
S3Fs Documentation

Release 0.4.0+2.g40a79bb

Continuum Analytics

Nov 19, 2019

Contents

| | | |
|----------|---------------------------------|-----------|
| 1 | Examples | 3 |
| 2 | Limitations | 5 |
| 3 | Logging | 7 |
| 4 | Credentials | 9 |
| 5 | Requester Pays Buckets | 11 |
| 6 | Serverside Encryption | 13 |
| 7 | Bucket Version Awareness | 15 |
| 8 | Contents | 17 |
| 8.1 | Installation | 17 |
| 8.2 | API | 17 |
| 8.3 | Changelog | 26 |
| 9 | Indices and tables | 27 |
| | Index | 29 |

S3Fs is a Pythonic file interface to S3. It builds on top of [boto3](#).

The top-level class `S3FileSystem` holds connection information and allows typical file-system style operations like `cp`, `mv`, `ls`, `du`, `glob`, etc., as well as `put/get` of local files to/from S3.

The connection can be anonymous - in which case only publicly-available, read-only buckets are accessible - or via credentials explicitly supplied or in configuration files.

Calling `open()` on a `S3FileSystem` (typically using a context manager) provides an `S3File` for read or write access to a particular key. The object emulates the standard `File` protocol (`read`, `write`, `tell`, `seek`), such that functions expecting a file can access S3. Only binary read and write modes are implemented, with blocked caching.

S3Fs uses and is based upon [fsspec](#).

CHAPTER 1

Examples

Simple locate and read a file:

```
>>> import s3fs
>>> fs = s3fs.S3FileSystem(anon=True)
>>> fs.ls('my-bucket')
['my-file.txt']
>>> with fs.open('my-bucket/my-file.txt', 'rb') as f:
...     print(f.read())
b'Hello, world'
```

(see also walk and glob)

Reading with delimited blocks:

```
>>> s3.read_block(path, offset=1000, length=10, delimiter=b'\n')
b'A whole line of text\n'
```

Writing with blocked caching:

```
>>> s3 = s3fs.S3FileSystem(anon=False) # uses default credentials
>>> with s3.open('mybucket/new-file', 'wb') as f:
...     f.write(2*2**20 * b'a')
...     f.write(2*2**20 * b'a') # data is flushed and file closed
>>> s3.du('mybucket/new-file')
{'mybucket/new-file': 4194304}
```

Because S3Fs faithfully copies the Python file interface it can be used smoothly with other projects that consume the file interface like gzip or pandas.

```
>>> with s3.open('mybucket/my-file.csv.gz', 'rb') as f:
...     g = gzip.GzipFile(fileobj=f) # Decompress data with gzip
...     df = pd.read_csv(g)         # Read CSV file with Pandas
```


CHAPTER 2

Limitations

This project is meant for convenience, rather than feature completeness. The following are known current omissions:

- file access is always binary (although `readline` and iterating by line are possible)
- no permissions/access-control (i.e., no `chmod/chown` methods)

CHAPTER 3

Logging

The logger `s3fs.core.logger` provides information about the operations of the file system. To see messages, set its level to `DEBUG`. You can also achieve this via an environment variable `S3FS_LOGGING_LEVEL=DEBUG`.

The AWS key and secret may be provided explicitly when creating an `S3FileSystem`. A more secure way, not including the credentials directly in code, is to allow boto to establish the credentials automatically. Boto will try the following methods, in order:

- `aws_access_key_id`, `aws_secret_access_key`, and `aws_session_token` environment variables
- configuration files such as `~/.aws/credentials`
- for nodes on EC2, the IAM metadata provider

In a distributed environment, it is not expected that raw credentials should be passed between machines. In the explicitly provided credentials case, the method `get_delegated_s3pars()` can be used to obtain temporary credentials. When not using explicit credentials, it should be expected that every machine also has the appropriate environment variables, config files or IAM roles available.

If none of the credential methods are available, only anonymous access will work, and `anon=True` must be passed to the constructor.

Furthermore, `S3FileSystem.current()` will return the most-recently created instance, so this method could be used in preference to the constructor in cases where the code must be agnostic of the credentials/config used.

Requester Pays Buckets

Some buckets, such as the [arXiv raw data](#), are configured so that the requester of the data pays any transfer fees. You must be authenticated to access these buckets and (because these charges may be unexpected) amazon requires an additional key on many of the API calls. To enable `RequesterPays` create your file system as

```
>>> s3 = s3fs.S3FileSystem(anon=False, requester_pays=True)
```

Serverside Encryption

For some buckets/files you may want to use some of s3's server side encryption features. `s3fs` supports these in a few ways

```
>>> s3 = s3fs.S3FileSystem(  
...     s3_additional_kwargs={'ServerSideEncryption': 'AES256'})
```

This will create an s3 filesystem instance that will append the `ServerSideEncryption` argument to all s3 calls (where applicable).

The same applies for `s3.open`. Most of the methods on the filesystem object will also accept and forward keyword arguments to the underlying calls. The most recently specified argument is applied last in the case where both `s3_additional_kwargs` and a method's `**kwargs` are used.

The `s3.utils.SSEParams` provides some convenient helpers for the serverside encryption parameters in particular. An instance can be passed instead of a regular python dictionary as the `s3_additional_kwargs` parameter.

Bucket Version Awareness

If your bucket has object versioning enabled then you can add version-aware support to `s3fs`. This ensures that if a file is opened at a particular point in time that version will be used for reading.

This mitigates the issue where more than one user is concurrently reading and writing to the same object.

```
>>> s3 = s3fs.S3FileSystem(version_aware=True)
# Open the file at the latest version
>>> fo = s3.open('versioned_bucket/object')
>>> versions = s3.object_version_info('versioned_bucket/object')
# Open the file at a particular version
>>> fo_old_version = s3.open('versioned_bucket/object', version_id='SOMEVERSIONID')
```

In order for this to function the user must have the necessary IAM permissions to perform a `GetObjectVersion`

8.1 Installation

8.1.1 Conda

The `s3fs` library and its dependencies can be installed from the [conda-forge](#) repository using `conda`:

```
$ conda install s3fs -c conda-forge
```

8.1.2 PyPI

You can install `s3fs` with `pip`:

```
pip install s3fs
```

8.1.3 Install from source

You can also download the `s3fs` library from Github and install normally:

```
git clone git@github.com:dask/s3fs
cd s3fs
python setup.py install
```

8.2 API

`S3FileSystem`([anon, key, secret, token, ...])

Access S3 as if it were a file system.

Continued on next page

Table 1 – continued from previous page

| | |
|--|--|
| <code>S3FileSystem.cat(self, path)</code> | Get the content of a file |
| <code>S3FileSystem.du(self, path[, total, maxdepth])</code> | Space used by files within a path |
| <code>S3FileSystem.exists(self, path)</code> | Is there a file at the given path |
| <code>S3FileSystem.get(self, rpath, lpath[, recursive])</code> | Copy file to local. |
| <code>S3FileSystem.glob(self, path, <i>**kwargs</i>)</code> | Find files by glob-matching. |
| <code>S3FileSystem.info(self, path[, version_id])</code> | Give details of entry at path |
| <code>S3FileSystem.ls(self, path[, detail, refresh])</code> | List single “directory” with or without details |
| <code>S3FileSystem.mkdir(self, path[, acl])</code> | Create directory entry at path |
| <code>S3FileSystem.mv(self, path1, path2, <i>**kwargs</i>)</code> | Move file from one location to another |
| <code>S3FileSystem.open(self, path[, mode, ...])</code> | Return a file-like object from the filesystem |
| <code>S3FileSystem.put(self, lpath, rpath[, recursive])</code> | Upload file from local |
| <code>S3FileSystem.read_block(self, fn, offset, length)</code> | Read a block of bytes from |
| <code>S3FileSystem.rm(self, path[, recursive])</code> | Remove keys and/or bucket. |
| <code>S3FileSystem.tail(self, path[, size])</code> | Get the last <code>size</code> bytes from file |
| <code>S3FileSystem.touch(self, path[, truncate, data])</code> | Create empty file or truncate |
| <hr/> | |
| <code>S3File(s3, path[, mode, block_size, acl, ...])</code> | Open S3 key as a file. |
| <code>S3File.close(self)</code> | Close file |
| <code>S3File.flush(self[, force])</code> | Write buffered data to backend store. |
| <code>S3File.info(self)</code> | File information about this path |
| <code>S3File.read(self[, length])</code> | Return data from cache, or fetch pieces as necessary |
| <code>S3File.seek(self, loc[, whence])</code> | Set current file location |
| <code>S3File.tell(self)</code> | Current file location |
| <code>S3File.write(self, data)</code> | Write data to buffer. |
| <hr/> | |
| <code>S3Map(root, s3[, check, create])</code> | Mirror previous class, not implemented in <code>fspec</code> |

```
class s3fs.core.S3FileSystem(anon=False, key=None, secret=None, token=None,
                             use_ssl=True, client_kwargs=None, requester_pays=False,
                             default_block_size=None, default_fill_cache=True, de-
                             fault_cache_type='bytes', version_aware=False, con-
                             fig_kwargs=None, s3_additional_kwargs=None, session=None,
                             username=None, password=None, **kwargs)
```

Access S3 as if it were a file system.

This exposes a filesystem-like API (`ls`, `cp`, `open`, etc.) on top of S3 storage.

Provide credentials either explicitly (`key=`, `secret=`) or depend on boto’s credential methods. See `boto3` documentation for more information. If no credentials are available, use `anon=True`.

Parameters

anon [bool (False)] Whether to use anonymous connection (public buckets only). If False, uses the key/secret given, or boto’s credential resolver (environment variables, config files, EC2 IAM server, in that order)

key [string (None)] If not anonymous, use this access key ID, if specified

secret [string (None)] If not anonymous, use this secret access key, if specified

token [string (None)] If not anonymous, use this security token, if specified

- use_ssl** [bool (True)] Whether to use SSL in connections to S3; may be faster without, but insecure
- s3_additional_kwargs** [dict of parameters that are used when calling s3 api] methods. Typically used for things like “ServerSideEncryption”.
- client_kwargs** [dict of parameters for the boto3 client]
- requester_pays** [bool (False)] If RequesterPays buckets are supported.
- default_block_size: int (None)** If given, the default block size value used for `open()`, if no specific value is given at all time. The built-in default is 5MB.
- default_fill_cache** [Bool (True)] Whether to use cache filling with `open` by default. Refer to `S3File.open`.
- default_cache_type** [string ('bytes')] If given, the default `cache_type` value used for `open()`. Set to “none” if no caching is desired. See `fsspec`'s documentation for other available `cache_type` values. Default `cache_type` is 'bytes'.
- version_aware** [bool (False)] Whether to support bucket versioning. If enable this will require the user to have the necessary IAM permissions for dealing with versioned objects.
- config_kwargs** [dict of parameters passed to `botocore.client.Config`]
- kwargs** [other parameters for boto3 session]
- session** [botocore Session object to be used for all connections.] This session will be used in place of creating a new session inside `S3FileSystem`.

Examples

```
>>> s3 = S3FileSystem(anon=False) # doctest: +SKIP
>>> s3.ls('my-bucket/') # doctest: +SKIP
['my-file.txt']
```

```
>>> with s3.open('my-bucket/my-file.txt', mode='rb') as f: # doctest: +SKIP
...     print(f.read()) # doctest: +SKIP
b'Hello, world!'
```

Attributes

- transaction** A context within which files are committed together upon exit

Methods

| | |
|---|---|
| <code>bulk_delete(self, pathlist, **kwargs)</code> | Remove multiple keys with one call |
| <code>cat(self, path)</code> | Get the content of a file |
| <code>checksum(self, path)</code> | Unique value for current version of file |
| <code>chmod(self, path, acl, **kwargs)</code> | Set Access Control on a bucket/key |
| <code>clear_instance_cache()</code> | Clear the cache of filesystem instances. |
| <code>connect(self[, refresh])</code> | Establish S3 connection object. |
| <code>copy(self, path1, path2, **kwargs)</code> | Copy within two locations in the filesystem |
| <code>copy_basic(self, path1, path2, **kwargs)</code> | Copy file between locations on S3 |
| <code>cp(self, path1, path2, **kwargs)</code> | Alias of <code>FilesystemSpec.copy</code> . |

Continued on next page

Table 4 – continued from previous page

| | |
|--|---|
| <code>current()</code> | Return the most recently created FileSystem |
| <code>delete(self, path[, recursive, maxdepth])</code> | Alias of FileSystemSpec.rm. |
| <code>disk_usage(self, path[, total, maxdepth])</code> | Alias of FileSystemSpec.du. |
| <code>download(self, rpath, lpath[, recursive])</code> | Alias of FileSystemSpec.get. |
| <code>du(self, path[, total, maxdepth])</code> | Space used by files within a path |
| <code>end_transaction(self)</code> | Finish write transaction, non-context version |
| <code>exists(self, path)</code> | Is there a file at the given path |
| <code>find(self, path[, maxdepth, withdirs])</code> | List all files below path. |
| <code>get(self, rpath, lpath[, recursive])</code> | Copy file to local. |
| <code>get_delegated_s3pars(self[, exp])</code> | Get temporary credentials from STS, appropriate for sending across a network. |
| <code>get_mapper(self, root[, check, create])</code> | Create key/value store based on this file-system |
| <code>get_tags(self, path)</code> | Retrieve tag key/values for the given path |
| <code>getxattr(self, path, attr_name, **kwargs)</code> | Get an attribute from the metadata. |
| <code>glob(self, path, **kwargs)</code> | Find files by glob-matching. |
| <code>head(self, path[, size])</code> | Get the first <code>size</code> bytes from file |
| <code>info(self, path[, version_id])</code> | Give details of entry at path |
| <code>invalidate_cache(self[, path])</code> | Discard any cached directory information |
| <code>isdir(self, path)</code> | Is this entry directory-like? |
| <code>isfile(self, path)</code> | Is this entry file-like? |
| <code>listdir(self, path[, detail])</code> | Alias of FileSystemSpec.ls. |
| <code>ls(self, path[, detail, refresh])</code> | List single “directory” with or without details |
| <code>makedir(self, path[, create_parents])</code> | Alias of FileSystemSpec.mkdir. |
| <code>makedirs(self, path[, exist_ok])</code> | Recursively make directories |
| <code>merge(self, path, filelist, **kwargs)</code> | Create single S3 file from list of S3 files |
| <code>metadata(self, path[, refresh])</code> | Return metadata of path. |
| <code>mkdir(self, path[, acl])</code> | Create directory entry at path |
| <code>makedirs(self, path[, exist_ok])</code> | Alias of FileSystemSpec.makedirs. |
| <code>move(self, path1, path2, **kwargs)</code> | Alias of FileSystemSpec.mv. |
| <code>mv(self, path1, path2, **kwargs)</code> | Move file from one location to another |
| <code>open(self, path[, mode, block_size, ...])</code> | Return a file-like object from the filesystem |
| <code>put(self, lpath, rpath[, recursive])</code> | Upload file from local |
| <code>put_tags(self, path, tags[, mode])</code> | Set tags for given existing key |
| <code>read_block(self, fn, offset, length[, delimiter])</code> | Read a block of bytes from |
| <code>rename(self, path1, path2, **kwargs)</code> | Alias of FileSystemSpec.mv. |
| <code>rm(self, path[, recursive])</code> | Remove keys and/or bucket. |
| <code>rmdir(self, path)</code> | Remove a directory, if empty |
| <code>setxattr(self, path[, copy_kwargs])</code> | Set metadata. |
| <code>size(self, path)</code> | Size in bytes of file |
| <code>start_transaction(self)</code> | Begin write transaction for deferring files, non-context version |
| <code>stat(self, path, **kwargs)</code> | Alias of FileSystemSpec.info. |
| <code>tail(self, path[, size])</code> | Get the last <code>size</code> bytes from file |
| <code>touch(self, path[, truncate, data])</code> | Create empty file or truncate |
| <code>ukey(self, path)</code> | Hash of file properties, to tell if it has changed |
| <code>upload(self, lpath, rpath[, recursive])</code> | Alias of FileSystemSpec.put. |
| <code>url(self, path[, expires])</code> | Generate presigned URL to access path by HTTP |
| <code>walk(self, path[, maxdepth])</code> | Return all files belows path |

| | |
|----------------------------------|--|
| <code>copy_managed</code> | |
| <code>object_version_info</code> | |

bulk_delete (*self*, *pathlist*, ***kwargs*)

Remove multiple keys with one call

Parameters

pathlist [listof strings] The keys to remove, must all be in the same bucket.

chmod (*self*, *path*, *acl*, ***kwargs*)

Set Access Control on a bucket/key

See <http://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html#canned-acl>

Parameters

path [string] the object to set

acl [string] the value of ACL to apply

connect (*self*, *refresh=True*)

Establish S3 connection object.

Parameters

refresh [bool] Whether to create new session/client, even if a previous one with the same parameters already exists. If False (default), an existing one will be used if possible

copy (*self*, *path1*, *path2*, ***kwargs*)

Copy within two locations in the filesystem

copy_basic (*self*, *path1*, *path2*, ***kwargs*)

Copy file between locations on S3

exists (*self*, *path*)

Is there a file at the given path

get_delegated_s3pars (*self*, *exp=3600*)

Get temporary credentials from STS, appropriate for sending across a network. Only relevant where the key/secret were explicitly provided.

Parameters

exp [int] Time in seconds that credentials are good for

Returns

dict of parameters

get_tags (*self*, *path*)

Retrieve tag key/values for the given path

Returns

{str: str}

getxattr (*self*, *path*, *attr_name*, ***kwargs*)

Get an attribute from the metadata.

Examples

```
>>> mys3fs.getxattr('mykey', 'attribute_1') # doctest: +SKIP
'value_1'
```

info (*self, path, version_id=None*)

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 (*self, 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 (*self, path*)

Is this entry directory-like?

ls (*self, path, detail=False, refresh=False, **kwargs*)

List single "directory" with or without details

Parameters

path [string/bytes] location at which to list files

detail [bool (=True)] if True, each list item is a dict of file properties; otherwise, returns list of filenames

refresh [bool (=False)] if False, look in local cache for file details first

kwargs [dict] additional arguments passed on

merge (*self, path, filelist, **kwargs*)

Create single S3 file from list of S3 files

Uses multi-part, no data is downloaded. The original files are not deleted.

Parameters

path [str] The final file to produce

filelist [list of str] The paths, in order, to assemble into the final file.

metadata (*self, path, refresh=False, **kwargs*)

Return metadata of path.

Metadata is cached unless `refresh=True`.

Parameters

path [string/bytes] filename to get metadata for

refresh [bool (=False)] if False, look in local cache for file metadata first

mkdir (*self, path, acl="", **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.**put_tags** (*self, path, tags, mode='o'*)

Set tags for given existing key

Tags are a `str:str` mapping that can be attached to any key, see <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/allocation-tag-restrictions.html>

This is similar to, but distinct from, key metadata, which is usually set at key creation time.

Parameters**path:** **str** Existing key to attach tags to**tags:** **dict str, str** Tags to apply.**mode:** One of 'o' or 'm' 'o': Will over-write any existing tags. 'm': Will merge in new tags with existing tags. Incurs two remote calls.**rm** (*self, path, recursive=False, **kwargs*)

Remove keys and/or bucket.

Parameters**path** [string] The location to remove.**recursive** [bool (True)] Whether to remove also all entries below, i.e., which are returned by `walk()`.**rmdir** (*self, path*)

Remove a directory, if empty

setxattr (*self, path, copy_kwargs=None, **kw_args*)

Set metadata.

Attributes have to be of the form documented in the '[Metadata Reference](#)'.**Parameters****kw_args** [key-value pairs like `field="value"`, where the values must be] strings. Does not alter existing fields, unless the field appears here - if the value is `None`, delete the field.**copy_kwargs** [dict, optional] dictionary of additional params to use for the underlying `s3.copy_object`.**Examples**

```
>>> mys3file.setxattr(attribute_1='value1', attribute_2='value2') # doctest:
↳ +SKIP
# Example for use with copy_args
>>> mys3file.setxattr(copy_kwargs={'ContentType': 'application/pdf'},
...     attribute_1='value1') # doctest: +SKIP
```

<http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html#object-metadata>**touch** (*self, path, truncate=True, data=None, **kwargs*)

Create empty file or truncate

url (*self*, *path*, *expires=3600*, ***kwargs*)
Generate presigned URL to access path by HTTP

Parameters

path [string] the key path we are interested in
expires [int] the number of seconds this signature will be good for.

walk (*self*, *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 s3fs.core.S3File(s3, path, mode='rb', block_size=5242880, acl="", version_id=None,  
                    fill_cache=True, s3_additional_kwargs=None, autocommit=True,  
                    cache_type='bytes', requester_pays=False)
```

Open S3 key as a file. Data is only loaded and cached on demand.

Parameters

s3 [S3FileSystem] boto3 connection
path [string] S3 bucket/key to access
mode [str] One of ‘rb’, ‘wb’, ‘ab’. These have the same meaning as they do for the built-in `open` function.
block_size [int] read-ahead size for finding delimiters
fill_cache [bool] If seeking to new a part of the file beyond the current buffer, with this True, the buffer will be filled between the sections to best support random access. When reading only a few specific chunks out of a file, performance may be better if False.
acl: str Canned ACL to apply
version_id [str] Optional version to read the file at. If not specified this will default to the current version of the object. This is only used for reading.
requester_pays [bool (False)] If RequesterPays buckets are supported.

See also:

S3FileSystem.open used to create `S3File` objects

Examples

```
>>> s3 = S3FileSystem() # doctest: +SKIP
>>> with s3.open('my-bucket/my-file.txt', mode='rb') as f: # doctest: +SKIP
...     ... # doctest: +SKIP
```

Attributes

closed

Methods

| | |
|--|---|
| <code>close(self)</code> | Close file |
| <code>commit(self)</code> | Move from temp to final destination |
| <code>discard(self)</code> | Throw away temporary file |
| <code>fileno(self, /)</code> | Returns underlying file descriptor if one exists. |
| <code>flush(self[, force])</code> | Write buffered data to backend store. |
| <code>getxattr(self, xattr_name, **kwargs)</code> | Get an attribute from the metadata. |
| <code>info(self)</code> | File information about this path |
| <code>isatty(self, /)</code> | Return whether this is an 'interactive' stream. |
| <code>metadata(self[, refresh])</code> | Return metadata of file. |
| <code>read(self[, length])</code> | Return data from cache, or fetch pieces as necessary |
| <code>readable(self)</code> | Whether opened for reading |
| <code>readinto(self, b)</code> | mirrors builtin file's readinto method |
| <code>readline(self)</code> | Read until first occurrence of newline character |
| <code>readlines(self)</code> | Return all data, split by the newline character |
| <code>readuntil(self[, char, blocks])</code> | Return data between current position and first occurrence of char |
| <code>seek(self, loc[, whence])</code> | Set current file location |
| <code>seekable(self)</code> | Whether is seekable (only in read mode) |
| <code>setxattr(self[, copy_kwargs])</code> | Set metadata. |
| <code>tell(self)</code> | Current file location |
| <code>truncate()</code> | Truncate file to size bytes. |
| <code>url(self, **kwargs)</code> | HTTP URL to read this file (if it already exists) |
| <code>writable(self)</code> | Whether opened for writing |
| <code>write(self, data)</code> | Write data to buffer. |

| | |
|-------------------|--|
| readinto1 | |
| writelines | |

commit (*self*)

Move from temp to final destination

discard (*self*)

Throw away temporary file

getxattr (*self*, *xattr_name*, ***kwargs*)

Get an attribute from the metadata. See `getxattr()`.

Examples

```
>>> mys3file.getxattr('attribute_1') # doctest: +SKIP
'value_1'
```

metadata (*self*, *refresh=False*, ***kwargs*)

Return metadata of file. See `metadata()`.

Metadata is cached unless *refresh=True*.

setattr (*self*, *copy_kwargs=None*, ***kwargs*)
Set metadata. See `setattr()`.

Examples

```
>>> mys3file.setattr(attribute_1='value1', attribute_2='value2') # doctest:␣  
↪+SKIP
```

url (*self*, ***kwargs*)
HTTP URL to read this file (if it already exists)

`s3fs.mapping.S3Map` (*root*, *s3*, *check=False*, *create=False*)
Mirror previous class, not implemented in `fsspec`

class `s3fs.utils.ParamKwargsHelper` (*s3*)
Utility class to help extract the subset of keys that an `s3` method is actually using

Parameters

`s3` [boto S3FileSystem]

Methods

| | |
|--------------------------|--|
| <code>filter_dict</code> | |
|--------------------------|--|

class `s3fs.utils.SSEParams` (*server_side_encryption=None*, *sse_customer_algorithm=None*,
sse_customer_key=None, *sse_kms_key_id=None*)

Methods

| | |
|------------------------|--|
| <code>to_kwargs</code> | |
|------------------------|--|

8.3 Changelog

8.3.1 Version 0.4.0

- New instances no longer need reconnect (PR #244) by Martin Durant
- Always use multipart uploads when not autocommitting (PR #243) by Marius van Niekerk
- Create `CONTRIBUTING.md` (PR #248) by Jacob Tomlinson
- Use autofunction for `S3Map` sphinx autosummary (PR #251) by James Bourbeau
- Miscellaneous doc updates (PR #252) by James Bourbeau
- Support for Python 3.8 (PR #264) by Tom Augspurger
- Improved performance for `isdir` (PR #259) by Nate Yoder
- Increased the minimum required version of `fsspec` to 0.6.0

CHAPTER 9

Indices and tables

- `genindex`
- `modindex`
- `search`

B

bulk_delete() (*s3fs.core.S3FileSystem method*), 21

C

chmod() (*s3fs.core.S3FileSystem method*), 21

commit() (*s3fs.core.S3File method*), 25

connect() (*s3fs.core.S3FileSystem method*), 21

copy() (*s3fs.core.S3FileSystem method*), 21

copy_basic() (*s3fs.core.S3FileSystem method*), 21

D

discard() (*s3fs.core.S3File method*), 25

E

exists() (*s3fs.core.S3FileSystem method*), 21

G

get_delegated_s3pars() (*s3fs.core.S3FileSystem method*), 21

get_tags() (*s3fs.core.S3FileSystem method*), 21

getxattr() (*s3fs.core.S3File method*), 25

getxattr() (*s3fs.core.S3FileSystem method*), 21

I

info() (*s3fs.core.S3FileSystem method*), 21

invalidate_cache() (*s3fs.core.S3FileSystem method*), 22

isdir() (*s3fs.core.S3FileSystem method*), 22

L

ls() (*s3fs.core.S3FileSystem method*), 22

M

merge() (*s3fs.core.S3FileSystem method*), 22

metadata() (*s3fs.core.S3File method*), 25

metadata() (*s3fs.core.S3FileSystem method*), 22

mkdir() (*s3fs.core.S3FileSystem method*), 22

P

ParamKwargsHelper (*class in s3fs.utils*), 26

put_tags() (*s3fs.core.S3FileSystem method*), 23

R

rm() (*s3fs.core.S3FileSystem method*), 23

rmdir() (*s3fs.core.S3FileSystem method*), 23

S

S3File (*class in s3fs.core*), 24

S3FileSystem (*class in s3fs.core*), 18

S3Map() (*in module s3fs.mapping*), 26

setxattr() (*s3fs.core.S3File method*), 25

setxattr() (*s3fs.core.S3FileSystem method*), 23

SSEParams (*class in s3fs.utils*), 26

T

touch() (*s3fs.core.S3FileSystem method*), 23

U

url() (*s3fs.core.S3File method*), 26

url() (*s3fs.core.S3FileSystem method*), 23

W

walk() (*s3fs.core.S3FileSystem method*), 24