PyFilesystem Documentation

Release 0.5.0

Will McGugan

Aug 09, 2017

Contents

PyFilesystem provides a simplified common interface to a variety of different filesystems, such as the local filesystem, zip files, ftp servers etc.

CHAPTER 1

Guide

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` class. Just as *file-like* objects abstract a single file, FS objects abstract the whole filesystem by providing a common interface to operations such as reading directories, getting file information, opening/copying/deleting files etc.

Even if you only want to work with the local filesystem, PyFilesystem simplifies a number of common operations and reduces the chance of error.

About PyFilesystem

PyFilesystem was initially created by Will McGugan and is now a joint effort from the following contributors:

- Will McGugan (http://www.willmcgugan.com)
- Ryan Kelly (http://www.rfk.id.au)
- Andrew Scheller (http://www.andrewscheller.co.uk/)
- Ben Timby (http://ben.timby.com/)

And many others who have contributed bug reports and patches.

Need Help?

If you have any problems or questions, please contact the developers through one of the following channels:

Bugs

If you find a bug in PyFilesystem, please file an issue: https://github.com/PyFilesystem/pyfilesystem/issues

Discussion Group

There is also a discussion group for PyFilesystem: http://groups.google.com/group/pyfilesystem-discussion

Getting Started

PyFilesystem is a Python-only module and can be installed from source or with pip. PyFilesystem works on Linux, Mac and OSX.

Installing

To install with pip, use the following:

```
pip install fs
```

Or to upgrade to the most recent version:

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

You can also install the cutting edge release by cloning the source from GIT:

```
git clone https://github.com/PyFilesystem/pyfilesystem.git
cd pyfilesystem
python setup.py install
```

Whichever method you use, you should now have the *fs* module on your path (version number may vary):

```
>>> import fs
>>> fs.__version__
'0.5.0'
```

You should also have the command line applications installed. If you enter the following in the command line, you should see a tree display of the current working directory:

```
fstree -l 2
```

Because the command line utilities use PyFilesystem, they also work with any of the supported filesystems. For example:

```
fstree ftp://ftp.mozilla.org -l 2
```

See *Command Line Applications* for more information on the command line applications.

Prerequisites

PyFilesystem requires at least **Python 2.6**. There are a few other dependencies if you want to use some of the more advanced filesystem interfaces, but for basic use all that is needed is the Python standard library.

- Boto (required for *s3fs*) https://github.com/boto/boto
- Paramiko (required for *sftpfs*) https://github.com/paramiko/paramiko
- wxPython (required for `browsewin`) http://www.wxpython.org/

Quick Examples

Before you dive in to the API documentation, here are a few interesting things you can do with PyFilesystem.

The following will list all the files in your home directory:

```
>>> from fs.osfs import OSFS
>>> home_fs = OSFS('~/') # 'c:\Users\<login name>' on Windows
>>> home_fs.listdir()
```

Here's how to browse your home folder with a graphical interface:

```
>>> home_fs.browse()
```

This will display the total number of bytes store in '.py' files your home directory:

```
>>> sum(home_fs.getsize(f) for f in home_fs.walkfiles(wildcard='*.py'))
```

Concepts

Working with PyFilesystem is generally easier than working with lower level interfaces, as long as you are aware these simple concepts.

Sandboxing

FS objects are not permitted to work with any files / directories outside of the Filesystem they represent. If you attempt to open a file or directory outside the root of the FS (e.g. by using "../" in the path) you will get a `ValueError`.

There is no concept of a current working directory in PyFilesystem, since it is a common source of bugs and not all filesystems even have such a notion. If you want to work with a sub-directory of a FS object, you can use the *opendir()* method which returns another FS object representing the 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, especially if we are passing *foo_fs* to an untrusted function or to a function that has the potential to delete files. Fortunately we can isolate a single sub-directory with then *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-positioned, 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
```

PyFilesystem will catch any attempts to read outside of the root directory. For example, the following will not work:

```
bar_fs.open('../private.txt') # throws a ValueError
```

Paths

Paths used within an FS object use the same common format, regardless of the underlying file system it represents (or the platform it resides on).

When working with paths in FS objects, keep in mind the following:

- Path components are separated by a forward slash (/)
- Paths beginning with a forward slash are absolute (start at the root of the FS)
- 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 or additional methods 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.

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* from an FS object you are working with. You can do this with the *getsyspath()* method which converts a valid path in the context of the FS object to an absolute path on the system, should one exist.

For example:

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

Not all FS implementation will map to a valid system path (e.g. the FTP FS object). If you call *getsyspath()* on such FS objects you will either get a *NoSysPathError* exception or a return value of `None`, if you call `getsyspath` with `allow_none=True`.

Errors

PyFilesystem converts all exceptions to a common type, so that you need only write your exception handling code once. For example, if you try to open a file that doesn't exist, PyFilesystem will throw a *ResourceNotFoundError* regardless of whether the filesystem is local, on a ftp server or in a zip file:

```
>>> from fs.osfs import OSFS
>>> root_fs = OSFS('/')
>>> root_fs.open('doesnotexist.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.6/dist-packages/fs/errors.py", line 181, in wrapper
    return func(self,*args,**kwds)
  File "/usr/local/lib/python2.6/dist-packages/fs/osfs/__init__.py", line 107, in open
    return open(self.getsyspath(path), mode, kwargs.get("buffering", -1))
fs.errors.ResourceNotFoundError: Resource not found: doesnotexist.txt
```

All PyFilesystem exceptions are derived from *FSError*, so you may use that if you want to catch all possible filesystem related exceptions.

Opening Filesystems

Generally, when you want to work with the files and directories of any of the supported filesystems, you create an instance of the appropriate class. For example, the following opens the directory `/foo/bar`:

```
from fs.osfs import OSFS
my_fs = OSFS('/foo/bar')
```

This is fine if you know beforehand where the directory you want to work with is located, and on what medium. However, there are occasions where the location of the files may change at runtime or should be specified in a config file or from the command line.

In these situations you can use an *opener*, which is a generic way of specifying a filesystem. For example, the following is equivalent to the code above:

```
from fs.opener import fsopendir
my_fs = fsopendir('/foo/bar')
```

The `fsopendir` callable takes a string that identifies the filesystem with a URI syntax, but if called with a regular path will return an *OSFS* instance. To open a different kind of filesystem, precede the path with the required protocol. For example, the following code opens an FTP filesystem rather than a directory on your hard-drive:

```
from fs.opener import fsopendir
my_fs = fsopendir('ftp://example.org/foo/bar')
```

For further information regarding filesystem openers see *fs.opener*.

Filesystem Interface

The following methods are available in all PyFilesystem implementation:

- *close()* Close the filesystem and free any resources
- *copy()* Copy a file to a new location
- *copydir()* Recursively copy a directory to a new location
- *cachehint()* Permit implementation to use aggressive caching for performance reasons
- *createfile()* Create a file with data

- *desc()* Return a short descriptive text regarding a path
- *exists()* Check whether a path exists as file or directory
- *getcontents()* Returns the contents of a file as a string
- *getinfo()* Return information about the path e.g. size, mtime
- *getmeta()* Get the value of a filesystem meta value, if it exists
- *getmmap()* Gets an mmap object for the given resource, if supported
- *getpathurl()* Get an external URL at which the given file can be accessed, if possible
- *getsize()* Returns the number of bytes used for a given file or directory
- *getsyspath()* Get a file's name in the local filesystem, if possible
- *hasmeta()* Check if a filesystem meta value exists
- *haspathurl()* Check if a path maps to an external URL
- *hassyspath()* Check if a path maps to a system path (recognized by the OS)
- *ilistdir()* Generator version of the *listdir()* method
- *ilistdirinfo()* Generator version of the *listdirinfo()* method
- *isdir()* Check whether a path exists and is a directory
- *isdirempty()* Checks if a directory contains no files
- *isfile()* Check whether the path exists and is a file
- *listdir()* List the contents of a directory
- *listdirinfo()* Get a directory listing along with the info dict for each entry
- *makedir()* Create a new directory
- *makeopendir()* Make a directory and returns the FS object that represents it
- *move()* Move a file to a new location
- *movedir()* Recursively move a directory to a new location
- *open()* Opens a file for read/writing
- *opendir()* Opens a directory and returns a FS object that represents it
- *remove()* Remove an existing file
- *removedir()* Remove an existing directory
- *rename()* Atomically rename a file or directory
- *safeopen()* Like *open()* but returns a `NullFile` if the file could not be opened
- *setcontents()* Sets the contents of a file as a string or file-like object
- *setcontents_async()* Sets the contents of a file asynchronously
- *settimes()* Sets the accessed and modified times of a path
- *tree()* Display an ascii rendering of the directory structure
- *walk()* Like *listdir()* but descends in to sub-directories
- *walkdirs()* Returns an iterable of paths to sub-directories
- *walkfiles()* Returns an iterable of file paths in a directory, and its sub-directories

See *FS* for the method signature and full details.

If you intend to implement an FS object, see *A Guide For Filesystem Implementers*.

Filesystems

This page lists the builtin filesystems.

FTP (File Transfer Protocol)

An interface to FTP servers. See *FTPFS*

Memory

A filesystem that exists entirely in memory. See *memoryfs*

Mount

A filesystem that can map directories in to other filesystems (like a symlink). See *mountfs*

Multi

A filesystem that overlays other filesystems. See *multifs*

OS

An interface to the OS Filesystem. See *osfs*

RPCFS (Remote Procedure Call)

An interface to a file-system served over XML RPC, See *rpcfs* and *xmlrpc*

SFTP (Secure FTP)

A secure FTP filesystem. See *sftpfs*

S3

A filesystem to access an Amazon S3 service. See *s3fs*

Temporary

Creates a temporary filesystem in an OS provided location. See *tempfs*

Wrap

A collection of wrappers that add new behavior / features to existing FS instances. See *wrapfs*

Zip

An interface to zip files. See *zipfs*

3rd-Party Filesystems

This page lists filesystem implementations that have been contributed by third parties. They may be less well tested than those found in the main module namespace.

DAV (WebDAV Protocol)

An interface to WebDAV file servers. See *davfs*

Tahoe LAFS

An interface to Tahoe Least-Authority File System. See *tahoelafs*

BIG (BIG Archive File Format)

A read-only interface to the BIG archive file format used in some EA games titles (e.g. Command & Conquer 4). See *bigfs*

Exposing FS objects

The `fs.expose` module offers a number of ways of making an FS implementation available over an Internet connection, or to other processes on the system.

FUSE

Makes an FS object available to other applications on the system. See `fuse`.

Dokan

Makes an FS object available to other applications on the system. See *dokan*.

Secure FTP

Makes an FS object available via Secure FTP. See *sftp*.

XMLRPC

Makes an FS object available over XMLRPC. See *xmlrpc*

Import Hook

Allows importing python modules from the files in an FS object. See *importhook*

Django Storage

Connects FS objects to Django. See *django_storage*

Utility Modules

PyFilesystem also contains some miscellaneous utility modules to make writing new FS implementations easier.

fs.path

Contains many utility functions for manipulating filesystem paths. See *path*.

fs.errors

Contains all the standard error classes used by PyFilesystem, along with some useful error-handling decorators. See *errors*.

fs.filelike

Takes care of a lot of the groundwork for implementing and manipulating objects that support Python's standard "file-like" interface. See *filelike*.

fs.remote

Contains useful functions and classes for implementing remote filesystems. See *remote*.

Command Line Applications

PyFilesystem adds a number of applications that expose some of the PyFilesystem functionality to the command line. These commands use the opener syntax, as described in *Opening Filesystems*, to refer to filesystems.

Most of these applications shadow existing shell commands and work in similar ways.

All of the command line application support the *-h* (or *–help*) switch which will display a full list of options.

Custom Filesystem Openers

When opening filesystems, the command line applications will use the default openers. You can also 'point' the command line applications at an opener to add it to a list of available openers. For example, the following uses a custom opener to list the contents of a directory served with the 'myfs' protocol:

```
fsls --fs mypackage.mymodule.myfs.MyFSOpener myfs://127.0.0.1
```

Listing Supported Filesystems

All of the command line applications support the `--listopeners` switch, which lists all available installed openers:

```
fsls --listopeners
```

fsls

Lists the contents of a directory, similar to the `ls` command, e.g.:

```
fsls
fsls ../
fsls ftp://example.org/pub
fsls zip://photos.zip
```

fstree

Displays an ASCII tree of a directory. e.g.:

```
fstree
fstree -g
fstree rpc://192.168.1.64/foo/bar -l3
fstree zip://photos.zip
```

fscat

Writes a file to stdout, e.g.:

```
fscat ~/.bashrc
fscat http://www.willmcgugan.com
fscat ftp://ftp.mozilla.org/pub/README
```

fsinfo

Displays information regarding a file / directory, e.g.:

```
fsinfo C:\autoexec.bat
fsinfo ftp://ftp.mozilla.org/pub/README
```

fsmv

Moves a file from one location to another, e.g.:

```
fsmv foo bar
fsmv *.jpg zip://photos.zip
```

fsmkdir

Makes a directory on a filesystem, e.g.:

```
fsmkdir foo
fsmkdir ftp://ftp.mozilla.org/foo
fsmkdir rpc://127.0.0.1/foo
```

fscp

Copies a file from one location to another, e.g.:

```
fscp foo bar
fscp ftp://ftp.mozilla.org/pub/README readme.txt
```

fsrm

Removes (deletes) a file from a filesystem, e.g.:

```
fsrm foo
fsrm -r mydir
```

fsserve

Serves the contents of a filesystem over a network with one of a number of methods; HTTP, RPC or SFTP, e.g.:

```
fsserve
fsserve --type rpc
fsserve --type http zip://photos.zip
```

fsmount

Mounts a filesystem with FUSE (on Linux) and Dokan (on Windows), e.g.:

```
fsmount mem:// ram
fsserve mem:// M
fsserve ftp://ftp.mozilla.org/pub ftpgateway
```

A Guide For Filesystem Implementers

PyFilesystem objects are designed to be as generic as possible and still expose the full filesystem functionality. With a little care, you can write a wrapper for your filesystem that allows it to work interchangeably with any of the built-in FS classes and tools.

To create a working PyFilesystem interface, derive a class from *FS* and implement the 9 *Essential Methods*. The base class uses these essential methods as a starting point for providing a lot of extra functionality, but in some cases the default implementation may not be the most efficient. For example, most filesystems have an atomic way of moving a file from one location to another without having to copy data, whereas the default implementation of *move()* method must copy all the bytes in the source file to the destination file. Any of the *Non - Essential Methods* may be overridden, but efficient custom versions of the following methods will have the greatest impact on performance:

- *copy()* copy a file
- *copydir()* copy a directory
- *exists()* check if a file / directory exists
- *getsyspath()* get a system path for a given resource, if it exists
- *move()* move a file
- *movedir()* move a directory

For network based filesystems (i.e. where the physical data is pulled over a network), there are a few methods which can reduce the number of round trips to the server, if an efficient implementation is provided:

- *listdirinfo()* returns the directory contents and info dictionary in one call
- *ilistdir()* a generator version of *listdir()*
- *ilistdirinfo()* a generator version of *listdirinfo()*

The generator methods (beginning with `i`) are intended for use with filesystems that contain a lot of files, where reading the directory in one go may be expensive.

Other methods in the *Filesystem Interface* are unlikely to require a non-default implementation, but there is nothing preventing you from implementing them – just be careful to use the same signature and replicate expected functionality.

Filesystem Errors

With the exception of the constructor, FS methods should throw *FSError* exceptions in preference to any implementation-specific exception classes, so that generic exception handling can be written. The constructor *may* throw a non-FSError exception, if no appropriate FSError exists. The rationale for this is that creating an FS interface may require specific knowledge, but this shouldn't prevent it from working with more generic code.

If specific exceptions need to be translated in to an equivalent FSError, pass the original exception class to the FSError constructor with the 'details' keyword argument.

For example, the following translates some fictitious exception in to an FSError exception, and passes the original exception as an argument.:

```
try:
    someapi.open(path, mode)
except someapi.UnableToOpen, e:
    raise errors.ResourceNotFoundError(path=path, details=e)
```

Any code written to catch the generic error, can also retrieve the original exception if it contains additional information.

Thread Safety

All PyFilesystem methods, other than the constructor, should be thread-safe where-ever possible. One way to do this is to pass `threads_synchronize=True` to the base constructor and use the `synchronize()` decorator to lock the FS object when a method is called.

If the implementation cannot be made thread-safe for technical reasons, ensure that `getmeta("thread_safe")` returns `False`.

Meta Values

The *getmeta()* method is designed to return implementation specific information. PyFilesystem implementations should return as much of the standard set of meta values as possible.

Implementations are also free to reserve a dotted namespace notation for themselves, to provide an interface to highly specific information. If you do this, please avoid generic terms as they may conflict with existing or future implementations. For example `"bobs_ftpfs.author"`, rather than `"ftpfs.author"`.

If your meta values are static, i.e. they never change, then create a dictionary class attribute called `_meta` in your implementation that contains all the meta keys and values. The default `getmeta` implementation will pull the meta values from this dictionary.

Essential Methods

The following methods are required for a minimal Filesystem interface:

- *open()* Opens a file for read/writing
- *isfile()* Check whether the path exists and is a file
- *isdir()* Check whether a path exists and is a directory
- *listdir()* List the contents of a directory
- *makedir()* Create a new directory
- *remove()* Remove an existing file
- *removedir()* Remove an existing directory
- *rename()* Atomically rename a file or directory
- *getinfo()* Return information about the path e.g. size, mtime

Non - Essential Methods

The following methods have default implementations in *FS* and aren't required for a functional FS interface. They may be overridden if an alternative implementation can be supplied:

- *copy()* Copy a file to a new location
- *copydir()* Recursively copy a directory to a new location
- *desc()* Return a short descriptive text regarding a path
- *exists()* Check whether a path exists as file or directory
- *listdirinfo()* Get a directory listing along with the info dict for each entry
- *ilistdir()* Generator version of the listdir method

- *ilistdirinfo()* Generator version of the listdirinfo method
- *getpathurl()* Get an external URL at which the given file can be accessed, if possible
- *getsyspath()* Get a file's name in the local filesystem, if possible
- *getmeta()* Get the value of a filesystem meta value, if it exists
- *getmmap()* Gets an mmap object for the given resource, if supported
- *hassyspath()* Check if a path maps to a system path (recognized by the OS)
- *haspathurl()* Check if a path maps to an external URL
- *hasmeta()* Check if a filesystem meta value exists
- *move()* Move a file to a new location
- *movedir()* Recursively move a directory to a new location
- *settimes()* Sets the accessed and modified times of a path

Release Notes

PyFilesystem has reached a point where the interface is relatively stable. The were some backwards incompatibilities introduced with version 0.5.0, due to Python 3 support.

Changes from 0.4.0

Python 3.X support was added. The interface has remained much the same, but the `open` method now works like Python 3's builtin, which handles text encoding more elegantly. i.e. if you open a file in text mode, you get a stream that reads or writes unicode rather than binary strings.

The new signature to the `open` method (and `safeopen`) is as follows:

```
def open(self, path, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
↪ line_buffering=False, **kwargs):
```

In order to keep the same signature across both Python 2 and 3, PyFilesystems uses the `io` module from the standard library. Unfortunately this is only available from Python 2.6 onwards, so Python 2.5 support has been dropped. If you need Python 2.5 support, consider sticking to PyFilesystem 0.4.0.

By default the new `open` method now returns a unicode text stream, whereas 0.4.0 returned a binary file-like object. If you have code that runs on 0.4.0, you will probably want to either modify your code to work with unicode or explicitly open files in binary mode. The latter is as simple as changing the mode from "r" to "rb" (or "w" to "wb"), but if you were working with unicode, the new text streams will likely save you a few lines of code.

The `setcontents` and `getcontents` methods have also grown a few parameters in order to work with text files. So you won't require an extra encode / decode step for text files.

CHAPTER 2

Code Documentation

fs.appdirfs

A collection of filesystems that map to application specific locations.

These classes abstract away the different requirements for user data across platforms, which vary in their conventions. They are all subclasses of *OSFS*, all that differs from *OSFS* is the constructor which detects the appropriate location given the name of the application, author name and other parameters.

Uses *appdirs* (https://github.com/ActiveState/appdirs), written by Trent Mick and Sridhar Ratnakumar <trentm at gmail com; github at srid name>

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

A filesystem for per-user application data.

Parameters

- **appname** – the name of the application
- **appauthor** – the name of the author (used on Windows)
- **version** – optional version string, if a unique location per version of the application is required
- **roaming** – if True, use a *roaming* profile on Windows, see http://technet.microsoft.com/en-us/library/cc766489(WS.10).aspx
- **create** – if True (the default) the directory will be created if it does not exist

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

A filesystem for application site data.

Parameters

- **appname** – the name of the application
- **appauthor** – the name of the author (not used on linux)

- **version** – optional version string, if a unique location per version of the application is required
- **roaming** – if True, use a *roaming* profile on Windows, see http://technet.microsoft.com/en-us/library/cc766489(WS.10).aspx
- **create** – if True (the default) the directory will be created if it does not exist

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

A filesystem for per-user application cache data.

Parameters

- **appname** – the name of the application
- **appauthor** – the name of the author (not used on linux)
- **version** – optional version string, if a unique location per version of the application is required
- **roaming** – if True, use a *roaming* profile on Windows, see http://technet.microsoft.com/en-us/library/cc766489(WS.10).aspx
- **create** – if True (the default) the directory will be created if it does not exist

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

A filesystem for per-user application log data.

Parameters

- **appname** – the name of the application
- **appauthor** – the name of the author (not used on linux)
- **version** – optional version string, if a unique location per version of the application is required
- **roaming** – if True, use a *roaming* profile on Windows, see http://technet.microsoft.com/en-us/library/cc766489(WS.10).aspx
- **create** – if True (the default) the directory will be created if it does not exist

fs.base

This module contains the basic FS interface and a number of other essential interfaces.

fs.base.FS

All Filesystem objects inherit from this class.

class fs.base.**FS**(*thread_synchronize=True*)

The base class for Filesystem abstraction objects. An instance of a class derived from FS is an abstraction on some kind of filesystem, such as the OS filesystem or a zip file.

The base class for Filesystem objects.

Parameters **thread_synconize** – If True, a lock object will be created for the object, otherwise a dummy lock will be used.

browse(*hide_dotfiles=False*)
Displays the FS tree in a graphical window (requires wxPython)

Parameters **hide_dotfiles** – If True, files and folders that begin with a dot will be hidden

cache_hint(*enabled*)

Recommends the use of caching. Implementations are free to use or ignore this value.

Parameters **enabled** – If True the implementation is permitted to aggressively cache directory structure / file information. Caching such information can speed up many operations, particularly for network based filesystems. The downside of caching is that changes made to directories or files outside of this interface may not be picked up immediately.

cachehint(*enabled*)

Recommends the use of caching. Implementations are free to use or ignore this value.

Parameters **enabled** – If True the implementation is permitted to aggressively cache directory structure / file information. Caching such information can speed up many operations, particularly for network based filesystems. The downside of caching is that changes made to directories or files outside of this interface may not be picked up immediately.

close()
Close the filesystem. This will perform any shutdown related operations required. This method will be called automatically when the filesystem object is garbage collected, but it is good practice to call it explicitly so that any attached resourced are freed when they are no longer required.

copy(*src*, *dst*, *overwrite=False*, *chunk_size=65536*)
Copies a file from src to dst.

Parameters

- **src** (*string*) – the source path
- **dst** (*string*) – the destination path
- **overwrite** (*bool*) – if True, then an existing file at the destination may be overwritten; If False then DestinationExistsError will be raised.
- **chunk_size** (*bool*) – size of chunks to use if a simple copy is required (defaults to 64K).

copydir(*src*, *dst*, *overwrite=False*, *ignore_errors=False*, *chunk_size=16384*)
copies a directory from one location to another.

Parameters

- **src** (*string*) – source directory path
- **dst** (*string*) – destination directory path
- **overwrite** (*bool*) – if True then any existing files in the destination directory will be overwritten
- **ignore_errors** (*bool*) – if True, exceptions when copying will be ignored
- **chunk_size** – size of chunks to use when copying, if a simple copy is required (defaults to 16K)

createfile(*path*, *wipe=False*)
Creates an empty file if it doesn't exist

Parameters

- **path** – path to the file to create
- **wipe** – if True, the contents of the file will be erased

desc(*path*)
Returns short descriptive text regarding a path. Intended mainly as a debugging aid.

Parameters **path** – A path to describe

Return type str

exists(*path*)
Check if a path references a valid resource.

Parameters **path** (*string*) – A path in the filesystem

Return type bool

getcontents(*path*, *mode='rb'*, *encoding=None*, *errors=None*, *newline=None*)
Returns the contents of a file as a string.

Parameters

- **path** – A path of file to read
- **mode** – Mode to open file with (should be 'rb' for binary or 't' for text)
- **encoding** – Encoding to use when reading contents in text mode
- **errors** – Unicode errors parameter if text mode is use
- **newline** – Newlines parameter for text mode decoding

Return type str

Returns file contents

getinfo(*path*)
Returns information for a path as a dictionary. The exact content of this dictionary will vary depending on the implementation, but will likely include a few common values. The following values will be found in info dictionaries for most implementations:

•"size" - Number of bytes used to store the file or directory

•"created_time" - A datetime object containing the time the resource was created

•"accessed_time" - A datetime object containing the time the resource was last accessed

•"modified_time" - A datetime object containing the time the resource was modified

Parameters **path** (*string*) – a path to retrieve information for

Return type dict

Raises

- ***fs.errors.ParentDirectoryMissingError*** – if an intermediate directory is missing
- ***fs.errors.ResourceInvalidError*** – if the path is not a directory
- ***fs.errors.ResourceNotFoundError*** – if the path does not exist

getinfokeys(*path*, **keys*)

Get specified keys from info dict, as returned from *getinfo*. The returned dictionary may not contain all the keys that were asked for, if they aren't available.

This method allows a filesystem to potentially provide a faster way of retrieving these info values if you are only interested in a subset of them.

Parameters

- **path** – a path to retrieve information for
- **keys** – the info keys you would like to retrieve

Return type dict

getmeta(*meta_name*, *default=<class 'fs.base.NoDefaultMeta'>*)

Retrieve a meta value associated with an FS object.

Meta values are a way for an FS implementation to report potentially useful information associated with the file system.

A meta key is a lower case string with no spaces. Meta keys may also be grouped in namespaces in a dotted notation, e.g. 'atomic.namespaces'. FS implementations aren't obliged to return any meta values, but the following are common:

•*read_only* True if the file system cannot be modified

•*thread_safe* True if the implementation is thread safe

•*network* True if the file system requires network access

•*unicode_paths* True if the file system supports unicode paths

•*case_insensitive_paths* True if the file system ignores the case of paths

•*atomic.makedir* True if making a directory is an atomic operation

•*atomic.rename* True if rename is an atomic operation, (and not implemented as a copy followed by a delete)

•*atomic.setcontents* True if the implementation supports setting the contents of a file as an atomic operation (without opening a file)

•*free_space* The free space (in bytes) available on the file system

•*total_space* The total space (in bytes) available on the file system

•*virtual* True if the filesystem defers to other filesystems

•*invalid_path_chars* A string containing characters that may not be used in paths

FS implementations may expose non-generic meta data through a self-named namespace. e.g. `"somefs.some_meta"`

Since no meta value is guaranteed to exist, it is advisable to always supply a default value to `getmeta`.

Parameters

- **meta_name** – The name of the meta value to retrieve
- **default** – An option default to return, if the meta value isn't present

Raises ***fs.errors.NoMetaError*** – If specified meta value is not present, and there is no default

getmmap (*path*, *read_only=False*, *copy=False*)
Returns a mmap object for this path.

See http://docs.python.org/library/mmap.html for more details on the mmap module.

Parameters

- **path** – A path on this filesystem
- **read_only** – If True, the mmap may not be modified
- **copy** – If False then changes wont be written back to the file

Raises ***fs.errors.NoMMapError*** – Only paths that have a syspath can be opened as a mmap

getpathurl (*path*, *allow_none=False*)
Returns a url that corresponds to the given path, if one exists.

If the path does not have an equivalent URL form (and allow_none is False) then a *NoPathURLError* exception is thrown. Otherwise the URL will be returns as an unicode string.

Parameters

- **path** – a path within the filesystem
- **allow_none** (*bool*) – if true, this method can return None if there is no URL form of the given path

Raises ***fs.errors.NoPathURLError*** – If no URL form exists, and allow_none is False (the default)

Return type unicode

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

Parameters **path** (*string*) – a path to the resource

Returns the size of the file

Return type integer

getsyspath (*path*, *allow_none=False*)
Returns the system path (a path recognized by the OS) if one is present.

If the path does not map to a system path (and *allow_none* is False) then a NoSysPathError exception is thrown. Otherwise, the system path will be returned as a unicode string.

Parameters

- **path** – a path within the filesystem
- **allow_none** (*bool*) – if True, this method will return None when there is no system path, rather than raising NoSysPathError

Raises ***fs.errors.NoSysPathError*** – if the path does not map on to a system path, and allow_none is set to False (default)

Return type unicode

hasmeta (*meta_name*)
Check that a meta value is supported

Parameters **meta_name** – The name of a meta value to check

Return type bool

haspathurl (*path*)
Check if the path has an equivalent URL form

Parameters **path** – path to check

Returns True if *path* has a URL form

Return type bool

hassyspath (*path*)
Check if the path maps to a system path (a path recognized by the OS).

Parameters **path** – path to check

Returns True if *path* maps to a system path

Return type bool

ilistdir (*path='./'*, *wildcard=None*, *full=False*, *absolute=False*, *dirs_only=False*, *files_only=False*)
Generator yielding the files and directories under a given path.

This method behaves identically to *listdir()* but returns an generator instead of a list. Depending on the filesystem this may be more efficient than calling *listdir()* and iterating over the resulting list.

ilistdirinfo (*path='./'*, *wildcard=None*, *full=False*, *absolute=False*, *dirs_only=False*, *files_only=False*)
Generator yielding paths and path info under a given path.

This method behaves identically to `listdirinfo()` but returns an generator instead of a list. Depending on the filesystem this may be more efficient than calling `listdirinfo()` and iterating over the resulting list.

isdir (*path*)
Check if a path references a directory.

Parameters **path** (*string*) – a path in the filesystem

Return type bool

isdirempty (*path*)
Check if a directory is empty (contains no files or sub-directories)

Parameters **path** – a directory path

Return type bool

isfile (*path*)
Check if a path references a file.

Parameters **path** (*string*) – a path in the filesystem

Return type bool

isvalidpath (*path*)
Check if a path is valid on this filesystem

Parameters **path** – an fs path

listdir (*path='./'*, *wildcard=None*, *full=False*, *absolute=False*, *dirs_only=False*, *files_only=False*)
Lists the the files and directories under a given path.

The directory contents are returned as a list of unicode paths.

Parameters

- **path** (*string*) – root of the path to list

- **wildcard** (*string containing a wildcard, or a callable that accepts a path and returns a boolean*) – Only returns paths that match this wildcard
- **full** (*bool*) – returns full paths (relative to the root)
- **absolute** (*bool*) – returns absolute paths (paths beginning with /)
- **dirs_only** (*bool*) – if True, only return directories
- **files_only** (*bool*) – if True, only return files

Return type iterable of paths

Raises

- ***fs.errors.ParentDirectoryMissingError*** – if an intermediate directory is missing
- ***fs.errors.ResourceInvalidError*** – if the path exists, but is not a directory
- ***fs.errors.ResourceNotFoundError*** – if the path is not found

listdirinfo(*path='./'*, *wildcard=None*, *full=False*, *absolute=False*, *dirs_only=False*, *files_only=False*)

Retrieves a list of paths and path info under a given path.

This method behaves like listdir() but instead of just returning the name of each item in the directory, it returns a tuple of the name and the info dict as returned by getinfo.

This method may be more efficient than calling *getinfo()* on each individual item returned by *listdir()*, particularly for network based filesystems.

Parameters

- **path** – root of the path to list
- **wildcard** – filter paths that match this wildcard
- **dirs_only** (*bool*) – only retrieve directories
- **files_only** (*bool*) – only retrieve files

Raises

- ***fs.errors.ResourceNotFoundError*** – If the path is not found
- ***fs.errors.ResourceInvalidError*** – If the path exists, but is not a directory

makedir(*path*, *recursive=False*, *allow_recreate=False*)

Make a directory on the filesystem.

Parameters

- **path** (*string*) – path of directory
- **recursive** (*bool*) – if True, any intermediate directories will also be created
- **allow_recreate** – if True, re-creating a directory wont be an error

Raises

- ***fs.errors.DestinationExistsError*** – if the path is already a directory, and allow_recreate is False
- ***fs.errors.ParentDirectoryMissingError*** – if a containing directory is missing and recursive is False
- ***fs.errors.ResourceInvalidError*** – if a path is an existing file

- **fs.errors.ResourceNotFoundError** – if the path is not found

makeopendir(*path*, *recursive=False*)
makes a directory (if it doesn't exist) and returns an FS object for the newly created directory.

Parameters

- **path** – path to the new directory
- **recursive** – if True any intermediate directories will be created

Returns the opened dir

Return type an FS object

move(*src*, *dst*, *overwrite=False*, *chunk_size=16384*)
moves a file from one location to another.

Parameters

- **src** (*string*) – source path
- **dst** (*string*) – destination path
- **overwrite** (*bool*) – When True the destination will be overwritten (if it exists), otherwise a DestinationExistsError will be thrown
- **chunk_size** (*integer*) – Size of chunks to use when copying, if a simple copy is required

Raises **fs.errors.DestinationExistsError** – if destination exists and *overwrite* is False

movedir(*src*, *dst*, *overwrite=False*, *ignore_errors=False*, *chunk_size=16384*)
moves a directory from one location to another.

Parameters

- **src** (*string*) – source directory path
- **dst** (*string*) – destination directory path
- **overwrite** (*bool*) – if True then any existing files in the destination directory will be overwritten
- **ignore_errors** (*bool*) – if True then this method will ignore FSError exceptions when moving files
- **chunk_size** (*integer*) – size of chunks to use when copying, if a simple copy is required

Raises **fs.errors.DestinationExistsError** – if destination exists and *overwrite* is False

open(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, ***kwargs*)
Open a the given path as a file-like object.

Parameters

- **path** (*string*) – a path to file that should be opened
- **mode** (*string*) – mode of file to open, identical to the mode string used in 'file' and 'open' builtins
- **kwargs** (*dict*) – additional (optional) keyword parameters that may be required to open the file

Return type a file-like object

Raises

- ***fs.errors.ParentDirectoryMissingError*** – if an intermediate directory is missing
- ***fs.errors.ResourceInvalidError*** – if an intermediate directory is an file
- ***fs.errors.ResourceNotFoundError*** – if the path is not found

opendir(*path*)
Opens a directory and returns a FS object representing its contents.

Parameters **path** (*string*) – path to directory to open

Returns the opened dir

Return type an FS object

printtree(*max_levels=5*)
Prints a tree structure of the FS object to the console

Parameters **max_levels** – The maximum sub-directories to display, defaults to 5. Set to None for no limit

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

Parameters **path** (*string*) – Path of the resource to remove

Raises

- ***fs.errors.ParentDirectoryMissingError*** – if an intermediate directory is missing
- ***fs.errors.ResourceInvalidError*** – if the path is a directory
- ***fs.errors.ResourceNotFoundError*** – if the path does not exist

removedir(*path*, *recursive=False*, *force=False*)
Remove a directory from the filesystem

Parameters

- **path** (*string*) – path of the directory to remove
- **recursive** (*bool*) – if True, empty parent directories will be removed
- **force** (*bool*) – if True, any directory contents will be removed

Raises

- ***fs.errors.DirectoryNotEmptyError*** – if the directory is not empty and force is False
- ***fs.errors.ParentDirectoryMissingError*** – if an intermediate directory is missing
- ***fs.errors.ResourceInvalidError*** – if the path is not a directory
- ***fs.errors.ResourceNotFoundError*** – if the path does not exist

rename(*src*, *dst*)
Renames a file or directory

Parameters

- **src** (*string*) – path to rename
- **dst** (*string*) – new name

Raises

- ***ParentDirectoryMissingError*** – if a containing directory is missing
- ***ResourceInvalidError*** – if the path or a parent path is not a directory or src is a parent of dst or one of src or dst is a dir and the other don't
- ***ResourceNotFoundError*** – if the src path does not exist

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

Like *open()*, but returns a `NullFile` if the file could not be opened.

A `NullFile` is a dummy file which has all the methods of a file-like object, but contains no data.

Parameters

- **path** (*string*) – a path to file that should be opened
- **mode** (*string*) – mode of file to open, identical to the mode string used in 'file' and 'open' builtins
- **kwargs** (*dict*) – additional (optional) keyword parameters that may be required to open the file

Return type a file-like object

setcontents(*path*, *data=''*, *encoding=None*, *errors=None*, *chunk_size=65536*)

A convenience method to create a new file from a string or file-like object

Parameters

- **path** – a path of the file to create
- **data** – a string or bytes object containing the contents for the new file
- **encoding** – if *data* is a file open in text mode, or a text string, then use this *encoding* to write to the destination file
- **errors** – if *data* is a file open in text mode or a text string, then use *errors* when opening the destination file
- **chunk_size** – Number of bytes to read in a chunk, if the implementation has to resort to a read / copy loop

setcontents_async(*path*, *data*, *encoding=None*, *errors=None*, *chunk_size=65536*, *progress_callback=None*, *finished_callback=None*, *error_callback=None*)

Create a new file from a string or file-like object asynchronously

This method returns a `threading.Event` object. Call the `wait` method on the event object to block until all data has been written, or simply ignore it.

Parameters

- **path** – a path of the file to create
- **data** – a string or a file-like object containing the contents for the new file
- **encoding** – if *data* is a file open in text mode, or a text string, then use this *encoding* to write to the destination file
- **errors** – if *data* is a file open in text mode or a text string, then use *errors* when opening the destination file

- **chunk_size** – Number of bytes to read and write in a chunk
- **progress_callback** – A function that is called periodically with the number of bytes written.
- **finished_callback** – A function that is called when all data has been written
- **error_callback** – A function that is called with an exception object if any error occurs during the copy process.

Returns An event object that is set when the copy is complete, call the *wait* method of this object to block until the data is written

settimes (**args*, ***kwds*)
Set the accessed time and modified time of a file

Parameters

- **path** (*string*) – path to a file
- **accessed_time** (*datetime*) – the datetime the file was accessed (defaults to current time)
- **modified_time** (*datetime*) – the datetime the file was modified (defaults to current time)

tree (*max_levels=5*)
Prints a tree structure of the FS object to the console

Parameters **max_levels** – The maximum sub-directories to display, defaults to 5. Set to None for no limit

validatepath (*path*)
Validate an fs path, throws an *InvalidPathError* exception if validation fails.

A path is invalid if it fails to map to a path on the underlaying filesystem. The default implementation checks for the presence of any of the characters in the meta value 'invalid_path_chars', but implementations may have other requirements for paths.

Parameters **path** – an fs path to validatepath

Raises ***fs.errors.InvalidPathError*** – if *path* does not map on to a valid path on this filesystem

walk (*path='/'*, *wildcard=None*, *dir_wildcard=None*, *search='breadth'*, *ignore_errors=False*)
Walks a directory tree and yields the root path and contents. Yields a tuple of the path of each directory and a list of its file contents.

Parameters

- **path** (*string*) – root path to start walking
- **wildcard** (a string containing a wildcard (e.g. **.txt*) or a callable that takes the file path and returns a boolean) – if given, only return files that match this wildcard
- **dir_wildcard** (a string containing a wildcard (e.g. **.txt*) or a callable that takes the directory name and returns a boolean) – if given, only walk directories that match the wildcard
- **search** – a string identifying the method used to walk the directories. There are two such methods:
  - `"breadth"` yields paths in the top directories first
  - `"depth"` yields the deepest paths first

- **ignore_errors** (*bool*) – ignore any errors reading the directory

Return type iterator of (current_path, paths)

walkdirs(*path='/'*, *wildcard=None*, *search='breadth'*, *ignore_errors=False*)
Like the 'walk' method but yields directories.

Parameters

- **path** (*string*) – root path to start walking
- **wildcard** (A string containing a wildcard (e.g. **.txt*) or a callable that takes the directory name and returns a boolean) – if given, only return directories that match this wildcard
- **search** – a string identifying the method used to walk the directories. There are two such methods:
  - `"breadth"` yields paths in the top directories first
  - `"depth"` yields the deepest paths first
- **ignore_errors** (*bool*) – ignore any errors reading the directory

Return type iterator of dir paths

walkfiles(*path='/'*, *wildcard=None*, *dir_wildcard=None*, *search='breadth'*, *ignore_errors=False*)
Like the 'walk' method, but just yields file paths.

Parameters

- **path** (*string*) – root path to start walking
- **wildcard** (A string containing a wildcard (e.g. **.txt*) or a callable that takes the file path and returns a boolean) – if given, only return files that match this wildcard
- **dir_wildcard** (A string containing a wildcard (e.g. **.txt*) or a callable that takes the directory name and returns a boolean) – if given, only walk directories that match the wildcard
- **search** – a string identifying the method used to walk the directories. There are two such methods:
  - `"breadth"` yields paths in the top directories first
  - `"depth"` yields the deepest paths first
- **ignore_errors** (*bool*) – ignore any errors reading the directory

Return type iterator of file paths

fs.base.SubFS

A SubFS is an FS implementation that represents a directory on another Filesystem. When you use the *opendir()* method it will return a SubFS instance. You should not need to instantiate a SubFS directly.

For example:

```
from fs.osfs import OSFS
home_fs = OSFS('foo')
bar_fs = home_fs.opendir('bar')
```

fs.base.NullFile

A NullFile is a file-like object with no functionality. It is used in situations where a file-like object is required but the caller doesn't have any data to read or write.

The *safeopen()* method returns an NullFile instance, which can reduce error-handling code.

For example, the following code may be written to append some text to a log file:

```
logfile = None
try:
    logfile = myfs.open('log.txt', 'a')
    logfile.writeline('operation successful!')
finally:
    if logfile is not None:
        logfile.close()
```

This could be re-written using the *safeopen* method:

```
myfs.safeopen('log.txt', 'a').writeline('operation successful!')
```

If the file doesn't exist then the call to writeline will be a null-operation (i.e. not do anything).

fs.browsewin

fs.contrib

The `fs.contrib` module contains a number of filesystem implementations provided by third parties.

fs.contrib.davfs

FS implementation accessing a WebDAV server.

This module provides a relatively-complete WebDAV Level 1 client that exposes a WebDAV server as an FS object. Locks are not currently supported.

Requires the dexml module:

http://pypi.python.org/pypi/dexml/

class `fs.contrib.davfs.`**`DAVFS`**(*url*, *credentials=None*, *get_credentials=None*, *thread_synchronize=True*, *connection_classes=None*, *timeout=None*)

Access a remote filesystem via WebDAV.

This FS implementation provides access to a remote filesystem via the WebDAV protocol. Basic Level 1 WebDAV is supported; locking is not currently supported, but planned for the future.

HTTP Basic authentication is supported; provide a dict giving username and password in the "credentials" argument, or a callback for obtaining one in the "get_credentials" argument.

To use custom HTTP connector classes (e.g. to implement proper certificate checking for SSL connections) you can replace the factory functions in the DAVFS.connection_classes dictionary, or provide the "connection_classes" argument.

DAVFS constructor.

The only required argument is the root url of the remote server. If authentication is required, provide the 'credentials' keyword argument and/or the 'get_credentials' keyword argument. The former is a dict of credentials info, while the latter is a callback function returning such a dict. Only HTTP Basic Auth is supported at this stage, so the only useful keys in a credentials dict are 'username' and 'password'.

getpathurl (*path*, *allow_none=False*)
Convert a client-side path into a server-side URL.

fs.contrib.tahoelafs

This modules provides a PyFilesystem interface to the Tahoe Least Authority File System. Tahoe-LAFS is a distributed, encrypted, fault-tolerant storage system:

http://tahoe-lafs.org/

You will need access to a Tahoe-LAFS "web api" service.

Example (it will use publicly available (but slow) Tahoe-LAFS cloud):

```
from fs.contrib.tahoelafs import TahoeLAFS, Connection
dircap = TahoeLAFS.createdircap(webapi='http://insecure.tahoe-lafs.org')
print "Your dircap (unique key to your storage directory) is", dircap
print "Keep it safe!"
fs = TahoeLAFS(dircap, autorun=False, webapi='http://insecure.tahoe-lafs.org')
f = fs.open("foo.txt", "a")
f.write('bar!')
f.close()
print "Now visit %s and enjoy :-)" % fs.getpathurl('foo.txt')
```

When any problem occurred, you can turn on internal debugging messages:

```
import logging
l = logging.getLogger()
l.setLevel(logging.DEBUG)
l.addHandler(logging.StreamHandler(sys.stdout))

... your Python code using TahoeLAFS ...
```

TODO:

- unicode support
- try network errors / bad happiness
- exceptions
- tests
- sanitize all path types (., /)
- support for extra large file uploads (poster module)
- Possibility to block write until upload done (Tahoe mailing list)
- Report something sane when Tahoe crashed/unavailable
- solve failed unit tests (makedir_winner, ...)
- file times
- docs & author
- python3 support

- remove creating blank files (depends on FileUploadManager)

TODO (Not TahoeLAFS specific tasks):

- RemoteFileBuffer on the fly buffering support
- RemoteFileBuffer unit tests
- RemoteFileBuffer submit to trunk
- Implement FileUploadManager + faking isfile/exists of just processing file
- pyfilesystem docs is outdated (rename, movedir, ...)

class `fs.contrib.tahoelafs.`**`TahoeLAFS`**(**args*, ***kwds*)
FS providing cached access to a Tahoe Filesystem.

This class is the preferred means to access a Tahoe filesystem. It maintains an internal cache of recently-accessed metadata to speed up operations.

fs.contrib.bigfs

A FS object that represents the contents of a BIG file (C&C Generals, BfME C&C3, C&C Red Alert 3, C&C4 file format)

Written by Koen van de Sande http://www.tibed.net

Contributed under the terms of the BSD License: http://www.opensource.org/licenses/bsd-license.php

class `fs.contrib.bigfs.`**`BigFS`**(*filename*, *mode='r'*, *thread_synchronize=True*)
A FileSystem that represents a BIG file.

Create a FS that maps on to a big file.

Parameters

- **`filename`** – A (system) path, or a file-like object
- **`mode`** – Mode to open file: 'r' for reading, 'w' and 'a' not supported
- **`thread_synchronize`** – – Set to True (default) to enable thread-safety

`close`()
Finalizes the zip file so that it can be read. No further operations will work after this method is called.

fs.errors

Defines the Exception classes thrown by PyFilesystem objects. Exceptions relating to the underlying filesystem are translated in to one of the following Exceptions. Exceptions that relate to a path store that path in *self.path*.

All Exception classes are derived from *FSError* which can be used as a catch-all exception.

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

exception `fs.errors.`**`CreateFailedError`**(*msg=None*, *details=None*)
An exception thrown when a FS could not be created

exception `fs.errors.`**`PathError`**(*path=''*, ***kwds*)
Exception for errors to do with a path string.

exception fs.errors.**InvalidPathError**(*path=''*, ***kwds*)
Base exception for fs paths that can't be mapped on to the underlaying filesystem.

exception fs.errors.**InvalidCharsInPathError**(*path=''*, ***kwds*)
The path contains characters that are invalid on this filesystem

exception fs.errors.**OperationFailedError**(*opname=''*, *path=None*, ***kwds*)
Base exception class for errors associated with a specific operation.

exception fs.errors.**UnsupportedError**(*opname=''*, *path=None*, ***kwds*)
Exception raised for operations that are not supported by the FS.

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

exception fs.errors.**StorageSpaceError**(*opname=''*, *path=None*, ***kwds*)
Exception raised when operations encounter storage space trouble.

exception fs.errors.**ResourceError**(*path=''*, ***kwds*)
Base exception class for error associated with a specific resource.

exception fs.errors.**NoSysPathError**(*path=''*, ***kwds*)
Exception raised when there is no syspath for a given path.

exception fs.errors.**NoMetaError**(*meta_name*, *msg=None*)
Exception raised when there is no meta value available.

exception fs.errors.**NoPathURLError**(*path=''*, ***kwds*)
Exception raised when there is no URL form for a given path.

exception fs.errors.**ResourceNotFoundError**(*path=''*, ***kwds*)
Exception raised when a required resource is not found.

exception fs.errors.**ResourceInvalidError**(*path=''*, ***kwds*)
Exception raised when a resource is the wrong type.

exception fs.errors.**DestinationExistsError**(*path=''*, ***kwds*)
Exception raised when a target destination already exists.

exception fs.errors.**DirectoryNotEmptyError**(*path=''*, ***kwds*)
Exception raised when a directory to be removed is not empty.

exception fs.errors.**ParentDirectoryMissingError**(*path=''*, ***kwds*)
Exception raised when a parent directory is missing.

exception fs.errors.**ResourceLockedError**(*path=''*, ***kwds*)
Exception raised when a resource can't be used because it is locked.

exception fs.errors.**NoMMapError**(*path=''*, ***kwds*)
Exception raise when getmmap fails to create a mmap

exception fs.errors.**BackReferenceError**
Exception raised when too many backrefs exist in a path (ex: '/..', '/docs/../..').

fs.errors.**convert_fs_errors**(*func*)
Function wrapper to convert FSError instances into OSError.

fs.errors.**convert_os_errors**(*func*)
Function wrapper to convert OSError/IOError instances into FSError.

fs.expose

The `fs.expose` module contains a number of options for making an FS implementation available over the Internet, or to other applications.

fs.expose.dokan

Expose an FS object to the native filesystem via Dokan.

This module provides the necessary interfaces to mount an FS object into the local filesystem using Dokan on win32:

```
http://dokan-dev.github.io/
```

For simple usage, the function 'mount' takes an FS object and new device mount point or an existing empty folder and exposes the given FS as that path:

```
>>> from fs.memoryfs import MemoryFS
>>> from fs.expose import dokan
>>> fs = MemoryFS()
>>> # Mount device mount point
>>> mp = dokan.mount(fs, "Q:\")
>>> mp.path
'Q:\'
>>> mp.unmount()
>>> fs = MemoryFS()
>>> # Mount in an existing empty folder.
>>> mp = dokan.mount(fs, "C:\test")
>>> mp.path
'C:\test'
>>> mp.unmount()
```

The above spawns a new background process to manage the Dokan event loop, which can be controlled through the returned subprocess.Popen object. To avoid spawning a new process, set the 'foreground' option:

```
>>> #  This will block until the filesystem is unmounted
>>> dokan.mount(fs, "Q:\", foreground=True)
```

Any additional options for the Dokan process can be passed as keyword arguments to the 'mount' function.

If you require finer control over the creation of the Dokan process, you can instantiate the MountProcess class directly. It accepts all options available to subprocess.Popen:

```
>>> from subprocess import PIPE
>>> mp = dokan.MountProcess(fs, "Q:\", stderr=PIPE)
>>> dokan_errors = mp.communicate()[1]
```

If you are exposing an untrusted filesystem, you may like to apply the wrapper class Win32SafetyFS before passing it into dokan. This will take a number of steps to avoid suspicious operations on windows, such as hiding autorun files.

The binding to Dokan is created via ctypes. Due to the very stable ABI of win32, this should work without further configuration on just about all systems with Dokan installed.

class `fs.expose.dokan.`**FSOperations**(*fs*, *fsname='NTFS'*, *volname='Dokan Volume'*, *securityfolder='/home/docs'*)

Object delegating all DOKAN_OPERATIONS pointers to an FS object.

get_ops_struct()

Get a DOKAN_OPERATIONS struct mapping to our methods.

class fs.expose.dokan.**MountProcess**(*fs*, *path*, *dokan_opts={}*, *nowait=False*, ***kwds*)
subprocess.Popen subclass managing a Dokan mount.

This is a subclass of subprocess.Popen, designed for easy management of a Dokan mount in a background process. Rather than specifying the command to execute, pass in the FS object to be mounted, the target path and a dictionary of options for the Dokan process.

In order to be passed successfully to the new process, the FS object must be pickleable. Since win32 has no fork() this restriction is not likely to be lifted (see also the "multiprocessing" module)

This class has an extra attribute 'path' giving the path of the mounted filesystem, and an extra method 'unmount' that will cleanly unmount it and terminate the process.

unmount()
Cleanly unmount the Dokan filesystem, terminating this subprocess.

class fs.expose.dokan.**Win32SafetyFS**(*wrapped_fs*, *allow_autorun=False*)
FS wrapper for extra safety when mounting on win32.

This wrapper class provides some safety features when mounting untrusted filesystems on win32. Specifically:

- hiding autorun files
- removing colons from paths

fs.expose.dokan.**handle_fs_errors**(*func*)
Method decorator to report FS errors in the appropriate way.

This decorator catches all FS errors and translates them into an equivalent OSError, then returns the negated error number. It also makes the function return zero instead of None as an indication of successful execution.

fs.expose.dokan.**mount**(*fs*, *path*, *foreground=False*, *ready_callback=None*, *unmount_callback=None*, ***kwds*)
Mount the given FS at the given path, using Dokan.

By default, this function spawns a new background process to manage the Dokan event loop. The return value in this case is an instance of the 'MountProcess' class, a subprocess.Popen subclass.

If the keyword argument 'foreground' is given, we instead run the Dokan main loop in the current process. In this case the function will block until the filesystem is unmounted, then return None.

If the keyword argument 'ready_callback' is provided, it will be called when the filesystem has been mounted and is ready for use. Any additional keyword arguments control the behavior of the final dokan mount point. Some interesting options include:

- numthreads: number of threads to use for handling Dokan requests
- fsname: name to display in explorer etc
- flags: DOKAN_OPTIONS bitmask
- securityfolder: folder path used to duplicate security rights on all folders
- FSOperationsClass: custom FSOperations subclass to use

fs.expose.dokan.**timeout_protect**(*func*)
Method decorator to enable timeout protection during call.

This decorator adds an entry to the timeout protect queue before executing the function, and marks it as finished when the function exits.

fs.expose.dokan.**unmount**(*path*)
Unmount the given path.

This function unmounts the dokan path mounted at the given path. It works but may leave dangling processes; its better to use the "unmount" method on the MountProcess class if you have one.

fs.expose.sftp

Expose an FS object over SFTP (via paramiko).

This module provides the necessary interfaces to expose an FS object over SFTP, plugging into the infrastructure provided by the 'paramiko' module.

For simple usage, the class 'BaseSFTPServer' provides an all-in-one server class based on the standard SocketServer module. Use it like so:

```
server = BaseSFTPServer((hostname,port),fs)
server.serve_forever()
```

Note that the base class allows UNAUTHENTICATED ACCESS by default. For more serious work you will probably want to subclass it and override methods such as check_auth_password() and get_allowed_auths().

To integrate this module into an existing server framework based on paramiko, the 'SFTPServerInterface' class provides a concrete implementation of the paramiko.SFTPServerInterface protocol. If you don't understand what this is, you probably don't want to use it.

class fs.expose.sftp.**BaseSFTPServer**(*address*, *fs=None*, *encoding=None*, *host_key=None*, *RequestHandlerClass=None*)

SocketServer.TCPServer subclass exposing an FS via SFTP.

Operation is in the standard SocketServer style. The target FS object can be passed into the constructor, or set as an attribute on the server:

```
server = BaseSFTPServer((hostname,port),fs)
server.serve_forever()
```

It is also possible to specify the host key used by the sever by setting the 'host_key' attribute. If this is not specified, it will default to the key found in the DEFAULT_HOST_KEY variable.

class fs.expose.sftp.**BaseServerInterface**

Paramiko ServerInterface implementation that performs user authentication.

Note that this base class allows UNAUTHENTICATED ACCESS to the exposed FS. This is intentional, since we can't guess what your authentication needs are. To protect the exposed FS, override the following methods:

- get_allowed_auths Determine the allowed auth modes
- check_auth_none Check auth with no credentials
- check_auth_password Check auth with a password
- check_auth_publickey Check auth with a public key

check_auth_none(*username*)

Check whether the user can proceed without authentication.

check_auth_password(*username*, *password*)

Check whether the given password is valid for authentication.

check_auth_publickey(*username*, *key*)

Check whether the given public key is valid for authentication.

get_allowed_auths(*username*)

Return string containing a comma separated list of allowed auth modes.

The available modes are "node", "password" and "publickey".

class fs.expose.sftp.**SFTPHandle**(*owner*, *path*, *flags*)

SFTP file handler pointing to a file in an FS object.

This is a simple file wrapper for SFTPServerInterface, passing read and write requests directly through the to underlying file from the FS.

class fs.expose.sftp.**SFTPRequestHandler**(*request*, *client_address*, *server*)
SocketServer RequestHandler subclass for BaseSFTPServer.

This RequestHandler subclass creates a paramiko Transport, sets up the sftp subsystem, and hands off to the transport's own request handling thread.

handle()
Start the paramiko server, this will start a thread to handle the connection.

setup()
Creates the SSH transport. Sets security options.

class fs.expose.sftp.**SFTPServer**(*channel*, *name*, *server*, *sftp_si=<class 'paramiko.sftp_si.SFTPServerInterface'>*, **largs*, ***kwargs*)
An SFTPServer class that closes the filesystem when done.

The constructor for SFTPServer is meant to be called from within the *.Transport* as a subsystem handler. `server` and any additional parameters or keyword parameters are passed from the original call to *.Transport.set_subsystem_handler*.

Parameters

- **channel** (*Channel*) – channel passed from the *.Transport*.
- **name** (*str*) – name of the requested subsystem.
- **server** (*ServerInterface*) – the server object associated with this channel and subsystem
- **sftp_si** – a subclass of *.SFTPServerInterface* to use for handling individual requests.

class fs.expose.sftp.**SFTPServerInterface**(*server*, *fs*, *encoding=None*, **args*, ***kwds*)
SFTPServerInterface implementation that exposes an FS object.

This SFTPServerInterface subclass expects a single additional argument, the fs object to be exposed. Use it to set up a transport subsystem handler like so:

```
t.set_subsystem_handler("sftp",SFTPServer,SFTPServerInterface,fs)
```

If this all looks too complicated, you might consider the BaseSFTPServer class also provided by this module - it automatically creates the enclosing paramiko server infrastructure.

fs.expose.sftp.**report_sftp_errors**(*func*)
Decorator to catch and report FS errors as SFTP error codes.

Any FSError exceptions are caught and translated into an appropriate return code, while other exceptions are passed through untouched.

fs.expose.xmlrpc

Server to expose an FS via XML-RPC

This module provides the necessary infrastructure to expose an FS object over XML-RPC. The main class is 'RPCFSServer', a SimpleXMLRPCServer subclass designed to expose an underlying FS.

If you need to use a more powerful server than SimpleXMLRPCServer, you can use the RPCFSInterface class to provide an XML-RPC-compatible wrapper around an FS object, which can then be exposed using whatever server you choose (e.g. Twisted's XML-RPC server).

class fs.expose.xmlrpc.**RPCFSInterface**(*fs*)
Wrapper to expose an FS via a XML-RPC compatible interface.

The only real trick is using xmlrpclib.Binary objects to transport the contents of files.

decode_path(*path*)
Decode paths arriving over the wire.

encode_path(*path*)
Encode a filesystem path for sending over the wire.

Unfortunately XMLRPC only supports ASCII strings, so this method must return something that can be represented in ASCII. The default is base64-encoded UTF-8.

class fs.expose.xmlrpc.**RPCFSServer**(*fs*, *addr*, *requestHandler=None*, *logRequests=None*)
Server to expose an FS object via XML-RPC.

This class takes as its first argument an FS instance, and as its second argument a (hostname,port) tuple on which to listen for XML-RPC requests. Example:

```
fs = OSFS('/var/srv/myfiles')
s = RPCFSServer(fs,("",8080))
s.serve_forever()
```

To cleanly shut down the server after calling serve_forever, set the attribute "serve_more_requests" to False.

serve_forever()
Override serve_forever to allow graceful shutdown.

fs.expose.importhook

Expose an FS object to the python import machinery, via a PEP-302 loader.

This module allows you to import python modules from an arbitrary FS object, by placing FS urls on sys.path and/or inserting objects into sys.meta_path.

The main class in this module is FSImportHook, which is a PEP-302-compliant module finder and loader. If you place an instance of FSImportHook on sys.meta_path, you will be able to import modules from the exposed filesystem:

```
>>> from fs.memoryfs import MemoryFS
>>> m = MemoryFS()
>>> m.setcontents("helloworld.py","print 'hello world!'")
>>>
>>> import sys
>>> from fs.expose.importhook import FSImportHook
>>> sys.meta_path.append(FSImportHook(m))
>>> import helloworld
hello world!
```

It is also possible to install FSImportHook as an import path handler. This allows you to place filesystem URLs on sys.path and have them automagically opened for importing. This example would allow modules to be imported from an SFTP server:

```
>>> from fs.expose.importhook import FSImportHook
>>> FSImportHook.install()
>>> sys.path.append("sftp://some.remote.machine/mypath/")
```

class fs.expose.importhook.**FSImportHook**(*fs_or_url*)
PEP-302-compliant module finder and loader for FS objects.

FSImportHook is a module finder and loader that takes its data from an arbitrary FS object. The FS must have .py or .pyc files stored in the standard module structure.

For easy use with sys.path, FSImportHook will also accept a filesystem URL, which is automatically opened using fs.opener.

find_module (*fullname*, *path=None*)
Find the FS loader for the given module.

This object is always its own loader, so this really just checks whether it's a valid module on the exposed filesystem.

get_code (*fullname*, *info=None*)
Get the bytecode for the specified module.

get_data (*path*)
Read the specified data file.

get_filename (*fullname*, *info=None*)
Get the __file__ attribute for the specified module.

get_source (*fullname*, *info=None*)
Get the sourcecode for the specified module, if present.

classmethod install ()
Install this class into the import machinery.

This classmethod installs the custom FSImportHook class into the import machinery of the running process, if it is not already installed.

is_package (*fullname*, *info=None*)
Check whether the specified module is a package.

load_module (*fullname*)
Load the specified module.

This method locates the file for the specified module, loads and executes it and returns the created module object.

classmethod uninstall ()
Uninstall this class from the import machinery.

This classmethod uninstalls the custom FSImportHook class from the import machinery of the running process.

fs.expose.django

Use an FS object for Django File Storage

This module exposes the class "FSStorage", a simple adapter for using FS objects as Django storage objects. Simply include the following lines in your settings.py:

```
DEFAULT_FILE_STORAGE = fs.expose.django_storage.FSStorage
DEFAULT_FILE_STORAGE_FS = OSFS('foo/bar')  # Or whatever FS
```

class fs.expose.django_storage.**FSStorage** (*fs=None*, *base_url=None*)
Expose an FS object as a Django File Storage object.

Parameters

- **fs** – an FS object

- **base_url** – The url to prepend to the path

fs.filelike

This module takes care of the groundwork for implementing and manipulating objects that provide a rich file-like interface, including reading, writing, seeking and iteration.

The main class is FileLikeBase, which implements the entire file-like interface on top of primitive _read(), _write(), _seek(), _tell() and _truncate() methods. Subclasses may implement any or all of these methods to obtain the related higher-level file behaviors.

Other useful classes include:

- **StringIO: a version of the builtin StringIO class, patched to more** closely preserve the semantics of a standard file.
- **FileWrapper: a generic base class for wrappers around a filelike object** (think e.g. compression or decryption).
- **SpooledTemporaryFile: a version of the builtin SpooledTemporaryFile** class, patched to more closely preserve the semantics of a standard file.
- **LimitBytesFile: a filelike wrapper that limits the total bytes read** from a file; useful for turning a socket into a file without reading past end-of-data.

class fs.filelike.**FileLikeBase**(*bufsize=65536*)
Base class for implementing file-like objects.

This class takes a lot of the legwork out of writing file-like objects with a rich interface. It implements the higher-level file-like methods on top of five primitive methods: _read, _write, _seek, _tell and _truncate. See their docstrings for precise details on how these methods behave.

Subclasses then need only implement some subset of these methods for rich file-like interface compatibility. They may of course override other methods as desired.

The class is missing the following attributes and methods, which don't really make sense for anything but real files:

•fileno()

•isatty()

•encoding

•mode

•name

•newlines

Unlike standard file objects, all read methods share the same buffer and so can be freely mixed (e.g. read(), readline(), next(), ...).

This class understands and will accept the following mode strings, with any additional characters being ignored:

•r - open the file for reading only.

•r+ - open the file for reading and writing.

•r- - open the file for streamed reading; do not allow seek/tell.

•**w - open the file for writing only; create the file if** it doesn't exist; truncate it to zero length.

•**w+ - open the file for reading and writing; create the file** if it doesn't exist; truncate it to zero length.

- w- - open the file for streamed writing; do not allow seek/tell.
- **a - open the file for writing only; create the file if it** doesn't exist; place pointer at end of file.
- **a+ - open the file for reading and writing; create the file** if it doesn't exist; place pointer at end of file.

These are mostly standard except for the "-" indicator, which has been added for efficiency purposes in cases where seeking can be expensive to simulate (e.g. compressed files). Note that any file opened for both reading and writing must also support seeking.

FileLikeBase Constructor.

The optional argument 'bufsize' specifies the number of bytes to read at a time when looking for a newline character. Setting this to a larger number when lines are long should improve efficiency.

close()
Flush write buffers and close the file.

The file may not be accessed further once it is closed.

flush()
Flush internal write buffer, if necessary.

next()
next() method complying with the iterator protocol.

File-like objects are their own iterators, with each call to next() returning subsequent lines from the file.

read(*size=-1*)
Read at most 'size' bytes from the file.

Bytes are returned as a string. If 'size' is negative, zero or missing, the remainder of the file is read. If EOF is encountered immediately, the empty string is returned.

readline(*size=-1*)
Read a line from the file, or at most <size> bytes.

readlines(*sizehint=-1*)
Return a list of all lines in the file.

seek(*offset*, *whence=0*)
Move the internal file pointer to the given location.

tell()
Determine current position of internal file pointer.

truncate(*size=None*)
Truncate the file to the given size.

If <size> is not specified or is None, the current file position is used. Note that this method may fail at runtime if the underlying filelike object is not truncatable.

write(*string*)
Write the given string to the file.

writelines(*seq*)
Write a sequence of lines to the file.

xreadlines()
Iterator over lines in the file - equivalent to iter(self).

class fs.filelike.**FileWrapper**(*wrapped_file*, *mode=None*)
Base class for objects that wrap a file-like object.

This class provides basic functionality for implementing file-like objects that wrap another file-like object to alter its functionality in some way. It takes care of house-keeping duties such as flushing and closing the wrapped file.

Access to the wrapped file is given by the attribute wrapped_file. By convention, the subclass's constructor should accept this as its first argument and pass it to its superclass's constructor in the same position.

This class provides a basic implementation of _read() and _write() which just calls read() and write() on the wrapped object. Subclasses will probably want to override these.

FileWrapper constructor.

'wrapped_file' must be a file-like object, which is to be wrapped in another file-like object to provide additional functionality.

If given, 'mode' must be the access mode string under which the wrapped file is to be accessed. If not given or None, it is looked up on the wrapped file if possible. Otherwise, it is not set on the object.

close()
Close the object for reading/writing.

flush()
Flush the write buffers of the file.

class fs.filelike.**LimitBytesFile**(*size*, *fileobj*, **args*, ***kwds*)
Filelike wrapper to limit bytes read from a stream.

class fs.filelike.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *bufsize=-1*, **args*, ***kwds*)
SpooledTemporaryFile wrapper with some compatibility fixes.

This is a simple compatibility wrapper around the native class of the same name, fixing some corner-cases of its behavior. Specifically:

- have truncate() accept a size argument
- roll to disk is seeking past the max in-memory size
- use improved StringIO class from this module

class fs.filelike.**StringIO**(*data=None*, *mode=None*)
StringIO wrapper that more closely matches standard file behavior.

This is a simple compatibility wrapper around the native StringIO class which fixes some corner-cases of its behavior. Specifically:

- adding __enter__ and __exit__ methods
- having truncate(size) zero-fill when growing the file

fs.ftpfs

class fs.ftpfs.**FTPFS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=<object object>*, *port=21*, *dircache=True*, *follow_symlinks=False*)
Connect to a FTP server.

Parameters

- **host** – Host to connect to
- **user** – Username, or a blank string for anonymous
- **passwd** – Password, if required

- **acct** – Accounting information (few servers require this)
- **timeout** – Timeout in seconds
- **port** – Port to connection (default is 21)
- **dircache** – If True then directory information will be cached, speeding up operations such as *getinfo*, *isdir*, *isfile*, but changes to the ftp file structure will not be visible until *clear_dircache()* is called

clear_dircache(**args*, ***kwargs*)
Clear cached directory information.

Parameters **path** – Path of directory to clear cache for, or all directories if

None (the default)

fs.httpfs

class fs.httpfs.**HTTPFS**(*url*)
Can barely be called a filesystem, because HTTP servers generally don't support typical filesystem functionality. This class exists to allow the *fs.opener* system to read files over HTTP.

If you do need filesystem like functionality over HTTP, see *davfs*.

Parameters **url** – The base URL

fs.memoryfs

A Filesystem that exists in memory only. Which makes them extremely fast, but non-permanent.

If you open a file from a *memoryfs* you will get back a StringIO object from the standard library.

class fs.memoryfs.**MemoryFS**(*file_factory=None*)
An in-memory filesystem.

fs.mountfs

Contains MountFS class which is a virtual filesystem which can have other filesystems linked as branched directories.

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.mountdir('config', config_fs)
combined_fs.mountdir('resources', resources_fs)
```

This will create a single 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 can be accessed with the same path structure:

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

MountFS.**mountdir**(*self*, *path*, *fs*)
Mounts a host FS object on a given path.

Parameters

- **path** – A path within the MountFS
- **fs** – A filesystem object to mount

MountFS.**mountfile**(*self*, *path*, *open_callable=None*, *info_callable=None*)
Mounts a single file path.

Parameters

- **path** – A path within the MountFS
- **open_callable** – A callable that returns a file-like object, *open_callable* should have the same signature as *open()*
- **info_callable** – A callable that returns a dictionary with information regarding the file-like object, *info_callable* should have the same signagture as *getinfo()*

MountFS.**unmount**(*path*)
Unmounts a path.

Parameters **path** – Path to unmount

Returns True if a path was unmounted, False if the path was already unmounted

Return type bool

fs.multifs

A MultiFS is a filesystem composed of a sequence of other filesystems, where the directory structure of each filesystem is overlaid over the previous filesystem. When you attempt to access a file from the MultiFS it will try each 'child' FS in order, until it either finds a path that exists or raises a ResourceNotFoundError.

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 looks for files in *templates* if they don't exist in *theme*. We can do this with the following code:

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

themed_template_fs = MultiFS()
themed_template_fs.addfs('templates', OSFS('templates'))
themed_template_fs.addfs('theme', OSFS('theme'))
```

Now we have a *themed_template_fs* FS object presents a single view of both directories:

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

A MultiFS is generally read-only, and any operation that may modify data (including opening files for writing) will fail. However, you can set a writeable fs with the *setwritefs* method – which does not have to be one of the FS objects set with *addfs*.

The reason that only one FS object is ever considered for write access is that otherwise it would be ambiguous as to which filesystem you would want to modify. If you need to be able to modify more than one FS in the MultiFS, you can always access them directly.

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.

Parameters **auto_close** – If True the child filesystems will be closed when the MultiFS is closed

addfs(**args*, ***kwargs*)

Adds a filesystem to the MultiFS.

Parameters

- **name** – A unique name to refer to the filesystem being added. The filesystem can later be retrieved by using this name as an index to the MultiFS, i.e. multifs['myfs']
- **fs** – The filesystem to add
- **write** – If this value is True, then the *fs* will be used as the writeable FS
- **priority** – A number that gives the priorty 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

clearwritefs(**args*, ***kwargs*)
Clears the writeable filesystem (operations that modify the multifs will fail)

removefs(**args*, ***kwargs*)
Removes a filesystem from the sequence.

Parameters **name** – The name of the filesystem, as used in addfs

setwritefs(**args*, ***kwargs*)
Sets the filesystem to use when write access is required. Without a writeable FS, any operations that could modify data (including opening files for writing / appending) will fail.

Parameters **fs** – An FS object that will be used to open writeable files

which(**args*, ***kwargs*)
Retrieves the filesystem that a given path would delegate to. Returns a tuple of the filesystem's name and the filesystem object itself.

Parameters **path** – A path in MultiFS

fs.osfs

Exposes the OS Filesystem as an FS object.

For example, to print all the files and directories in the OS root:

```
>>> from fs.osfs import OSFS
>>> home_fs = OSFS('/')
>>> print home_fs.listdir()
```

class fs.osfs.**OSFS**(*root_path*, *thread_synchronize=True*, *encoding=None*, *create=False*, *dir_mode=448*, *use_long_paths=True*)
Expose the underlying operating-system filesystem as an FS object.

This is the most basic of filesystems, which simply shadows the underlaying filesystem of the OS. Most of its methods simply defer to the matching methods in the os and os.path modules.

Creates an FS object that represents the OS Filesystem under a given root path

Parameters

- **root_path** – The root OS path
- **thread_synchronize** – If True, this object will be thread-safe by use of a threading.Lock object
- **encoding** – The encoding method for path strings
- **create** – If True, then root_path will be created if it doesn't already exist
- **dir_mode** – The mode to use when creating the directory

unsyspath (*path*)
Convert a system-level path into an FS-level path.

This basically the reverse of getsyspath(). If the path does not refer to a location within this filesystem, ValueError is raised.

Parameters **path** – a system path

Returns a path within this FS object

Return type string

fs.opener

Open filesystems via a URI.

There are occasions when you want to specify a filesystem from the command line or in a config file. This module enables that functionality, and can return an FS object given a filesystem specification in a URI-like syntax (inspired by the syntax of http://commons.apache.org/vfs/filesystems.html).

The *OpenerRegistry* class maps the protocol (file, ftp etc.) on to an Opener object, which returns an appropriate filesystem object and path. You can create a custom opener registry that opens just the filesystems you require, or use the opener registry defined here (also called *opener*) that can open any supported filesystem.

The *parse* method of an *OpenerRegsitry* object returns a tuple of an FS object a path. Here's an example of how to use the default opener registry:

```
>>> from fs.opener import opener
>>> opener.parse('ftp://ftp.mozilla.org/pub')
(<fs.ftpfs.FTPFS object at 0x96e66ec>, u'pub')
```

You can use use the *opendir* method, which just returns an FS object. In the example above, *opendir* will return a FS object for the directory *pub*:

```
>>> opener.opendir('ftp://ftp.mozilla.org/pub')
<SubFS: <FTPFS ftp.mozilla.org>/pub>
```

If you are just interested in a single file, use the *open* method of a registry which returns a file-like object, and has the same signature as FS objects and the *open* builtin:

```
>>> opener.open('ftp://ftp.mozilla.org/pub/README')
<fs.ftpfs._FTPFile object at 0x973764c>
```

The *opendir* and *open* methods can also be imported from the top-level of this module for sake of convenience. To avoid shadowing the builtin *open* method, they are named *fsopendir* and *fsopen*. Here's how you might import them:

```
from fs.opener import fsopendir, fsopen
```

exception fs.opener.**OpenerError**
The base exception thrown by openers

exception fs.opener.**NoOpenerError**
Thrown when there is no opener for the given protocol

class fs.opener.**OpenerRegistry** (*openers=[]*)
An opener registry that stores a number of opener objects used to parse FS URIs

add (*opener*)
Adds an opener to the registry

Parameters **opener** – a class derived from fs.opener.Opener

get_opener(*name*)
Retrieve an opener for the given protocol

Parameters **name** – name of the opener to open

Raises ***NoOpenerError*** – if no opener has been registered of that name

getcontents(*fs_url*, *mode='rb'*, *encoding=None*, *errors=None*, *newline=None*)
Gets the contents from a given FS url (if it references a file)

Parameters **fs_url** – a FS URL e.g. ftp://ftp.mozilla.org/README

open(*fs_url*, *mode='r'*, ***kwargs*)
Opens a file from a given FS url

If you intend to do a lot of file manipulation, it would likely be more efficient to do it directly through the an FS instance (from *parse* or *opendir*). This method is fine for one-offs though.

Parameters

- **fs_url** – a FS URL, e.g. ftp://ftp.mozilla.org/README
- **mode** – mode to open file file

Return type a file

opendir(*fs_url*, *writeable=True*, *create_dir=False*)
Opens an FS object from an FS URL

Parameters

- **fs_url** – an FS URL e.g. ftp://ftp.mozilla.org
- **writeable** – set to True (the default) if the FS must be writeable
- **create_dir** – create the directory references by the FS URL, if it doesn't already exist

parse(*fs_url*, *default_fs_name=None*, *writeable=False*, *create_dir=False*, *cache_hint=True*)
Parses a FS url and returns an fs object a path within that FS object (if indicated in the path). A tuple of (<FS instance>, <path>) is returned.

Parameters

- **fs_url** – an FS url
- **default_fs_name** – the default FS to use if none is indicated (defaults is OSFS)
- **writeable** – if True, a writeable FS will be returned
- **create_dir** – if True, then the directory in the FS will be created

class fs.opener.**OpenerRegistry**(*openers=[]*)
An opener registry that stores a number of opener objects used to parse FS URIs

add(*opener*)
Adds an opener to the registry

Parameters **opener** – a class derived from fs.opener.Opener

get_opener(*name*)
Retrieve an opener for the given protocol

Parameters **name** – name of the opener to open

Raises ***NoOpenerError*** – if no opener has been registered of that name

getcontents (*fs_url*, *mode='rb'*, *encoding=None*, *errors=None*, *newline=None*)
Gets the contents from a given FS url (if it references a file)

Parameters **fs_url** – a FS URL e.g. ftp://ftp.mozilla.org/README

open (*fs_url*, *mode='r'*, ***kwargs*)
Opens a file from a given FS url

If you intend to do a lot of file manipulation, it would likely be more efficient to do it directly through the an FS instance (from *parse* or *opendir*). This method is fine for one-offs though.

Parameters

- **fs_url** – a FS URL, e.g. ftp://ftp.mozilla.org/README
- **mode** – mode to open file file

Return type a file

opendir (*fs_url*, *writeable=True*, *create_dir=False*)
Opens an FS object from an FS URL

Parameters

- **fs_url** – an FS URL e.g. ftp://ftp.mozilla.org
- **writeable** – set to True (the default) if the FS must be writeable
- **create_dir** – create the directory references by the FS URL, if it doesn't already exist

parse (*fs_url*, *default_fs_name=None*, *writeable=False*, *create_dir=False*, *cache_hint=True*)
Parses a FS url and returns an fs object a path within that FS object (if indicated in the path). A tuple of (<FS instance>, <path>) is returned.

Parameters

- **fs_url** – an FS url
- **default_fs_name** – the default FS to use if none is indicated (defaults is OSFS)
- **writeable** – if True, a writeable FS will be returned
- **create_dir** – if True, then the directory in the FS will be created

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

Opener follow a very simple protocol. To create an opener, derive a class from *Opener* and define a classmethod called *get_fs*, which should have the following signature:

```
@classmethod
def get_fs(cls, registry, fs_name, fs_name_params, fs_path, writeable, create_
↪dir):
```

The parameters of *get_fs* are as follows:

•*fs_name* the name of the opener, as extracted from the protocol part of the url,

•*fs_name_params* reserved for future use

•*fs_path* the path part of the url

•*writeable* if True, then *get_fs* must return an FS that can be written to

•*create_dir* if True then *get_fs* should attempt to silently create the directory references in path

In addition to *get_fs* an opener class should contain two class attributes: names and desc. *names* is a list of protocols that list opener will opener. *desc* is an English description of the individual opener syntax.

fs.path

Useful functions for FS path manipulation.

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).

class `fs.path.`**PathMap**

Dict-like object with paths for keys.

A PathMap is like a dictionary where the keys are all FS paths. It has two main advantages over a standard dictionary. First, keys are normalized automatically:

```
>>> pm = PathMap()
>>> pm["hello/world"] = 42
>>> print pm["/hello/there/../world"]
42
```

Second, various dictionary operations (e.g. listing or clearing values) can be efficiently performed on a subset of keys sharing some common prefix:

```
# list all values in the map
pm.values()

# list all values for paths starting with "/foo/bar"
pm.values("/foo/bar")
```

Under the hood, a PathMap is a trie-like structure where each level is indexed by path name component. This allows lookups to be performed in O(number of path components) while permitting efficient prefix-based operations.

clear(*root='/'*)

Clear all entries beginning with the given root path.

get(*path*, *default=None*)

Get the value stored under the given path, or the given default.

iteritems(*root='/'*, *m=None*)

Iterate over all (key,value) pairs beginning with the given root.

iterkeys(*root='/'*, *m=None*)

Iterate over all keys beginning with the given root path.

iternames(*root='/'*)

Iterate over all names beneath the given root path.

This is basically the equivalent of listdir() for a PathMap - it yields the next level of name components beneath the given path.

itervalues(*root='/'*, *m=None*)

Iterate over all values whose keys begin with the given root path.

pop(*path*, *default=None*)

Pop the value stored under the given path, or the given default.

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.

fs.path.**basename**(*path*)
Returns the basename of the resource referenced by a path.

This is always equivalent to the 'tail' component of the value returned by pathsplit(path).

Parameters **path** – A FS path

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

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

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

fs.path.**dirname**(*path*)
Returns the parent directory of a path.

This is always equivalent to the 'head' component of the value returned by pathsplit(path).

Parameters **path** – A FS path

```
>>> 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** – An FS path

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

fs.path.**isabs**(*path*)
Return True if path is an absolute path.

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

Parameters **path** – Path to check

```
>>> isdotfile('.baz')
True
```

```
>>> isdotfile('foo/bar/.baz')
True
```

```
>>> isdotfile('foo/bar.baz')
False
```

fs.path.**isprefix**(*path1*, *path2*)
Return true is path1 is a prefix of path2.

Parameters

- **path1** – An FS path
- **path2** – An FS path

```
>>> isprefix("foo/bar", "foo/bar/spam.txt")
True
>>> isprefix("foo/bar/", "foo/bar")
True
>>> isprefix("foo/barry", "foo/baz/bar")
False
>>> isprefix("foo/bar/baz/", "foo/baz/bar")
False
```

fs.path.**issamedir**(*path1*, *path2*)
Return true if two paths reference a resource in the same directory.

Parameters

- **path1** – An FS path
- **path2** – An FS path

```
>>> 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

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

fs.path.**iteratepath**(*path*, *numsplits=None*)
Iterate over the individual components of a path.

Parameters **path** – Path to iterate over

Numsplits Maximum number of splits

fs.path.**join**(**paths*)
Joins any number of paths together, returning a new path string.

This is a simple alias for the pathjoin function, allowing it to be used as fs.path.join in direct correspondence with os.path.join.

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

fs.path.**normpath**(*path*)
Normalizes a path to be in the format expected by FS objects.

This function removes trailing slashes, collapses duplicate slashes, and generally tries very hard to return a new path in the canonical FS format. If the path is invalid, ValueError will be raised.

Parameters **path** – path to normalize

Returns a valid FS path

```
>>> normpath("/foo//bar/frob/../baz")
'/foo/bar/baz'
```

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

fs.path.**ospath**(*path*)
Replace path separators in an OS path if required

fs.path.**pathcombine**(*path1*, *path2*)
Joins two paths together.

This is faster than *pathjoin*, but only works when the second path is relative, and there are no backreferences in either path.

```
>>> pathcombine("foo/bar", "baz")
'foo/bar/baz'
```

fs.path.**pathjoin**(**paths*)
Joins any number of paths together, returning a new path string.

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

```
>>> pathjoin('foo', 'bar', 'baz')
'foo/bar/baz'
```

```
>>> pathjoin('foo/bar', '../baz')
'foo/baz'
```

```
>>> pathjoin('foo/bar', '/baz')
'/baz'
```

fs.path.**pathsplit**(*path*)
Splits 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** – Path to split

```
>>> pathsplit("foo/bar")
('foo', 'bar')
```

```
>>> pathsplit("foo/bar/baz")
('foo/bar', 'baz')
```

```
>>> pathsplit("/foo/bar/baz")
('/foo/bar', 'baz')
```

fs.path.**recursepath**(*path*, *reverse=False*)
Returns intermediate paths from the root to the given path

Parameters **reverse** – reverses the order of the paths

```
>>> recursepath('a/b/c')
['/', u'/a', u'/a/b', u'/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_path** – Path to a directory
- **path** – 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** – Path to adjust

```
>>> relpath('/a/b')
'a/b'
```

fs.path.**split**(*path*)
Splits a path into (head, tail) pair.

This is a simple alias for the `pathsplit` function, allowing it to be used as `fs.path.split` in direct correspondence with `os.path.split`.

Parameters **path** – Path to split

fs.path.**splitext**(*path*)
Splits the extension from the path, and returns the path (up to the last '.' and the extension).

Parameters **path** – A path to split

```
>>> splitext('baz.txt')
('baz', 'txt')
```

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

fs.remote

Utilities for interfacing with remote filesystems

This module provides reusable utility functions that can be used to construct FS subclasses interfacing with a remote filesystem. These include:

- **RemoteFileBuffer: a file-like object that locally buffers the contents of** a remote file, writing them back on flush() or close().
- **ConnectionManagerFS: a WrapFS subclass that tracks the connection state** of a remote FS, and allows client code to wait for a connection to be re-established.
- **CacheFS: a WrapFS subclass that caches file and directory meta-data in** memory, to speed access to a remote FS.

class `fs.remote.`**`CacheFS`**(**args*, ***kwds*)
Simple FS wrapper to cache meta-data of a remote filesystems.

This FS mixin implements a simplistic cache that can help speed up access to a remote filesystem. File and directory meta-data is cached but the actual file contents are not.

CacheFSMixin constructor.

The optional keyword argument 'cache_timeout' specifies the cache timeout in seconds. The default timeout is 1 second. To prevent cache entries from ever timing out, set it to None.

The optional keyword argument 'max_cache_size' specifies the maximum number of entries to keep in the cache. To allow the cache to grow without bound, set it to None. The default is 1000.

class `fs.remote.`**`CacheFSMixin`**(**args*, ***kwds*)
Simple FS mixin to cache meta-data of a remote filesystems.

This FS mixin implements a simplistic cache that can help speed up access to a remote filesystem. File and directory meta-data is cached but the actual file contents are not.

If you want to add caching to an existing FS object, use the CacheFS class instead; it's an easy-to-use wrapper rather than a mixin. This mixin class is provided for FS implementors who want to use caching internally in their own classes.

FYI, the implementation of CacheFS is this:

class CacheFS(CacheFSMixin,WrapFS): pass

CacheFSMixin constructor.

The optional keyword argument 'cache_timeout' specifies the cache timeout in seconds. The default timeout is 1 second. To prevent cache entries from ever timing out, set it to None.

The optional keyword argument 'max_cache_size' specifies the maximum number of entries to keep in the cache. To allow the cache to grow without bound, set it to None. The default is 1000.

class `fs.remote.`**`CachedInfo`**(*info={}*, *has_full_info=True*, *has_full_children=False*)
Info objects stored in cache for CacheFS.

class `fs.remote.`**`ConnectionManagerFS`**(*wrapped_fs*, *poll_interval=None*, *connected=True*)
FS wrapper providing simple connection management of a remote FS.

The ConnectionManagerFS class is designed to wrap a remote FS object and provide some convenience methods for dealing with its remote connection state.

The boolean attribute 'connected' indicates whether the remote filesystem has an active connection, and is initially True. If any of the remote filesystem methods raises a RemoteConnectionError, 'connected' will switch to False and remain so until a successful remote method call.

Application code can use the method 'wait_for_connection' to block until the connection is re-established. Currently this reconnection is checked by a simple polling loop; eventually more sophisticated operating-system integration may be added.

Since some remote FS classes can raise RemoteConnectionError during initialization, this class makes use of lazy initialization. The remote FS can be specified as an FS instance, an FS subclass, or a (class,args) or (class,args,kwds) tuple. For example:

```
>>> fs = ConnectionManagerFS(MyRemoteFS("http://www.example.com/"))
Traceback (most recent call last):
    ...
RemoteConnectionError: couldn't connect to "http://www.example.com/"
>>> fs = ConnectionManagerFS((MyRemoteFS,["http://www.example.com/"]))
>>> fs.connected
False
>>>
```

class fs.remote.**RemoteFileBuffer**(*fs*, *path*, *mode*, *rfile=None*, *write_on_flush=True*)

File-like object providing buffer for local file operations.

Instances of this class manage a local tempfile buffer corresponding to the contents of a remote file. All reads and writes happen locally, with the content being copied to the remote file only on flush() or close(). Writes to the remote file are performed using the setcontents() method on the owning FS object.

The intended use-case is for a remote filesystem (e.g. S3FS) to return instances of this class from its open() method, and to provide the file-uploading logic in its setcontents() method, as in the following pseudo-code:

```
def open(self,path,mode="r"):
    rf = self._get_remote_file(path)
    return RemoteFileBuffer(self,path,mode,rf)

def setcontents(self,path,file):
    self._put_remote_file(path,file)
```

The contents of the remote file are read into the buffer on-demand.

RemoteFileBuffer constructor.

The owning filesystem, path and mode must be provided. If the optional argument 'rfile' is provided, it must be a read()-able object or a string containing the initial file contents.

fs.rpcfs

This module provides the class 'RPCFS' to access a remote FS object over XML-RPC. You probably want to use this in conjunction with the 'RPCFSServer' class from the *xmlrpc* module.

class fs.rpcfs.**RPCFS**(*uri*, *transport=None*)

Access a filesystem exposed via XML-RPC.

This class provides the client-side logic for accessing a remote FS object, and is dual to the RPCFSServer class defined in fs.expose.xmlrpc.

Example:

```
fs = RPCFS("http://my.server.com/filesystem/location/")
```

Constructor for RPCFS objects.

The only required argument is the URI of the server to connect to. This will be passed to the underlying XML-RPC server proxy object, along with the 'transport' argument if it is provided.

Parameters **uri** – address of the server

decode_path(*path*)
Decode paths arriving over the wire.

encode_path(*path*)
Encode a filesystem path for sending over the wire.

Unfortunately XMLRPC only supports ASCII strings, so this method must return something that can be represented in ASCII. The default is base64-encoded UTF8.

class fs.rpcfs.**ReRaiseFaults**(*obj*)
XML-RPC proxy wrapper that re-raises Faults as proper Exceptions.

fs.rpcfs.**re_raise_faults**(*func*)
Decorator to re-raise XML-RPC faults as proper exceptions.

fs.s3fs

Currently only avaiable on Python2 due to boto not being available for Python3

FS subclass accessing files in Amazon S3

This module provides the class 'S3FS', which implements the FS filesystem interface for objects stored in Amazon Simple Storage Service (S3).

class fs.s3fs.**S3FS**(*bucket*, *prefix=''*, *aws_access_key=None*, *aws_secret_key=None*, *separator='/'*, *thread_synchronize=True*, *key_sync_timeout=1*, ***conn_kwargs*)
A filesystem stored in Amazon S3.

This class provides the FS interface for files stored in Amazon's Simple Storage Service (S3). It should be instantiated with the name of the S3 bucket to use, and optionally a prefix under which the files should be stored.

Local temporary files are used when opening files from this filesystem, and any changes are only pushed back into S3 when the files are closed or flushed.

Constructor for S3FS objects.

S3FS objects require the name of the S3 bucket in which to store files, and can optionally be given a prefix under which the files should be stored. AWS connection arguments (e.g. aws_access_key_id, aws_secret_access_key, etc) may be specified as additional keyword arguments to be passed to boto.s3.connection.S3Connection; if they are not specified boto will try to read them from various locations (see http://boto.cloudhackers.com/en/latest/boto_config_tut.html). Note that legacy arguments 'aws_access_key' and 'aws_secret_key' will be passed to boto as 'aws_access_key_id' and 'aws_secret_access_key'.

The keyword argument 'key_sync_timeout' specifies the maximum time in seconds that the filesystem will spend trying to confirm that a newly-uploaded S3 key is available for reading. For no timeout set it to zero. To disable these checks entirely (and thus reduce the filesystem's consistency guarantees to those of S3's "eventual consistency" model) set it to None.

By default the path separator is "/", but this can be overridden by specifying the keyword 'separator' in the constructor.

copy(*src*, *dst*, *overwrite=False*, *chunk_size=16384*)
Copy a file from 'src' to 'dst'.

src – The source path dst – The destination path overwrite – If True, then the destination may be overwritten (if a file exists at that location). If False then an exception will be thrown if the destination exists chunk_size – Size of chunks to use in copy (ignored by S3)

exists (*path*)
Check whether a path exists.

getpathurl (*path*, *allow_none=False*, *expires=3600*)
Returns a url that corresponds to the given path.

ilistdir (*path='./'*, *wildcard=None*, *full=False*, *absolute=False*, *dirs_only=False*, *files_only=False*)
List contents of a directory.

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

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

listdir (*path='./'*, *wildcard=None*, *full=False*, *absolute=False*, *dirs_only=False*, *files_only=False*)
List contents of a directory.

makedir (*path*, *recursive=False*, *allow_recreate=False*)
Create a directory at the given path.

The 'mode' argument is accepted for compatibility with the standard FS interface, but is currently ignored.

makepublic (*path*)
Mark given path as publicly accessible using HTTP(S)

move (*src*, *dst*, *overwrite=False*, *chunk_size=16384*)
Move a file from one location to another.

open (*path*, *mode=u'rt'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, ***kwargs*)
Open the named file in the given mode.

This method downloads the file contents into a local temporary file so that it can be worked on efficiently. Any changes made to the file are only sent back to S3 when the file is flushed or closed.

remove (*path*)
Remove the file at the given path.

removedir (*path*, *recursive=False*, *force=False*)
Remove the directory at the given path.

rename (*src*, *dst*)
Rename the file at 'src' to 'dst'.

fs.sftpfs

Currently only avaiable on Python2 due to paramiko not being available for Python3

Filesystem accessing an SFTP server (via paramiko)

class fs.sftpfs.**SFTPFS** (*connection*, *root_path='/'*, *encoding=None*, *hostkey=None*, *username=''*, *password=None*, *pkey=None*, *agent_auth=True*, *no_auth=False*, *look_for_keys=True*)
A filesystem stored on a remote SFTP server.

This is basically a compatibility wrapper for the excellent SFTPClient class in the paramiko module.

SFTPFS constructor.

The only required argument is 'connection', which must be something from which we can construct a paramiko.SFTPClient object. Possible values include:

- a hostname string
- a (hostname,port) tuple
- a paramiko.Transport instance
- a paramiko.Channel instance in "sftp" mode

The keyword argument 'root_path' specifies the root directory on the remote machine - access to files outside this root will be prevented.

Parameters

- **connection** – a connection string
- **root_path** – The root path to open
- **encoding** – String encoding of paths (defaults to UTF-8)
- **hostkey** – the host key expected from the server or None if you don't require server validation
- **username** – Name of SFTP user
- **password** – Password for SFTP user
- **pkey** – Public key
- **agent_auth** – attempt to authorize with the user's public keys
- **no_auth** – attempt to log in without any kind of authorization
- **look_for_keys** – Look for keys in the same locations as ssh, if other authentication is not succesful

close(**args*, ***kwargs*)
Close the connection to the remote server.

fs.tempfs

Make a temporary file system that exists in a folder provided by the OS. All files contained in a TempFS are removed when the *close* method is called (or when the TempFS is cleaned up by Python).

class fs.tempfs.**TempFS**(*identifier=None*, *temp_dir=None*, *dir_mode=448*, *thread_synchronize=True*)
Create a Filesystem in a temporary directory (with tempfile.mkdtemp), and removes it when the TempFS object is cleaned up.

Creates a temporary Filesystem

identifier – A string that is included in the name of the temporary directory, default uses "TempFS"

close(**args*, ***kwargs*)
Removes the temporary directory.

This will be called automatically when the object is cleaned up by Python, although it is advisable to call it manually. Note that once this method has been called, the FS object may no longer be used.

fs.utils

The *utils* module provides a number of utility functions that don't belong in the Filesystem interface. Generally the functions in this module work with multiple Filesystems, for instance moving and copying between non-similar Filesystems.

fs.utils.**copyfile**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *overwrite=True*, *chunk_size=65536*)
Copy a file from one filesystem to another. Will use system copyfile, if both files have a syspath. Otherwise file will be copied a chunk at a time.

Parameters

- **src_fs** – Source filesystem object
- **src_path** – Source path
- **dst_fs** – Destination filesystem object
- **dst_path** – Destination path
- **chunk_size** – Size of chunks to move if system copyfile is not available (default 64K)

fs.utils.**movefile**(*src_fs*, *src_path*, *dst_fs*, *dst_path*, *overwrite=True*, *chunk_size=65536*)
Move a file from one filesystem to another. Will use system copyfile, if both files have a syspath. Otherwise file will be copied a chunk at a time.

Parameters

- **src_fs** – Source filesystem object
- **src_path** – Source path
- **dst_fs** – Destination filesystem object
- **dst_path** – Destination path
- **chunk_size** – Size of chunks to move if system copyfile is not available (default 64K)

fs.utils.**movedir**(*fs1*, *fs2*, *create_destination=True*, *ignore_errors=False*, *chunk_size=65536*)
Moves contents of a directory from one filesystem to another.

Parameters

- **fs1** – A tuple of (<filesystem>, <directory path>)
- **fs2** – Destination filesystem, or a tuple of (<filesystem>, <directory path>)
- **create_destination** – If True, the destination will be created if it doesn't exist
- **ignore_errors** – If True, exceptions from file moves are ignored
- **chunk_size** – Size of chunks to move if a simple copy is used

fs.utils.**copydir**(*fs1*, *fs2*, *create_destination=True*, *ignore_errors=False*, *chunk_size=65536*)
Copies contents of a directory from one filesystem to another.

Parameters

- **fs1** – Source filesystem, or a tuple of (<filesystem>, <directory path>)
- **fs2** – Destination filesystem, or a tuple of (<filesystem>, <directory path>)
- **create_destination** – If True, the destination will be created if it doesn't exist
- **ignore_errors** – If True, exceptions from file moves are ignored
- **chunk_size** – Size of chunks to move if a simple copy is used

fs.utils.**countbytes**(*fs*)
Returns the total number of bytes contained within files in a filesystem.

Parameters **fs** – A filesystem object

fs.utils.**isfile**(*fs*, *path*, *info=None*)
Check whether a path within a filesystem is a file.

If you're able to provide the info dict for the path, this may be possible without querying the filesystem (e.g. by checking st_mode).

fs.utils.**isdir**(*fs*, *path*, *info=None*)
Check whether a path within a filesystem is a directory.

If you're able to provide the info dict for the path, this may be possible without querying the filesystem (e.g. by checking st_mode).

fs.utils.**find_duplicates**(*fs*, *compare_paths=None*, *quick=False*, *signature_chunk_size=16384*, *signature_size=163840*)
A generator that yields the paths of duplicate files in an FS object. Files are considered identical if the contents are the same (dates or other attributes not take in to account).

Parameters

- **fs** – A filesystem object
- **compare_paths** – An iterable of paths within the FS object, or all files if omitted
- **quick** – If set to True, the quick method of finding duplicates will be used, which can potentially return false positives if the files have the same size and start with the same data. Do not use when deleting files!
- **signature_chunk_size** – The number of bytes to read before generating a signature checksum value
- **signature_size** – The total number of bytes read to generate a signature

For example, the following will list all the duplicate .jpg files in "~/Pictures":

```
>>> from fs.utils import find_duplicates
>>> from fs.osfs import OSFS
>>> fs = OSFS('~/Pictures')
>>> for dups in find_duplicates(fs, fs.walkfiles('*.jpg')):
...     print list(dups)
```

fs.utils.**print_fs**(*fs*, *path='/'*, *max_levels=5*, *file_out=None*, *terminal_colors=None*, *hide_dotfiles=False*, *dirs_first=False*, *files_wildcard=None*, *dirs_only=False*)
Prints a filesystem listing to stdout (including sub directories).

This mostly useful as a debugging aid. Be careful about printing a OSFS, or any other large filesystem. Without max_levels set, this function will traverse the entire directory tree.

For example, the following will print a tree of the files under the current working directory:

```
>>> from fs.osfs import *
>>> from fs.utils import *
>>> fs = OSFS('.')
>>> print_fs(fs)
```

Parameters

- **fs** – A filesystem object

- **path** – Path of a directory to list (default "/")
- **max_levels** – Maximum levels of dirs to list (default 5), set to None for no maximum
- **file_out** – File object to write output to (defaults to sys.stdout)
- **terminal_colors** – If True, terminal color codes will be written, set to False for non-console output. The default (None) will select an appropriate setting for the platform.
- **hide_dotfiles** – if True, files or directories beginning with '.' will be removed

fs.utils.**open_atomic_write**(*fs*, *path*, *mode='w'*)
Open a file for 'atomic' writing

This returns a context manager which ensures that a file is written in its entirety or not at all.

fs.watch

Change notification support for FS.

This module defines a standard interface for FS subclasses that support change notification callbacks. It also offers some WrapFS subclasses that can simulate such an ability on top of an ordinary FS object.

An FS object that wants to be "watchable" must provide the following methods:

- `add_watcher(callback,path="/",events=None,recursive=True)`

  Request that the given callback be executed in response to changes to the given path. A specific set of change events can be specified. This method returns a Watcher object.

- `del_watcher(watcher_or_callback)`

  Remove the given watcher object, or any watchers associated with the given callback.

If you would prefer to read changes from a filesystem in a blocking fashion rather than using callbacks, you can use the function 'iter_changes' to obtain an iterator over the change events.

class fs.watch.**ACCESSED**(*fs*, *path*)
Event fired when a file's contents are accessed.

class fs.watch.**CLOSED**(*fs*, *path*)
Event fired when the filesystem is closed.

class fs.watch.**CREATED**(*fs*, *path*)
Event fired when a new file or directory is created.

class fs.watch.**ERROR**(*fs*, *path*)
Event fired when some miscellaneous error occurs.

class fs.watch.**EVENT**(*fs*, *path*)
Base class for change notification events.

class fs.watch.**MODIFIED**(*fs*, *path*, *data_changed=False*, *closed=False*)
Event fired when a file or directory is modified.

class fs.watch.**MOVED_DST**(*fs*, *path*, *source=None*)
Event fired when a file or directory is the target of a move.

class fs.watch.**MOVED_SRC**(*fs*, *path*, *destination=None*)
Event fired when a file or directory is the source of a move.

class fs.watch.**OVERFLOW**(*fs*, *path*)
Event fired when some events could not be processed.

class fs.watch.**PollingWatchableFS**(*wrapped_fs*, *poll_interval=300*)
FS wrapper simulating watcher callbacks by periodic polling.

This FS wrapper augments the functionality of WatchableFS by periodically polling the underlying FS for changes. It is thus capable of detecting changes made to the underlying FS via other interfaces, albeit with a (configurable) delay to account for the polling interval.

class fs.watch.**REMOVED**(*fs*, *path*)
Event fired when a file or directory is removed.

class fs.watch.**WatchableFS**(**args*, ***kwds*)
FS wrapper simulating watcher callbacks.

This FS wrapper intercepts method calls that modify the underlying FS and generates appropriate notification events. It thus allows watchers to monitor changes made through the underlying FS object, but not changes that might be made through other interfaces to the same filesystem.

class fs.watch.**WatchableFSMixin**(**args*, ***kwds*)
Mixin class providing watcher management functions.

add_watcher(*callback*, *path='/'*, *events=None*, *recursive=True*)
Add a watcher callback to the FS.

del_watcher(*watcher_or_callback*)
Delete a watcher callback from the FS.

notify_watchers(*event_or_class*, *path=None*, **args*, ***kwds*)
Notify watchers of the given event data.

class fs.watch.**WatchedFile**(*file*, *fs*, *path*, *mode=None*)
File wrapper for use with WatchableFS.

This file wrapper provides access to a file opened from a WatchableFS instance, and fires MODIFIED events when the file is modified.

class fs.watch.**Watcher**(*fs*, *callback*, *path='/'*, *events=None*, *recursive=True*)
Object encapsulating filesystem watch info.

fs.watch.**ensure_watchable**(*fs*, *wrapper_class=<class 'fs.watch.PollingWatchableFS'>*, **args*, ***kwds*)
Ensure that the given fs supports watching, simulating it if necessary.

Given an FS object, this function returns an equivalent FS that has support for watcher callbacks. This may be the original object if it supports them natively, or a wrapper class if they must be simulated.

class fs.watch.**iter_changes**(*fs=None*, *path='/'*, *events=None*, ***kwds*)
Blocking iterator over the change events produced by an FS.

This class can be used to transform the callback-based watcher mechanism into a blocking stream of events. It operates by having the callbacks push events onto a queue as they come in, then reading them off one at a time.

fs.wrapfs

The fs.wrapfs module adds additional functionality to existing FS objects.

fs.wrapfs

A class for wrapping an existing FS object with additional functionality.

This module provides the class WrapFS, a base class for objects that wrap another FS object and provide some transformation of its contents. It could be very useful for implementing e.g. transparent encryption or compression services.

For a simple example of how this class could be used, see the 'HideDotFilesFS' class in the module fs.wrapfs.hidedotfilesfs. This wrapper implements the standard unix shell functionality of hiding dot-files in directory listings.

class `fs.wrapfs.`**`WrapFS`**(*fs*)

FS that wraps another FS, providing translation etc.

This class allows simple transforms to be applied to the names and/or contents of files in an FS. It could be used to implement e.g. compression or encryption in a relatively painless manner.

The following methods can be overridden to control how files are accessed in the underlying FS object:

- **_file_wrap(file, mode): called for each file that is opened from** the underlying FS; may return a modified file-like object.
- _encode(path): encode a path for access in the underlying FS
- _decode(path): decode a path from the underlying FS

If the required path translation proceeds one component at a time, it may be simpler to override the _encode_name() and _decode_name() methods.

`fs.wrapfs.`**`rewrite_errors`**(*func*)

Re-write paths in errors raised by wrapped FS objects.

`fs.wrapfs.`**`wrap_fs_methods`**(*decorator*, *cls=None*, *exclude=[]*)

Apply the given decorator to all FS methods on the given class.

This function can be used in two ways. When called with two arguments it applies the given function 'decorator' to each FS method of the given class. When called with just a single argument, it creates and returns a class decorator which will do the same thing when applied. So you can use it like this:

```
wrap_fs_methods(mydecorator,MyFSClass)
```

Or on more recent Python versions, like this:

```
@wrap_fs_methods(mydecorator)
class MyFSClass(FS):
    ...
```

fs.wrapfs.hidedotfilesfs

An FS wrapper class for hiding dot-files in directory listings.

class `fs.wrapfs.hidedotfilesfs.`**`HideDotFilesFS`**(*fs*)

FS wrapper class that hides dot-files in directory listings.

The listdir() function takes an extra keyword argument 'hidden' indicating whether hidden dot-files should be included in the output. It is False by default.

`is_hidden`(*path*)

Check whether the given path should be hidden.

fs.wrapfs.lazyfs

A class for lazy initialization of an FS object.

This module provides the class LazyFS, an FS wrapper class that can lazily initialize its underlying FS object.

class fs.wrapfs.lazyfs.**LazyFS**(*fs*)
Simple 'lazy initialization' for FS objects.

This FS wrapper can be created with an FS instance, an FS class, or a (class,args,kwds) tuple. The actual FS instance will be created on demand the first time it is accessed.

wrapped_fs
Obtain the wrapped FS instance, creating it if necessary.

fs.wrapfs.limitsizefs

An FS wrapper class for limiting the size of the underlying FS.

This module provides the class LimitSizeFS, an FS wrapper that can limit the total size of files stored in the wrapped FS.

class fs.wrapfs.limitsizefs.**LimitSizeFS**(*fs*, *max_size*)
FS wrapper class to limit total size of files stored.

class fs.wrapfs.limitsizefs.**LimitSizeFile**(*file*, *mode*, *size*, *fs*, *path*)
Filelike wrapper class for use by LimitSizeFS.

fs.wrapfs.readonlyfs

An FS wrapper class for blocking operations that would modify the FS.

class fs.wrapfs.readonlyfs.**ReadOnlyFS**(*fs*)
Makes a FS object read only. Any operation that could potentially modify the underlying file system will throw an UnsupportedError

Note that this isn't a secure sandbox, untrusted code could work around the read-only restrictions by getting the base class. Its main purpose is to provide a degree of safety if you want to protect an FS object from accidental modification.

copy(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

copydir(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

createfile(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

delxattr(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

getsyspath(*path*, *allow_none=False*)
Doesn't technically modify the filesystem but could be used to work around read-only restrictions.

makedir(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

move(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

movedir(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

open(*path*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, ***kwargs*)
Only permit read access

remove(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

removedir(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

rename(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

setcontents(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

settimes(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

setxattr(**args*, ***kwargs*)
Replacement method for methods that can modify the file system

fs.zipfs

A FS object that represents the contents of a Zip file

class fs.zipfs.**ZipFS**(*zip_file*, *mode='r'*, *compression='deflated'*, *allow_zip_64=False*, *encoding='CP437'*, *thread_synchronize=True*)
A FileSystem that represents a zip file.

Create a FS that maps on to a zip file.

Parameters

- **zip_file** – a (system) path, or a file-like object
- **mode** – mode to open zip file, 'r' for reading, 'w' for writing or 'a' for appending
- **compression** – can be 'deflated' (default) to compress data or 'stored' to just store date
- **allow_zip_64** – set to True to use zip files greater than 2 GB, default is False
- **encoding** – the encoding to use for unicode filenames
- **thread_synchronize** – set to True (default) to enable thread-safety

Raises

- **fs.errors.ZipOpenError** – thrown if the zip file could not be opened
- **fs.errors.ZipNotFoundError** – thrown if the zip file does not exist (derived from ZipOpenError)

close()
Finalizes the zip file so that it can be read. No further operations will work after this method is called.

exception fs.zipfs.**ZipNotFoundError**(*msg=None*, *details=None*)
Thrown when the requested zip file does not exist

exception fs.zipfs.**ZipOpenError**(*msg=None*, *details=None*)
Thrown when the zip file could not be opened

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

G

H

I

S

T

U

V

W

X

Z