fsspec Documentation

Release 2021.10.1+0.g1fafe90.dirty

Joseph Crail

Oct 15, 2021

CONTENTS:

Filesystem Spec (`fsspec`) is a project to provide a unified pythonic interface to local, remote and embedded file systems and bytes storage.

CHAPTER

ONE

BRIEF OVERVIEW

There are many places to store bytes, from in memory, to the local disk, cluster distributed storage, to the cloud. Many files also contain internal mappings of names to bytes, maybe in a hierarchical directory-oriented tree. Working with all these different storage media, and their associated libraries, is a pain. `fsspec` exists to provide a familiar API that will work the same whatever the storage backend. As much as possible, we iron out the quirks specific to each implementation, so you need do no more than provide credentials for each service you access (if needed) and thereafter not have to worry about the implementation again.

CHAPTER

TWO

WHY

`fsspec` provides two main concepts: a set of filesystem classes with uniform APIs (i.e., functions such as `cp`, `rm`, `cat`, `mkdir`, ...) supplying operations on a range of storage systems; and top-level convenience functions like *`fsspec.open()`*, to allow you to quickly get from a URL to a file-like object that you can use with a third-party library or your own code.

The section *Background* gives motivation and history of this project, but most users will want to skip straight to *Usage* to find out how to use the package and *Features of fsspec* to see the long list of added functionality included along with the basic file-system interface.

CHAPTER

THREE

WHO USES FSSPEC?

You can use `fsspec`'s file objects with any python function that accepts file objects, because of *duck typing*.

You may well be using `fsspec` already without knowing it. The following libraries use `fsspec` internally for path and file handling:

1. Dask, the parallel, out-of-core and distributed programming platform
2. Intake, the data source cataloguing and loading library and its plugins
3. pandas, the tabular data analysis package
4. xarray and zarr, multidimensional array storage and labelled operations
5. DVC, version control system for machine learning projects
6. Kedro, a Python framework for reproducible, maintainable and modular data science code

`fsspec` filesystems are also supported by:

1. pyarrow, the in-memory data layout engine

… plus many more that we don't know about.

CHAPTER

FOUR

INSTALLATION

fsspec can be installed from PyPI or conda and has no dependencies of its own

```
pip install fsspec
conda install -c conda-forge fsspec
```

Not all filesystem implementations are available without installing extra dependencies. For example to be able to access data in S3, you can use the optional pip install syntax below, or install the specific package required

```
pip install fsspec[gcs]
conda install -c conda-forge gcsfs
```

fsspec attempts to provide the right message when you attempt to use a filesystem for which you need additional dependencies. The current list of known implementations can be found as follows

```
from fsspec.registry import known_implementations

known_implementations
```

4.1 Background

Python provides a standard interface for open files, so that alternate implementations of file-like object can work seamlessly with many function which rely only on the methods of that standard interface. A number of libraries have implemented a similar concept for file-systems, where file operations can be performed on a logical file-system which may be local, structured data store or some remote service.

This repository is intended to be a place to define a standard interface that such file-systems should adhere to, such that code using them should not have to know the details of the implementation in order to operate on any of a number of backends. With hope, the community can come together to define an interface that is the best for the highest number of users, and having the specification, makes developing other file-system implementations simpler.

4.1.1 History

We have been involved in building a number of remote-data file-system implementations, principally in the context of the Dask project. In particular, several are listed in docs with links to the specific repositories. With common authorship, there is much that is similar between the implementations, for example posix-like naming of the operations, and this has allowed Dask to be able to interact with the various backends and parse generic URLs in order to select amongst them. However, *some* extra code was required in each case to adapt the peculiarities of each implementation with the generic usage that Dask demanded. People may find the code which parses URLs and creates file-system instances interesting.

At the same time, the Apache Arrow project was also concerned with a similar problem, particularly a common interface to local and HDFS files, for example the hdfs interface (which actually communicated with HDFS with a choice of driver). These are mostly used internally within Arrow, but Dask was modified in order to be able to use the alternate HDFS interface (which solves some security issues with `hdfs3`). In the process, a conversation was started, and I invite all interested parties to continue the conversation in this location.

There is a good argument that this type of code has no place in Dask, which is concerned with making graphs representing computations, and executing those graphs on a scheduler. Indeed, the file-systems are generally useful, and each has a user-base wider than just those that work via Dask.

4.1.2 Influences

The following places to consider, when choosing the definitions of how we would like the file-system specification to look:

1. python's os module and its `path` namespace; also other file-connected functionality in the standard library
2. posix/bash method naming conventions that linux/unix/osx users are familiar with; or perhaps their Windows variants
3. the existing implementations for the various backends (e.g., gcsfs or Arrow's hdfs)
4. pyfilesystems, an attempt to do something similar, with a plugin architecture. This conception has several types of local file-system, and a lot of well-thought-out validation code.

4.1.3 Other similar work

It might have been conceivable to reuse code in `pyfilesystems`, which has an established interface and several implementations of its own. However, it supports none of the highlight, critical to cloud and parallel access, and would not be easy to coerce. Following on the success of `s3fs` and `gcsfs`, and their use within Dask, it seemed best to have an interface as close to those as possible. See a discussion on the topic.

Other newer technologies such as smart_open and `pyarrow`'s newer file-system rewrite also have some parts of the functionality presented here, that might suit some use cases better.

4.1.4 Structure of the package

The best place to get a feel for the contents of `fsspec` is by looking through the *Usage* and *API Reference* sections. In addition, the source code will be interesting for those who wish to subclass and develop new file-system implementations. `fsspec/spec.py` contains the main abstract file-system class to derive from, `AbstractFileSystem`.

4.2 Usage

This is quick-start documentation to help people get familiar with the layout and functioning of `fsspec`.

4.2.1 Instantiate a file-system

`fsspec` provides an abstract file-system interface as a base class, to be used by other filesystems. A file-system instance is an object for manipulating files on some remote store, local files, files within some wrapper, or anything else that is capable of producing file-like objects.

Some concrete implementations are bundled with `fsspec` and others can be installed separately. They can be instantiated directly, or the `registry` can be used to find them.

Direct instantiation:

```
from fsspec.implementations.local import LocalFileSystem

fs = LocalFileSystem()
```

Look-up via registry:

```
import fsspec

fs = fsspec.filesystem('file')
```

Many filesystems also take extra parameters, some of which may be options - see *API Reference*, or use *`fsspec.get_filesystem_class()`* to get the class object and inspect its docstring.

```
import fsspec

fs = fsspec.filesystem('ftp', host=host, port=port, username=user, password=pw)
```

4.2.2 Use a file-system

File-system instances offer a large number of methods for getting information about and manipulating files for the given back-end. Although some specific implementations may not offer all features (e.g., `http` is read-only), generally all normal operations, such as `ls`, `rm`, should be expected to work (see the full list: *`fsspec.spec.AbstractFileSystem`*). Note that this quick-start will prefer posix-style naming, but many common operations are aliased: `cp()` and `copy()` are identical, for instance. Functionality is generally chosen to be as close to the builtin `os` module's working for things like `glob` as possible. The following block of operations should seem very familiar.

```
fs.mkdir("/remote/output")
fs.touch("/remote/output/success")  # creates empty file
assert fs.exists("/remote/output/success")
assert fs.isfile("/remote/output/success")
assert fs.cat("/remote/output/success") == b""  # get content as bytestring
fs.copy("/remote/output/success", "/remote/output/copy")
assert fs.ls("/remote/output", detail=False) == ["/remote/output/success", "/remote/
↪output/copy"]
fs.rm("/remote/output", recursive=True)
```

The `open()` method will return a file-like object which can be passed to any other library that expects to work with python files, or used by your own code as you would a normal python file object. These will normally be binary-mode

only, but may implement internal buffering in order to limit the number of reads from a remote source. They respect the use of `with` contexts. If you have `pandas` installed, for example, you can do the following:

```
f = fs.open("/remote/path/notes.txt", "rb")
lines = f.readline()  # read to first b"\n"
f.seek(-10, 2)
foot = f.read()  # read last 10 bytes of file
f.close()

import pandas as pd
with fs.open('/remote/data/myfile.csv') as f:
    df = pd.read_csv(f, sep='|', header=None)
```

4.2.3 Higher-level

For many situations, the only function that will be needed is *fsspec.open_files()*, which will return *fsspec.core.OpenFile* instances created from a single URL and parameters to pass to the backend(s). This supports text-mode and compression on the fly, and the objects can be serialized for passing between processes or machines (so long as each has access to the same backend file-system). The protocol (i.e., backend) is inferred from the URL passed, and glob characters are expanded in read mode (search for files) or write mode (create names). Critically, the file on the backend system is not actually opened until the `OpenFile` instance is used in a `with` context.

```
of = fsspec.open("github://dask:fastparquet@main/test-data/nation.csv", "rt")
# of is an OpenFile container object. The "with" context below actually opens it
with of as f:
    # now f is a text-mode file
    for line in f:
        # iterate text lines
        print(line)
        if "KENYA" in line:
            break
```

4.3 Features of fsspec

Here follows a brief description of some features of note of `fsspec` that provides to make it an interesting project beyond some other file-system abstractions.

4.3.1 Serialisability

Coming out of the Dask stable, it was an important design decision that file-system instances be serialisable, so that they could be created in one process (e.g., the client) and used in other processes (typically the workers). These other processes may even be on other machines, so in many cases they would need to be able to re-establish credentials, ideally without passing sensitive tokens in the pickled binary data.

`fsspec` instances, generally speaking, abide by these rules, do not include locks, files and other thread-local material, and where possible, use local credentials (such as a token file) for re-establishing sessions upon de-serialisation. (While making use of cached instances, where they exist, see below).

4.3.2 `OpenFile` instances

The *fsspec.core.OpenFile()* class provides a convenient way to prescribe the manner to open some file (local, remote, in a compressed store, etc.) which is portable, and can also apply any compression and text-mode to the file. These instances are also serialisable, because they do not contain any open files.

The way to work with `OpenFile` s is to isolate interaction with in a `with` context. It is the initiation of the context which actually does the work of creating file-like instances.

```
of = fsspec.open(url, ...)
# of is just a place-holder
with of as f:
    # f is now a real file-like object holding resources
    f.read(...)
```

4.3.3 File Buffering and random access

Most implementations create file objects which derive from `fsspec.spec.AbstractBufferedFile`, and have many behaviours in common. A subclass of `AbstractBufferedFile` provides random access for the underlying file-like data (without downloading the whole thing). This is a critical feature in the big-data access model, where each sub-task of an operation may need on a small part of a file, and does not, therefore want to be forced into downloading the whole thing.

These files offer buffering of both read and write operations, so that communication with the remote resource is limited. The size of the buffer is generally configured with the `blocksize=` kwarg at open time, although the implementation may have some minimum or maximum sizes that need to be respected.

For reading, a number of buffering schemes are available, listed in `fsspec.caching.caches` (see *Read Buffering*), or "none" for no buffering at all, e.g., for a simple read-ahead buffer, you can do

```
fs = fsspec.filesystem(...)
with fs.open(path, mode='rb', cache_type='readahead') as f:
    use_for_something(f)
```

4.3.4 Transparent text-mode and compression

As mentioned above, the `OpenFile` class allows for the opening of files on a binary store, which appear to be in text mode and/or allow for a compression/decompression layer between the caller and the back-end storage system. From the user's point of view, this is achieved simply by passing arguments to the *fsspec.open_files()* or *fsspec.open()* functions, and thereafter happens transparently.

4.3.5 Key-value stores

File-systems are naturally like dict-like key-value mappings: each (string) path corresponds to some binary data on the storage back-end. For some use-cases, it is very convenient to be able to view some path within the file-system as a dict-like store, and the function *fsspec.get_mapper()* gives a one-stop way to return such an object. This has become useful, for example, in the context of the zarr project, which stores it array chunks in keys in any arbitrary mapping-like object.

```
mapper = fsspec.get_mapper('protocol://server/path', args)
list(mapper)
mapper[k] = b'some data'
```

4.3.6 PyArrow integration

pyarrow has its own internal idea of what a file-system is (`pyarrow.fs.FileSystem`), and some functions, particularly the loading of parquet, require that the target be compatible. As it happens, the design of the file-system interface in `pyarrow` *is* compatible with `fsspec` (this is not by accident).

At import time, `fsspec` checks for the existence of `pyarrow`, and, if `pyarrow < 2.0` is found, adds its base filesystem to the superclasses of the spec base-class. For `pyarrow >= 2.0`, `fsspec` file systems can simply be passed to `pyarrow` functions that expect `pyarrow` filesystems, and `pyarrow` will automatically wrap them.

In this manner, all `fsspec`-derived file-systems are also `pyarrow` file-systems, and can be used by `pyarrow` functions.

4.3.7 Transactions

`fsspec` supports *transactions*, during which writing to files on a remote store are deferred (typically put into a temporary location) until the transaction is over, whereupon the whole transaction is finalised in a semi-atomic way, and all the files are moved/committed to their final destination. The implementation of the details is file-system specific (and not all support it yet), but the idea is, that all files should get written or none, to mitigate against data corruption. The feature can be used like

```
fs = fsspec.filesystem(...)
with fs.transaction:
    with fs.open('file1', 'wb') as f:
        f.write(b'some data')
    with fs.open('file2', 'wb') as f:
        f.write(b'more data')
```

Here, files 1 and 2 do not get moved to the target location until the transaction context finishes. If the context finishes due to an (uncaught) exception, then the files are discarded and the file target locations untouched.

The class *`fsspec.spec.Transaction()`* allows for fine-tuning of the operation, and every `fsspec` instance has an instance of this as an attribute `.transaction` to give access.

Note that synchronising transactions across multiple instances, perhaps across a cluster, is a harder problem to solve, and the implementation described here is only part of the solution.

4.3.8 Mount anything with FUSE

Any path of any file-system can be mapped to a local directory using fusepy and *`fsspec.fuse.run()`*. This feature is experimental, but basic file listing with details, and read/write should generally be available to the extent that the remote file-system provides enough information. Naturally, if a file-system is read-only, then write operations will fail - but they will tend to fail late and with obscure error messages such as "bad address".

Some specific quirks of some file-systems may cause confusion for FUSE. For example, it is possible for a given path on s3 to be both a valid key (i.e., containing binary data, like a file) and a valid prefix (i.e., can be listed to find subkeys, like a directory). Since this breaks the assumptions of a normal file-system, it may not be possible to reach all paths on the remote.

4.3.9 Instance Caching

In a file-system implementation class is marked as *cachable* (attribute `.cachable`), then its instances will get stored in a class attribute, to enable quick look-up instead of needing to regenerate potentially expensive connections and sessions. They key in the cache is a tokenisation of the arguments to create the instance. The cache itself (attribute `._cache`) is currently a simple dict, but could in the future be LRU, or something more complicated, to fine-tune instance lifetimes.

Since files can hold on to write caches and read buffers, the instance cache may cause excessive memory usage in some situations; but normally, files' `close` methods will be called, discarding the data. Only when there is also an unfinalised transaction or captured traceback might this be anticipated becoming a problem.

To disable instance caching, i.e., get a fresh instance which is not in the cache even for a cachable class, pass `skip_instance_cache=True`.

4.3.10 Listings Caching

For some implementations, getting file listings (i.e., `ls` and anything that depends on it) is expensive. These implementations use dict-like instances of *fsspec.dircache.DirCache* to manage the listings.

The cache allows for time-based expiry of entries with the `listings_expiry_time` parameter, or LRU expiry with the `max_paths` parameter. These can be set on any implementation instance that uses listings caching; or to skip the caching altogether, use `use_listings_cache=False`. That would be appropriate when the target location is known to be volatile because it is being written to from other sources.

When the `fsspec` instance writes to the backend, the method `invalidate_cache` is called, so that subsequent listing of the given paths will force a refresh. In addition, some methods like `ls` have a `refresh` parameter to force fetching the listing again.

4.3.11 URL chaining

Some implementations proxy or otherwise make use of another filesystem implementation, such as locally caching remote files, i.e., finding out what files exist using the remote implementation, but actually opening the local copies upon access. Other examples include reading from a Dask worker which can see file-systems the client cannot, and accessing a zip file which is being read from another backend.

In such cases, you can specify the parameters exactly as specified in the implementation docstrings, for the dask case something like

```
of = fsspec.open('dask://bucket/key', target_protocol='s3', target_options={'anon': True}
↪)
```

As a shorthand, particularly useful where you have multiple hops, is to "chain" the URLs with the special separator `"::"`. The arguments to be passed on to each of the implementations referenced are keyed by the protocol names included in the URL. Here is the equivalent to the line above:

```
of = fsspec.open('dask::s3://bucket/key', s3={'anon': True})
```

A couple of more complicates cases:

```
of = fsspec.open_files('zip://*.csv::simplecache::gcs://bucket/afile.zip',
                       simplecache={'cache_storage': '/stored/zip/files'},
                       gcs={'project': 'my-project'})
```

reads a zip-file from google, stores it locally, and gives access to the contained CSV files. Conversely,

```
of = fsspec.open_files('simplecache::zip://*.csv::gcs://bucket/afile.zip',
                       simplecache={'cache_storage': '/stored/csv/files'},
                       gcs={'project': 'my-project'})
```

reads the same zip-file, but extracts the CSV files and stores them locally in the cache.

For developers: this "chaining" methods works by formatting the arguments passed to `open_*` into `target_protocol` (a simple string) and `target_options` (a dict) and also optionally `fo` (target path, if a specific file is required). In order for an implementation to chain successfully like this, it must look for exactly those named arguments.

4.3.12 Caching Files Locally

`fsspec` allows you to access data on remote file systems, that is its purpose. However, such access can often be rather slow compared to local storage, so as well as buffering (see above), the option exists to copy files locally when you first access them, and thereafter to use the local data. This local cache of data might be temporary (i.e., attached to the process and discarded when the process ends) or at some specific location in your local storage.

Two mechanisms are provided, and both involve wrapping a `target` filesystem. The following example creates a file-based cache.

```
fs = fsspec.filesystem("filecache", target_protocol='s3', target_options={'anon': True},
                       cache_storage='/tmp/files/')
```

Each time you open a remote file on S3, it will first copy it to a local temporary directory, and then all further access will use the local file. Since we specify a particular local location, the files will persist and can be reused from future sessions, although you can also set policies to have cached files expire after some time, or to check the remote file system on each open, to see if the target file has changed since it was copied.

With the top-level functions `open`, `open_local` and `open_files`, you can use the same set of kwargs as the example above, or you can chain the URL - the following would be the equivalent

```
of = fsspec.open("filecache::s3://bucket/key",
                 s3={'anon': True}, filecache={'cache_storage':'/tmp/files'})
```

With the "blockcache" variant, data is downloaded block-wise: only the specific parts of the remote file which are accessed. This means that the local copy of the file might end up being much smaller than the remote one, if only certain parts of it are required.

Whereas "filecache" works for all file system implementations, and provides a real local file for other libraries to use, "blockcache" has restrictions: that you have a storage/OS combination which supports sparse files, that the backend implementation uses files which derive `from AbstractBufferedFile`, and that the library you pass the resultant object to accepts generic python file-like objects. You should not mix block- and file-caches in the same directory. "simplecache" is the same as "filecache", except without the options for cache expiry and to check the original source - it can be used where the target can be considered static, and particularly where a large number of target files are expected (because no metadata is written to disc). Only "simplecache" is guaranteed thread/process-safe.

4.3.13 Remote Write Caching

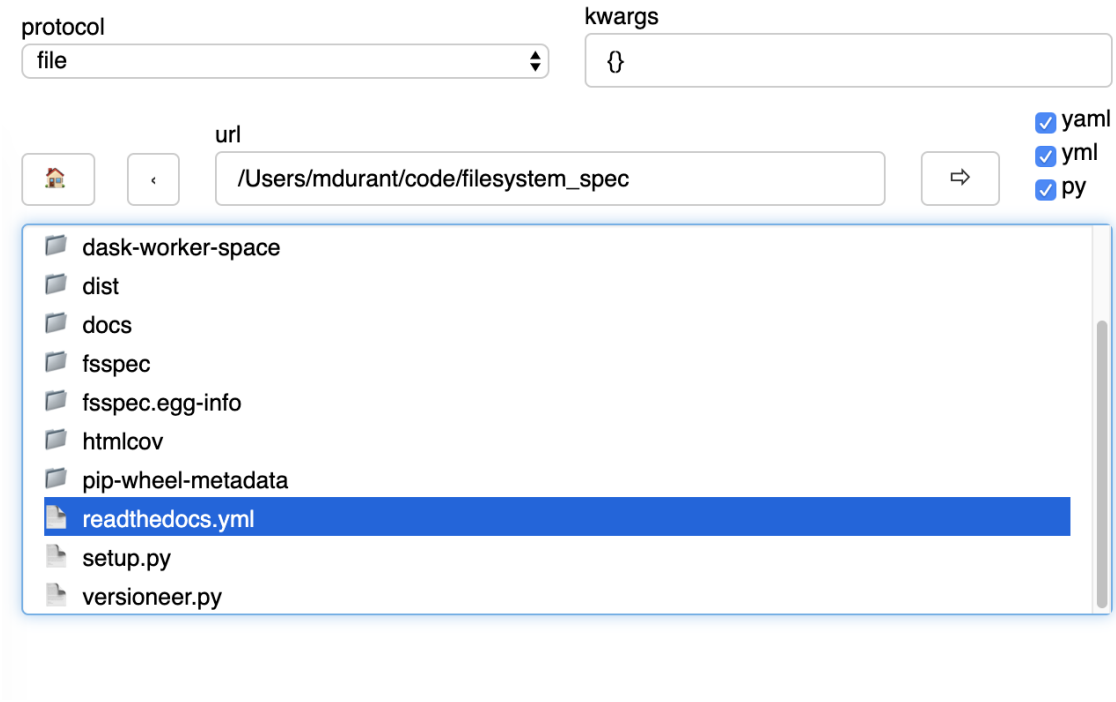
You can cache files to local files to send to remote using the "simplecache" protocol. The following example demonstrates how this might look

```
with fsspec.open('simplecache::s3://mybucket/myfile', 'wb',
                 s3={"profile": "writer"}) as f:
    f.write(b"some data")
```

This will open a local file for writing, and when this file is closed, it will be uploaded to the target URL, in this case on S3. The file-like object `f` can be passed to any library expecting to write to a file. Note that we pass parameters to `S3FileSystem` using the key `"s3"`, the same as the name of the protocol.

4.3.14 File Selector (GUI)

The module `fsspec.gui` contains a graphical file selector interface. It is built using panel, which must be installed in order to use the GUI. Upon instantiation, you can provide the initial URL location (which can be returned to with the "" button), arguments and filters.



Clicking on a directory will descend into it, and selecting a file will mark it as the output of the interface. You can select any of the known protocols, but should provide any required arguments in the "kwargs" box (as a dictionary) and any absolute URL location before clicking "" to go to that location. If using file filters, they will appear as a list of checkboxes; only those file-endings selected will be shown (or if none are selected, all files are shown).

The interface provides the following outputs:

1. `.urlpath`: the currently selected item (if any)
2. `.storage_options`: the value of the kwargs box
3. `.fs`: the current filesystem instance
4. `.open_file()`: produces an `OpenFile` instance for the current selection

4.3.15 Configuration

You can set default keyword arguments to pass to any fsspec backend by editing config files, providing environment variables, or editing the contents of the dictionary `fsspec.config.conf`.

Files are stored in the directory pointed to by `FSSPEC_CONFIG_DIR`, `"~/.config/fsspec/` by default. All *.ini and *.json files will be loaded and parsed from their respective formats and fed into the config dict at import time. For example, if there is a file "~/.config/fsspec/conf.json" containing

```
{"file": {"auto_mkdir": true}}
```

then any instance of the file system whose protocol is "file" (i.e., `LocalFileSystem`) with be passed the kwargs `auto_mkdir=True` **unless** the user supplies the kwarg themselves.

For instance:

```
import fsspec
fs = fsspec.filesystem("file")
assert fs.auto_mkdir == True
fs = fsspec.filesystem("file", auto_mkdir=False)
assert fs.auto_mkdir == False
```

Obviously, you should only define default values that are appropriate for a given file system implementation. INI files only support string values.

Alternatively, you can provide overrides with environment variables of the style `FSSPEC_{protocol}_{kwargname}=value`.

Configuration is determined in the following order, with later items winning:

1. the contents of ini files, and json files in the config directory, sorted alphabetically
2. environment variables
3. the contents of `fsspec.config.conf`, which can be edited at runtime
4. kwargs explicitly passed, whether with `fsspec.open`, `fsspec.filesystem` or directly instantiating the implementation class.

4.3.16 Asynchronous

Some implementations, those deriving from `fsspec.asyn.AsyncFileSystem`, have async/coroutine implementations of some file operations. The async methods have names beginning with `_`, and listed in the `asyn` module; synchronous or blocking functions are automatically generated, which will operate via an event loop in another thread, by default.

See *Async* for modes of operation and how to implement such file systems.

4.3.17 Callbacks

Some methods support a `callback=` argument, which is the entry point to providing feedback on transfers to the user or any other logging service. This feature is new and experimental and supported by varying amounts in the backends.

See the docstrings in the callbacks module for further details.

4.4 Developing with fsspec

Whereas the majority of the documentation describes the use of `fsspec` from the end-user's point of view, `fsspec` is used by many libraries as the primary/only interface to file operations.

4.4.1 Clients of the library

The most common entrance point for libraries which wish to rely on `fsspec` will be `open` or `open_files`, as a way of generating an object compatible with the python file interface. This actually produces an `OpenFile` instance, which can be serialised across a network, and resources are only engaged when entering a context, e.g.

```
with fsspec.open("protocol://path", 'rb', param=value) as f:
    process_file(f)
```

Note the backend-specific parameters that can be passed in this call.

In cases where the caller wants to control the context directly, they can use the `open` method of the `OpenFile`, or get the filesystem object directly, skipping the `OpenFile` route. In the latter case, text encoding and compression or **not** handled for you. The file-like object can also be used as a context manager, or the `close()` method must be called explicitly to release resources.

```
# OpenFile route
of = fsspec.open("protocol://path", 'rb', param=value)
f = of.open()
process_file(f)
f.close()

# filesystem class route, context
fs = fsspec.filesystem("protocol", param=value)
with fs.open("path", "rb") as f:
    process_file(f)

# filesystem class route, explicit close
fs = fsspec.filesystem("protocol", param=value)
f = fs.open("path", "rb")
process_file(f)
f.close()
```

4.4.2 Implementing a backend

The class `AbstractFileSystem` provides a template of the methods that a potential implementation should supply, as well as default implementation of functionality that depends on these. Methods that *could* be implemented are marked with `NotImplementedError` or `pass` (the latter specifically for directory operations that might not be required for some backends where directories are emulated.

Note that not all of the methods need to be implemented: for example, some implementations may be read-only, in which case things like `pipe`, `put`, `touch`, `rm`, etc., can be left as not-implemented (or you might implement them are raise PermissionError, OSError 30 or some read-only exception).

We may eventually refactor `AbstractFileSystem` to split the default implementation, the set of methods that you might implement in a new backend, and the documented end-user API.

In order to register a new backend with fsspec, new backends should register themselves using the entry_points facility from setuptools. In particular, if you want to register a new filesystem protocol `myfs` which is provided by the `MyFS` class in the `myfs` package, add the following to your `setup.py`:

```
setuptools.setup(
    ...
    entry_points={
        'fsspec.specs': [
            'myfs=myfs.MyFS',
        ],
    },
    ...
)
```

Alternatively, the previous method of registering a new backend can be used. That is, new backends must register themselves on import (`register_implementation`) or post a PR to the `fsspec` repo asking to be included in `fsspec.registry.known_implementations`.

4.4.3 Implementing async

Starting in version 0.7.5, we provide async operations for some methods of some implementations. Async support in storage implementations is optional. Special considerations are required for async development, see *Async*.

4.4.4 Developing the library

The following can be used to install `fsspec` in development mode

```
git clone https://github.com/intake/filesystem_spec
cd filesystem_spec
pip install -e .
```

A number of additional dependencies are required to run tests, see "ci/environment*.yml", as well as Docker. Most implementation-specific tests should skip if their requirements are not met.

Development happens by submitting pull requests (PRs) on github. This repo adheres for flake8 and black coding conventions. You may wish to install commit hooks if you intend to make PRs, as linting is done as part of the CI.

Docs use sphinx and the numpy docstring style. Please add an entry to the changelog along with any PR.

4.5 Async

`fsspec` supports asynchronous operations on certain implementations. This allows for concurrent calls within bulk operations such as `cat` (fetch the contents of many files at once) even from normal code, and for the direct use of fsspec in async code without blocking. Async implementations derive from the class `fsspec.async.AsyncFileSystem`. The class attribute `async_impl` can be used to test whether an implementation is async of not.

`AsyncFileSystem` contains `async def` coroutine versions of the methods of `AbstractFileSystem`. By convention, these methods are prefixed with "_" to indicate that they are not to called directly in normal code, only when you know what you are doing. In most cases, the code is identical or slightly modified by replacing sync calls with `await` calls to async functions.

The only async implementation built into `fsspec` is `HTTPFileSystem`.

4.5.1 Synchronous API

The methods of `AbstractFileSystem` are available and can be called from normal code. They call and wait on the corresponding async function. The *work* is carried out in a separate threads, so if there are many fsspec operations in flight at once, launched from many threads, they will still all be processed on the same IO-dedicated thread.

Most users should not be aware that their code is running async.

Note that the sync functions are wrapped using `sync_wrapper`, which copies the docstrings from `AbstractFileSystem`, unless they are explicitly given in the implementation.

Example:

```
fs = fsspec.filesystem("http")
out = fs.cat([url1, url2, url3])  # fetches data concurrently
```

4.5.2 Using from Async

File system instances can be created with `asynchronous=True`. This implies that the instantiation is happening within a coroutine, so the various async method can be called directly with `await`, as is normal in async code.

Note that, because `__init__` is a blocking function, any creation of asynchronous resources will be deferred. You will normally need to explicitly `await` a coroutine to create them. Since garbage collection also happens in blocking code, you may wish to explicitly await resource destructors too. Example:

```
async def work_coroutine():
    fs = fsspec.filesystem("http", asynchronous=True)
    session = await fs.set_session()  # creates client
    out = await fs._cat([url1, url2, url3])  # fetches data concurrently
    await session.close()  # explicit destructor

asyncio.run(work_coroutine())
```

4.5.3 Bring your own loop

For the non-asynchronous case, `fsspec` will normally create an asyncio event loop on a specific thread. However, the calling application may prefer IO processes to run on a loop that is already around and running (in another thread). The loop needs to be asyncio compliant, but does not necessarily need to be an `ayncio.events.AbstractEventLoop`. Example:

```
loop = ...  # however a loop was made, running on another thread
fs = fsspec.filesystem("http", loop=loop)
out = fs.cat([url1, url2, url3])  # fetches data concurrently
```

4.5.4 Implementing new backends

Async file systems should derive from `AsyncFileSystem`, and implement the `async def _*` coroutines there. These functions will either have sync versions automatically generated is the name is in the `async_methods` list, or can be directly created using `sync_wrapper`.

```
class MyFileSystem(AsyncFileSystem):

    async def _my_method(self):
        ...

    my_method = sync_wrapper(_my_method)
```

These functions must **not call** methods or functions which themselves are synced, but should instead `await` other coroutines. Calling methods which do not require sync, such as `_strip_protocol` is fine.

Note that `__init__`, cannot be `async`, so it might need to allocate async resources using the `sync` function, but *only* if `asynchronous=False`. If it is `True`, you probably need to require the caller to await a coroutine that creates those resources. Similarly, any destructor (e.g., `__del__`) will run from normal code, and possibly after the loop has stopped/closed.

To call `sync`, you will need to pass the associated event loop, which will be available as the attribute `.loop`.

fsspec.asyn.AsyncFileSystem(*args, **kwargs)	Async file operations, default implementations
fsspec.asyn.sync(loop, func, *args[, timeout])	Make loop run coroutine until it returns.
fsspec.asyn.sync_wrapper(func[, obj])	Given a function, make so can be called in async or blocking contexts
fsspec.asyn.get_loop()	Create or return the default fsspec IO loop

class fsspec.asyn.**AsyncFileSystem**(**args*, ***kwargs*)

Async file operations, default implementations

Passes bulk operations to asyncio.gather for concurrent operation.

Implementations that have concurrent batch operations and/or async methods should inherit from this class instead of AbstractFileSystem. Docstrings are copied from the un-underscored method in AbstractFileSystem, if not given.

Attributes

loop

transaction A context within which files are committed together upon exit

Methods

cat(path[, recursive, on_error])	Fetch (potentially multiple) paths' contents
cat_file(path[, start, end])	Get the content of a file
checksum(path)	Unique value for current version of file
clear_instance_cache()	Clear the cache of filesystem instances.
copy(path1, path2[, recursive, on_error])	Copy within two locations in the filesystem
cp(path1, path2, **kwargs)	Alias of AbstractFileSystem.copy.
created(path)	Return the created timestamp of a file as a datetime.datetime
current()	Return the most recently created FileSystem
delete(path[, recursive, maxdepth])	Alias of AbstractFileSystem.rm.
disk_usage(path[, total, maxdepth])	Alias of AbstractFileSystem.du.
download(rpath, lpath[, recursive])	Alias of AbstractFileSystem.get.
du(path[, total, maxdepth])	Space used by files within a path
end_transaction()	Finish write transaction, non-context version
exists(path, **kwargs)	Is there a file at the given path
expand_path(path[, recursive, maxdepth])	Turn one or more globs or directories into a list of all matching paths to files or directories.
find(path[, maxdepth, withdirs])	List all files below path.
from_json(blob)	Recreate a filesystem instance from JSON representation
get(rpath, lpath[, recursive, callback])	Copy file(s) to local.
get_file(rpath, lpath[, callback])	Copy single remote file to local
get_mapper(root[, check, create])	Create key/value store based on this file-system
glob(path, **kwargs)	Find files by glob-matching.
head(path[, size])	Get the first size bytes from file
info(path, **kwargs)	Give details of entry at path
invalidate_cache([path])	Discard any cached directory information
isdir(path)	Is this entry directory-like?
isfile(path)	Is this entry file-like?
lexists(path, **kwargs)	If there is a file at the given path (including broken links)
listdir(path[, detail])	Alias of AbstractFileSystem.ls.
ls(path[, detail])	List objects at path.
makedir(path[, create_parents])	Alias of AbstractFileSystem.mkdir.
makedirs(path[, exist_ok])	Recursively make directories
mkdir(path[, create_parents])	Create directory entry at path
mkdirs(path[, exist_ok])	Alias of AbstractFileSystem.makedirs.
modified(path)	Return the modified timestamp of a file as a datetime.datetime
move(path1, path2, **kwargs)	Alias of AbstractFileSystem.mv.
mv(path1, path2[, recursive, maxdepth])	Move file(s) from one location to another
open(path[, mode, block_size, cache_options])	Return a file-like object from the filesystem
pipe(path[, value])	Put value into path
pipe_file(path, value, **kwargs)	Set the bytes of given file
put(lpath, rpath[, recursive, callback])	Copy file(s) from local.
put_file(lpath, rpath[, callback])	Copy single file to remote
read_block(fn, offset, length[, delimiter])	Read a block of bytes from
rename(path1, path2, **kwargs)	Alias of AbstractFileSystem.mv.
rm(path[, recursive, maxdepth])	Delete files.

continues on next page

Table 2 – continued from previous page

rm_file(path)	Delete a file
rmdir(path)	Remove a directory, if empty
sign(path[, expiration])	Create a signed URL representing the given path
size(path)	Size in bytes of file
start_transaction()	Begin write transaction for deferring files, non-context version
stat(path, **kwargs)	Alias of AbstractFileSystem.info.
tail(path[, size])	Get the last size bytes from file
to_json()	JSON representation of this filesystem instance
touch(path[, truncate])	Create empty file, or update timestamp
ukey(path)	Hash of file properties, to tell if it has changed
upload(lpath, rpath[, recursive])	Alias of AbstractFileSystem.put.
walk(path[, maxdepth])	Return all files belows path

cat_ranges	
cp_file	

fsspec.asyn.**sync**(*loop*, *func*, **args*, *timeout=None*, ***kwargs*)
Make loop run coroutine until it returns. Runs in other thread

fsspec.asyn.**sync_wrapper**(*func*, *obj=None*)
Given a function, make so can be called in async or blocking contexts

Leave obj=None if defining within a class. Pass the instance if attaching as an attribute of the instance.

fsspec.asyn.**get_loop**()
Create or return the default fsspec IO loop

The loop will be running on a separate thread.

fsspec.asyn.**fsspec_loop**()
Temporarily switch the current event loop to the fsspec's own loop, and then revert it back after the context gets terinated.

4.6 API Reference

4.6.1 User Functions

fsspec.open_files(urlpath[, mode, ...])	Given a path or paths, return a list of OpenFile objects.
fsspec.open(urlpath[, mode, compression, ...])	Given a path or paths, return one OpenFile object.
fsspec.open_local(url[, mode])	Open file(s) which can be resolved to local
fsspec.filesystem(protocol, **storage_options)	Instantiate filesystems for given protocol and arguments
fsspec.get_filesystem_class(protocol)	Fetch named protocol implementation from the registry
fsspec.get_mapper(url[, check, create, ...])	Create key-value interface for given URL and options
fsspec.fuse.run(fs, path, mount_point[, ...])	Mount stuff in a local directory
fsspec.gui.FileSelector([url, filters, ...])	Panel-based graphical file selector widget

fsspec.**open_files**(*urlpath*, *mode='rb'*, *compression=None*, *encoding='utf8'*, *errors=None*, *name_function=None*, *num=1*, *protocol=None*, *newline=None*, *auto_mkdir=True*, *expand=True*, ***kwargs*)

Given a path or paths, return a list of `OpenFile` objects.

For writing, a str path must contain the "*" character, which will be filled in by increasing numbers, e.g., "part*" -> "part1", "part2" if num=2.

For either reading or writing, can instead provide explicit list of paths.

Parameters

urlpath: string or list Absolute or relative filepath(s). Prefix with a protocol like `s3://` to read from alternative filesystems. To read from multiple files you can pass a globstring or a list of paths, with the caveat that they must all have the same protocol.

mode: 'rb', 'wt', etc.

compression: string Compression to use. See `dask.bytes.compression.files` for options.

encoding: str For text mode only

errors: None or str Passed to TextIOWrapper in text mode

name_function: function or None if opening a set of files for writing, those files do not yet exist, so we need to generate their names by formatting the urlpath for each sequence number

num: int [1] if writing mode, number of files we expect to create (passed to name+function)

protocol: str or None If given, overrides the protocol found in the URL.

newline: bytes or None Used for line terminator in text mode. If None, uses system default; if blank, uses no translation.

auto_mkdir: bool (True) If in write mode, this will ensure the target directory exists before writing, by calling `fs.mkdirs(exist_ok=True)`.

expand: bool

****kwargs: dict** Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns

An `OpenFiles` instance, which is a list of `OpenFile` objects that can

be used as a single context

Examples

```
>>> files = open_files('2015-*-*.csv')
>>> files = open_files(
...     's3://bucket/2015-*-*.csv.gz', compression='gzip'
... )
```

fsspec.**open**(*urlpath*, *mode='rb'*, *compression=None*, *encoding='utf8'*, *errors=None*, *protocol=None*, *newline=None*, ***kwargs*)

Given a path or paths, return one `OpenFile` object.

Parameters

urlpath: string or list Absolute or relative filepath. Prefix with a protocol like `s3://` to read from alternative filesystems. Should not include glob character(s).

mode: 'rb', 'wt', etc.

compression: string Compression to use. See `dask.bytes.compression.files` for options.

encoding: str For text mode only

errors: None or str Passed to TextIOWrapper in text mode

protocol: str or None If given, overrides the protocol found in the URL.

newline: bytes or None Used for line terminator in text mode. If None, uses system default; if blank, uses no translation.

****kwargs: dict** Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns

`OpenFile` object.

Examples

```
>>> openfile = open('2015-01-01.csv')
>>> openfile = open(
...     's3://bucket/2015-01-01.csv.gz', compression='gzip'
... )
>>> with openfile as f:
...     df = pd.read_csv(f)
...
```

`fsspec.`**`open_local`**(*url*, *mode='rb'*, ***storage_options*)
Open file(s) which can be resolved to local

For files which either are local, or get downloaded upon open (e.g., by file caching)

Parameters

url: str or list(str)

mode: str Must be read mode

storage_options: passed on to FS for or used by open_files (e.g., compression)

`fsspec.`**`filesystem`**(*protocol*, ***storage_options*)
Instantiate filesystems for given protocol and arguments

`storage_options` are specific to the protocol being chosen, and are passed directly to the class.

`fsspec.`**`get_filesystem_class`**(*protocol*)
Fetch named protocol implementation from the registry

The dict `known_implementations` maps protocol names to the locations of classes implementing the corresponding file-system. When used for the first time, appropriate imports will happen and the class will be placed in the registry. All subsequent calls will fetch directly from the registry.

Some protocol implementations require additional dependencies, and so the import may fail. In this case, the string in the "err" field of the `known_implementations` will be given as the error message.

fsspec.**get_mapper**(*url*, *check=False*, *create=False*, *missing_exceptions=None*, *alternate_root=None*, ***kwargs*)
Create key-value interface for given URL and options

The URL will be of the form "protocol://location" and point to the root of the mapper required. All keys will be file-names below this location, and their values the contents of each key.

Also accepts compound URLs like zip::s3://bucket/file.zip , see `fsspec.open`.

Parameters

url: str Root URL of mapping

check: bool Whether to attempt to read from the location before instantiation, to check that the mapping does exist

create: bool Whether to make the directory corresponding to the root before instantiating

missing_exceptions: None or tuple If given, these exception types will be regarded as missing keys and return KeyError when trying to read data. By default, you get (FileNotFoundError, IsADirectoryError, NotADirectoryError)

alternate_root: None or str In cases of complex URLs, the parser may fail to pick the correct part for the mapper root, so this arg can override

Returns

`FSMap` instance, the dict-like key-value store.

fsspec.fuse.**run**(*fs*, *path*, *mount_point*, *foreground=True*, *threads=False*, *ready_file=False*, *ops_class=<class 'fsspec.fuse.FUSEr'>*)
Mount stuff in a local directory

This uses fusepy to make it appear as if a given path on an fsspec instance is in fact resident within the local file-system.

This requires that fusepy by installed, and that FUSE be available on the system (typically requiring a package to be installed with apt, yum, brew, etc.).

Parameters

fs: file-system instance From one of the compatible implementations

path: str Location on that file-system to regard as the root directory to mount. Note that you typically should include the terminating "/" character.

mount_point: str An empty directory on the local file-system where the contents of the remote path will appear.

foreground: bool Whether or not calling this function will block. Operation will typically be more stable if True.

threads: bool Whether or not to create threads when responding to file operations within the mounter directory. Operation will typically be more stable if False.

ready_file: bool Whether the FUSE process is ready. The `fuse_ready` file will exist in the `mount_point` directory if True. Debugging purpose.

ops_class: FUSEr or Subclass of FUSEr To override the default behavior of FUSEr. For Example, logging to file.

class fsspec.gui.**FileSelector**(*url=None*, *filters=None*, *ignore=None*, *kwargs=None*)
Panel-based graphical file selector widget

Instances of this widget are interactive and can be displayed in jupyter by having them as the output of a cell, or in a separate browser tab using `.show()`.

Attributes

fs Current filesystem instance

storage_options Value of the kwargs box as a dictionary

urlpath URL of currently selected item

Methods

connect(signal, slot)	Associate call back with given event
ignore_events()	Temporarily turn off events processing in this instance
open_file([mode, compression, encoding])	Create OpenFile instance for the currently selected item
show([threads])	Open a new browser tab and display this instance's interface

filters_changed	
go_clicked	
home_clicked	
protocol_changed	
selection_changed	
set_filters	
up_clicked	

property fs
Current filesystem instance

open_file(*mode='rb'*, *compression=None*, *encoding=None*)
Create OpenFile instance for the currently selected item

For example, in a notebook you might do something like

```
[ ]: sel = FileSelector(); sel

# user selects their file

[ ]: with sel.open_file('rb') as f:
...      out = f.read()
```

Parameters

mode: str (optional) Open mode for the file.

compression: str (optional) The interact with the file as compressed. Set to 'infer' to guess compression from the file ending

encoding: str (optional) If using text mode, use this encoding; defaults to UTF8.

property storage_options
Value of the kwargs box as a dictionary

property urlpath
URL of currently selected item

4.6.2 Base Classes

fsspec.spec.AbstractFileSystem(*args, **kwargs)	An abstract super-class for pythonic file-systems
fsspec.spec.Transaction(fs)	Filesystem transaction write context
fsspec.spec.AbstractBufferedFile(fs, path[, ...])	Convenient class to derive from to provide buffering
fsspec.archive.AbstractArchiveFileSystem(...)	A generic superclass for implementing Archive-based filesystems.
fsspec.FSMap(root, fs[, check, create, ...])	Wrap a FileSystem instance as a mutable wrapping.
fsspec.asyn.AsyncFileSystem(*args, **kwargs)	Async file operations, default implementations
fsspec.core.OpenFile(fs, path[, mode, ...])	File-like object to be used in a context
fsspec.core.OpenFiles(*args[, mode, fs])	List of OpenFile instances
fsspec.core.BaseCache(blocksize, fetcher, size)	Pass-though cache: doesn't keep anything, calls every time
fsspec.core.get_fs_token_paths(urlpath[, ...])	Filesystem, deterministic token, and paths from a urlpath and options.
fsspec.core.url_to_fs(url, **kwargs)	Turn fully-qualified and potentially chained URL into filesystem instance
fsspec.dircache.DirCache([...])	Caching of directory listings, in a structure like.
fsspec.registry.ReadOnlyRegistry(target)	Dict-like registry, but immutable
fsspec.registry.register_implementation(...)	Add implementation class to the registry
fsspec.callbacks.Callback([size, value, hooks])	Base class and interface for callback mechanism
fsspec.callbacks.NoOpCallback([size, value, ...])	This implementation of Callback does exactly nothing
fsspec.callbacks.DotPrinterCallback([...])	Simple example Callback implementation

class fsspec.spec.**AbstractFileSystem**(**args*, ***kwargs*)

An abstract super-class for pythonic file-systems

Implementations are expected to be compatible with or, better, subclass from here.

Attributes

transaction A context within which files are committed together upon exit

Methods

cat(path[, recursive, on_error])	Fetch (potentially multiple) paths' contents
cat_file(path[, start, end])	Get the content of a file
checksum(path)	Unique value for current version of file
clear_instance_cache()	Clear the cache of filesystem instances.
copy(path1, path2[, recursive, on_error])	Copy within two locations in the filesystem
cp(path1, path2, **kwargs)	Alias of *AbstractFileSystem.copy*.
created(path)	Return the created timestamp of a file as a datetime.datetime
current()	Return the most recently created FileSystem
delete(path[, recursive, maxdepth])	Alias of *AbstractFileSystem.rm*.
disk_usage(path[, total, maxdepth])	Alias of *AbstractFileSystem.du*.
download(rpath, lpath[, recursive])	Alias of *AbstractFileSystem.get*.
du(path[, total, maxdepth])	Space used by files within a path
end_transaction()	Finish write transaction, non-context version
exists(path, **kwargs)	Is there a file at the given path

continues on next page

Table 6 – continued from previous page

expand_path(path[, recursive, maxdepth])	Turn one or more globs or directories into a list of all matching paths to files or directories.
find(path[, maxdepth, withdirs])	List all files below path.
from_json(blob)	Recreate a filesystem instance from JSON representation
get(rpath, lpath[, recursive, callback])	Copy file(s) to local.
get_file(rpath, lpath[, callback])	Copy single remote file to local
get_mapper(root[, check, create])	Create key/value store based on this file-system
glob(path, **kwargs)	Find files by glob-matching.
head(path[, size])	Get the first `size` bytes from file
info(path, **kwargs)	Give details of entry at path
invalidate_cache([path])	Discard any cached directory information
isdir(path)	Is this entry directory-like?
isfile(path)	Is this entry file-like?
lexists(path, **kwargs)	If there is a file at the given path (including broken links)
listdir(path[, detail])	Alias of *AbstractFileSystem.ls*.
ls(path[, detail])	List objects at path.
makedir(path[, create_parents])	Alias of *AbstractFileSystem.mkdir*.
makedirs(path[, exist_ok])	Recursively make directories
mkdir(path[, create_parents])	Create directory entry at path
mkdirs(path[, exist_ok])	Alias of *AbstractFileSystem.makedirs*.
modified(path)	Return the modified timestamp of a file as a datetime.datetime
move(path1, path2, **kwargs)	Alias of *AbstractFileSystem.mv*.
mv(path1, path2[, recursive, maxdepth])	Move file(s) from one location to another
open(path[, mode, block_size, cache_options])	Return a file-like object from the filesystem
pipe(path[, value])	Put value into path
pipe_file(path, value, **kwargs)	Set the bytes of given file
put(lpath, rpath[, recursive, callback])	Copy file(s) from local.
put_file(lpath, rpath[, callback])	Copy single file to remote
read_block(fn, offset, length[, delimiter])	Read a block of bytes from
rename(path1, path2, **kwargs)	Alias of *AbstractFileSystem.mv*.
rm(path[, recursive, maxdepth])	Delete files.
rm_file(path)	Delete a file
rmdir(path)	Remove a directory, if empty
sign(path[, expiration])	Create a signed URL representing the given path
size(path)	Size in bytes of file
start_transaction()	Begin write transaction for deferring files, non-context version
stat(path, **kwargs)	Alias of *AbstractFileSystem.info*.
tail(path[, size])	Get the last `size` bytes from file
to_json()	JSON representation of this filesystem instance
touch(path[, truncate])	Create empty file, or update timestamp
ukey(path)	Hash of file properties, to tell if it has changed
upload(lpath, rpath[, recursive])	Alias of *AbstractFileSystem.put*.
walk(path[, maxdepth])	Return all files belows path

cat_ranges	
cp_file	

cat(*path*, *recursive=False*, *on_error='raise'*, ***kwargs*)
Fetch (potentially multiple) paths' contents

Parameters

recursive: bool If True, assume the path(s) are directories, and get all the contained files

on_error ["raise", "omit", "return"] If raise, an underlying exception will be raised (converted to KeyError if the type is in self.missing_exceptions); if omit, keys with exception will simply not be included in the output; if "return", all keys are included in the output, but the value will be bytes or an exception instance.

kwargs: passed to cat_file

Returns

dict of {path: contents} if there are multiple paths

or the path has been otherwise expanded

cat_file(*path*, *start=None*, *end=None*, ***kwargs*)
Get the content of a file

Parameters

path: URL of file on this filesystems

start, end: int Bytes limits of the read. If negative, backwards from end, like usual python slices. Either can be None for start or end of file, respectively

kwargs: passed to ``open()``.

checksum(*path*)
Unique value for current version of file

If the checksum is the same from one moment to another, the contents are guaranteed to be the same. If the checksum changes, the contents *might* have changed.

This should normally be overridden; default will probably capture creation/modification timestamp (which would be good) or maybe access timestamp (which would be bad)

classmethod clear_instance_cache()
Clear the cache of filesystem instances.

Notes

Unless overridden by setting the `cachable` class attribute to False, the filesystem class stores a reference to newly created instances. This prevents Python's normal rules around garbage collection from working, since the instances refcount will not drop to zero until `clear_instance_cache` is called.

copy(*path1*, *path2*, *recursive=False*, *on_error=None*, ***kwargs*)
Copy within two locations in the filesystem

on_error ["raise", "ignore"] If raise, any not-found exceptions will be raised; if ignore any not-found exceptions will cause the path to be skipped; defaults to raise unless recursive is true, where the default is ignore

cp(*path1*, *path2*, ***kwargs*)
Alias of *AbstractFileSystem.copy*.

created(*path*)
Return the created timestamp of a file as a datetime.datetime

classmethod current()
Return the most recently created FileSystem

If no instance has been created, then create one with defaults

delete(*path*, *recursive=False*, *maxdepth=None*)
Alias of *AbstractFileSystem.rm*.

disk_usage(*path*, *total=True*, *maxdepth=None*, ***kwargs*)
Alias of *AbstractFileSystem.du*.

download(*rpath*, *lpath*, *recursive=False*, ***kwargs*)
Alias of *AbstractFileSystem.get*.

du(*path*, *total=True*, *maxdepth=None*, ***kwargs*)
Space used by files within a path

Parameters

path: str

total: bool whether to sum all the file sizes

maxdepth: int or None maximum number of directory levels to descend, None for unlimited.

kwargs: passed to ``ls``

Returns

Dict of {fn: size} if total=False, or int otherwise, where numbers

refer to bytes used.

end_transaction()
Finish write transaction, non-context version

exists(*path*, ***kwargs*)
Is there a file at the given path

expand_path(*path*, *recursive=False*, *maxdepth=None*)
Turn one or more globs or directories into a list of all matching paths to files or directories.

find(*path*, *maxdepth=None*, *withdirs=False*, ***kwargs*)
List all files below path.

Like posix `find` command without conditions

Parameters

path [str]

maxdepth: int or None If not None, the maximum number of levels to descend

withdirs: bool Whether to include directory paths in the output. This is True when used by glob, but users usually only want files.

kwargs are passed to ``ls``.

static from_json(*blob*)
Recreate a filesystem instance from JSON representation

See `.to_json()` for the expected structure of the input

Parameters

blob: str

Returns

file system instance, not necessarily of this particular class.

`get`(*rpath*, *lpath*, *recursive=False*, *callback=<fsspec.callbacks.NoOpCallback object>*, ***kwargs*)
Copy file(s) to local.

Copies a specific file or tree of files (if recursive=True). If lpath ends with a "/", it will be assumed to be a directory, and target files will go within. Can submit a list of paths, which may be glob-patterns and will be expanded.

Calls get_file for each source.

`get_file`(*rpath*, *lpath*, *callback=<fsspec.callbacks.NoOpCallback object>*, ***kwargs*)
Copy single remote file to local

`get_mapper`(*root*, *check=False*, *create=False*)
Create key/value store based on this file-system

Makes a MutableMapping interface to the FS at the given root path. See `fsspec.mapping.FSMap` for further details.

`glob`(*path*, ***kwargs*)
Find files by glob-matching.

If the path ends with '/' and does not contain "*", it is essentially the same as `ls(path)`, returning only files.

We support `"**"`, `"?"` and `"[..]"`. We do not support ^ for pattern negation.

Search path names that contain embedded characters special to this implementation of glob may not produce expected results; e.g., 'foo/bar/*starredfilename*'.

kwargs are passed to `ls`.

`head`(*path*, *size=1024*)
Get the first `size` bytes from file

`info`(*path*, ***kwargs*)
Give details of entry at path

Returns a single dictionary, with exactly the same information as `ls` would with `detail=True`.

The default implementation should calls ls and could be overridden by a shortcut. kwargs are passed on to `` `ls() ``.

Some file systems might not be able to measure the file's size, in which case, the returned dict will include `'size': None`.

Returns

dict with keys: name (full path in the FS), size (in bytes), type (file,

directory, or something else) and other FS-specific keys.

`invalidate_cache`(*path=None*)
Discard any cached directory information

Parameters

path: string or None If None, clear all listings cached else listings at or under given path.

`isdir`(*path*)
Is this entry directory-like?

isfile(*path*)
Is this entry file-like?

lexists(*path*, ***kwargs*)
If there is a file at the given path (including broken links)

listdir(*path*, *detail=True*, ***kwargs*)
Alias of *AbstractFileSystem.ls*.

ls(*path*, *detail=True*, ***kwargs*)
List objects at path.

This should include subdirectories and files at that location. The difference between a file and a directory must be clear when details are requested.

The specific keys, or perhaps a FileInfo class, or similar, is TBD, but must be consistent across implementations. Must include:

- full path to the entry (without protocol)
- size of the entry, in bytes. If the value cannot be determined, will be `None`.
- type of entry, "file", "directory" or other

Additional information may be present, aproriate to the file-system, e.g., generation, checksum, etc.

May use refresh=True|False to allow use of self._ls_from_cache to check for a saved listing and avoid calling the backend. This would be common where listing may be expensive.

Parameters

path: str

detail: bool if True, gives a list of dictionaries, where each is the same as the result of `info(path)`. If False, gives a list of paths (str).

kwargs: may have additional backend-specific options, such as version information

Returns

List of strings if detail is False, or list of directory information

dicts if detail is True.

makedir(*path*, *create_parents=True*, ***kwargs*)
Alias of *AbstractFileSystem.mkdir*.

makedirs(*path*, *exist_ok=False*)
Recursively make directories

Creates directory at path and any intervening required directories. Raises exception if, for instance, the path already exists but is a file.

Parameters

path: str leaf directory name

exist_ok: bool (False) If True, will error if the target already exists

mkdir(*path*, *create_parents=True*, ***kwargs*)
Create directory entry at path

For systems that don't have true directories, may create an for this instance only and not touch the real filesystem

Parameters

path: str location

create_parents: bool if True, this is equivalent to `makedirs`

kwargs: may be permissions, etc.

mkdirs(*path*, *exist_ok=False*)
Alias of *AbstractFileSystem.makedirs*.

modified(*path*)
Return the modified timestamp of a file as a datetime.datetime

move(*path1*, *path2*, ***kwargs*)
Alias of *AbstractFileSystem.mv*.

mv(*path1*, *path2*, *recursive=False*, *maxdepth=None*, ***kwargs*)
Move file(s) from one location to another

open(*path*, *mode='rb'*, *block_size=None*, *cache_options=None*, ***kwargs*)
Return a file-like object from the filesystem

The resultant instance must function correctly in a context `with` block.

Parameters

path: str Target file

mode: str like 'rb', 'w' See builtin `open()`

block_size: int Some indication of buffering - this is a value in bytes

cache_options [dict, optional] Extra arguments to pass through to the cache.

encoding, errors, newline: passed on to TextIOWrapper for text mode

pipe(*path*, *value=None*, ***kwargs*)
Put value into path

(counterpart to `cat`)

Parameters

path: string or dict(str, bytes) If a string, a single remote location to put `value` bytes; if a dict, a mapping of {path: bytesvalue}.

value: bytes, optional If using a single path, these are the bytes to put there. Ignored if `path` is a dict

pipe_file(*path*, *value*, ***kwargs*)
Set the bytes of given file

put(*lpath*, *rpath*, *recursive=False*, *callback=<fsspec.callbacks.NoOpCallback object>*, ***kwargs*)
Copy file(s) from local.

Copies a specific file or tree of files (if recursive=True). If rpath ends with a "/", it will be assumed to be a directory, and target files will go within.

Calls put_file for each source.

put_file(*lpath*, *rpath*, *callback=<fsspec.callbacks.NoOpCallback object>*, ***kwargs*)
Copy single file to remote

read_block(*fn*, *offset*, *length*, *delimiter=None*)
Read a block of bytes from

Starting at `offset` of the file, read `length` bytes. If `delimiter` is set then we ensure that the read starts and stops at delimiter boundaries that follow the locations `offset` and `offset + length`. If `offset` is zero then we start at zero. The bytestring returned WILL include the end delimiter string.

If offset+length is beyond the eof, reads to eof.

Parameters

fn: string Path to filename

offset: int Byte offset to start read

length: int Number of bytes to read

delimiter: bytes (optional) Ensure reading starts and stops at delimiter bytestring

See also:

utils.read_block

Examples

```
>>> fs.read_block('data/file.csv', 0, 13)
b'Alice, 100\nBo'
>>> fs.read_block('data/file.csv', 0, 13, delimiter=b'\n')
b'Alice, 100\nBob, 200\n'
```

Use `length=None` to read to the end of the file. >>> fs.read_block('data/file.csv', 0, None, delimiter=b'n') # doctest: +SKIP b'Alice, 100nBob, 200nCharlie, 300'

rename(*path1*, *path2*, ***kwargs*)
Alias of *AbstractFileSystem.mv*.

rm(*path*, *recursive=False*, *maxdepth=None*)
Delete files.

Parameters

path: str or list of str File(s) to delete.

recursive: bool If file(s) are directories, recursively delete contents and then also remove the directory

maxdepth: int or None Depth to pass to walk for finding files to delete, if recursive. If None, there will be no limit and infinite recursion may be possible.

rm_file(*path*)
Delete a file

rmdir(*path*)
Remove a directory, if empty

sign(*path*, *expiration=100*, ***kwargs*)
Create a signed URL representing the given path

Some implementations allow temporary URLs to be generated, as a way of delegating credentials.

Parameters

path [str] The path on the filesystem

expiration [int] Number of seconds to enable the URL for (if supported)

Returns

URL [str] The signed URL

Raises

NotImplementedError [if method is not implemented for a filesystem]

`size`(*path*)
Size in bytes of file

`start_transaction`()
Begin write transaction for deferring files, non-context version

`stat`(*path*, ***kwargs*)
Alias of *AbstractFileSystem.info*.

`tail`(*path*, *size=1024*)
Get the last `size` bytes from file

`to_json`()
JSON representation of this filesystem instance

Returns

str: JSON structure with keys cls (the python location of this class), protocol (text name of this class's protocol, first one in case of multiple), args (positional args, usually empty), and all other kwargs as their own keys.

`touch`(*path*, *truncate=True*, ***kwargs*)
Create empty file, or update timestamp

Parameters

path: str file location

truncate: bool If True, always set file size to 0; if False, update timestamp and leave file unchanged, if backend allows this

`property transaction`
A context within which files are committed together upon exit

Requires the file class to implement *commit()* and *discard()* for the normal and exception cases.

`ukey`(*path*)
Hash of file properties, to tell if it has changed

`upload`(*lpath*, *rpath*, *recursive=False*, ***kwargs*)
Alias of *AbstractFileSystem.put*.

`walk`(*path*, *maxdepth=None*, ***kwargs*)
Return all files belows path

List all files, recursing into subdirectories; output is iterator-style, like `os.walk()`. For a simple list of files, `find()` is available.

Note that the "files" outputted will include anything that is not a directory, such as links.

Parameters

path: str Root to recurse into

maxdepth: int Maximum recursion depth. None means limitless, but not recommended on link-based file-systems.

kwargs: passed to ``ls``

class fsspec.spec.**Transaction**(*fs*)
Filesystem transaction write context

Gathers files for deferred commit or discard, so that several write operations can be finalized semi-atomically. This works by having this instance as the `.transaction` attribute of the given filesystem

Methods

complete([commit])	Finish transaction: commit or discard all deferred files
start()	Start a transaction on this FileSystem

complete(*commit=True*)
Finish transaction: commit or discard all deferred files

start()
Start a transaction on this FileSystem

class fsspec.spec.**AbstractBufferedFile**(*fs*, *path*, *mode='rb'*, *block_size='default'*, *autocommit=True*, *cache_type='readahead'*, *cache_options=None*, *size=None*, ***kwargs*)
Convenient class to derive from to provide buffering

In the case that the backend does not provide a pythonic file-like object already, this class contains much of the logic to build one. The only methods that need to be overridden are `_upload_chunk`, `_initiate_upload` and `_fetch_range`.

Attributes

closed

details

full_name

Methods

close()	Close file
commit()	Move from temp to final destination
discard()	Throw away temporary file
fileno(/)	Returns underlying file descriptor if one exists.
flush([force])	Write buffered data to backend store.
info()	File information about this path
isatty(/)	Return whether this is an 'interactive' stream.
read([length])	Return data from cache, or fetch pieces as necessary
readable()	Whether opened for reading
readinto(b)	mirrors builtin file's readinto method
readline()	Read until first occurrence of newline character
readlines()	Return all data, split by the newline character
readuntil([char, blocks])	Return data between current position and first occurrence of char
seek(loc[, whence])	Set current file location
seekable()	Whether is seekable (only in read mode)

continues on next page

Table 8 – continued from previous page

tell()	Current file location
truncate	Truncate file to size bytes.
writable()	Whether opened for writing
write(data)	Write data to buffer.
writelines(lines, /)	Write a list of lines to stream.

readinto1	

close()
Close file

Finalizes writes, discards cache

commit()
Move from temp to final destination

discard()
Throw away temporary file

flush(*force=False*)
Write buffered data to backend store.

Writes the current buffer, if it is larger than the block-size, or if the file is being closed.

Parameters

force: bool When closing, write the last block even if it is smaller than blocks are allowed to be. Disallows further writing to this file.

info()
File information about this path

read(*length=- 1*)
Return data from cache, or fetch pieces as necessary

Parameters

length: int (-1) Number of bytes to read; if <0, all remaining bytes.

readable()
Whether opened for reading

readinto(*b*)
mirrors builtin file's readinto method

https://docs.python.org/3/library/io.html#io.RawIOBase.readinto

readline()
Read until first occurrence of newline character

Note that, because of character encoding, this is not necessarily a true line ending.

readlines()
Return all data, split by the newline character

readuntil(*char=b'\n'*, *blocks=None*)
Return data between current position and first occurrence of char

char is included in the output, except if the end of the tile is encountered first.

Parameters

char: bytes Thing to find

blocks: None or int How much to read in each go. Defaults to file blocksize - which may mean a new read on every call.

`seek`(*loc*, *whence=0*)
Set current file location

Parameters

loc: int byte location

whence: {0, 1, 2} from start of file, current location or end of file, resp.

`seekable`()
Whether is seekable (only in read mode)

`tell`()
Current file location

`writable`()
Whether opened for writing

`write`(*data*)
Write data to buffer.

Buffer only sent on flush() or if buffer is greater than or equal to blocksize.

Parameters

data: bytes Set of bytes to be written.

`class fsspec.archive.AbstractArchiveFileSystem`(**args*, ***kwargs*)
A generic superclass for implementing Archive-based filesystems.

Currently, it is shared amongst `ZipFileSystem`, `LibArchiveFileSystem` and `TarFileSystem`.

Attributes

`transaction` A context within which files are committed together upon exit

Methods

`cat`(path[, recursive, on_error])	Fetch (potentially multiple) paths' contents
`cat_file`(path[, start, end])	Get the content of a file
`checksum`(path)	Unique value for current version of file
`clear_instance_cache`()	Clear the cache of filesystem instances.
`copy`(path1, path2[, recursive, on_error])	Copy within two locations in the filesystem
`cp`(path1, path2, **kwargs)	Alias of `AbstractFileSystem.copy`.
`created`(path)	Return the created timestamp of a file as a datetime.datetime
`current`()	Return the most recently created FileSystem
`delete`(path[, recursive, maxdepth])	Alias of `AbstractFileSystem.rm`.
`disk_usage`(path[, total, maxdepth])	Alias of `AbstractFileSystem.du`.
`download`(rpath, lpath[, recursive])	Alias of `AbstractFileSystem.get`.
`du`(path[, total, maxdepth])	Space used by files within a path
`end_transaction`()	Finish write transaction, non-context version
`exists`(path, **kwargs)	Is there a file at the given path

continues on next page

Table 9 – continued from previous page

expand_path(path[, recursive, maxdepth])	Turn one or more globs or directories into a list of all matching paths to files or directories.
find(path[, maxdepth, withdirs])	List all files below path.
from_json(blob)	Recreate a filesystem instance from JSON representation
get(rpath, lpath[, recursive, callback])	Copy file(s) to local.
get_file(rpath, lpath[, callback])	Copy single remote file to local
get_mapper(root[, check, create])	Create key/value store based on this file-system
glob(path, **kwargs)	Find files by glob-matching.
head(path[, size])	Get the first size bytes from file
info(path, **kwargs)	Give details of entry at path
invalidate_cache([path])	Discard any cached directory information
isdir(path)	Is this entry directory-like?
isfile(path)	Is this entry file-like?
lexists(path, **kwargs)	If there is a file at the given path (including broken links)
listdir(path[, detail])	Alias of AbstractFileSystem.ls.
ls(path[, detail])	List objects at path.
makedir(path[, create_parents])	Alias of AbstractFileSystem.mkdir.
makedirs(path[, exist_ok])	Recursively make directories
mkdir(path[, create_parents])	Create directory entry at path
mkdirs(path[, exist_ok])	Alias of AbstractFileSystem.makedirs.
modified(path)	Return the modified timestamp of a file as a datetime.datetime
move(path1, path2, **kwargs)	Alias of AbstractFileSystem.mv.
mv(path1, path2[, recursive, maxdepth])	Move file(s) from one location to another
open(path[, mode, block_size, cache_options])	Return a file-like object from the filesystem
pipe(path[, value])	Put value into path
pipe_file(path, value, **kwargs)	Set the bytes of given file
put(lpath, rpath[, recursive, callback])	Copy file(s) from local.
put_file(lpath, rpath[, callback])	Copy single file to remote
read_block(fn, offset, length[, delimiter])	Read a block of bytes from
rename(path1, path2, **kwargs)	Alias of AbstractFileSystem.mv.
rm(path[, recursive, maxdepth])	Delete files.
rm_file(path)	Delete a file
rmdir(path)	Remove a directory, if empty
sign(path[, expiration])	Create a signed URL representing the given path
size(path)	Size in bytes of file
start_transaction()	Begin write transaction for deferring files, non-context version
stat(path, **kwargs)	Alias of AbstractFileSystem.info.
tail(path[, size])	Get the last size bytes from file
to_json()	JSON representation of this filesystem instance
touch(path[, truncate])	Create empty file, or update timestamp
ukey(path)	Hash of file properties, to tell if it has changed
upload(lpath, rpath[, recursive])	Alias of AbstractFileSystem.put.
walk(path[, maxdepth])	Return all files belows path

cat_ranges	
cp_file	

`info`(*path*, ***kwargs*)
Give details of entry at path

Returns a single dictionary, with exactly the same information as `ls` would with `detail=True`.

The default implementation should calls ls and could be overridden by a shortcut. kwargs are passed on to `` `ls() ``.

Some file systems might not be able to measure the file's size, in which case, the returned dict will include `'size': None`.

Returns

dict with keys: name (full path in the FS), size (in bytes), type (file,

directory, or something else) and other FS-specific keys.

`ls`(*path*, *detail=False*, ***kwargs*)
List objects at path.

This should include subdirectories and files at that location. The difference between a file and a directory must be clear when details are requested.

The specific keys, or perhaps a FileInfo class, or similar, is TBD, but must be consistent across implementations. Must include:

- full path to the entry (without protocol)
- size of the entry, in bytes. If the value cannot be determined, will be `None`.
- type of entry, "file", "directory" or other

Additional information may be present, aproriate to the file-system, e.g., generation, checksum, etc.

May use refresh=True|False to allow use of self._ls_from_cache to check for a saved listing and avoid calling the backend. This would be common where listing may be expensive.

Parameters

path: str

detail: bool if True, gives a list of dictionaries, where each is the same as the result of `info(path)`. If False, gives a list of paths (str).

kwargs: may have additional backend-specific options, such as version information

Returns

List of strings if detail is False, or list of directory information

dicts if detail is True.

`ukey`(*path*)
Hash of file properties, to tell if it has changed

`class` `fsspec.FSMap`(*root*, *fs*, *check=False*, *create=False*, *missing_exceptions=None*)
Wrap a FileSystem instance as a mutable wrapping.

The keys of the mapping become files under the given root, and the values (which must be bytes) the contents of those files.

Parameters

root: string prefix for all the files

fs: FileSystem instance

check: bool (=True) performs a touch at the location, to check for write access.

Examples

```
>>> fs = FileSystem(**parameters)
>>> d = FSMap('my-data/path/', fs)
or, more likely
>>> d = fs.get_mapper('my-data/path/')
```

```
>>> d['loc1'] = b'Hello World'
>>> list(d.keys())
['loc1']
>>> d['loc1']
b'Hello World'
```

Methods

clear()	Remove all keys below root - empties out mapping
delitems(keys)	Remove multiple keys from the store
get(k[,d])	
getitems(keys[, on_error])	Fetch multiple items from the store
items()	
keys()	
pop(k[,d])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem()	as a 2-tuple; but raise KeyError if D is empty.
setdefault(k[,d])	
setitems(values_dict)	Set the values of multiple items in the store
update([E,]**F)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values()	

clear()
Remove all keys below root - empties out mapping

delitems(*keys*)
Remove multiple keys from the store

getitems(*keys*, *on_error='raise'*)
Fetch multiple items from the store

If the backend is async-able, this might proceed concurrently

Parameters

keys: list(str) They keys to be fetched

on_error ["raise", "omit", "return"] If raise, an underlying exception will be raised (converted to KeyError if the type is in self.missing_exceptions); if omit, keys with exception

will simply not be included in the output; if "return", all keys are included in the output, but the value will be bytes or an exception instance.

Returns

dict(key, bytes|exception)

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

setitems(*values_dict*)
Set the values of multiple items in the store

Parameters

values_dict: dict(str, bytes)

class fsspec.core.**OpenFile**(*fs*, *path*, *mode='rb'*, *compression=None*, *encoding=None*, *errors=None*, *newline=None*)
File-like object to be used in a context

Can layer (buffered) text-mode and compression over any file-system, which are typically binary-only.

These instances are safe to serialize, as the low-level file object is not created until invoked using `with`.

Parameters

fs: FileSystem The file system to use for opening the file. Should match the interface of `dask.bytes.local.LocalFileSystem`.

path: str Location to open

mode: str like 'rb', optional Mode of the opened file

compression: str or None, optional Compression to apply

encoding: str or None, optional The encoding to use if opened in text mode.

errors: str or None, optional How to handle encoding errors if opened in text mode.

newline: None or str Passed to TextIOWrapper in text mode, how to handle line endings.

Attributes

full_name

Methods

close()	Close all encapsulated file objects
open()	Materialise this as a real open file without context

close()
Close all encapsulated file objects

open()
Materialise this as a real open file without context

The file should be explicitly closed to avoid enclosed file instances persisting. This code-path monkey-patches the file-like objects, so they can close even if the parent OpenFile object has already been deleted; but a with-context is better style.

class fsspec.core.**OpenFiles**(**args*, *mode='rb'*, *fs=None*)
List of OpenFile instances

Can be used in a single context, which opens and closes all of the contained files. Normal list access to get the elements works as normal.

A special case is made for caching filesystems - the files will be down/uploaded together at the start or end of the context, and this may happen concurrently, if the target filesystem supports it.

Methods

append(object, /)	Append object to the end of the list.
clear(/)	Remove all items from list.
copy(/)	Return a shallow copy of the list.
count(value, /)	Return number of occurrences of value.
extend(iterable, /)	Extend list by appending elements from the iterable.
index(value[, start, stop])	Return first index of value.
insert(index, object, /)	Insert object before index.
pop([index])	Remove and return item at index (default last).
remove(value, /)	Remove first occurrence of value.
reverse(/)	Reverse *IN PLACE*.
sort(*[, key, reverse])	Sort the list in ascending order and return None.

class fsspec.core.**BaseCache**(*blocksize*, *fetcher*, *size*)

Pass-though cache: doesn't keep anything, calls every time

Acts as base class for other cachers

Parameters

blocksize: int How far to read ahead in numbers of bytes

fetcher: func Function of the form f(start, end) which gets bytes from remote as specified

size: int How big this file is

fsspec.core.**get_fs_token_paths**(*urlpath*, *mode='rb'*, *num=1*, *name_function=None*, *storage_options=None*, *protocol=None*, *expand=True*)

Filesystem, deterministic token, and paths from a urlpath and options.

Parameters

urlpath: string or iterable Absolute or relative filepath, URL (may include protocols like `s3://`), or globstring pointing to data.

mode: str, optional Mode in which to open files.

num: int, optional If opening in writing mode, number of files we expect to create.

name_function: callable, optional If opening in writing mode, this callable is used to generate path names. Names are generated for each partition by `urlpath.replace('*', name_function(partition_index))`.

storage_options: dict, optional Additional keywords to pass to the filesystem class.

protocol: str or None To override the protocol specifier in the URL

expand: bool Expand string paths for writing, assuming the path is a directory

fsspec.core.**url_to_fs**(*url*, ***kwargs*)

Turn fully-qualified and potentially chained URL into filesystem instance

Parameters

url [str] The fsspec-compatible URL

****kwargs: dict** Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc.

Returns

filesystem [FileSystem] The new filesystem discovered from `url` and created with `**kwargs`.

urlpath [str] The file-systems-specific URL for `url`.

class fsspec.dircache.**DirCache**(*use_listings_cache=True*, *listings_expiry_time=None*, *max_paths=None*, ***kwargs*)

Caching of directory listings, in a structure like:

```
{"path0": [
    {"name": "path0/file0",
     "size": 123,
     "type": "file",
     ...
    },
    {"name": "path0/file1",
    },
    ...
    ],
 "path1": [...]
}
```

Parameters to this class control listing expiry or indeed turn caching off

__init__(*use_listings_cache=True*, *listings_expiry_time=None*, *max_paths=None*, ***kwargs*)

Parameters

use_listings_cache: bool If False, this cache never returns items, but always reports KeyError, and setting items has no effect

listings_expiry_time: int (optional) Time in seconds that a listing is considered valid. If None, listings do not expire.

max_paths: int (optional) The number of most recent listings that are considered valid; 'recent' refers to when the entry was set.

class fsspec.registry.**ReadOnlyRegistry**(*target*)

Dict-like registry, but immutable

Maps backend name to implementation class

To add backend implementations, use `register_implementation`

__init__(*target*)

fsspec.registry.**register_implementation**(*name*, *cls*, *clobber=True*, *errtxt=None*)

Add implementation class to the registry

Parameters

name: str Protocol name to associate with the class

cls: class or str if a class: fsspec-compliant implementation class (normally inherits from `fsspec.AbstractFileSystem`, gets added straight to the registry. If a str, the

full path to an implementation class like package.module.class, which gets added to known_implementations, so the import is deferred until the filesystem is actually used.

clobber: bool (optional) Whether to overwrite a protocol with the same name; if False, will raise instead.

errtxt: str (optional) If given, then a failure to import the given class will result in this text being given.

`class fsspec.callbacks.Callback(size=None, value=0, hooks=None, **kwargs)`

Base class and interface for callback mechanism

This class can be used directly for monitoring file transfers by providing `callback=Callback(hooks=...)` (see the `hooks` argument, below), or subclassed for more specialised behaviour.

Parameters

size: int (optional) Nominal quantity for the value that corresponds to a complete transfer, e.g., total number of tiles or total number of bytes

value: int (0) Starting internal counter value

hooks: dict or None A dict of named functions to be called on each update. The signature of these must be `f(size, value, **kwargs)`

Methods

absolute_update(value)	Set the internal value state
as_callback([maybe_callback])	Transform callback=.
branch(path_1, path_2, kwargs)	Set callbacks for child transfers
call([hook_name])	Execute hook(s) with current state
relative_update([inc])	Delta increment the internal cuonter
set_size(size)	Set the internal maximum size attribute
wrap(iterable)	Wrap an iterable to call `relative_update` on each iterations

no_op	

`absolute_update(value)`

Set the internal value state

Triggers `call()`

Parameters

value: int

`classmethod as_callback(maybe_callback=None)`

Transform callback=... into Callback instance

For the special value of `None`, return the global instance of `NoOpCallback`. This is an alternative to including `callback=_DEFAULT_CALLBACK` directly in a method signature.

`branch(path_1, path_2, kwargs)`

Set callbacks for child transfers

If this callback is operating at a higher level, e.g., put, which may trigger transfers that can also be monitored. The passed kwargs are to be *mutated* to add `callback=`, if this class supports branching to children.

Parameters

path_1: str Child's source path

path_2: str Child's destination path

kwargs: dict arguments passed to child method, e.g., put_file.

`call`(*hook_name=None*, ***kwargs*)
Execute hook(s) with current state

Each function is passed the internal size and current value

Parameters

hook_name: str or None If given, execute on this hook

kwargs: passed on to (all) hook(s)

`relative_update`(*inc=1*)
Delta increment the internal cuonter

Triggers `call()`

Parameters

inc: int

`set_size`(*size*)
Set the internal maximum size attribute

Usually called if not initially set at instantiation. Note that this triggers a `call()`.

Parameters

size: int

`wrap`(*iterable*)
Wrap an iterable to call `relative_update` on each iterations

Parameters

iterable: Iterable The iterable that is being wrapped

`class fsspec.callbacks.`**`NoOpCallback`**(*size=None*, *value=0*, *hooks=None*, ***kwargs*)
This implementation of Callback does exactly nothing

Methods

`absolute_update`(value)	Set the internal value state
`as_callback`([maybe_callback])	Transform callback=.
`branch`(path_1, path_2, kwargs)	Set callbacks for child transfers
`call`(*args, **kwargs)	Execute hook(s) with current state
`relative_update`([inc])	Delta increment the internal cuonter
`set_size`(size)	Set the internal maximum size attribute
`wrap`(iterable)	Wrap an iterable to call `relative_update` on each iterations

no_op	

call(**args*, ***kwargs*)
Execute hook(s) with current state

Each function is passed the internal size and current value

Parameters

hook_name: str or None If given, execute on this hook

kwargs: passed on to (all) hook(s)

class fsspec.callbacks.**DotPrinterCallback**(*chr_to_print='#'*, ***kwargs*)
Simple example Callback implementation

Almost identical to Callback with a hook that prints a char; here we demonstrate how the outer layer may print "#" and the inner layer "."

Methods

absolute_update(value)	Set the internal value state
as_callback([maybe_callback])	Transform callback=.
branch(path_1, path_2, kwargs)	Mutate kwargs to add new instance with different print char
call(**kwargs)	Just outputs a character
relative_update([inc])	Delta increment the internal cuonter
set_size(size)	Set the internal maximum size attribute
wrap(iterable)	Wrap an iterable to call `relative_update` on each iterations

no_op	

branch(*path_1*, *path_2*, *kwargs*)
Mutate kwargs to add new instance with different print char

call(***kwargs*)
Just outputs a character

4.6.3 Built-in Implementations

fsspec.implementations.ftp.FTPFileSystem(...)	A filesystem over classic
fsspec.implementations.hdfs.PyArrowHDFS(...)	Adapted version of Arrow's HadoopFileSystem
fsspec.implementations.arrow.ArrowFSWrapper(...)	FSSpec-compatible wrapper of pyarrow.fs.FileSystem.
fsspec.implementations.arrow.HadoopFileSystem(...)	A wrapper on top of the pyarrow.fs.HadoopFileSystem to connect it's interface with fsspec
fsspec.implementations.dask.DaskWorkerFileSystem(...)	View files accessible to a worker as any other remote file-system
fsspec.implementations.http.HTTPFileSystem(...)	Simple File-System for fetching data via HTTP(S)
fsspec.implementations.local.LocalFileSystem(...)	Interface to files on local storage

continues on next page

Table 16 – continued from previous page

fsspec.implementations.memory.MemoryFileSystem(...)	A filesystem based on a dict of BytesIO objects
fsspec.implementations.github.GithubFileSystem(...)	Interface to files in github
fsspec.implementations.sftp.SFTPFileSystem(...)	Files over SFTP/SSH
fsspec.implementations.webhdfs.WebHDFS(...)	Interface to HDFS over HTTP using the WebHDFS API.
fsspec.implementations.zip.ZipFileSystem(...)	Read contents of ZIP archive as a file-system
fsspec.implementations.cached.CachingFileSystem(...)	Locally caching filesystem, layer over any other FS
fsspec.implementations.cached.WholeFileCacheFileSystem(...)	Caches whole remote files on first access
fsspec.implementations.cached.SimpleCacheFileSystem(...)	Caches whole remote files on first access
fsspec.implementations.git.GitFileSystem(...)	Browse the files of a local git repo at any hash/tag/branch
fsspec.implementations.jupyter.JupyterFileSystem(...)	View of the files as seen by a Jupyter server (notebook or lab)
fsspec.implementations.libarchive.LibArchiveFileSystem(...)	Compressed archives as a file-system (read-only)
fsspec.implementations.dbfs.DatabricksFileSystem(...)	Get access to the Databricks filesystem implementation over HTTP.
fsspec.implementations.reference.ReferenceFileSystem(...)	View byte ranges of some other file as a file system

class fsspec.implementations.ftp.**FTPFileSystem**(**args*, ***kwargs*)

A filesystem over classic

__init__(*host*, *port=21*, *username=None*, *password=None*, *acct=None*, *block_size=None*, *tempdir='/tmp'*, *timeout=30*, ***kwargs*)

You can use _get_kwargs_from_urls to get some kwargs from a reasonable FTP url.

Authentication will be anonymous if username/password are not given.

Parameters

host: str The remote server name/ip to connect to

port: int Port to connect with

username: str or None If authenticating, the user's identifier

password: str of None User's password on the server, if using

acct: str or None Some servers also need an "account" string for auth

block_size: int or None If given, the read-ahead or write buffer size.

tempdir: str Directory on remote to put temporary files when in a transaction

timeout: int Timeout of the ftp connection in seconds

class fsspec.implementations.hdfs.**PyArrowHDFS**(**args*, ***kwargs*)

Adapted version of Arrow's HadoopFileSystem

This is a very simple wrapper over the pyarrow.hdfs.HadoopFileSystem, which passes on all calls to the underlying class.

__init__(*host='default'*, *port=0*, *user=None*, *kerb_ticket=None*, *driver='libhdfs'*, *extra_conf=None*, ***kwargs*)

Parameters

host: str Hostname, IP or "default" to try to read from Hadoop config

port: int Port to connect on, or default from Hadoop config if 0

user: str or None If given, connect as this username

kerb_ticket: str or None If given, use this ticket for authentication

driver: 'libhdfs' or 'libhdfs3' Binary driver; libhdfs if the JNI library and default

extra_conf: None or dict Passed on to HadoopFileSystem

class fsspec.implementations.arrow.**ArrowFSWrapper**(**args*, ***kwargs*)

FSSpec-compatible wrapper of pyarrow.fs.FileSystem.

Parameters

fs [pyarrow.fs.FileSystem]

__init__(*fs*, ***kwargs*)

Create and configure file-system instance

Instances may be cachable, so if similar enough arguments are seen a new instance is not required. The token attribute exists to allow implementations to cache instances if they wish.

A reasonable default should be provided if there are no arguments.

Subclasses should call this method.

Parameters

use_listings_cache, listings_expiry_time, max_paths: passed to DirCache, if the implementation supports directory listing caching. Pass use_listings_cache=False to disable such caching.

skip_instance_cache: bool If this is a cachable implementation, pass True here to force creating a new instance even if a matching instance exists, and prevent storing this instance.

asynchronous: bool

loop: asyncio-compatible IOLoop or None

fsspec.implementations.hdfs.**HadoopFileSystem**

alias of pyarrow.hdfs.

class fsspec.implementations.dask.**DaskWorkerFileSystem**(**args*, ***kwargs*)

View files accessible to a worker as any other remote file-system

When instances are run on the worker, uses the real filesystem. When run on the client, they call the worker to provide information or data.

Warning this implementation is experimental, and read-only for now.

__init__(*target_protocol=None*, *target_options=None*, *fs=None*, *client=None*, ***kwargs*)

Create and configure file-system instance

Instances may be cachable, so if similar enough arguments are seen a new instance is not required. The token attribute exists to allow implementations to cache instances if they wish.

A reasonable default should be provided if there are no arguments.

Subclasses should call this method.

Parameters

use_listings_cache, listings_expiry_time, max_paths: passed to `DirCache`, if the implementation supports directory listing caching. Pass use_listings_cache=False to disable such caching.

skip_instance_cache: bool If this is a cachable implementation, pass True here to force creating a new instance even if a matching instance exists, and prevent storing this instance.

asynchronous: bool

loop: asyncio-compatible IOLoop or None

`class fsspec.implementations.http.HTTPFileSystem(*args, **kwargs)`

Simple File-System for fetching data via HTTP(S)

`ls()` is implemented by loading the parent page and doing a regex match on the result. If simple_link=True, anything of the form "http(s)://server.com/stuff?thing=other"; otherwise only links within HTML href tags will be used.

`__init__(simple_links=True, block_size=None, same_scheme=True, size_policy=None, cache_type='bytes', cache_options=None, asynchronous=False, loop=None, client_kwargs=None, get_client=<function get_client>, **storage_options)`

NB: if this is called async, you must await set_client

Parameters

block_size: int Blocks to read bytes; if 0, will default to raw requests file-like objects instead of HTTPFile instances

simple_links: bool If True, will consider both HTML <a> tags and anything that looks like a URL; if False, will consider only the former.

same_scheme: True When doing ls/glob, if this is True, only consider paths that have http/https matching the input URLs.

size_policy: this argument is deprecated

client_kwargs: dict Passed to aiohttp.ClientSession, see https://docs.aiohttp.org/en/stable/client_reference.html For example, `{'auth': aiohttp.BasicAuth('user', 'pass')}`

get_client: Callable[..., aiohttp.ClientSession] A callable which takes keyword arguments and constructs an aiohttp.ClientSession. It's state will be managed by the HTTPFileSystem class.

storage_options: key-value Any other parameters passed on to requests

cache_type, cache_options: defaults used in open

`class fsspec.implementations.local.LocalFileSystem(*args, **kwargs)`

Interface to files on local storage

Parameters

auto_mkdirs: bool Whether, when opening a file, the directory containing it should be created (if it doesn't already exist). This is assumed by pyarrow code.

`__init__(auto_mkdir=False, **kwargs)`

Create and configure file-system instance

Instances may be cachable, so if similar enough arguments are seen a new instance is not required. The token attribute exists to allow implementations to cache instances if they wish.

A reasonable default should be provided if there are no arguments.

Subclasses should call this method.

Parameters

use_listings_cache, listings_expiry_time, max_paths: passed to `DirCache`, if the implementation supports directory listing caching. Pass use_listings_cache=False to disable such caching.

skip_instance_cache: bool If this is a cachable implementation, pass True here to force creating a new instance even if a matching instance exists, and prevent storing this instance.

asynchronous: bool

loop: asyncio-compatible IOLoop or None

`class fsspec.implementations.memory.MemoryFileSystem(*args, **kwargs)`

A filesystem based on a dict of BytesIO objects

This is a global filesystem so instances of this class all point to the same in memory filesystem.

`__init__(*args, **storage_options)`

Create and configure file-system instance

Instances may be cachable, so if similar enough arguments are seen a new instance is not required. The token attribute exists to allow implementations to cache instances if they wish.

A reasonable default should be provided if there are no arguments.

Subclasses should call this method.

Parameters

use_listings_cache, listings_expiry_time, max_paths: passed to `DirCache`, if the implementation supports directory listing caching. Pass use_listings_cache=False to disable such caching.

skip_instance_cache: bool If this is a cachable implementation, pass True here to force creating a new instance even if a matching instance exists, and prevent storing this instance.

asynchronous: bool

loop: asyncio-compatible IOLoop or None

`class fsspec.implementations.sftp.SFTPFileSystem(*args, **kwargs)`

Files over SFTP/SSH

Peer-to-peer filesystem over SSH using paramiko.

Note: if using this with the `open` or `open_files`, with full URLs, there is no way to tell if a path is relative, so all paths are assumed to be absolute.

`__init__(host, **ssh_kwargs)`

Parameters

host: str Hostname or IP as a string

temppath: str Location on the server to put files, when within a transaction

ssh_kwargs: dict Parameters passed on to connection. See details in http://docs.paramiko.org/en/2.4/api/client.html#paramiko.client.SSHClient.connect May include port, username, password...

class fsspec.implementations.webhdfs.**WebHDFS**(**args*, ***kwargs*)
Interface to HDFS over HTTP using the WebHDFS API. Supports also HttpFS gateways.

Three auth mechanisms are supported:

insecure: no auth is done, and the user is assumed to be whoever they say they are (parameter `user`), or a predefined value such as "dr.who" if not given

spnego: when kerberos authentication is enabled, auth is negotiated by requests_kerberos https://github.com/requests/requests-kerberos . This establishes a session based on existing kinit login and/or specified principal/password; paraneters are passed with `kerb_kwargs`

token: uses an existing Hadoop delegation token from another secured service. Indeed, this client can also generate such tokens when not insecure. Note that tokens expire, but can be renewed (by a previously specified user) and may allow for proxying.

__init__(*host*, *port=50070*, *kerberos=False*, *token=None*, *user=None*, *proxy_to=None*, *kerb_kwargs=None*, *data_proxy=None*, *use_https=False*, ***kwargs*)

Parameters

host: str Name-node address

port: int Port for webHDFS

kerberos: bool Whether to authenticate with kerberos for this connection

token: str or None If given, use this token on every call to authenticate. A user and user-proxy may be encoded in the token and should not be also given

user: str or None If given, assert the user name to connect with

proxy_to: str or None If given, the user has the authority to proxy, and this value is the user in who's name actions are taken

kerb_kwargs: dict Any extra arguments for HTTPKerberosAuth, see https://github.com/requests/requests-kerberos/blob/master/requests_kerberos/kerberos_.py

data_proxy: dict, callable or None If given, map data-node addresses. This can be necessary if the HDFS cluster is behind a proxy, running on Docker or otherwise has a mismatch between the host-names given by the name-node and the address by which to refer to them from the client. If a dict, maps host names `host->data_proxy[host]`; if a callable, full URLs are passed, and function must conform to `url->data_proxy(url)`.

use_https: bool Whether to connect to the Name-node using HTTPS instead of HTTP

kwargs

class fsspec.implementations.zip.**ZipFileSystem**(**args*, ***kwargs*)
Read contents of ZIP archive as a file-system

Keeps file object open while instance lives.

This class is pickleable, but not necessarily thread-safe

__init__(*fo=''*, *mode='r'*, *target_protocol=None*, *target_options=None*, *block_size=5242880*, ***kwargs*)

Parameters

fo: str or file-like Contains ZIP, and must exist. If a str, will fetch file using `open_files()`, which must return one file exactly.

mode: str Currently, only 'r' accepted

target_protocol: str (optional) If `fo` is a string, this value can be used to override the FS protocol inferred from a URL

target_options: dict (optional) Kwargs passed when instantiating the target FS, if `fo` is a string.

class fsspec.implementations.cached.**CachingFileSystem**(**args*, ***kwargs*)

Locally caching filesystem, layer over any other FS

This class implements chunk-wise local storage of remote files, for quick access after the initial download. The files are stored in a given directory with random hashes for the filenames. If no directory is given, a temporary one is used, which should be cleaned up by the OS after the process ends. The files themselves as sparse (as implemented in MMapCache), so only the data which is accessed takes up space.

Restrictions:

- the block-size must be the same for each access of a given file, unless all blocks of the file have already been read
- caching can only be applied to file-systems which produce files derived from fsspec.spec.AbstractBufferedFile ; LocalFileSystem is also allowed, for testing

__init__(*target_protocol=None*, *cache_storage='TMP'*, *cache_check=10*, *check_files=False*, *expiry_time=604800*, *target_options=None*, *fs=None*, *same_names=False*, *compression=None*, ***kwargs*)

Parameters

target_protocol: str (optional) Target filesystem protocol. Provide either this or `fs`.

cache_storage: str or list(str) Location to store files. If "TMP", this is a temporary directory, and will be cleaned up by the OS when this process ends (or later). If a list, each location will be tried in the order given, but only the last will be considered writable.

cache_check: int Number of seconds between reload of cache metadata

check_files: bool Whether to explicitly see if the UID of the remote file matches the stored one before using. Warning: some file systems such as HTTP cannot reliably give a unique hash of the contents of some path, so be sure to set this option to False.

expiry_time: int The time in seconds after which a local copy is considered useless. Set to falsy to prevent expiry. The default is equivalent to one week.

target_options: dict or None Passed to the instantiation of the FS, if fs is None.

fs: filesystem instance The target filesystem to run against. Provide this or `protocol`.

same_names: bool (optional) By default, target URLs are hashed, so that files from different backends with the same basename do not conflict. If this is true, the original basename is used.

compression: str (optional) To decompress on download. Can be 'infer' (guess from the URL name), one of the entries in `fsspec.compression.compr`, or None for no decompression.

class fsspec.implementations.cached.**WholeFileCacheFileSystem**(**args*, ***kwargs*)

Caches whole remote files on first access

This class is intended as a layer over any other file system, and will make a local copy of each file accessed, so that all subsequent reads are local. This is similar to `CachingFileSystem`, but without the block-wise functionality and so can work even when sparse files are not allowed. See its docstring for definition of the init arguments.

The class still needs access to the remote store for listing files, and may refresh cached files.

__init__(*target_protocol=None*, *cache_storage='TMP'*, *cache_check=10*, *check_files=False*, *expiry_time=604800*, *target_options=None*, *fs=None*, *same_names=False*, *compression=None*, ***kwargs*)

Parameters

target_protocol: str (optional) Target filesystem protocol. Provide either this or `fs`.

cache_storage: str or list(str) Location to store files. If "TMP", this is a temporary directory, and will be cleaned up by the OS when this process ends (or later). If a list, each location will be tried in the order given, but only the last will be considered writable.

cache_check: int Number of seconds between reload of cache metadata

check_files: bool Whether to explicitly see if the UID of the remote file matches the stored one before using. Warning: some file systems such as HTTP cannot reliably give a unique hash of the contents of some path, so be sure to set this option to False.

expiry_time: int The time in seconds after which a local copy is considered useless. Set to falsy to prevent expiry. The default is equivalent to one week.

target_options: dict or None Passed to the instantiation of the FS, if fs is None.

fs: filesystem instance The target filesystem to run against. Provide this or `protocol`.

same_names: bool (optional) By default, target URLs are hashed, so that files from different backends with the same basename do not conflict. If this is true, the original basename is used.

compression: str (optional) To decompress on download. Can be 'infer' (guess from the URL name), one of the entries in `fsspec.compression.compr`, or None for no decompression.

class fsspec.implementations.cached.**SimpleCacheFileSystem**(**args*, ***kwargs*)

Caches whole remote files on first access

This class is intended as a layer over any other file system, and will make a local copy of each file accessed, so that all subsequent reads are local. This implementation only copies whole files, and does not keep any metadata about the download time or file details. It is therefore safer to use in multi-threaded/concurrent situations.

This is the only of the caching filesystems that supports write: you will be given a real local open file, and upon close and commit, it will be uploaded to the target filesystem; the writability or the target URL is not checked until that time.

__init__(***kwargs*)

Parameters

target_protocol: str (optional) Target filesystem protocol. Provide either this or `fs`.

cache_storage: str or list(str) Location to store files. If "TMP", this is a temporary directory, and will be cleaned up by the OS when this process ends (or later). If a list, each location will be tried in the order given, but only the last will be considered writable.

cache_check: int Number of seconds between reload of cache metadata

check_files: bool Whether to explicitly see if the UID of the remote file matches the stored one before using. Warning: some file systems such as HTTP cannot reliably give a unique hash of the contents of some path, so be sure to set this option to False.

expiry_time: int The time in seconds after which a local copy is considered useless. Set to falsy to prevent expiry. The default is equivalent to one week.

target_options: dict or None Passed to the instantiation of the FS, if fs is None.

fs: filesystem instance The target filesystem to run against. Provide this or `protocol`.

same_names: bool (optional) By default, target URLs are hashed, so that files from different backends with the same basename do not conflict. If this is true, the original basename is used.

compression: str (optional) To decompress on download. Can be 'infer' (guess from the URL name), one of the entries in `fsspec.compression.compr`, or None for no decompression.

class fsspec.implementations.github.**GithubFileSystem**(**args*, ***kwargs*)

Interface to files in github

An instance of this class provides the files residing within a remote github repository. You may specify a point in the repos history, by SHA, branch or tag (default is current master).

Given that code files tend to be small, and that github does not support retrieving partial content, we always fetch whole files.

When using fsspec.open, allows URIs of the form:

- "github://path/file", in which case you must specify org, repo and may specify sha in the extra args
- 'github://org:repo@/precip/catalog.yml', where the org and repo are part of the URI
- 'github://org:repo@sha/precip/catalog.yml', where tha sha is also included

`sha` can be the full or abbreviated hex of the commit you want to fetch from, or a branch or tag name (so long as it doesn't contain special characters like "/", "?", which would have to be HTTP-encoded).

For authorised access, you must provide username and token, which can be made at https://github.com/settings/tokens

__init__(*org*, *repo*, *sha=None*, *username=None*, *token=None*, ***kwargs*)

Create and configure file-system instance

Instances may be cachable, so if similar enough arguments are seen a new instance is not required. The token attribute exists to allow implementations to cache instances if they wish.

A reasonable default should be provided if there are no arguments.

Subclasses should call this method.

Parameters

use_listings_cache, listings_expiry_time, max_paths: passed to `DirCache`, if the implementation supports directory listing caching. Pass use_listings_cache=False to disable such caching.

skip_instance_cache: bool If this is a cachable implementation, pass True here to force creating a new instance even if a matching instance exists, and prevent storing this instance.

asynchronous: bool

loop: asyncio-compatible IOLoop or None

class fsspec.implementations.git.**GitFileSystem**(**args*, ***kwargs*)

Browse the files of a local git repo at any hash/tag/branch

(experimental backend)

__init__(*path=None*, *ref=None*, ***kwargs*)

Parameters

path: str (optional) Local location of the repo (uses current directory if not given)

ref: str (optional) Reference to work with, could be a hash, tag or branch name. Defaults to current working tree. Note that `ls` and `open` also take hash, so this becomes the default for those operations

kwargs

class fsspec.implementations.jupyter.**JupyterFileSystem**(**args*, ***kwargs*)

View of the files as seen by a Jupyter server (notebook or lab)

__init__(*url*, *tok=None*, ***kwargs*)

Parameters

url [str] Base URL of the server, like "http://127.0.0.1:8888". May include token in the string, which is given by the process when starting up

tok [str] If the token is obtained separately, can be given here

kwargs

class fsspec.implementations.libarchive.**LibArchiveFileSystem**(**args*, ***kwargs*)

Compressed archives as a file-system (read-only)

Supports the following formats: tar, pax , cpio, ISO9660, zip, mtree, shar, ar, raw, xar, lha/lzh, rar Microsoft CAB, 7-Zip, WARC

See the libarchive documentation for further restrictions. https://www.libarchive.org/

Keeps file object open while instance lives. It only works in seekable file-like objects. In case the filesystem does not support this kind of file object, it is recommended to cache locally.

This class is pickleable, but not necessarily thread-safe (depends on the platform). See libarchive documentation for details.

__init__(*fo=''*, *mode='r'*, *target_protocol=None*, *target_options=None*, *block_size=5242880*, ***kwargs*)

Parameters

fo: str or file-like Contains ZIP, and must exist. If a str, will fetch file using `open_files()`, which must return one file exactly.

mode: str Currently, only 'r' accepted

target_protocol: str (optional) If `fo` is a string, this value can be used to override the FS protocol inferred from a URL

target_options: dict (optional) Kwargs passed when instantiating the target FS, if `fo` is a string.

class fsspec.implementations.dbfs.**DatabricksFileSystem**(**args*, ***kwargs*)

Get access to the Databricks filesystem implementation over HTTP. Can be used inside and outside of a databricks cluster.

__init__(*instance*, *token*, ***kwargs*)

Create a new DatabricksFileSystem.

Parameters

instance: str The instance URL of the databricks cluster. For example for an Azure databricks cluster, this has the form adb-<some-number>.<two digits>.azuredatabricks.net.

token: str Your personal token. Find out more here: https://docs.databricks.com/dev-tools/api/latest/authentication.html

class fsspec.implementations.reference.**ReferenceFileSystem**(**args*, ***kwargs*)

View byte ranges of some other file as a file system

Initial version: single file system target, which must support async, and must allow start and end args in _cat_file. Later versions may allow multiple arbitrary URLs for the targets.

This FileSystem is read-only. It is designed to be used with async targets (for now). This FileSystem only allows whole-file access, no `open`. We do not get original file details from the target FS.

Configuration is by passing a dict of references at init, or a URL to a JSON file containing the same; this dict can also contain concrete data for some set of paths.

Reference dict format: {path0: bytes_data, path1: (target_url, offset, size)}

https://github.com/intake/fsspec-reference-maker/blob/main/README.md

__init__(*fo*, *target=None*, *ref_storage_args=None*, *target_protocol=None*, *target_options=None*, *remote_protocol=None*, *remote_options=None*, *fs=None*, *template_overrides=None*, *simple_templates=False*, *loop=None*, *ref_type=None*, ***kwargs*)

Parameters

fo [dict or str] The set of references to use for this instance, with a structure as above. If str, will use fsspec.open, in conjunction with ref_storage_args to open and parse JSON at this location.

target [str] For any references having target_url as None, this is the default file target to use

ref_storage_args [dict] If references is a str, use these kwargs for loading the JSON file

target_protocol [str] Used for loading the reference file, if it is a path. If None, protocol will be derived from the given path

target_options [dict] Extra FS options for loading the reference file, if given as a path

remote_protocol [str] The protocol of the filesystem on which the references will be evaluated (unless fs is provided)

remote_options [dict] kwargs to go with remote_protocol

fs [file system instance] Directly provide a file system, if you want to configure it beforehand. This takes precedence over target_protocol/target_options

template_overrides [dict] Swap out any templates in the references file with these - useful for testing.

ref_type ["json" | "parquet" | "zarr"] If None, guessed from URL suffix, defaulting to JSON. Ignored if fo is not a string.

simple_templates: bool

kwargs [passed to parent class]

4.6.4 Other Known Implementations

- s3fs for Amazon S3 and other compatible stores
- gcsfs for Google Cloud Storage
- adl for Azure DataLake storage
- abfs for Azure Blob service
- dropbox for access to dropbox shares
- ocifs for access to Oracle Cloud Object Storage
- gdrive to access Google Drive and shares (experimental)
- wandbfs to access Wandb run data (experimental)

4.6.5 Read Buffering

fsspec.caching.ReadAheadCache(blocksize, ...)	Cache which reads only when we get beyond a block of data
fsspec.caching.BytesCache(blocksize, ...[, trim])	Cache which holds data in a in-memory bytes object
fsspec.caching.MMapCache(blocksize, fetcher, ...)	memory-mapped sparse file cache
fsspec.caching.BlockCache(blocksize, ...[, ...])	Cache holding memory as a set of blocks.

class fsspec.caching.**ReadAheadCache**(*blocksize*, *fetcher*, *size*)
Cache which reads only when we get beyond a block of data

This is a much simpler version of BytesCache, and does not attempt to fill holes in the cache or keep fragments alive. It is best suited to many small reads in a sequential order (e.g., reading lines from a file).

class fsspec.caching.**BytesCache**(*blocksize*, *fetcher*, *size*, *trim=True*)
Cache which holds data in a in-memory bytes object

Implements read-ahead by the block size, for semi-random reads progressing through the file.

Parameters

trim: bool As we read more data, whether to discard the start of the buffer when we are more than a blocksize ahead of it.

class fsspec.caching.**MMapCache**(*blocksize*, *fetcher*, *size*, *location=None*, *blocks=None*)
memory-mapped sparse file cache

Opens temporary file, which is filled blocks-wise when data is requested. Ensure there is enough disc space in the temporary location.

This cache method might only work on posix

class fsspec.caching.**BlockCache**(*blocksize*, *fetcher*, *size*, *maxblocks=32*)
Cache holding memory as a set of blocks.

Requests are only ever made `blocksize` at a time, and are stored in an LRU cache. The least recently accessed block is discarded when more than `maxblocks` are stored.

Parameters

blocksize [int] The number of bytes to store in each block. Requests are only ever made for `blocksize`, so this should balance the overhead of making a request against the granularity of the blocks.

fetcher [Callable]

size [int] The total size of the file being cached.

maxblocks [int] The maximum number of blocks to cache for. The maximum memory use for this cache is then `blocksize * maxblocks`.

Methods

cache_info()	The statistics on the block cache.

cache_info()
The statistics on the block cache.

Returns

NamedTuple Returned directly from the LRU Cache used internally.

4.7 Changelog

4.7.1 2021.10.1

Fixes

- Removed inaccurate `ZipFileSystem.cat()` override so that the base class' version is used (#789)
- fix entrypoint processing (#784)
- case where no blocks of a block-cache have yet been loaded (#801)
- don't fetch empty ranges (#802, 803)

Other

- simplify doc deps (#786, 791)

4.7.2 2021.10.0

Fixes

- only close http connector if present (#779)
- hdfs strip protocol (#778)
- fix filecache with check_files (#772)
- put_file to use _parent (#771)

Other

- add kedro link (#781)

4.7.3 2021.09.0

Enhancement

- http put from file-like (#764)
- implement webhdfs cp/rm_file (#762)
- multiple (and concurrent) cat_ranges (#744)

Fixes

- sphinx warnings (#769)
- lexists for links (#757)
- update versioneer (#750)
- hdfs updates (#749)
- propagate async timeout error (#746)
- fix local file seekable (#743)
- fix http isdir when does not exist (#741)

Other

- ocifs, arrow added (#754, #765)
- promote url_to_fs to top level (#753)

4.7.4 2021.08.1

Enhancements

- HTTP get_file/put_file APIs now support callbacks (#731)
- New HTTP put_file method for transferring data to the remote server (chunked) (#731)
- Customizable HTTP client initializers (through passing `get_client` argument) (#731, #701)
- Support for various checksum / fingerprint headers in HTTP `info()` (#731)
- local implementation of rm_file (#736)
- local speed improvements (#711)
- sharing options in SMB (#706)
- streaming cat/get for ftp (#700)

Fixes

- check for remote directory when putting (#737)
- storage_option update handling (#734(
- await HTTP call before checking status (#726)
- ftp connect (#722)
- bytes conversion of times in mapper (#721)
- variable overwrite in WholeFileCache cat (#719)
- http file size again (#718)

- rm and create directories in ftp (#716, #703)
- list of files in async put (#713)
- bytes to dict in cat (#710)

4.7.5 2021.07.0

Enhancements

- callbacks (#697)

4.7.6 2021.06.1

Enhancements

- Introduce `fsspec.asyn.fsspec_loop` to temporarily switch to the fsspec loop. (#671)
- support list for local rm (#678)

Fixes

- error when local mkdir twice (#679)
- fix local info regression for pathlike (#667)

Other

- link to wandbfs (#664)

4.7.7 2021.06.0

Enhancements

- Better testing and folder handling for Memory (#654)
- Negative indexes for cat_file (#653)
- optimize local file listing (#647)

Fixes

- FileNoteFound in http and range exception subclass (#649, 646)
- async timeouts (#643, 645)
- stringify path for pyarrow legacy (#630)

Other

- The `fsspec.asyn.get_loop()` will always return a loop of a selector policy (#658)
- add helper to construct Range headers for cat_file (#655)

4.7.8 2021.05.0

Enhancements

- Enable listings cache for HTTP filesystem (#560)
- Fold ZipFileSystem and LibArchiveFileSystem into a generic implementation and add new TarFileSystem (#561)
- Use throttling for the `get/put` methods of `AsyncFileSystem` (#629)
- rewrite for archive filesystems (#624)
- HTTP listings caching (#623)

Fixes

- gcsfs tests (#638)
- stringify_path for arrow (#630)

Other

- s3a:// alias

4.7.9 2021.04.0

Major changes

- calendar versioning

Enhancements

- better link and size finding for HTTP (#610, %99)
- link following in Local (#608)
- ReferenceFileSystem dev (#606, #604, #602)

Fixes

- drop metadata dep (#605)

4.7.10 0.9.0

Major Changes:

- avoid nested sync calls by copying code (#581, #586, docs #593)
- release again for py36 (#564, #575)

Enhancements:

- logging in mmap cacher, explicitly close files (#559)
- make LocalFileOpener an IOBase (#589)
- better reference file system (#568, #582, #584, #585)
- first-chunk cache (#580)
- sftp listdir (#571)
- http logging and fetch all (#551, #558)
- doc: entry points (#548)

Fixes:

- get_mapper for caching filesystems (#559)
- fix cross-device file move (#547)
- store paths without trailing "/" for DBFS (#557)
- errors that happen on `_initiate_upload` when closing the `AbstractBufferedFile` will now be propagated (#587)
- infer_compressions with upper case suffix ($595)
- file initialiser errors (#587)
- CI fix (#563)
- local file commit cross-device (#547)

4.7.11 Version 0.8.7

Fixes:

- fix error with pyarrow metadata for some pythons (#546)

4.7.12 Version 0.8.6

Features:

- Add dbfs:// support (#504, #514)

Enhancements

- don't import pyarrow (#503)
- update entry points syntax (#515)
- ci precommit hooks (#534)

Fixes:

- random appending of a directory within the filesystems `find()` method (#507, 537)
- fix git tests (#501)
- fix recursive memfs operations (#502)
- fix recorsive/maxdepth for cp (#508)
- fix listings cache timeout (#513)
- big endian bytes tests (#519)
- docs syntax (#535, 524, 520, 542)
- transactions and reads (#533)

4.7.13 Version 0.8.5

Features:

- config system
- libarchive implementation
- add reference file system implementation

4.7.14 Version 0.8.4

Features:

- function `can_be_local` to see whether URL is compatible with `open_local`
- concurrent cat with filecaches, if backend supports it
- jupyter FS

Fixes:

- dircache expiry after transaction
- blockcache garbage collection
- close for HDFS
- windows tests
- glob depth with "**"

4.7.15 Version 0.8.3

Features:

- error options for cat
- memory fs created time in detailed `ls`

Fixes:

- duplicate directories could appear in MemoryFileSystem
- Added support for hat dollar lbrace rbrace regex character escapes in glob
- Fix blockcache (was doing unnecessary work)
- handle multibyte dtypes in readinto
- Fix missing kwargs in call to _copy in asyn

Other:

- Stop inheriting from pyarrow.filesystem for pyarrow>=2.0
- Raise low-level program friendly OSError.
- Guard against instance reuse in new processes
- Make hash_name a method on CachingFileSystem to make it easier to change.
- Use get_event_loop for py3.6 compatibility

4.7.16 Version 0.8.2

Fixes:

- More careful strip for caching

4.7.17 Version 0.8.1

Features:

- add sign to base class
- Allow calling of coroutines from normal code when running async
- Implement writing for cached many files
- Allow concurrent caching of remote files
- Add gdrive:// protocol

Fixes:

- Fix memfs with exact ls
- HTTPFileSystem requires requests and aiohttp in registry

Other:

- Allow http kwargs to clientSession
- Use extras_require in setup.py for optional dependencies
- Replacing md5 with sha256 for hash (CVE req)
- Test against Python 3.8, drop 3.5 testing
- add az alias for abfs

4.7.18 Version 0.8.0

Major release allowing async implementations with concurrent batch operations.

Features:

- async filesystem spec, first applied to HTTP
- OpenFiles cContext for multiple files
- Document async, and ensure docstrings
- Make LocalFileOpener iterable
- handle smb:// protocol using smbprotocol package
- allow Path object in open
- simplecache write mode

Fixes:

- test_local: fix username not in home path
- Tighten cacheFS if dir deleted
- Fix race condition of lzma import when using threads

- properly rewind MemoryFile
- OpenFile newline in reduce

Other:

- Add aiobotocore to deps for s3fs check
- Set default clobber=True on impl register
- Use _get_kwargs_from_url when unchaining
- Add cache_type and cache_options to HTTPFileSystem constructor

4.7.19 Version 0.7.5

- async implemented for HTTP as prototype (read-only)
- write for simplecache
- added SMB (Samba, protocol >=2) implementation

4.7.20 Version 0.7.4

- panel-based GUI

4.7.21 0.7.3 series

- added `git` and `github` interfaces
- added chained syntax for open, open_files and get_mapper
- adapt webHDFS for HttpFS
- added open_local
- added `simplecache`, and compression to both file caches

4.7.22 Version 0.6.2

- Added `adl` and `abfs` protocols to the known implementations registry (GH#209)
- Fixed issue with whole-file caching and implementations providing multiple protocols (GH#219)

4.7.23 Version 0.6.1

- `LocalFileSystem` is now considered a filestore by pyarrow (GH#211)
- Fixed bug in HDFS filesystem with `cache_options` (GH#202)
- Fixed instance caching bug with multiple instances (GH#203)

4.7.24 Version 0.6.0

- Fixed issues with filesystem instance caching. This was causing authorization errors in downstream libraries like `gcsfs` and `s3fs` in multi-threaded code (GH#155, GH#181)
- Changed the default file caching strategy to *fsspec.caching.ReadAheadCache* (GH#193)
- Moved file caches to the new `fsspec.caching` module. They're still available from their old location in `fsspec.core`, but we recommend using the new location for new code (GH#195)
- Added a new file caching strategy, *fsspec.caching.BlockCache* for fetching and caching file reads in blocks (GH#191).
- Fixed equality checks for file system instance to return `False` when compared to objects other than file systems (GH#192)
- Fixed a bug in `fsspec.FSMap.keys()` returning a generator, which was consumed upon iteration (GH#189).
- Removed the magic addition of aliases in `AbstractFileSystem.__init__`. Now alias methods are always present (GH#177)
- Deprecated passing `trim` to *fsspec.spec.AbstractBufferedFile*. Pass it in `storage_options` instead (GH#188)
- Improved handling of requests for *fsspec.implementations.http.HTTPFileSystem* when the HTTP server responds with an (incorrect) content-length of 0 (GH#163)
- Added a `detail=True` parameter to *fsspec.spec.AbstractFileSystem.ls()* (GH#168)
- Fixed handling of UNC/DFS paths (GH#154)

INDEX

Symbols

A

B

C

O

P

R

S

T

U

W

Z