PyFilesystem Documentation

Release 2.0.10a2

Will McGugan

Sep 22, 2017

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.

Installing

You can install PyFilesystem with `pip` as follows:

```
pip install fs
```

Or to upgrade to the most recent version:

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

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.

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.

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']
```

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
|   - data.xml
```

```
|   - mountpoints.xml
|   - readme.txt
- lib.ini
- readme.txt
```

This can be a useful debugging aid!

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.settext('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.settext('reminder.txt', 'buy coffee')
```

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

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.

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.settext('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.

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, *getbytes()* will return the file contents as a bytes, and *gettext()* will read unicode text. These methods is generally preferable to explicitly opening files, as the FS object may have an optimized implementation.

Other *shortcut* methods are `setbin()`, *setbytes()*, *settext()*.

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.

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']))
```

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.

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.

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()*

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.

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

Info Objects

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

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.

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.

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.

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 the aren't supported by the filesystem.

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.

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.

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.

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.

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.

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.

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.

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.

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')
```

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.

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))
```

```
    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 compliment 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
)
```

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

Builtin Filesystems

App Filesystems

Filesystems for platform-specific application directories. A collection of filesystems that map to application specific locations defined by the OS.

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')`.

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')`.

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')`.

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')`.

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')`.

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')`.

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.

FTP Filesystem

Manage resources on a FTP server.

class `fs.ftpfs.`**FTPFS**(*host*, *user=u'anonymous'*, *passwd=u''*, *acct=u''*, *timeout=10*, *port=21*)

A FTP (File Transport Protocol) Filesystem.

Parameters

- **host** (*str*) – A FTP host, e.g. `'ftp.mirror.nl'`.
- **user** (*str*) – A username (default is `'anonymous'`)
- **passwd** – Password for the server, or `None` for anon.
- **acct** – FTP account.
- **timeout** (*int*) – Timeout for contacting server (in seconds).
- **port** (*int*) – Port number (default 21).

features

Get features dict from FTP server.

Memory Filesystem

Create and manage an in-memory filesystems.

class `fs.memoryfs.`**MemoryFS**

A filesystem that stores all file and directory information in memory. This makes them very fast, but non-permanent.

Memory filesystems are useful for caches, temporary data stores, unit testing, etc.

Memory filesystems require no parameters to their constructor. The following is how you would create a `MemoryFS` instance:

```
mem_fs = MemoryFS()
```

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')
```

exception `fs.mountfs.`**MountError**
Thrown when mounts conflict.

class `fs.mountfs.`**MountFS**(*auto_close=True*)
A virtual filesystem that maps directories on to other file-systems.

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

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

Parameters

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

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.

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

Adds a filesystem to the MultiFS.

Parameters

- **name** (*str*) – A unique name to refer to the filesystem being added.
- **fs** (*Filesystem or FS URL.*) – The filesystem to add.
- **write** (*bool*) – If this value is True, then the `fs` will be used as the writeable FS.
- **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.

get_fs(*name*)

Get a filesystem from its name.

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

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

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

Parameters

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

OS Filesystem

Manage the filesystem provided by your OS.

In essence an `OSFS` is a thin layer over the `io` and `os` modules in the Python library.

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

Parameters

- **root_path** (*str or path-like*) – 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.
- **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`.

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

Here are some examples of creating `OSFS` objects:

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

Sub Filesystem

A SubFS represents a directory in a 'parent' filesystem.

class `fs.subfs.`**ClosingSubFS**(*parent_fs*, *path*)
A version of SubFS which will close its parent automatically.

class `fs.subfs.`**SubFS**(*parent_fs*, *path*)
A sub-directory on another 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()*.

Tar Filesystem

A filesystem implementation for .tar files.

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

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.gettext('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.settext(
        'readme.txt',
        'This tar file was written by PyFilesystem'
    )
```

Parameters

- **file** (*str or file*) – An OS filename, or a open file object.
- **write** (*bool*) – Set to `True` to write a new tar file, or `False` to read an existing tar file.
- **compression** (*str*) – Compression to use (one of the formats supported by `tarfile`: `xz`, `gz`, `bz2`, or None).
- **temp_fs** (*str*) – An opener string for the temporary filesystem used to store data prior to tarring.

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

write_tar(*file=None*, *compression=None*, *encoding=None*)
Write tar to a file.

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

Parameters

- **file** (*str or file-like*) – Destination file, may be a file name or an open file object.
- **compression** – Compression to use (one of the constants defined in the tarfile module in the stdlib).

Temporary Filesystem

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=None*, *temp_dir=None*, *auto_clean=True*, *ignore_clean_errors=True*)

Create a temporary filesystem.

Parameters

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

`clean`()

Clean (delete) temporary files created by this filesystem.

Zip Filesystem

A filesystem implementation for .zip files.

class `fs.zipfs.`**`ReadZipFS`**(*file*, *encoding=u'utf-8'*)

A readable zip file.

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

A writable zip file.

`write_zip`(*file=None*, *compression=None*, *encoding=None*)

Write zip to a file.

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

Parameters

- **`file`** (*`str or file-like`*) – Destination file, may be a file name or an open file object.
- **`compression`** – Compression to use (one of the constants defined in the zipfile module in the stdlib).

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.gettext('readme.txt')
```

If you open the ZipFS with `write` set to `True`, then the ZipFS will be a 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.settext(
        'readme.txt',
        'This zip file was written by PyFilesystem'
    )
```

Parameters

- **file** (*str or file*) – An OS filename, or a open file object.
- **write** (*bool*) – Set to `True` to write a new zip file, or `False` 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*) – An opener string for the temporary filesystem used to store data prior to zipping.

CHAPTER 8

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*.

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.

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.

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.

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`:

```
from fs.test import FSTestCases

class TestMyFS(FSTestCases):

    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_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 object.

Parameters **fs** – A FS instance previously opened by *~fs.test.FSTestCases.make_fs*.

make_fs()
Return an FS instance.

test_geturl_purpose()
Check an unknown purpose raises a NoURL error

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.

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()*
- *exists()*

- *filterdir()*
- *getbytes()*
- *gettext()*
- *getmeta()*
- *getsize()*
- *getsyspath()*
- *gettype()*
- *geturl()*
- *hassyspath()*
- *hasurl()*
- *isclosed()*
- *isempty()*
- *isfile()*
- *lock()*
- *movedir()*
- *makedirs()*
- *move()*
- *open()*
- *opendir()*
- *removetree()*
- *scandir()*
- *setbytes()*
- `setbin()`
- *setfile()*
- *settimes()*
- *settext()*
- *touch()*
- *validatepath()*

Helper Methods

These methods SHOULD NOT be implemented.

Implementing these is highly unlikely to be worthwhile.

- *getbasic()*
- *getdetails()*
- *check()*

- *match()*
- *tree()*

CHAPTER 9

Creating an extension

Once a filesystem has been implemented, it can be integrated with other applications and projects using PyFilesystem.

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`.

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, 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(
                "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.

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",
)
```

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.

Let us Know

Contact us to add your filesystem to the PyFilesystem Wiki.

Live Example

See fs.sshfs for a functioning PyFilesystem2 extension implementing a Pyfilesystem2 filesystem over SSH.

CHAPTER 10

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 11

Reference

fs.base.FS

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.

appendbytes(*path*, *data*)
Append bytes to the end of a file. Creating the file if it doesn't already exists.

Parameters

- **path** (*str*) – Path to a file.
- **data** (*bytes*) – Bytes to append.

Raises

- **ValueError** – if `data` is not bytes.
- ***fs.errors.ResourceNotFound*** – if a parent directory of `path` does not exist.

appendtext(*path*, *text*, *encoding=u'utf-8'*, *errors=None*, *newline=u''*)
Append text to a file. Creating the file if it doesn't already exists.

Parameters

- **path** (*str*) – Path to a file.
- **text** (*str*) – Text to append.

Raises

- **ValueError** – if `text` is not bytes.
- ***fs.errors.ResourceNotFound*** – if a parent directory of `path` does not exist.

check()
Check a filesystem may be used.

Will throw a *FilesystemClosed* if the filesystem is closed.

Returns None

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.

Here's an example of automatically closing a filesystem:

```
with OSFS('~/Desktop') as desktop_fs:
    desktop_fs.settext(
        '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*)
Copy file contents from `src_path` to `dst_path`.

Parameters

- **src_path** (*str*) – Path of source file.
- **dst_path** (*str*) – Path to destination file.

Raises

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

copydir(*src_path*, *dst_path*, *create=False*)
Copy the contents of `src_path` to `dst_path`.

Parameters

- **src_path** (*str*) – Source directory.
- **dst_path** (*str*) – Destination directory.
- **create** (*bool*) – If `True` then `src_path` will be created if it doesn't already exist.

Raises ***fs.errors.ResourceNotFound*** – If the destination directory does not exist, and `create` is not True.

create(*path*, *wipe=False*)
Create an empty file.

Parameters

- **path** (*str*) – Path to new file in filesystem.
- **wipe** (*bool*) – Truncate any existing file to 0 bytes.

Returns `True` if file was created, `False` if it already existed.

Return type bool

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

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

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

Return type str

exists(*path*)
Check if a path maps to a resource.

Parameters **path** (*str*) – Path to a resource

Return type bool

A `path` exists if it maps to any resource (including a directory).

filterdir(*path*, *files=None*, *dirs=None*, *exclude_dirs=None*, *exclude_files=None*, *namespaces=None*, *page=None*)
Get an iterator of resource info, filtered by file patterns.

Parameters

- **path** (*str*) – A path to a directory on the filesystem.
- **files** (*list*) – A list of unix shell-style patterns to filter file names, e.g. `['*.py']`.
- **dirs** (*list*) – A list of unix shell-style wildcards to filter directory names.
- **exclude_dirs** (*list*) – An optional list of patterns used to exclude directories
- **exclude_files** (*list*) – An optional list of patterns used to exclude files.
- **namespaces** (*list*) – A list of namespaces to include in the resource information.
- **page** (*tuple or None*) – 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

This method enhances *scandir()* with additional filtering functionality.

getbasic(*path*)
Get the *basic* resource info.

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

Returns Resource information object for `path`.

Return type *Info*

This method is shorthand for the following:

```
fs.getinfo(path, namespaces=['basic'])
```

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

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

Returns file contents

Return type bytes

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

getdetails(*path*)
Get the *details* resource info.

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

Returns Resource information object for `path`.

Return type *Info*

This method is shorthand for the following:

```
fs.getinfo(path, namespaces=['details'])
```

getinfo(*path*, *namespaces=None*)
Get information regarding a resource (file or directory) on a filesystem.

Parameters

- **path** (*str*) – A path to a resource on the filesystem.
- **namespaces** (*list or None*) – Info namespaces to query (defaults to 'basic').

Returns Resource information object.

Return type *Info*

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

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

Parameters

- **keys** (*list or None*) – A list of keys to retrieve, or None for all keys.
- **namespace** (*str*) – The meta namespace (default is *"standard"*).

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 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.

Return type int

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 an *system path* to a resource.

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

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.

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=u''*)
Get the contents of a file as a string.

Parameters

- **path** (*str*) – A path to a readable file on the filesystem.
- **encoding** (*str*) – Encoding to use when reading contents in text mode.
- **errors** (*str*) – Unicode errors parameter.
- **newline** (*str*) – Newlines parameter.

Returns file contents.

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

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

Parameters **path** – A path in the filesystem.

Returns *ResourceType*

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=u'download'*)
Get a URL to the 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`.

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

Return type bool

hasurl (*path*, *purpose=u'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()*.

Return type bool

isclosed ()
Check if the filesystem is closed.

isdir (*path*)
Check a path exists and is a directory.

isempty (*path*)
Check if a directory is empty (contains no files or directories).

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

Return type bool

isfile(*path*)
Check a path exists and is a file.

islink(*path*)
Check if a path is a symlink.

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

Return type bool

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

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.

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*.

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. Here's how you would use it:

```
with my_fs.lock():  # May block
    # code here has exclusive access to the filesystem
```

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, and return a *SubFS* for the new directory.

Parameters

- **path** (*str*) – Path to directory from root.
- **permissions** (Permissions) – *Permissions* instance.
- **recreate** (*bool*) – Do not raise an error if the directory exists.

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.
- **recreate** (*bool*) – If `False` (default), it is an error to attempt to create a directory that already exists. Set to *True* to allow directories to be re-created without errors.
- **permissions** – Initial permissions.

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 isn't a directory.

match(*patterns*, *name*)
Check if a name matches any of a list of wildcards.

Parameters

- **patterns** (*list*) – A list of patterns, e.g. `['*.py']`
- **name** (*str*) – A file or directory name (not a path)

Return type bool

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).

```
>>> home_fs.match(['*.py'], '__init__.py')
True
>>> home_fs.match(['*.jpg', '*.png'], 'foo.gif')
False
```

If `patterns` is `None`, or (`['*']`), then this method will always return True.

move(*src_path*, *dst_path*, *overwrite=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.

Raises

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

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

Parameters

- **src_path** (*str*) – Path to 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 already exist.

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

Parameters

- **path** (*str*) – A path to a file on the filesystem.
- **mode** (*str*) – Mode to open file object.
- **buffering** (*int*) – Buffering policy: `0` to switch buffering off, `1` to select line buffering, `>1` to select a fixed-size buffer, `-1` to auto-detect.
- **encoding** (*str*) – Encoding for text files (defaults to `utf-8`)
- **errors** (*str*) – What to do with unicode decode errors (see stdlib docs)
- **newline** (*str*) – New line parameter (See stdlib docs).
- **options** – Additional keyword parameters to set implementation specific options (if required). See implementation docs for details.

Return type file object

openbin (*path*, *mode=u'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). 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 positive integer to indicate buffer size).
- **options** – Keyword parameters for any additional information required by the filesystem (if any).

Return type file object

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 *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** – 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()* will be used.

Returns A filesystem object representing a sub-directory.

Return type *SubFS*

Raises ***fs.errors.DirectoryExpected*** – If `dst_path` does not exist or is not a directory.

remove(*path*)
Remove a file.

Parameters **path** (*str*) – Path to the file you want 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()` if you want to remove the directory contents).
- ***fs.errors.DirectoryExpected*** – If the path is not a directory.
- ***fs.errors.ResourceNotFound*** – If the path does not exist.
- ***fs.errors.RemoveRootError*** – If an attempt is made to remove the root directory (i.e. *'/'*).

removetree(*dir_path*)
Recursively remove the contents of a directory.

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.

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

Parameters

- **path** (*str*) – A path on the filesystem
- **namespaces** (*list*) – A sequence of info namespaces.
- **page** (*tuple or None*) – May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or `None` to iterator the entire directory. Paging a directory scan may be necessary for very large directories.

Return type iterator

setbinfile(*path*, *file*)
Set a file to the contents of a binary file object.

Parameters

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

- **file** (*file object*) – A file object open for reading in binary mode.

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

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). For example:

```
with open('myfile.bin') as read_file:
    my_fs.setbinfile('myfile.bin', read_file)
```

setbytes(*path*, *contents*)

Copy (bytes) data to a file.

Parameters

- **path** (*str*) – Destination path on the filesystem.
- **contents** (*bytes*) – A bytes object with data to be written

setfile(*path*, *file*, *encoding=None*, *errors=None*, *newline=None*)

Set a file to the contents of a file object.

Parameters

- **path** (*str*) – A path on the filesystem.
- **file** (*file object*) – A file object open for reading.
- **encoding** (*str*) – Encoding of destination file, or `None` for binary.
- **errors** (*str*) – How encoding errors should be treated (same as `io.open`).
- **newline** (*str*) – Newline parameter (same is `io.open`).

This method will read the contents of a supplied file object, and write to a file on the filesystem. If the destination exists, it will first be truncated.

If *encoding* is supplied, the destination will be opened in text mode.

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). For example:

```
with open('myfile.bin') as read_file:
    my_fs.setfile('myfile.bin', read_file)
```

setinfo(*path*, *info*)

Set info on a resource.

Parameters

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

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

This method is the compliment to `getinfo` and is used to set info values on a resource.

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`. Here's an example:

```
details_info = {
    "details":
    {
```

```
        "modified_time": time.time()
    }
}
my_fs.setinfo('file.txt', details_info)
```

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

Parameters

- **contents** (*str*) – Path on the filesystem.
- **encoding** (*str*) – Encoding of destination file (default 'UTF-8).
- **errors** (*str*) – Error parameter for encoding (same as `io.open`).
- **newline** (*str*) – Newline parameter for encoding (same as `io.open`).

settimes (*path*, *accessed=None*, *modified=None*)
Set the accessed and modified time on a resource.

Parameters

- **accessed** – The accessed time, as a datetime, or None to use the current rime.
- **modified** – The modified time, or `None` (the default) to use the same time as `accessed` parameter.

touch (*path*)
Create 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()*.

validatepath (*path*)
Check if a path is valid on this filesystem, and return a normalized absolute path.

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.

walk
Get a *BoundWalker* object for this filesystem.

fs.compress

This module can compress the contents of a filesystem.

Currently zip and tar are supported.

fs.compress.**write_tar**(*src_fs*, *file*, *compression=None*, *encoding=u'utf-8'*, *walker=None*)
Write the contents of a filesystem to a tar file.

Parameters

- **file** (*str or file-like.*) – Destination file, may be a file name or an open file object.
- **compression** (*str*) – Compression to use.
- **encoding** (*str*) – The encoding to use for filenames. The default is "utf-8".
- **walker** (*Walker or None*) – 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=u'utf-8'*, *walker=None*)
Write the contents of a filesystem to a zip file.

Parameters

- **file** (*str or file-like.*) – Destination file, may be a file name or an open file object.
- **compression** (*str*) – Compression to use (one of the constants defined in the zipfile module in the stdlib).
- **encoding** (*str*) – The encoding to use for filenames. The default is "utf-8", use "CP437" if compatibility with WinZip is desired.
- **walker** (*Walker or None*) – A *Walker* instance, or None to use default walker. You can use this to specify which files you want to compress.

fs.copy

Copying files from one filesystem to another. Functions for copying resources *between* filesystem.

fs.copy.**copy_dir**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *walker=None*, *on_copy=None*)
Copy a directory from one filesystem to another.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **src_path** (*str*) – A path to a directory on src_fs.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **dst_path** (*str*) – A path to a directory on dst_fs.
- **walker** (*Walker*) – 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** (Function, with signature (src_fs, src_path, dst_fs, dst_path).) – A function callback called after a single file copy is executed.

fs.copy.**copy_dir_if_newer**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *walker=None*, *on_copy=None*)
Copy a directory from one filesystem to another. If both source and destination files exist, the copy is executed only if the source file is newer than the destination file. In case modification times of source or destination files are not available, copy is always executed.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **src_path** (*str*) – A path to a directory on src_fs.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **dst_path** (*str*) – A path to a directory on dst_fs.
- **walker** (*Walker*) – 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** (Function, with signature (src_fs, src_path, dst_fs, dst_path).) – A function callback called after a single file copy is executed.

fs.copy.**copy_file**(*src_fs*, *src_path*, *dst_fs*, *dst_path*)
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 URL or instance*) – Source filesystem.
- **src_path** (*str*) – Path to a file on src_fs.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **dst_path** (*str*) – Path to a file on dst_fs.

fs.copy.**copy_file_if_newer**(*src_fs*, *src_path*, *dst_fs*, *dst_path*)
Copy a file from one filesystem to another. If the destination exists, and is a file, it will be first truncated. If both source and destination files exist, the copy is executed only if the source file is newer than the destination file. In case modification times of source or destination files are not available, copy is always executed.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **src_path** (*str*) – Path to a file on src_fs.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **dst_path** (*str*) – Path to a file on dst_fs.

Returns True if the file copy was executed, False otherwise.

fs.copy.**copy_fs**(*src_fs*, *dst_fs*, *walker=None*, *on_copy=None*)
Copy the contents of one filesystem to another.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **walker** (*Walker*) – 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** (Function, with signature (src_fs, src_path, dst_fs, dst_path).) – A function callback called after a single file copy is executed.

fs.copy.**copy_fs_if_newer**(*src_fs*, *dst_fs*, *walker=None*, *on_copy=None*)
Copy the contents of one filesystem to another. If both source and destination files exist, the copy is executed only if the source file is newer than the destination file. In case modification times of source or destination files are not available, copy file is always executed.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **walker** (*Walker*) – 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** (Function, with signature (src_fs, src_path, dst_fs, dst_path).) – A function callback called after a single file copy is executed.

fs.copy.**copy_structure**(*src_fs*, *dst_fs*, *walker=None*)
Copy directories (but not files) from src_fs to dst_fs.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **walker** (*Walker*) – 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.

fs.enums

class fs.**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.

class fs.**Seek**
Constants used by file.seek.

These match os.SEEK_CUR, os.SEEK_END, and os.SEEK_SET from the standard library.

fs.errors

Defines the Exception classes thrown by PyFilesystem objects.

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.**CreateFailed**(*msg=None*)
An exception thrown when a FS could not be created.

exception fs.errors.**DestinationExists**(*path*, *exc=None*, *msg=None*)
Exception raised when a target destination already exists.

exception fs.errors.**DirectoryExists**(*path*, *exc=None*, *msg=None*)
Exception raised when trying to make a directory that already exists.

exception fs.errors.**DirectoryExpected**(*path*, *exc=None*, *msg=None*)
Exception raises when a directory was expected.

exception fs.errors.**DirectoryNotEmpty**(*path*, *exc=None*, *msg=None*)
Exception raised when a directory to be removed is not empty.

exception fs.errors.**FileExists**(*path*, *exc=None*, *msg=None*)
Exception raises when opening a file in exclusive mode.

exception fs.errors.**FileExpected**(*path*, *exc=None*, *msg=None*)
Exception raises when a file was expected.

exception fs.errors.**FilesystemClosed**(*msg=None*)
An exception thrown when attempting to use a closed filesystem.

exception fs.errors.**FSError**(*msg=None*)
Base exception class for the FS module.

exception fs.errors.**IllegalBackReference**(*path*)
Exception raised when 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.

---

exception fs.errors.**InsufficientStorage**(*path=None*, *exc=None*, *msg=None*)
Exception raised when operations encounter storage space trouble.

exception fs.errors.**InvalidCharsInPath**(*path*, *msg=None*)
The path contains characters that are invalid on this filesystem.

exception fs.errors.**InvalidPath**(*path*, *msg=None*)
Base exception for fs paths that can't be mapped on to the underlaying filesystem.

exception fs.errors.**MissingInfoNamespace**(*namespace*)
Raised when an expected namespace was missing.

exception fs.errors.**NoSysPath**(*path*, *msg=None*)
Exception raised when there is no sys path.

exception fs.errors.**NoURL**(*path*, *purpose*, *msg=None*)
Raised when there is no URL for a given path.

exception fs.errors.**OperationFailed**(*path=None*, *exc=None*, *msg=None*)
Base exception class for errors associated with a specific operation.

exception fs.errors.**OperationTimeout**(*path=None*, *exc=None*, *msg=None*)
Filesystem took too long.

exception fs.errors.**PathError**(*path*, *msg=None*)
Exception for errors to do with a path string.

exception fs.errors.**PermissionDenied**(*path=None*, *exc=None*, *msg=None*)
Permissions error.

exception fs.errors.**RemoteConnectionError**(*path=None*, *exc=None*, *msg=None*)
Exception raised when operations encounter remote connection trouble.

exception fs.errors.**RemoveRootError**(*path=None*, *exc=None*, *msg=None*)
Attempt to remove the root directory.

exception fs.errors.**ResourceError**(*path*, *exc=None*, *msg=None*)
Base exception class for error associated with a specific resource.

exception fs.errors.**ResourceInvalid**(*path*, *exc=None*, *msg=None*)
Exception raised when a resource is the wrong type.

exception fs.errors.**ResourceLocked**(*path*, *exc=None*, *msg=None*)
Exception raised when a resource can't be used because it is locked.

exception fs.errors.**ResourceNotFound**(*path*, *exc=None*, *msg=None*)
Exception raised when a required resource is not found.

exception fs.errors.**Unsupported**(*path=None*, *exc=None*, *msg=None*)
Exception raised for operations that are not supported by the FS.

fs.info

class fs.info.**Info**(*raw_info*, *to_datetime=<function epoch_to_datetime>*)
Container for *Resource Info*, returned by the following methods:

- *getinfo()*
- *scandir()*
- filterfir()

Parameters

- **raw_info** (*dict*) – A dict containing resource info.
- **to_datetime** – A callable that converts an epoch time to a datetime object. The default uses epoch_to_datetime().

accessed
Get the time this resource was last accessed, or None if not available.

Requires the "details" namespace.

Return type datetime

Raises ***MissingInfoNamespace*** – if the 'details' namespace is not in the Info.

copy(*to_datetime=None*)
Create a copy of this resource info object.

created
Get the time this resource was created, or None if not available.

Requires the "details" namespace.

Return type datetime

Raises ***MissingInfoNamespace*** – if the 'details' namespace is not in the Info.

get(*namespace*, *key*, *default=None*)
Get a raw info value.

```
>>> info.get('access', 'permissions')
['u_r', 'u_w', '_wx']
```

Parameters

- **namespace** (*str*) – A namespace identifier.
- **key** (*str*) – A key within the namespace.
- **default** – A default value to return if either the namespace or namespace + key is not found.

gid
Get the group id of a resource, or `None` if not available.

Requires the `"access"` namespace.

Return type int

Raises ***MissingInfoNamespace*** – if the 'access' namespace is not in the Info.

group
Get the group of the resource owner, or `None` if not available.

Requires the `"access"` namespace.

Return type str

Raises ***MissingInfoNamespace*** – if the 'access' namespace is not in the Info.

has_namespace(*namespace*)
Check if the resource info contains a given namespace.

Parameters **namespace** (*str*) – A namespace name.

Return type bool

is_dir
Check if the resource references a directory.

Return type bool

is_file
Check if a resource references a file.

Return type bool

is_link
Check if a resource is a symlink.

Return type bool

is_writeable(*namespace*, *key*)
Check if a given key in a namespace is writable (with *setinfo()*).

Parameters

- **namespace** (*str*) – A namespace identifier.
- **key** (*str*) – A key within the namespace.

Return type bool

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.

Return type str

metadata_changed
Get the time the metadata changed, or `None` if not available.

Requires the `"details"` namespace.

Return type datetime

Raises ***MissingInfoNamespace*** – if the 'details' namespace is not in the Info.

modified
Get the time the resource was modified, or `None` if not available.

Requires the `"details"` namespace.

Return type datetime

Raises ***MissingInfoNamespace*** – if the 'details' namespace is not in the Info.

name
Get the resource name.

Return type str

permissions
Get a permissions object, or `None` if not available.

Requires the `"access"` namespace.

Return type *fs.permissions.Permissions*

Raises ***MissingInfoNamespace*** – if the 'access' namespace is not in the Info.

size
Get the size of the resource, in bytes.

Requires the `"details"` namespace.

Return type int

Raises ***MissingInfoNamespace*** – if the 'details' namespace is not in the Info.

target
Get the link target, if this is a symlink, or `None` if this is not a symlink.

Requires the `"link"` namespace.

Return type bool

Raises ***MissingInfoNamespace*** – if the 'link' namespace is not in the Info.

type
Get the resource type enumeration.

Requires the `"details"` namespace.

Type *ResourceType*

Raises ***MissingInfoNamespace*** – if the 'details' namespace is not in the Info.

uid
Get the user id of a resource, or `None` if not available.

Requires the `"access"` namespace.

Return type int

Raises ***MissingInfoNamespace*** – if the 'access' namespace is not in the Info.

user
Get the owner of a resource, or `None` if not available.

Requires the `"access"` namespace.

Return type str

Raises ***MissingInfoNamespace*** – if the 'access' namespace is not in the Info.

fs.move

Moving files from one filesystem to another. Functions for moving files between filesystems.

`fs.move.`**move_dir**(*src_fs*, *src_path*, *dst_fs*, *dst_path*)
Move a directory from one filesystem to another.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **src_path** (*str*) – A path to a directory on `src_fs`.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.
- **dst_path** (*str*) – A path to a directory on `dst_fs`.

`fs.move.`**move_file**(*src_fs*, *src_path*, *dst_fs*, *dst_path*)
Move a file from one filesystem to another.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **src_path** (*str*) – Path to a file on `src_fs`.
- **dst_fs** (*str*) – Destination filesystem.
- **dst_fs** – Path to a file on `dst_fs`.

`fs.move.`**move_fs**(*src_fs*, *dst_fs*)
Move the contents of a filesystem to another filesystem.

Parameters

- **src_fs** (*FS URL or instance*) – Source filesystem.
- **dst_fs** (*FS URL or instance*) – Destination filesystem.

fs.mode

Tools for managing mode strings (as used in *open()* and *openbin()*).

class `fs.mode.`**Mode**(*mode*)
A mode object provides properties that can be used to interrogate the mode strings used when opening files.

Parameters **mode** (*str*) – A *mode* string, as used by `io.open`.

Raises **ValueError** – If the mode string is invalid.

Here's an example of typical use:

```
>>> mode = Mode('rb')
>>> mode.reading
True
>>> mode.writing
False
>>> mode.binary
True
>>> mode.text
False
```

appending
Check if a mode permits appending.

binary
Check if a mode specifies binary.

create
Check if the mode would create a file.

exclusive
Check if the mode require exclusive creation.

reading
Check if the mode permits reading.

text
Check if a mode specifies text.

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.

Currently, this just removes the 'x' on PY2 because PY2 doesn't support exclusive mode.

truncate
Check if a mode would truncate an existing file.

updating
Check if a mode permits updating (reading and writing).

validate(*_valid_chars=frozenset([u'a', u'b', u'+', u'r', u't', u'w', u'x'])*)
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
Check if a mode permits writing.

fs.mode.**check_readable**(*mode*)
Check a mode string allows reading.

Parameters **mode** (*str*) – A mode string, e.g. `"rt"`

Return type bool

fs.mode.**check_writable**(*mode*)
Check a mode string allows writing.

Parameters **mode** (*str*) – A mode string, e.g. `"wt"`

Return type bool

fs.mode.**validate_open_mode**(*mode*)
Check `mode` parameter of *open()* is valid.

Parameters **mode** (*str*) – Mode parameter.

Raises *ValueError* if mode is not valid.

fs.mode.**validate_openbin_mode**(*mode*, *_valid_chars=frozenset([u'a', u'b', u'+', u'r', u'w', u'x'])*)
Check `mode` parameter of *openbin()* is valid.

Parameters **mode** (*str*) – Mode parameter.

Raises *ValueError* if mode is not valid.

fs.opener

Open filesystems from a URL.

fs.opener.base

Defines the Opener abstract base class.

class fs.opener.base.**Opener**
The opener base class.

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 writeable.
- **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).

Returns *FS* object

fs.opener.parse

Parses 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`.

- `username` A username, or `None` .
- `password` An password, or `None`.
- `resource` A *resource*, typically a domain and path, e.g. `ftp.example.org/dir`
- `params` An dictionary of parameters extracted from the query string.
- `path` An optional path within the filesystem.

`fs.opener.parse.`**`parse_fs_url`**(*fs_url*)

Parse a Filesystem URL and return a *ParseResult*, or raise `ParseError` (subclass of ValueError) if the FS URL is not value.

Parameters **`fs_url`** (*str*) – A filesystem URL

Return type *ParseResult*

fs.opener.registry

Defines the Registry, which maps protocols and FS URLs to their respective Opener.

class `fs.opener.registry.`**`Registry`**(*default_opener=u'osfs'*)

A registry for *Opener* instances.

`get_opener`(*protocol*)

Get the opener class associated to a given protocol.

Parameters **`protocol`** (*str*) – A filesystem protocol.

Return type `Opener`.

Raises

- ***`UnsupportedProtocol`*** – If no opener could be found.
- **`EntryPointLoadingError`** – If the returned entry point is not an `Opener` subclass or could not be loaded successfully.

`manage_fs`(**args*, ***kwds*)

A context manager opens / closes a filesystem.

Parameters

- **`fs_url`** (*str or FS*) – A FS instance or a FS URL.
- **`create`** (*bool*) – If `True`, then create the filesystem if it doesn't already exist.
- **`writeable`** (*bool*) – If `True`, then the filesystem should 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.

Here's an example:

```
def print_ls(list_fs):
    """List a directory."""
    with manage_fs(list_fs) as fs:
        print(" ".join(fs.listdir()))
```

This function may be used in two ways. You may either pass either a `str`, as follows:

```
print_list('zip://projects.zip')
```

Or, an FS instance:

```
from fs.osfs import OSFS
projects_fs = OSFS('~/')
print_list(projects_fs)
```

open (*fs_url*, *writeable=True*, *create=False*, *cwd=u'.'*, *default_protocol=u'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*) – True if the filesystem must be writeable.
- **create** (*bool*) – True if the filesystem should be created if it does not exist.
- **cwd** (*str or None*) – The current working directory.

Return type Tuple of `(<filesystem>, <path from url>)`

open_fs (*fs_url*, *writeable=True*, *create=False*, *cwd=u'.'*, *default_protocol=u'osfs'*)
Open a filesystem object from a FS URL (ignoring the path component).

Parameters

- **fs_url** (*str*) – A filesystem URL.
- **writeable** (*bool*) – True if the filesystem must be writeable.
- **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).
- **default_protocol** (*str*) – The protocol to use if one is not supplied in the FS URL (defaults to `"osfs"`).

Returns *FS* object

protocols
A list of supported protocols.

fs.opener.errors

Errors raised when attempting to open a filesystem.

exception `fs.opener.errors.`**EntryPointError**
Raised by the registry when an entry point cannot be loaded.

exception `fs.opener.errors.`**OpenerError**
Base class for opener related errors.

exception `fs.opener.errors.`**ParseError**
Raised when attempting to parse an invalid FS URL.

exception `fs.opener.errors.`**UnsupportedProtocol**
May be raised if no opener could be found for a given protocol.

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.

Return type str

```
>>> basename('foo/bar/baz')
'baz'
>>> basename('foo/bar')
'bar'
>>> basename('foo/bar/')
''
```

`fs.path.`**`combine`**(*path1*, *path2*)
Join two paths together.

Parameters

- **`path1`** (*str*) – A PyFilesytem path.
- **`path2`** (*str*) – A PyFilesytem path.

Return type str

This is faster than *join()*, but only works when the second path is relative, and there are no back references in either path.

```
>>> 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.

Return type str

```
>>> 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** – A PyFilesytem path.

Return type bool

```
>>> forcedir("foo/bar")
'foo/bar/'
>>> forcedir("foo/bar/")
'foo/bar/'
```

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.

Return type str

```
>>> 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.

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.

Return type bool

fs.path.**isdotfile**(*path*)
Detect if a path references a dot file, i.e. a resource who's name starts with a '.'

Parameters **path** (*str*) – Path to check.

Return type bool

```
>>> 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.

Return type bool

```
>>> 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.

Return type bool

```
>>> 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** (*int*) – An FS path.

Return type bool

```
>>> iswildcard('foo/bar/baz.*')
True
>>> iswildcard('foo/bar')
False
```

fs.path.**iteratepath**(*path*)

Iterate over the individual components of a path.

```
>>> iteratepath('/foo/bar/baz')
['foo', 'bar', 'baz']
```

Parameters **path** (*str*) – Path to iterate over.

Returns A list of path components.

Return type list

fs.path.**join**(**paths*)
Join any number of paths together.

Parameters **paths** – Paths to join are given in positional arguments.

Return type str

```
>>> 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.

Type str

```
>>> normpath("/foo//bar/frob/../baz")
'/foo/bar/baz'
>>> normpath("foo/../../bar")
Traceback (most recent call last)
    ...
IllegalBackReference: Too many backrefs in 'foo/../../bar'
```

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.

Returns A list of paths.

Return type list

```
>>> recursepath('a/b/c')
['/', '/a', '/a/b', '/a/b/c']
```

fs.path.**relativefrom**(*base*, *path*)
Return a path relative from a given base path, i.e. insert backrefs as appropriate to reach the path from the base.

Parameters

- **base** (*str*) – Path to a directory.
- **path** (*atr*) – Path you wish to make relative.

```
>>> 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*) – Path to adjust

Return type str

```
>>> 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 tuple of `(head, tail)`

Return type tuple

```
>>> 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, and returns the path (up to the last '.' and the extension).

Parameters **path** – A path to split

Returns tuple of `(path, extension)`

Return type tuple

```
>>> splitext('baz.txt')
('baz', '.txt')
>>> splitext('foo/bar/baz.txt')
('foo/bar/baz', '.txt')
```

fs.permissions

An 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.

Parameters

- **names** (*list*) – A list of permissions.
- **mode** (*int*) – A mode integer.
- **user** (*str*) – A triplet of *user* permissions, e.g. `"rwx"` or `"r--"`
- **group** (*str*) – A triplet of *group* permissions, e.g. `"rwx"` or `"r--"`

- **other** (*str*) – A triplet of *other* permissions, e.g. `"rwx"` or `"r--"`
- **sticky** (*bool*) – A boolean for the *sticky* bit.
- **setuid** (*bool*) – A boolean for the *setuid* bit.
- **setguid** (*bool*) – A boolean for the *setuid* bit.

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.

```
>>> from fs.permissions import Permissions
>>> p = Permissions(user='rwx', group='rw-', other='r--')
>>> print(p)
rwxrw-r--
>>> p.mode
500
>>> oct(p.mode)
'0764'
```

add(**permissions*)
Add permission(s).

Parameters **permissions** – Permission name(s).

as_str()
Get a linux-style string representation of permissions.

check(**permissions*)
Check if one or more permissions are enabled.

Parameters **permissions** – Permission name(s).

Returns True if all given permissions are set.

Rtype bool

copy()
Make a copy of this permissions object.

classmethod **create**(*init=None*)
Create a permissions object from an initial value.

Parameters **init** – May be None for equivalent for 0o777 permissions, a mode integer, or a list of permission names.

Returns mode integer, that may be used by *os.makedir* (amongst others).

```
>>> 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.

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** – Permission name(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.

fs.tools

A collection of functions that operate on filesystems and tools.

fs.tools.**copy_file_data**(*src_file*, *dst_file*, *chunk_size=None*)
Copy data from one file object to another.

Parameters

- **src_file** (*file-like*) – File open for reading.
- **dst_file** (*file-like*) – 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 paths of intermediate directories required to create a new directory.

Parameters

- **fs** – A FS object.
- **dir_path** (*str*) – A path to a new directory on the filesystem.

Returns A list of paths.

Return type list

Raises ***fs.errors.DirectoryExpected*** – If a path component references a file and not a directory.

fs.tools.**remove_empty**(*fs*, *path*)
Remove all empty parents.

Parameters

- **fs** – A FS object.
- **path** (*str*) – Path to a directory on the filesystem.

fs.tree

Render a text tree view, with optional color in terminals. Render a FS object as text tree views.

fs.tree.**render**(*fs*, *path=u'/'*, *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** (A *FS* instance) – A filesystem.
- **file** (*file or None*) – An open file-like object to render the tree, or None for stdout.
- **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** – Optional list of files patterns to match in the tree render.

Return type tuple

Returns A tuple of (<directory count>, <file count>).

fs.walk

The machinery for walking a filesystem. 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 object.

Parameters

- **fs** – A FS object.
- **walker_class** – A *WalkerBase* sub-class. The default uses *Walker*.

You will typically not need to create instances of this class explicitly. Filesystems have a `walk` property which returns a `BoundWalker` object.

```
>>> import fs
>>> home_fs = fs.open_fs('~/')
>>> home_fs.walk
BoundWalker(OSFS('/Users/will', encoding='utf-8'))
```

A BoundWalker is callable. Calling it is an alias for *walk()*.

dirs(*path=u'/'*, ***kwargs*)

Walk a filesystem, yielding absolute paths to directories.

Parameters

- **path** (*str*) – A path to a directory.
- **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*.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.

Returns An iterable of directory paths (absolute from the FS root).

This method invokes `dirs()` with the bound FS object.

files(*path=u'/'*, ***kwargs*)

Walk a filesystem, yielding absolute paths to files.

Parameters

- **path** (*str*) – A path to a directory.
- **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_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.

Returns An iterable of file paths (absolute from the filesystem root).

This method invokes `files()` with the bound FS object.

info(*path=u'/'*, *namespaces=None*, ***kwargs*)
Walk a filesystem, yielding tuples of `(<absolute path>, <resource info>)`.

Parameters

- **path** (*str*) – A path to a directory.
- **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_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.

Returns An iterable *Info* objects.

This method invokes `info()` with the bound FS object.

walk(*path=u'/'*, *namespaces=None*, ***kwargs*)
Walk the directory structure of a filesystem.

Parameters

- **path** (*str*) – A path to a directory.
- **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_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk, e.g. `['*.svn', '*.git']`.

Returns *Step* 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>`.

Here's an example:

```
home_fs = open_fs('~/')
walker = Walker(filter=['*.py'])
for path, dirs, files in walker.walk(home_fs, namespaces=['details']):
    print("[{}]".format(path))
    print("{} directories".format(len(dirs)))
```

```
    total = sum(info.size for info in files)
    print("{} bytes {}".format(total))
```

This method invokes *walk()* with bound FS object.

class fs.walk.**Step**(*path*, *dirs*, *files*)

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=u'breadth'*, *filter=None*, *exclude_dirs=None*)
A walker object recursively lists directories in a filesystem.

Parameters

- **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 filename patterns, e.g. `['*.py']`. Files will only be returned if the final component matches one of the patterns.
- **exclude_dirs** (*list*) – A list of patterns that will be used to filter out directories from the walk. e.g. `['*.svn', '*.git']`.

classmethod bind(*fs*)
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. Here's an example of binding:

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

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. Here's how you might use it:

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

Parameters **fs** – A filesystem object.

Returns a *BoundWalker*

check_file (*fs*, *info*)
Check if a filename should be included in the walk. Override to exclude files from the walk.

Parameters

- **fs** (*FS*) – A FS object.
- **info** – A *Info* object.

Return type bool

check_open_dir (*fs*, *info*)
Check if a directory should be opened. Override to exclude directories from the walk.

Parameters

- **fs** (*FS*) – A FS object.
- **info** – A *Info* object.

Return type bool

filter_files (*fs*, *infos*)
Filters a sequence of resource Info objects.

The default implementation filters those files for which *check_file()* returns True.

Parameters

- **fs** (*FS*) – A FS object.
- **infos** (*list*) – A list of *Info* instances.

Return type list

walk (*fs*, *path=u'/'*, *namespaces=None*)
Walk the directory structure of a filesystem.

Parameters

- **fs** – A FS object.
- **path** (*str*) – a path to a directory.
- **namespaces** (*list*) – A list of additional namespaces to add to the Info objects.

Returns *Step* 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>`.

Here's an example:

```
home_fs = open_fs('~/')
walker = Walker(filter=['*.py'])
for path, dirs, files in walker.walk(home_fs, namespaces=['details']):
    print("[{}]".format(path))
    print("{} directories".format(len(dirs)))
    total = sum(info.size for info in files)
    print("{} bytes {}".format(total))
```

class fs.walk.**WalkerBase**
The base class for a Walker.

To create a custom walker, implement *walk()* in a sub-class.

See *Walker()* for a fully featured walker object that should be adequate for all but your most exotic directory walking needs.

dirs (*fs*, *path=u'/'*)
Walk a filesystem, yielding absolute paths to directories.

Parameters

- **fs** (*str*) – A FS object.
- **path** (*str*) – A path to a directory.

files (*fs*, *path=u'/'*)
Walk a filesystem, yielding absolute paths to files.

Parameters

- **fs** – A FS object.
- **path** (*str*) – A path to a directory.

Returns An iterable of file paths.

info (*fs*, *path=u'/'*, *namespaces=None*)
Walk a filesystem, yielding tuples of `(<absolute path>, <resource info>)`.

Parameters

- **fs** (*str*) – A FS object.
- **path** (*str*) – A path to a directory.
- **namespaces** (*list*) – A list of additional namespaces to add to the Info objects.

Returns An iterable of *Info* objects.

walk (*fs*, *path=u'/'*, *namespaces=None*)
Walk the directory structure of a filesystem.

Parameters

- **fs** – A FS object.
- **path** (*str*) – a path to a directory.
- **namespaces** (*list*) – A list of additional namespaces to add to the Info objects.

Returns Iterator of *Step* named tuples.

fs.wildcard

Match wildcard filenames.

fs.wildcard.**get_matcher** (*patterns*, *case_sensitive*)
Get a callable that checks a list of names matches the given wildcard 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.

Return type callable

Here's an example:

```
>>> import wildcard
>>> is_python = wildcard.get_macher(['*.py'])
>>> is_python('__init__.py')
>>> True
>>> is_python('foo.txt')
>>> False
```

fs.wildcard.**imatch**(*pattern*, *name*)

Test whether `name` matches `pattern`, ignoring case differences.

Parameters

- **pattern** (*str*) – A wildcard pattern. e.g. `"*.py"`
- **name** (*str*) – A filename

Return type bool

fs.wildcard.**imatch_any**(*patterns*, *name*)

Test if a name matches at least one of a list of patterns, ignoring case differences. 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.

Return type bool

fs.wildcard.**match**(*pattern*, *name*)

Test whether `name` matches `pattern`.

Parameters

- **pattern** (*str*) – A wildcard pattern. e.g. `"*.py"`
- **name** (*str*) – A filename

Return type bool

fs.wildcard.**match_any**(*patterns*, *name*)

Test if a name matches at least one 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.

Return type bool

fs.wrap

fs.wrap

A collection of *WrapFS* objects that modify the behavior of another filesystem.

Here's an example that opens a filesystem then makes it *read only*:

```
from fs import open_fs
from fs.wrap import read_only

projects_fs = open_fs('~/projects')
read_only_projects_fs = read_only(projects_fs)

# Will raise ResourceReadOnly exception
read_only_projects_fs.remove('__init__.py')
```

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.

---

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.

fs.wrap.**cache_directory**(*fs*)
Make a filesystem that caches directory information.

Parameters **fs** – A FS object.

Returns A filesystem that caches results of `scandir`, `isdir` and other methods which read directory information.

fs.wrap.**read_only**(*fs*)
Make a read-only filesystem.

Parameters **fs** – A FS object.

Returns A read only version of `fs`.

fs.wrapfs

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.

delegate_fs()
Get the 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 fileystem.

Returns a tuple of <filesystem>, <new path>

Return type tuple

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z