PyFilesystem2 Documentation

2.0.0

Will McGugan

3 12, 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.

: 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(files=['*.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.

: 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
```

: 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 as positional arguments to `getinfo()`. For example, the following retrieves the `details` and `access` namespaces for a file:

```
resource_info = fs.getinfo('myfile.txt', '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.

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.

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

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.

: 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 *class*:fs.permissions.Permissions‘ objects.

CHAPTER 5

FS URLs

PyFilesystem can open filesystems via a FS URL, which are similar to the URLs you might enter in to a browser.

Using FS URLs can be useful if you want to be able to specify a filesystem dynamically, in a conf file (for instance).

FS URLs are parsed in to the following format:

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

The components are as follows:

- `<type>` 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://
```

If `<type>` is not specified then it is assumed to be an `OSFS`. The following FS URLs are equivalent:

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

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))
```

: 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.

FTP Filesystem

Manage resources on a FTP server.

Memory Filesystem

Create and manage an in-memory filesystems.

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'))
```

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
```

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.

Sub Filesystem

Tar Filesystem

A filesystem implementation for .tar files.

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.

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, when writing a new FS class, 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`:

```python
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:

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

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.

fs.compress

fs.copy

Copying files from one filesystem to another.

fs.enums

fs.errors

fs.info

fs.move

Moving files from one filesystem to another.

fs.mode

fs.opener

Open filesystems from a URL.

fs.path

fs.permissions

fs.tools

fs.tree

Render a text tree view, with optional color in terminals.

fs.walk

fs.wildcard

fs.wrap

fs.wrapfs

CHAPTER 10

Indices and tables

- genindex
- modindex
- search