of the given path.

**Return type** str

### Example

```
>>> dirname('foo/bar/baz')
'foo/bar'
>>> dirname('/foo/bar')
'/foo'
>>> dirname('/foo')
'/'
```

fs.path.**forcedir**(*path*)

Ensure the path ends with a trailing forward slash.

**Parameters** **path** (*str*) – A PyFilesytem path.

**Returns** The path, ending with a slash.

**Return type** str

### Example

```
>>> forcedir("foo/bar")
'foo/bar/'
>>> forcedir("foo/bar/")
'foo/bar/'
>>> forcedir("foo/spam.txt")
'foo/spam.txt/'
```

fs.path.**frombase**(*path1*, *path2*)

Get the final path of `path2` that isn't in `path1`.

**Parameters**

- **path1** (*str*) – A PyFilesytem path.
- **path2** (*str*) – A PyFilesytem path.

**Returns** the final part of `path2`.

**Return type** str

### Example

```
>>> frombase('foo/bar/', 'foo/bar/baz/egg')
'baz/egg'
```

fs.path.**isabs**(*path*)

Check if a path is an absolute path.

**Parameters** **path** (*str*) – A PyFilesytem path.

**Returns** `True` if the path is absolute (starts with a `'/'`).

**Return type** bool

fs.path.**isbase**(*path1*, *path2*)

Check if `path1` is a base of `path2`.

**Parameters**

- **path1** (*str*) – A PyFilesytem path.
- **path2** (*str*) – A PyFilesytem path.

**Returns** `True` if `path2` starts with `path1`

**Return type** bool

### Example

```
>>> isbase('foo/bar', 'foo/bar/baz/egg.txt')
True
```

fs.path.**isdotfile**(*path*)
Detect if a path references a dot file.

**Parameters** **path** (*str*) – Path to check.

**Returns** `True` if the resource name starts with a `'.'`.

**Return type** bool

### Example

```
>>> isdotfile('.baz')
True
>>> isdotfile('foo/bar/.baz')
True
>>> isdotfile('foo/bar.baz')
False
```

fs.path.**isparent**(*path1*, *path2*)
Check if `path1` is a parent directory of `path2`.

**Parameters**

- **path1** (*str*) – A PyFilesytem path.
- **path2** (*str*) – A PyFilesytem path.

**Returns** `True` if `path1` is a parent directory of `path2`

**Return type** bool

### Example

```
>>> isparent("foo/bar", "foo/bar/spam.txt")
True
>>> isparent("foo/bar/", "foo/bar")
True
>>> isparent("foo/barry", "foo/baz/bar")
False
>>> isparent("foo/bar/baz/", "foo/baz/bar")
False
```

fs.path.**issamedir**(*path1*, *path2*)
Check if two paths reference a resource in the same directory.

**Parameters**

- **path1** (*str*) – A PyFilesytem path.
- **path2** (*str*) – A PyFilesytem path.

**Returns** `True` if the two resources are in the same directory.

**Return type** bool

### Example

```
>>> issamedir("foo/bar/baz.txt", "foo/bar/spam.txt")
True
>>> issamedir("foo/bar/baz/txt", "spam/eggs/spam.txt")
False
```

fs.path.**iswildcard**(*path*)

Check if a path ends with a wildcard.

**Parameters** **path** (*str*) – A PyFilesystem path.

**Returns** `True` if path ends with a wildcard.

**Return type** bool

### Example

```
>>> iswildcard('foo/bar/baz.*')
True
>>> iswildcard('foo/bar')
False
```

fs.path.**iteratepath**(*path*)

Iterate over the individual components of a path.

**Parameters** **path** (*str*) – Path to iterate over.

**Returns** A list of path components.

**Return type** list

### Example

```
>>> iteratepath('/foo/bar/baz')
['foo', 'bar', 'baz']
```

fs.path.**join**(**paths*)

Join any number of paths together.

**Parameters** ***paths** (*str*) – Paths to join, given as positional arguments.

**Returns** The joined path.

**Return type** str

### Example

```
>>> join('foo', 'bar', 'baz')
'foo/bar/baz'
>>> join('foo/bar', '../baz')
'foo/baz'
>>> join('foo/bar', '/baz')
'/baz'
```

fs.path.**normpath**(*path*)

Normalize a path.

This function simplifies a path by collapsing back-references and removing duplicated separators.

**Parameters** **path** (*str*) – Path to normalize.

**Returns** A valid FS path.

**Return type** str

### Example

```
>>> normpath("/foo//bar/frob/../baz")
'/foo/bar/baz'
>>> normpath("foo/../../bar")
Traceback (most recent call last):
    ...
fs.errors.IllegalBackReference: path 'foo/../../bar' contains back-references␣
↪outside of filesystem
```

fs.path.**parts**(*path*)

Split a path in to its component parts.

**Parameters** **path** (*str*) – Path to split in to parts.

**Returns** List of components

**Return type** list

### Example

```
>>> parts('/foo/bar/baz')
['/', 'foo', 'bar', 'baz']
```

fs.path.**recursepath**(*path*, *reverse=False*)

Get intermediate paths from the root to the given path.

**Parameters**

- **path** (*str*) – A PyFilesystem path
- **reverse** (*bool*) – Reverses the order of the paths (default `False`).

**Returns** A list of paths.

**Return type** list

### Example

```
>>> recursepath('a/b/c')
['/', '/a', '/a/b', '/a/b/c']
```

fs.path.**relativefrom**(*base*, *path*)

Return a path relative from a given base path.

Insert backrefs as appropriate to reach the path from the base.

**Parameters**

- **base** (*str*) – Path to a directory.
- **path** (*str*) – Path to make relative.

**Returns** the path to `base` from `path`.

**Return type** str

```
>>> relativefrom("foo/bar", "baz/index.html")
'../../baz/index.html'
```

fs.path.**relpath**(*path*)

Convert the given path to a relative path.

This is the inverse of *abspath*, stripping a leading `'/'` from the path if it is present.

**Parameters** **path** (*str*) – A path to adjust.

**Returns** A relative path.

**Return type** str

### Example

```
>>> relpath('/a/b')
'a/b'
```

fs.path.**split**(*path*)

Split a path into (head, tail) pair.

This function splits a path into a pair (head, tail) where 'tail' is the last pathname component and 'head' is all preceding components.

**Parameters** **path** (*str*) – Path to split

**Returns** a tuple containing the head and the tail of the path.

**Return type** (str, str)

### Example

```
>>> split("foo/bar")
('foo', 'bar')
>>> split("foo/bar/baz")
('foo/bar', 'baz')
>>> split("/foo/bar/baz")
('/foo/bar', 'baz')
```

fs.path.**splitext**(*path*)

Split the extension from the path.

**Parameters** **path** (*str*) – A path to split.

**Returns** A tuple containing the path and the extension.

**Return type** (str, str)

### Example

```
>>> splitext('baz.txt')
('baz', '.txt')
>>> splitext('foo/bar/baz.txt')
('foo/bar/baz', '.txt')
>>> splitext('foo/bar/.foo')
('foo/bar/.foo', '')
```

## 13.14 fs.permissions

Abstract permissions container.

**class** fs.permissions.**Permissions**(*names=None*, *mode=None*, *user=None*, *group=None*, *other=None*, *sticky=None*, *setuid=None*, *setguid=None*)

An abstraction for file system permissions.

Permissions objects store information regarding the permissions on a resource. It supports Linux permissions, but is generic enough to manage permission information from almost any filesystem.

### Example

```
>>> from fs.permissions import Permissions
>>> p = Permissions(user='rwx', group='rw-', other='r--')
>>> print(p)
rwxrw-r--
>>> p.mode
500
>>> oct(p.mode)
'0o764'
```

**__init__**(*names=None*, *mode=None*, *user=None*, *group=None*, *other=None*, *sticky=None*, *setuid=None*, *setguid=None*)

Create a new *Permissions* instance.

**Parameters**

- **names** (*list, optional*) – A list of permissions.
- **mode** (*int, optional*) – A mode integer.
- **user** (*str, optional*) – A triplet of *user* permissions, e.g. `"rwx"` or `"r--"`
- **group** (*str, optional*) – A triplet of *group* permissions, e.g. `"rwx"` or `"r--"`
- **other** (*str, optional*) – A triplet of *other* permissions, e.g. `"rwx"` or `"r--"`
- **sticky** (*bool, optional*) – A boolean for the *sticky* bit.
- **setuid** (*bool, optional*) – A boolean for the *setuid* bit.
- **setguid** (*bool, optional*) – A boolean for the *setguid* bit.

**add**(**permissions*)

Add permission(s).

**Parameters** ***permissions** (*str*) – Permission name(s), such as `'u_w'` or `'u_x'`.

**as_str**()

Get a Linux-style string representation of permissions.

**check**(**permissions*)
Check if one or more permissions are enabled.

**Parameters** ***permissions** (*str*) – Permission name(s), such as `'u_w'` or `'u_x'`.

**Returns** `True` if all given permissions are set.

**Return type** bool

**copy**()
Make a copy of this permissions object.

**classmethod create**(*init=None*)
Create a permissions object from an initial value.

**Parameters** **init** (*int or list, optional*) – May be None to use `0o777` permissions, a mode integer, or a list of permission names.

**Returns** mode integer that may be used for instance by `os.makedir`.

**Return type** int

### Example

```
>>> Permissions.create(None)
Permissions(user='rwx', group='rwx', other='rwx')
>>> Permissions.create(0o700)
Permissions(user='rwx', group='', other='')
>>> Permissions.create(['u_r', 'u_w', 'u_x'])
Permissions(user='rwx', group='', other='')
```

**dump**()
Get a list suitable for serialization.

**g_r**
Boolean for 'g_r' permission.

**g_w**
Boolean for 'g_w' permission.

**g_x**
Boolean for 'g_x' permission.

**classmethod get_mode**(*init*)
Convert an initial value to a mode integer.

**classmethod load**(*permissions*)
Load a serialized permissions object.

**mode**
mode integer.

**Type** int

**o_r**
Boolean for 'o_r' permission.

**o_w**
Boolean for 'o_w' permission.

**o_x**
Boolean for 'o_x' permission.

**classmethod parse**(*ls*)
Parse permissions in Linux notation.

**remove**(**permissions*)
Remove permission(s).

**Parameters** ***permissions** (*str*) – Permission name(s), such as `'u_w'` or `'u_x'`.s

**setguid**
Boolean for 'setguid' permission.

**setuid**
Boolean for 'setuid' permission.

**sticky**
Boolean for 'sticky' permission.

**u_r**
Boolean for 'u_r' permission.

**u_w**
Boolean for 'u_w' permission.

**u_x**
Boolean for 'u_x' permission.

fs.permissions.**make_mode**(*init*)
Make a mode integer from an initial value.

## 13.15 fs.tools

Miscellaneous tools for operating on filesystems.

fs.tools.**copy_file_data**(*src_file*, *dst_file*, *chunk_size=None*)
Copy data from one file object to another.

**Parameters**

- **src_file** (*io.IOBase*) – File open for reading.
- **dst_file** (*io.IOBase*) – File open for writing.
- **chunk_size** (*int*) – Number of bytes to copy at a time (or `None` to use sensible default).

fs.tools.**get_intermediate_dirs**(*fs*, *dir_path*)
Get a list of non-existing intermediate directories.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **dir_path** (*str*) – A path to a new directory on the filesystem.

**Returns** A list of non-existing paths.

**Return type** list

**Raises** *DirectoryExpected* – If a path component references a file and not a directory.

fs.tools.**is_thread_safe**(**filesystems*)
Check if all filesystems are thread-safe.

**Parameters** **filesystems** (FS) – Filesystems instances to check.

**Returns** if all filesystems are thread safe.

**Return type** bool

`fs.tools.`**`remove_empty`**(*fs*, *path*)
Remove all empty parents.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – Path to a directory on the filesystem.

## 13.16 fs.tree

Render a FS object as text tree views.

Color is supported on UNIX terminals.

`fs.tree.`**`render`**(*fs*, *path='/'*, *file=None*, *encoding=None*, *max_levels=5*, *with_color=None*, *dirs_first=True*, *exclude=None*, *filter=None*)
Render a directory structure in to a pretty tree.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – The path of the directory to start rendering from (defaults to root folder, i.e. `'/'`).
- **file** (*io.IOBase*) – An open file-like object to render the tree, or None for stdout.
- **encoding** (*str, optional*) – Unicode encoding, or None to auto-detect.
- **max_levels** (*int, optional*) – Maximum number of levels to display, or None for no maximum.
- **with_color** (*bool, optional*) – Enable terminal color output, or None to auto-detect terminal.
- **dirs_first** (*bool*) – Show directories first.
- **exclude** (*list, optional*) – Option list of directory patterns to exclude from the tree render.
- **filter** (*list, optional*) – Optional list of files patterns to match in the tree render.

**Returns** A tuple of `(<directory count>, <file count>)`.

**Return type** (int, int)

## 13.17 fs.walk

Machinery for walking a filesystem.

*Walking* a filesystem means recursively visiting a directory and any sub-directories. It is a fairly common requirement for copying, searching etc. See *Walking* for details.

**class** `fs.walk.`**`BoundWalker`**(*fs*, *walker_class=<class 'fs.walk.Walker'>*)
A class that binds a *Walker* instance to a FS instance.

You will typically not need to create instances of this class explicitly. Filesystems have a `walk` property which returns a *BoundWalker* object.

### Example

```
>>> tmp_fs = fs.tempfs.TempFS()
>>> tmp_fs.walk
BoundWalker(TempFS())
```

A *BoundWalker* is callable. Calling it is an alias for the *walk* method.

**__call__**(*path='/'*, *namespaces=None*, ***kwargs*)

Walk the directory structure of a filesystem.

**Parameters**

- **path** (*str*) –
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']` (defaults to `['basic']`).

**Keyword Arguments**

- **ignore_errors** (*bool*) – If `True`, any errors reading a directory will be ignored, otherwise exceptions will be raised.
- **on_error** (*callable*) – If `ignore_errors` is `False`, then this callable will be invoked with a path and the exception object. It should return `True` to ignore the error, or `False` to re-raise it.
- **search** (*str*) – If `'breadth'` then the directory will be walked *top down*. Set to `'depth'` to walk *bottom up*.
- **filter** (*list*) – If supplied, this parameter should be a list of file name patterns, e.g. `['*.py']`. Files will only be returned if the final component matches one of the patterns.
- **exclude** (*list, optional*) – If supplied, this parameter should be a list of filename patterns, e.g. `['~*', '.*']`. Files matching any of these patterns will be removed from the walk.
- **filter_dirs** (*list, optional*) – A list of patterns that will be used to match directories paths. The walk will only open directories that match at least one of these patterns.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.
- **max_depth** (*int, optional*) – Maximum directory depth to walk.

**Returns** an iterator of `(<path>, <dirs>, <files>)` named tuples, where `<path>` is an absolute path to a directory, and `<dirs>` and `<files>` are a list of *Info* objects for directories and files in `<path>`.

**Return type** Iterator

### Example

```
>>> walker = Walker(filter=['*.py'])
>>> for path, dirs, files in walker.walk(my_fs, namespaces=['details']):
...     print("[{}]".format(path))
...     print("{} directories".format(len(dirs)))
...     total = sum(info.size for info in files)
...     print("{} bytes".format(total))
[/]
2 directories
55 bytes
...
```

This method invokes *Walker.walk* with bound FS object.

**__init__** (*fs*, *walker_class=<class 'fs.walk.Walker'>*)
Create a new walker bound to the given filesystem.

**Parameters**

- **fs** (*FS*) – A filesystem instance.
- **walker_class** (*type*) – A WalkerBase sub-class. The default uses *Walker*.

**dirs** (*path='/'*, ***kwargs*)
Walk a filesystem, yielding absolute paths to directories.

**Parameters** **path** (*str*) – A path to a directory.

**Keyword Arguments**

- **ignore_errors** (*bool*) – If True, any errors reading a directory will be ignored, otherwise exceptions will be raised.
- **on_error** (*callable*) – If ignore_errors is False, then this callable will be invoked with a path and the exception object. It should return True to ignore the error, or False to re-raise it.
- **search** (*str*) – If 'breadth' then the directory will be walked *top down*. Set to 'depth' to walk *bottom up*.
- **filter_dirs** (*list, optional*) – A list of patterns that will be used to match directories paths. The walk will only open directories that match at least one of these patterns.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. ['*.svn', '*.git'].
- **max_depth** (*int, optional*) – Maximum directory depth to walk.

**Returns** an iterator over directory paths (absolute from the filesystem root).

**Return type** Iterator

This method invokes *Walker.dirs* with the bound FS object.

**files** (*path='/'*, ***kwargs*)
Walk a filesystem, yielding absolute paths to files.

**Parameters** **path** (*str*) – A path to a directory.

**Keyword Arguments**

- **ignore_errors** (*bool*) – If True, any errors reading a directory will be ignored, otherwise exceptions will be raised.

- **on_error** (*callable*) – If `ignore_errors` is `False`, then this callable will be invoked with a path and the exception object. It should return `True` to ignore the error, or `False` to re-raise it.
- **search** (*str*) – If `'breadth'` then the directory will be walked *top down*. Set to `'depth'` to walk *bottom up*.
- **filter** (*list*) – If supplied, this parameter should be a list of file name patterns, e.g. `['*.py']`. Files will only be returned if the final component matches one of the patterns.
- **exclude** (*list, optional*) – If supplied, this parameter should be a list of filename patterns, e.g. `['~*', '.*']`. Files matching any of these patterns will be removed from the walk.
- **filter_dirs** (*list, optional*) – A list of patterns that will be used to match directories paths. The walk will only open directories that match at least one of these patterns.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.
- **max_depth** (*int, optional*) – Maximum directory depth to walk.

**Returns** An iterator over file paths (absolute from the filesystem root).

**Return type** Iterator

This method invokes *Walker.files* with the bound `FS` object.

**info**(*path='/'*, *namespaces=None*, ***kwargs*)

Walk a filesystem, yielding path and `Info` of resources.

**Parameters**

- **path** (*str*) – A path to a directory.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']` (defaults to `['basic']`).

**Keyword Arguments**

- **ignore_errors** (*bool*) – If `True`, any errors reading a directory will be ignored, otherwise exceptions will be raised.
- **on_error** (*callable*) – If `ignore_errors` is `False`, then this callable will be invoked with a path and the exception object. It should return `True` to ignore the error, or `False` to re-raise it.
- **search** (*str*) – If `'breadth'` then the directory will be walked *top down*. Set to `'depth'` to walk *bottom up*.
- **filter** (*list*) – If supplied, this parameter should be a list of file name patterns, e.g. `['*.py']`. Files will only be returned if the final component matches one of the patterns.
- **exclude** (*list, optional*) – If supplied, this parameter should be a list of filename patterns, e.g. `['~*', '.*']`. Files matching any of these patterns will be removed from the walk.
- **filter_dirs** (*list, optional*) – A list of patterns that will be used to match directories paths. The walk will only open directories that match at least one of these patterns.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.

- **max_depth** (*int, optional*) – Maximum directory depth to walk.

**Returns** an iterable yielding tuples of `(<absolute path>, <resource info>)`.

**Return type** Iterable

This method invokes *Walker.info* with the bound `FS` object.

**walk**(*path='/'*, *namespaces=None*, ***kwargs*)

Walk the directory structure of a filesystem.

**Parameters**

- **path** (*str*) –
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']` (defaults to `['basic']`).

**Keyword Arguments**

- **ignore_errors** (*bool*) – If `True`, any errors reading a directory will be ignored, otherwise exceptions will be raised.
- **on_error** (*callable*) – If `ignore_errors` is `False`, then this callable will be invoked with a path and the exception object. It should return `True` to ignore the error, or `False` to re-raise it.
- **search** (*str*) – If `'breadth'` then the directory will be walked *top down*. Set to `'depth'` to walk *bottom up*.
- **filter** (*list*) – If supplied, this parameter should be a list of file name patterns, e.g. `['*.py']`. Files will only be returned if the final component matches one of the patterns.
- **exclude** (*list, optional*) – If supplied, this parameter should be a list of filename patterns, e.g. `['~*', '.*']`. Files matching any of these patterns will be removed from the walk.
- **filter_dirs** (*list, optional*) – A list of patterns that will be used to match directories paths. The walk will only open directories that match at least one of these patterns.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.
- **max_depth** (*int, optional*) – Maximum directory depth to walk.

**Returns** an iterator of `(<path>, <dirs>, <files>)` named tuples, where `<path>` is an absolute path to a directory, and `<dirs>` and `<files>` are a list of *Info* objects for directories and files in `<path>`.

**Return type** Iterator

### Example

```
>>> walker = Walker(filter=['*.py'])
>>> for path, dirs, files in walker.walk(my_fs, namespaces=['details']):
...     print("[{}]".format(path))
...     print("{} directories".format(len(dirs)))
...     total = sum(info.size for info in files)
...     print("{} bytes".format(total))
[/]
2 directories
```

(continues on next page)

(continued from previous page)

```
55 bytes
...
```

This method invokes *Walker.walk* with bound FS object.

**class** fs.walk.**Step**(*path*, *dirs*, *files*)
type: a *step* in a directory walk.

**dirs**
Alias for field number 1

**files**
Alias for field number 2

**path**
Alias for field number 0

**class** fs.walk.**Walker**(*ignore_errors=False*, *on_error=None*, *search='breadth'*, *filter=None*, *exclude=None*, *filter_dirs=None*, *exclude_dirs=None*, *max_depth=None*)
A walker object recursively lists directories in a filesystem.

**__init__**(*ignore_errors=False*, *on_error=None*, *search='breadth'*, *filter=None*, *exclude=None*, *filter_dirs=None*, *exclude_dirs=None*, *max_depth=None*)
Create a new *Walker* instance.

**Parameters**

- **ignore_errors** (*bool*) – If True, any errors reading a directory will be ignored, otherwise exceptions will be raised.
- **on_error** (*callable, optional*) – If ignore_errors is False, then this callable will be invoked for a path and the exception object. It should return True to ignore the error, or False to re-raise it.
- **search** (*str*) – If "breadth" then the directory will be walked *top down*. Set to "depth" to walk *bottom up*.
- **filter** (*list, optional*) – If supplied, this parameter should be a list of filename patterns, e.g. ["*.py"]. Files will only be returned if the final component matches one of the patterns.
- **exclude** (*list, optional*) – If supplied, this parameter should be a list of filename patterns, e.g. ["~*"]. Files matching any of these patterns will be removed from the walk.
- **filter_dirs** (*list, optional*) – A list of patterns that will be used to match directories paths. The walk will only open directories that match at least one of these patterns.
- **exclude_dirs** (*list, optional*) – A list of patterns that will be used to filter out directories from the walk. e.g. ['*.svn', '*.git'].
- **max_depth** (*int, optional*) – Maximum directory depth to walk.

**classmethod bind**(*fs*)
Bind a *Walker* instance to a given filesystem.

This *binds* in instance of the Walker to a given filesystem, so that you won't need to explicitly provide the filesystem as a parameter.

**Parameters** **fs** (FS) – A filesystem object.

**Returns** a bound walker.

**Return type** *BoundWalker*

### Examples

Use this method to explicitly bind a filesystem instance:

```
>>> walker = Walker.bind(my_fs)
>>> for path in walker.files(filter=['*.py']):
...     print(path)
/foo.py
/bar.py
```

Unless you have written a customized walker class, you will be unlikely to need to call this explicitly, as filesystem objects already have a `walk` attribute which is a bound walker object:

```
>>> for path in my_fs.walk.files(filter=['*.py']):
...     print(path)
/foo.py
/bar.py
```

**check_file**(*fs*, *info*)
Check if a filename should be included.

Override to exclude files from the walk.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **info** (Info) – A resource info object.

**Returns** `True` if the file should be included.

**Return type** bool

**check_open_dir**(*fs*, *path*, *info*)
Check if a directory should be opened.

Override to exclude directories from the walk.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – Path to directory.
- **info** (Info) – A resource info object for the directory.

**Returns** `True` if the directory should be opened.

**Return type** bool

**check_scan_dir**(*fs*, *path*, *info*)
Check if a directory should be scanned.

Override to omit scanning of certain directories. If a directory is omitted, it will appear in the walk but its files and sub-directories will not.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – Path to directory.

- **info** (Info) – A resource info object for the directory.

**Returns** True if the directory should be scanned.

**Return type** bool

**dirs** (*fs*, *path='/'*)
Walk a filesystem, yielding absolute paths to directories.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – A path to a directory on the filesystem.

**Yields** *str* – absolute path to directories on the filesystem found recursively within the given directory.

**files** (*fs*, *path='/'*)
Walk a filesystem, yielding absolute paths to files.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – A path to a directory on the filesystem.

**Yields** *str* – absolute path to files on the filesystem found recursively within the given directory.

**info** (*fs*, *path='/'*, *namespaces=None*)
Walk a filesystem, yielding tuples of `(<path>, <info>)`.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of additional namespaces to add to the Info objects.

**Yields** *(str, Info)* – a tuple of `(<absolute path>, <resource info>)`.

**walk** (*fs*, *path='/'*, *namespaces=None*)
Walk the directory structure of a filesystem.

**Parameters**

- **fs** (FS) – A filesystem instance.
- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of additional namespaces to add to the Info objects.

**Returns** an iterator of *Step* instances.

**Return type** collections.Iterator

The return value is an iterator of `(<path>, <dirs>, <files>)` named tuples, where `<path>` is an absolute path to a directory, and `<dirs>` and `<files>` are a list of *Info* objects for directories and files in `<path>`.

### Example

```
>>> walker = Walker(filter=['*.py'])
>>> for path, dirs, files in walker.walk(my_fs, namespaces=["details"]):
...     print("[{}]".format(path))
...     print("{} directories".format(len(dirs)))
...     total = sum(info.size for info in files)
...     print("{} bytes".format(total))
[/]
2 directories
55 bytes
...
```

## 13.18 fs.wildcard

Match wildcard filenames.

`fs.wildcard.`**`get_matcher`**(*patterns*, *case_sensitive*)

Get a callable that matches names against the given patterns.

**Parameters**

- **`patterns`** (*list*) – A list of wildcard pattern. e.g. `["*.py", "*.pyc"]`
- **`case_sensitive`** (*bool*) – If `True`, then the callable will be case sensitive, otherwise it will be case insensitive.

**Returns** a matcher that will return `True` if the name given as an argument matches any of the given patterns.

**Return type** callable

### Example

```
>>> from fs import wildcard
>>> is_python = wildcard.get_matcher(['*.py'], True)
>>> is_python('__init__.py')
True
>>> is_python('foo.txt')
False
```

`fs.wildcard.`**`imatch`**(*pattern*, *name*)

Test whether a name matches a wildcard pattern (case insensitive).

**Parameters**

- **`pattern`** (*str*) – A wildcard pattern, e.g. `"*.py"`.
- **`name`** (*bool*) – A filename.

**Returns** `True` if the filename matches the pattern.

**Return type** bool

`fs.wildcard.`**`imatch_any`**(*patterns*, *name*)

Test if a name matches any of a list of patterns (case insensitive).

Will return `True` if `patterns` is an empty list.

**Parameters**

- **patterns** (*list*) – A list of wildcard pattern, e.g `["*.py", "*.pyc"]`
- **name** (*str*) – A filename.

**Returns** `True` if the name matches at least one of the patterns.

**Return type** bool

`fs.wildcard.`**match**(*pattern*, *name*)

Test whether a name matches a wildcard pattern.

**Parameters**

- **pattern** (*str*) – A wildcard pattern, e.g. `"*.py"`.
- **name** (*str*) – A filename.

**Returns** `True` if the filename matches the pattern.

**Return type** bool

`fs.wildcard.`**match_any**(*patterns*, *name*)

Test if a name matches any of a list of patterns.

Will return `True` if `patterns` is an empty list.

**Parameters**

- **patterns** (*list*) – A list of wildcard pattern, e.g `["*.py", "*.pyc"]`
- **name** (*str*) – A filename.

**Returns** `True` if the name matches at least one of the patterns.

**Return type** bool

## 13.19 fs.wrap

Collection of useful *WrapFS* subclasses.

Here's an example that opens a filesystem then makes it *read only*:

```
>>> home_fs = fs.open_fs('~')
>>> read_only_home_fs = fs.wrap.read_only(home_fs)
>>> read_only_home_fs.removedir('Desktop')
Traceback (most recent call last):
  ...
fs.errors.ResourceReadOnly: resource 'Desktop' is read only
```

**class** `fs.wrap.`**WrapCachedDir**(*wrap_fs*)

Caches filesystem directory information.

This filesystem caches directory information retrieved from a scandir call. This *may* speed up code that calls `isdir`, `isfile`, or `gettype` too frequently.

---

**Note:** Using this wrap will prevent changes to directory information being visible to the filesystem object. Consequently it is best used only in a fairly limited scope where you don't expected anything on the filesystem to change.

---

**__init__**(*wrap_fs*)
Create a filesystem. See help(type(self)) for accurate signature.

**getinfo**(*path*, *namespaces=None*)
Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**isdir**(*path*)
Check if a path maps to an existing directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a directory.

**Return type** bool

**isfile**(*path*)
Check if a path maps to an existing file.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a file.

**Return type** bool

**scandir**(*path*, *namespaces=None*, *page=None*)
Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**class** fs.wrap.**WrapReadOnly**(*wrap_fs*)
Makes a Filesystem read-only.

Any call that would would write data or modify the filesystem in any way will raise a *ResourceReadOnly* exception.

**appendbytes**(*path*, *data*)
Append bytes to the end of a file, creating it if needed.

**Parameters**

- **path** (*str*) – Path to a file.
- **data** (*bytes*) – Bytes to append.

**Raises**

- TypeError – If `data` is not a `bytes` instance.
- *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

**appendtext**(*path*, *text*, *encoding='utf-8'*, *errors=None*, *newline=''*)
Append text to the end of a file, creating it if needed.

**Parameters**

- **path** (*str*) – Path to a file.
- **text** (*str*) – Text to append.
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`).
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.

**Raises**

- TypeError – if `text` is not an unicode string.
- *fs.errors.ResourceNotFound* – if a parent directory of `path` does not exist.

**copy**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Copy file contents from `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source file.
- **dst_path** (*str*) – Path to destination file.
- **overwrite** (*bool*) – If `True`, overwrite the destination file if it exists (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Raises**

- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.
- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.
- *fs.errors.FileExpected* – If `src_path` is not a file.

**create**(*path*, *wipe=False*)
Create an empty file.

The default behavior is to create a new file if one doesn't already exist. If `wipe` is `True`, any existing file will be truncated.

**Parameters**

- **path** (*str*) – Path to a new file in the filesystem.
- **wipe** (*bool*) – If `True`, truncate any existing file to 0 bytes (defaults to `False`).

**Returns** `True` if a new file had to be created.

**Return type** bool

**getmeta**(*namespace='standard'*)

Get meta information regarding a filesystem.

**Parameters** **namespace** (*str*) – The meta namespace (defaults to `"standard"`).

**Returns** the meta information.

**Return type** dict

Meta information is associated with a *namespace* which may be specified with the `namespace` parameter. The default namespace, `"standard"`, contains common information regarding the filesystem's capabilities. Some filesystems may provide other namespaces which expose less common or implementation specific information. If a requested namespace is not supported by a filesystem, then an empty dictionary will be returned.

The `"standard"` namespace supports the following keys:

| key | Description |
|---|---|
| case_insensitive | `True` if this filesystem is case insensitive. |
| invalid_path_chars | A string containing the characters that may not be used on this filesystem. |
| max_path_length | Maximum number of characters permitted in a path, or `None` for no limit. |
| max_sys_path_length | Maximum number of characters permitted in a sys path, or `None` for no limit. |
| network | `True` if this filesystem requires a network. |
| read_only | `True` if this filesystem is read only. |
| supports_rename | `True` if this filesystem supports an `os.rename` operation. |

Most builtin filesystems will provide all these keys, and third- party filesystems should do so whenever possible, but a key may not be present if there is no way to know the value.

**Note:** Meta information is constant for the lifetime of the filesystem, and may be cached.

**makedir**(*path*, *permissions=None*, *recreate=False*)

Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (*Permissions, optional*) – a `Permissions` instance, or `None` to use default.
- **recreate** (*bool*) – Set to `True` to avoid raising an error if the directory already exists (defaults to `False`).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.

- *fs.errors.ResourceNotFound* – If the path is not found.

**makedirs**(*path*, *permissions=None*, *recreate=False*)
Make a directory, and any missing intermediate directories.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (Permissions, *optional*) – Initial permissions, or None to use defaults.
- **recreate** (*bool*) – If False (the default), attempting to create an existing directory will raise an error. Set to True to ignore existing directories.

**Returns** A sub-directory filesystem.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – if the path is already a directory, and recreate is False.
- *fs.errors.DirectoryExpected* – if one of the ancestors in the path is not a directory.

**move**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Move a file from src_path to dst_path.

**Parameters**

- **src_path** (*str*) – A path on the filesystem to move.
- **dst_path** (*str*) – A path on the filesystem where the source file will be written to.
- **overwrite** (*bool*) – If True, destination path will be overwritten if it exists.
- **preserve_time** (*bool*) – If True, try to preserve mtime of the resources (defaults to False).

**Raises**

- *fs.errors.FileExpected* – If src_path maps to a directory instead of a file.
- *fs.errors.DestinationExists* – If dst_path exists, and overwrite is False.
- *fs.errors.ResourceNotFound* – If a parent directory of dst_path does not exist.

**open**(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=''*, *line_buffering=False*, ***options*)
Open a file.

**Parameters**

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open the file object with (defaults to *r*).
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, 1 to select line buffering, of any positive integer to indicate a buffer size).
- **encoding** (*str*) – Encoding for text files (defaults to utf-8)
- **errors** (*str, optional*) – What to do with unicode decode errors (see codecs module for more information).

- **newline** (*str*) – Newline parameter.
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If the path is not a file.
- *fs.errors.FileExists* – If the file exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If the path does not exist.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)

Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**remove**(*path*)

Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)

Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).

- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**removetree**(*path*)

Recursively remove a directory and all its contents.

This method is similar to *removedir*, but will remove the contents of the directory if it is not empty.

**Parameters** **dir_path** (*str*) – Path to a directory on the filesystem.

**Raises**

- *fs.errors.ResourceNotFound* – If `dir_path` does not exist.
- *fs.errors.DirectoryExpected* – If `dir_path` is not a directory.

**Caution:** A filesystem should never delete its root folder, so `FS.removetree("/")` has different semantics: the contents of the root folder will be deleted, but the root will be untouched:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.removetree("/")
>>> home_fs.exists("/")
True
>>> home_fs.isempty("/")
True
```

Combined with *opendir*, this can be used to clear a directory without removing the directory itself:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.opendir("/Videos").removetree("/")
>>> home_fs.exists("/Videos")
True
>>> home_fs.isempty("/Videos")
True
```

**setinfo**(*path*, *info*)

Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**settimes** (*path*, *accessed=None*, *modified=None*)
Set the accessed and modified time on a resource.

**Parameters**

- **path** – A path to a resource on the filesystem.
- **accessed** (*datetime, optional*) – The accessed time, or `None` (the default) to use the current time.
- **modified** (*datetime, optional*) – The modified time, or `None` (the default) to use the same time as the `accessed` parameter.

**touch** (*path*)
Touch a file on the filesystem.

Touching a file means creating a new file if `path` doesn't exist, or update accessed and modified times if the path does exist. This method is similar to the linux command of the same name.

**Parameters** **path** (*str*) – A path to a file on the filesystem.

**upload** (*path*, *file*, *chunk_size=None*, ***options*)
Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**writebytes** (*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** `TypeError` – if contents is not bytes.

**writefile** (*path*, *file*, *encoding=None*, *errors=None*, *newline=''*)
Set a file to the contents of a file object.

**Parameters**

- **path** (*str*) – A path on the filesystem.

- **file** (*io.IOBase*) – A file object open for reading.
- **encoding** (*str, optional*) – Encoding of destination file, defaults to `None` for binary.
- **errors** (*str, optional*) – How encoding errors should be treated (same as `io.open`).
- **newline** (*str*) – Newline parameter (same as `io.open`).

This method is similar to `upload`, in that it copies data from a file-like object to a resource on the filesystem, but unlike `upload`, this method also supports creating files in text-mode (if the `encoding` argument is supplied).

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('myfile.txt') as read_file:
...     my_fs.writefile('myfile.txt', read_file)
```

**writetext**(*path*, *contents*, *encoding='utf-8'*, *errors=None*, *newline=''*)

Create or replace a file with text.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*str*) – Text to be written.
- **encoding** (*str, optional*) – Encoding of destination file (defaults to `'utf-8'`).
- **errors** (*str, optional*) – How encoding errors should be treated (same as `io.open`).
- **newline** (*str*) – Newline parameter (same as `io.open`).

**Raises** `TypeError` – if `contents` is not a unicode string.

`fs.wrap.`**cache_directory**(*fs*)

Make a filesystem that caches directory information.

**Parameters** **fs** (`FS`) – A filesystem instance.

**Returns** A filesystem that caches results of `scandir`, `isdir` and other methods which read directory information.

**Return type** *FS*

`fs.wrap.`**read_only**(*fs*)

Make a read-only filesystem.

**Parameters** **fs** (`FS`) – A filesystem instance.

**Returns** A read only version of `fs`

**Return type** *FS*

## 13.20 fs.wrapfs

Base class for filesystem wrappers.

**class** fs.wrapfs.**WrapFS**(*wrap_fs*)
A proxy for a filesystem object.

This class exposes an filesystem interface, where the data is stored on another filesystem(s), and is the basis for *SubFS* and other *virtual* filesystems.

**__init__**(*wrap_fs*)
Create a filesystem. See help(type(self)) for accurate signature.

**appendbytes**(*path*, *data*)
Append bytes to the end of a file, creating it if needed.

**Parameters**

- **path** (*str*) – Path to a file.
- **data** (*bytes*) – Bytes to append.

**Raises**

- TypeError – If `data` is not a `bytes` instance.
- *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

**appendtext**(*path*, *text*, *encoding='utf-8'*, *errors=None*, *newline=''*)
Append text to the end of a file, creating it if needed.

**Parameters**

- **path** (*str*) – Path to a file.
- **text** (*str*) – Text to append.
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`).
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.

**Raises**

- TypeError – if `text` is not an unicode string.
- *fs.errors.ResourceNotFound* – if a parent directory of `path` does not exist.

**copy**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Copy file contents from `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source file.
- **dst_path** (*str*) – Path to destination file.
- **overwrite** (*bool*) – If `True`, overwrite the destination file if it exists (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Raises**

- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.
- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.
- *fs.errors.FileExpected* – If `src_path` is not a file.

**copydir**(*src_path*, *dst_path*, *create=False*, *preserve_time=False*)
Copy the contents of `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source directory.
- **dst_path** (*str*) – Path to destination directory.
- **create** (*bool*) – If `True`, then `dst_path` will be created if it doesn't exist already (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resource (defaults to `False`).

**Raises**

- *fs.errors.ResourceNotFound* – If the `dst_path` does not exist, and `create` is not `True`.
- *fs.errors.DirectoryExpected* – If `src_path` is not a directory.

**create**(*path*, *wipe=False*)
Create an empty file.

The default behavior is to create a new file if one doesn't already exist. If `wipe` is `True`, any existing file will be truncated.

**Parameters**

- **path** (*str*) – Path to a new file in the filesystem.
- **wipe** (*bool*) – If `True`, truncate any existing file to 0 bytes (defaults to `False`).

**Returns** `True` if a new file had to be created.

**Return type** bool

**delegate_fs**()
Get the proxied filesystem.

This method should return a filesystem for methods not associated with a path, e.g. *getmeta*.

**delegate_path**(*path*)
Encode a path for proxied filesystem.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** a tuple of `(<filesystem>, <new_path>)`

**Return type** (*FS*, str)

**desc**(*path*)
Return a short descriptive text regarding a path.

**Parameters** **path** (*str*) – A path to a resource on the filesystem.

**Returns** a short description of the path.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**download**(*path*, *file*, *chunk_size=None*, ***options*)

Copy a file from the filesystem to a file-like object.

This may be more efficient that opening and copying files manually if the filesystem supplies an optimized method.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

**Parameters**

- **path** (*str*) – Path to a resource.
- **file** (*file-like*) – A file-like object open for writing in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

### Example

```
>>> with open('starwars.mov', 'wb') as write_file:
...     my_fs.download('/Videos/starwars.mov', write_file)
```

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

**exists**(*path*)

Check if a path maps to a resource.

**Parameters** **path** (*str*) – Path to a resource.

**Returns** `True` if a resource exists at the given path.

**Return type** bool

**filterdir**(*path*, *files=None*, *dirs=None*, *exclude_dirs=None*, *exclude_files=None*, *namespaces=None*, *page=None*)

Get an iterator of resource info, filtered by patterns.

This method enhances *scandir* with additional filtering functionality.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **files** (*list, optional*) – A list of UNIX shell-style patterns to filter file names, e.g. `['*.py']`.
- **dirs** (*list, optional*) – A list of UNIX shell-style patterns to filter directory names.
- **exclude_dirs** (*list, optional*) – A list of patterns used to exclude directories.
- **exclude_files** (*list, optional*) – A list of patterns used to exclude files.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.

- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**getinfo**(*path*, *namespaces=None*)

Get information about a resource on a filesystem.

**Parameters**

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list, optional*) – Info namespaces to query. The `"basic"` namespace is alway included in the returned info, whatever the value of `namespaces` may be.

**Returns** resource information object.

**Return type** *Info*

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

For more information regarding resource information, see *Resource Info*.

**getmeta**(*namespace='standard'*)

Get meta information regarding a filesystem.

**Parameters** **namespace** (*str*) – The meta namespace (defaults to `"standard"`).

**Returns** the meta information.

**Return type** dict

Meta information is associated with a *namespace* which may be specified with the `namespace` parameter. The default namespace, `"standard"`, contains common information regarding the filesystem's capabilities. Some filesystems may provide other namespaces which expose less common or implementation specific information. If a requested namespace is not supported by a filesystem, then an empty dictionary will be returned.

The `"standard"` namespace supports the following keys:

| key | Description |
|---|---|
| case_insensitive | `True` if this filesystem is case insensitive. |
| invalid_path_chars | A string containing the characters that may not be used on this filesystem. |
| max_path_length | Maximum number of characters permitted in a path, or `None` for no limit. |
| max_sys_path_length | Maximum number of characters permitted in a sys path, or `None` for no limit. |
| network | `True` if this filesystem requires a network. |
| read_only | `True` if this filesystem is read only. |
| supports_rename | `True` if this filesystem supports an `os.rename` operation. |

Most builtin filesystems will provide all these keys, and third- party filesystems should do so whenever possible, but a key may not be present if there is no way to know the value.

**Note:** Meta information is constant for the lifetime of the filesystem, and may be cached.

**getsize**(*path*)

Get the size (in bytes) of a resource.

**Parameters** **path** (*str*) – A path to a resource.

**Returns** the *size* of the resource.

**Return type** int

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

The *size* of a file is the total number of readable bytes, which may not reflect the exact number of bytes of reserved disk space (or other storage medium).

The size of a directory is the number of bytes of overhead use to store the directory entry.

**getsyspath**(*path*)

Get the *system path* of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the *system path* of the resource, if any.

**Return type** str

**Raises** *fs.errors.NoSysPath* – If there is no corresponding system path.

A system path is one recognized by the OS, that may be used outside of PyFilesystem (in an application or a shell for example). This method will get the corresponding system path that would be referenced by `path`.

Not all filesystems have associated system paths. Network and memory based filesystems, for example, may not physically store data anywhere the OS knows about. It is also possible for some paths to have a system path, whereas others don't.

This method will always return a str on Py3.* and unicode on Py2.7. See `getospath` if you need to encode the path as bytes.

If `path` doesn't have a system path, a *NoSysPath* exception will be thrown.

---

**Note:** A filesystem may return a system path even if no resource is referenced by that path – as long as it can be certain what that system path would be.

---

**gettype**(*path*)

Get the type of a resource.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** the type of the resource.

**Return type** *ResourceType*

**Raises** *fs.errors.ResourceNotFound* – if `path` does not exist.

A type of a resource is an integer that identifies the what the resource references. The standard type integers may be one of the values in the *ResourceType* enumerations.

The most common resource types, supported by virtually all filesystems are `directory` (1) and `file` (2), but the following types are also possible:

| ResourceType | value |
|---|---|
| unknown | 0 |
| directory | 1 |
| file | 2 |
| character | 3 |
| block_special_file | 4 |
| fifo | 5 |
| socket | 6 |
| symlink | 7 |

Standard resource types are positive integers, negative values are reserved for implementation specific resource types.

**geturl** (*path*, *purpose='download'*)
Get the URL to a given resource.

**Parameters**

- **path** (*str*) – A path on the filesystem
- **purpose** (*str*) – A short string that indicates which URL to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`: for instance, `OSFS` supports `'fs'`, which returns a FS URL (see *FS URLs*).

**Returns** a URL.

**Return type** str

**Raises** *fs.errors.NoURL* – If the path does not map to a URL.

**hash** (*path*, *name*)
Get the hash of a file's contents.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **name** (*str*) – One of the algorithms supported by the `hashlib` module, e.g. `"md5"` or `"sha256"`.

**Returns** The hex digest of the hash.

**Return type** str

**Raises**

- *fs.errors.UnsupportedHash* – If the requested hash is not supported.
- *fs.errors.ResourceNotFound* – If `path` does not exist.
- *fs.errors.FileExpected* – If `path` exists but is not a file.

**hassyspath** (*path*)
Check if a path maps to a system path.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if the resource at `path` has a *syspath*.

**Return type** bool

**hasurl** (*path*, *purpose='download'*)
Check if a path has a corresponding URL.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **purpose** (*str*) – A purpose parameter, as given in *geturl*.

**Returns** `True` if an URL for the given purpose exists.

**Return type** bool

**isdir**(*path*)
Check if a path maps to an existing directory.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a directory.

**Return type** bool

**isfile**(*path*)
Check if a path maps to an existing file.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a file.

**Return type** bool

**islink**(*path*)
Check if a path maps to a symlink.

**Parameters** **path** (*str*) – A path on the filesystem.

**Returns** `True` if `path` maps to a symlink.

**Return type** bool

**listdir**(*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *ResourceType*.

**Parameters** **path** (*str*) – A path to a directory on the filesystem

**Returns** list of names, relative to `path`.

**Return type** list

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**lock**()
Get a context manager that *locks* the filesystem.

Locking a filesystem gives a thread exclusive access to it. Other threads will block until the threads with the lock has left the context manager.

**Returns** a lock specific to the filesystem instance.

**Return type** threading.RLock

### Example

```
>>> with my_fs.lock():  # May block
...     # code here has exclusive access to the filesystem
...     pass
```

It is a good idea to put a lock around any operations that you would like to be *atomic*. For instance if you are copying files, and you don't want another thread to delete or modify anything while the copy is in progress.

Locking with this method is only required for code that calls multiple filesystem methods. Individual methods are thread safe already, and don't need to be locked.

---

**Note:** This only locks at the Python level. There is nothing to prevent other processes from modifying the filesystem outside of the filesystem instance.

---

**makedir** (*path*, *permissions=None*, *recreate=False*)
Make a directory.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (Permissions, *optional*) – a Permissions instance, or None to use default.
- **recreate** (*bool*) – Set to True to avoid raising an error if the directory already exists (defaults to False).

**Returns** a filesystem whose root is the new directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – If the path already exists.
- *fs.errors.ResourceNotFound* – If the path is not found.

**makedirs** (*path*, *permissions=None*, *recreate=False*)
Make a directory, and any missing intermediate directories.

**Parameters**

- **path** (*str*) – Path to directory from root.
- **permissions** (Permissions, *optional*) – Initial permissions, or None to use defaults.
- **recreate** (*bool*) – If False (the default), attempting to create an existing directory will raise an error. Set to True to ignore existing directories.

**Returns** A sub-directory filesystem.

**Return type** *SubFS*

**Raises**

- *fs.errors.DirectoryExists* – if the path is already a directory, and recreate is False.
- *fs.errors.DirectoryExpected* – if one of the ancestors in the path is not a directory.

**move**(*src_path*, *dst_path*, *overwrite=False*, *preserve_time=False*)
Move a file from `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – A path on the filesystem to move.
- **dst_path** (*str*) – A path on the filesystem where the source file will be written to.
- **overwrite** (*bool*) – If `True`, destination path will be overwritten if it exists.
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**Raises**

- *fs.errors.FileExpected* – If `src_path` maps to a directory instead of a file.
- *fs.errors.DestinationExists* – If `dst_path` exists, and `overwrite` is `False`.
- *fs.errors.ResourceNotFound* – If a parent directory of `dst_path` does not exist.

**movedir**(*src_path*, *dst_path*, *create=False*, *preserve_time=False*)
Move directory `src_path` to `dst_path`.

**Parameters**

- **src_path** (*str*) – Path of source directory on the filesystem.
- **dst_path** (*str*) – Path to destination directory.
- **create** (*bool*) – If `True`, then `dst_path` will be created if it doesn't exist already (defaults to `False`).
- **preserve_time** (*bool*) – If `True`, try to preserve mtime of the resources (defaults to `False`).

**Raises**

- *fs.errors.ResourceNotFound* – if `dst_path` does not exist, and `create` is `False`.
- *fs.errors.DirectoryExpected* – if `src_path` or one of its ancestors is not a directory.

**open**(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=''*, *line_buffering=False*, ***options*)
Open a file.

**Parameters**

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open the file object with (defaults to *r*).
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, 1 to select line buffering, of any positive integer to indicate a buffer size).
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`)
- **errors** (*str, optional*) – What to do with unicode decode errors (see `codecs` module for more information).
- **newline** (*str*) – Newline parameter.

- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If the path is not a file.
- *fs.errors.FileExists* – If the file exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If the path does not exist.

**openbin**(*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **mode** (*str*) – Mode to open file (must be a valid non-text mode, defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.
- **buffering** (*int*) – Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).
- ****options** – keyword arguments for any additional information required by the filesystem (if any).

**Returns** a *file-like* object.

**Return type** io.IOBase

**Raises**

- *fs.errors.FileExpected* – If `path` exists and is not a file.
- *fs.errors.FileExists* – If the `path` exists, and *exclusive mode* is specified (`x` in the mode).
- *fs.errors.ResourceNotFound* – If `path` does not exist and `mode` does not imply creating the file, or if any ancestor of `path` does not exist.

**opendir**(*path*, *factory=None*)
Get a filesystem object for a sub-directory.

**Parameters**

- **path** (*str*) – Path to a directory on the filesystem.
- **factory** (*callable, optional*) – A callable that when invoked with an FS instance and `path` will return a new FS object representing the sub-directory contents. If no `factory` is supplied then `subfs_class` will be used.

**Returns** A filesystem representing a sub-directory.

**Return type** *SubFS*

**Raises**

- *fs.errors.ResourceNotFound* – If `path` does not exist.
- *fs.errors.DirectoryExpected* – If `path` is not a directory.

**readbytes**(*path*)
Get the contents of a file as bytes.

**Parameters** **path** (*str*) – A path to a readable file on the filesystem.

**Returns** the file contents.

**Return type** bytes

**Raises**

- *fs.errors.FileExpected* – if `path` exists but is not a file.
- *fs.errors.ResourceNotFound* – if `path` does not exist.

**readtext**(*path*, *encoding=None*, *errors=None*, *newline=''*)
Get the contents of a file as a string.

**Parameters**

- **path** (*str*) – A path to a readable file on the filesystem.
- **encoding** (*str, optional*) – Encoding to use when reading contents in text mode (defaults to `None`, reading in binary mode).
- **errors** (*str, optional*) – Unicode errors parameter.
- **newline** (*str*) – Newlines parameter.

**Returns** file contents.

**Return type** str

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist.

**remove**(*path*)
Remove a file from the filesystem.

**Parameters** **path** (*str*) – Path of the file to remove.

**Raises**

- *fs.errors.FileExpected* – If the path is a directory.
- *fs.errors.ResourceNotFound* – If the path does not exist.

**removedir**(*path*)
Remove a directory from the filesystem.

**Parameters** **path** (*str*) – Path of the directory to remove.

**Raises**

- *fs.errors.DirectoryNotEmpty* – If the directory is not empty ( see *removetree* for a way to remove the directory contents).
- *fs.errors.DirectoryExpected* – If the path does not refer to a directory.
- *fs.errors.ResourceNotFound* – If no resource exists at the given path.
- *fs.errors.RemoveRootError* – If an attempt is made to remove the root directory (i.e. `'/'`)

**removetree**(*dir_path*)
Recursively remove a directory and all its contents.

This method is similar to *removedir*, but will remove the contents of the directory if it is not empty.

**Parameters** **dir_path** (*str*) – Path to a directory on the filesystem.

**Raises**

- *fs.errors.ResourceNotFound* – If `dir_path` does not exist.
- *fs.errors.DirectoryExpected* – If `dir_path` is not a directory.

**Caution:** A filesystem should never delete its root folder, so `FS.removetree("/")` has different semantics: the contents of the root folder will be deleted, but the root will be untouched:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.removetree("/")
>>> home_fs.exists("/")
True
>>> home_fs.isempty("/")
True
```

Combined with *opendir*, this can be used to clear a directory without removing the directory itself:

```
>>> home_fs = fs.open_fs("~")
>>> home_fs.opendir("/Videos").removetree("/")
>>> home_fs.exists("/Videos")
True
>>> home_fs.isempty("/Videos")
True
```

**scandir** (*path*, *namespaces=None*, *page=None*)

Get an iterator of resource info.

**Parameters**

- **path** (*str*) – A path to a directory on the filesystem.
- **namespaces** (*list, optional*) – A list of namespaces to include in the resource information, e.g. `['basic', 'access']`.
- **page** (*tuple, optional*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

**Returns** an iterator of `Info` objects.

**Return type** Iterator

**Raises**

- *fs.errors.DirectoryExpected* – If `path` is not a directory.
- *fs.errors.ResourceNotFound* – If `path` does not exist.

**setinfo** (*path*, *info*)

Set info on a resource.

This method is the complement to *getinfo* and is used to set info values on a resource.

**Parameters**

- **path** (*str*) – Path to a resource on the filesystem.
- **info** (*dict*) – Dictionary of resource info.

**Raises** *fs.errors.ResourceNotFound* – If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

### Example

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

**settimes** (*path*, *accessed=None*, *modified=None*)
Set the accessed and modified time on a resource.

**Parameters**

- **path** – A path to a resource on the filesystem.
- **accessed** (*datetime, optional*) – The accessed time, or `None` (the default) to use the current time.
- **modified** (*datetime, optional*) – The modified time, or `None` (the default) to use the same time as the `accessed` parameter.

**touch** (*path*)
Touch a file on the filesystem.

Touching a file means creating a new file if `path` doesn't exist, or update accessed and modified times if the path does exist. This method is similar to the linux command of the same name.

**Parameters** **path** (*str*) – A path to a file on the filesystem.

**upload** (*path*, *file*, *chunk_size=None*, ***options*)
Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – a file object open for reading in binary mode.
- **chunk_size** (*int, optional*) – Number of bytes to read at a time, if a simple copy is used, or `None` to use sensible default.
- ****options** – Implementation specific options required to open the source file.

**Raises** *fs.errors.ResourceNotFound* – If a parent directory of `path` does not exist.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

**validatepath** (*path*)
Validate a path, returning a normalized absolute path on sucess.

Many filesystems have restrictions on the format of paths they support. This method will check that `path` is valid on the underlaying storage mechanism and throw a *InvalidPath* exception if it is not.

**Parameters** **path** (*str*) – A path.

**Returns** A normalized, absolute path.

**Return type** str

**Raises**

- *fs.errors.InvalidPath* – If the path is invalid.
- *fs.errors.FilesystemClosed* – if the filesystem is closed.
- *fs.errors.InvalidCharsInPath* – If the path contains invalid characters.

**walk**
a walker bound to this filesystem.

**Type** *BoundWalker*

**writebytes**(*path*, *contents*)
Copy binary data to a file.

**Parameters**

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – Data to be written.

**Raises** TypeError – if contents is not bytes.

**writefile**(*path*, *file*, *encoding=None*, *errors=None*, *newline=''*)
Set a file to the contents of a file object.

**Parameters**

- **path** (*str*) – A path on the filesystem.
- **file** (*io.IOBase*) – A file object open for reading.
- **encoding** (*str, optional*) – Encoding of destination file, defaults to None for binary.
- **errors** (*str, optional*) – How encoding errors should be treated (same as io.open).
- **newline** (*str*) – Newline parameter (same as io.open).

This method is similar to upload, in that it copies data from a file-like object to a resource on the filesystem, but unlike upload, this method also supports creating files in text-mode (if the encoding argument is supplied).

Note that the file object file will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

### Example

```
>>> with open('myfile.txt') as read_file:
...     my_fs.writefile('myfile.txt', read_file)
```

CHAPTER 14

# Contributing to PyFilesystem

Pull Requests are very welcome for this project!

For bug fixes or new features, please file an issue before submitting a pull request. If the change isn't trivial, it may be best to wait for feedback. For a quicker response, contact Will McGugan directly.

## 14.1 `tox`

Most of the guidelines that follow can be checked with a particular `tox` environment. Having it installed will help you develop and verify your code locally without having to wait for our Continuous Integration pipeline to finish.

## 14.2 Tests

New code should have unit tests. We strive to have near 100% coverage. Get in touch, if you need assistance with the tests. You shouldn't refrain from opening a Pull Request even if all the tests were not added yet, or if not all of them are passing yet.

### 14.2.1 Dependencies

The dependency for running the tests can be found in the `tests/requirements.txt` file. If you're using `tox`, you won't have to install them manually. Otherwise, they can be installed with `pip`:

```
$ pip install -r tests/requirements.txt
```

### 14.2.2 Running (with `tox`)

Simply run in the repository folder to execute the tests for all available environments:

```
$ tox
```

Since this can take some time, you can use a single environment to run tests only once, for instance to run tests only with Python 3.9:

```
$ tox -e py39
```

### 14.2.3 Running (without `tox`)

Tests are written using the standard `unittest` framework. You should be able to run them using the standard library runner:

```
$ python -m unittest discover -vv
```

## 14.3 Coding Guidelines

This project runs on Python2.7 and Python3.X. Python2.7 will be dropped at some point, but for now, please maintain compatibility. PyFilesystem2 uses the `six` library to write version-agnostic Python code.

### 14.3.1 Style

The code (including the tests) should follow PEP8. You can check for the code style with:

```
$ tox -e codestyle
```

This will invoke `flake8` with some common plugins such as `flake8-comprehensions`.

### 14.3.2 Format

Please format new code with black, using the default settings. You can check whether the code is well-formatted with:

```
$ tox -e codeformat
```

### 14.3.3 Type annotations

The code is typechecked with `mypy`, and type annotations written as comments, to stay compatible with Python2. Run the typechecking with:

```
$ tox -e typecheck
```

## 14.4 Documentation

### 14.4.1 Dependencies

The documentation is built with Sphinx, using the ReadTheDocs theme. The dependencies are listed in `docs/requirements.txt` and can be installed with `pip`:

```
$ pip install -r docs/requirements.txt
```

### 14.4.2 Building

Run the following command to build the HTML documentation:

```
$ python setup.py build_sphinx
```

The documentation index will be written to the `build/sphinx/html/` directory.

### 14.4.3 Style

The API reference is written in the Python source, using docstrings in Google format. The documentation style can be checked with:

```
$ tox -e docstyle
```

CHAPTER 15

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## f

# Index

## Symbols



## A



## B

## C



## D



## E



## F

## G

## N



## O



## P



## R

## S



## T



## U

## V



## W



## Z