S3Fs Documentation

Release 0.2.0+11.g14765aa

Continuum Analytics

Dec 04, 2018

Contents

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.

This project was originally designed as a storage-layer interface for dask.distributed and has a very similar interface to hdfs3

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/chmown methods)

CHAPTER 3

Credentials

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.

CHAPTER 4

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 maybe 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)
```

CHAPTER 5

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.

CHAPTER 6

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.S3FileSytem(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

CHAPTER 7

Contents

7.1 Installation

7.1.1 Conda

The `s3fs` library and its dependencies can be installed from the conda-forge repository using conda:

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

7.1.2 PyPI

You can install `s3fs` with pip:

```
pip install s3fs
```

7.1.3 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
```

7.2 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

S3FileSystem.cat(path, **kwargs)	Returns contents of file
S3FileSystem.du(path[, total, deep])	Bytes in keys at path
S3FileSystem.exists(path)	Does such a file/directory exist?
S3FileSystem.get(path, filename, **kwargs)	Stream data from file at path to local filename
S3FileSystem.glob(path)	Find files by glob-matching.
S3FileSystem.info(path[, version_id, refresh])	Detail on the specific file pointed to by path.
S3FileSystem.ls(path[, detail, refresh])	List single "directory" with or without details
S3FileSystem.mkdir(path[, acl])	Make new bucket or empty key
S3FileSystem.mv(path1, path2, **kwargs)	Move file between locations on S3
S3FileSystem.open(path[, mode, block_size, ...])	Open a file for reading or writing
S3FileSystem.put(filename, path, **kwargs)	Stream data from local filename to file at path
S3FileSystem.read_block(fn, offset, length)	Read a block of bytes from an S3 file
S3FileSystem.rm(path[, recursive])	Remove keys and/or bucket.
S3FileSystem.tail(path[, size])	Return last bytes of file
S3FileSystem.touch(path[, acl])	Create empty key

S3File(s3, path[, mode, block_size, acl, ...])	Open S3 key as a file.
S3File.close()	Close file
S3File.flush([force, retries])	Write buffered data to S3.
S3File.info(**kwargs)	File information about this path
S3File.read([length])	Return data from cache, or fetch pieces as necessary
S3File.seek(loc[, whence])	Set current file location
S3File.tell()	Current file location
S3File.write(data)	Write data to buffer.

S3Map(root[, s3, check, create])	Wrap an S3FileSystem as a mutable wrapping.

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*, *version_aware=False*, *config_kwargs=None*, *s3_additional_kwargs=None*, *session=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: None, int 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`.

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 inplace 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!'
```

Methods

bulk_delete(pathlist, **kwargs)	Remove multiple keys with one call
cat(path, **kwargs)	Returns contents of file
chmod(path, acl, **kwargs)	Set Access Control on a bucket/key
connect([refresh])	Establish S3 connection object.
copy_basic(path1, path2, **kwargs)	Copy file between locations on S3
current()	Return the most recently created S3FileSystem
du(path[, total, deep])	Bytes in keys at path
exists(path)	Does such a file/directory exist?
get(path, filename, **kwargs)	Stream data from file at path to local filename
get_delegated_s3pars([exp])	Get temporary credentials from STS, appropriate for sending across a network.
get_tags(path)	Retrieve tag key/values for the given path
getxattr(path, attr_name, **kwargs)	Get an attribute from the metadata.
glob(path)	Find files by glob-matching.
head(path[, size])	Return first bytes of file
info(path[, version_id, refresh])	Detail on the specific file pointed to by path.
ls(path[, detail, refresh])	List single "directory" with or without details
merge(path, filelist, **kwargs)	Create single S3 file from list of S3 files
metadata(path[, refresh])	Return metadata of path.

Continued on next page

Table 4 – continued from previous page

mkdir(path[, acl])	Make new bucket or empty key
mv(path1, path2, **kwargs)	Move file between locations on S3
open(path[, mode, block_size, acl, ...])	Open a file for reading or writing
put(filename, path, **kwargs)	Stream data from local filename to file at path
put_tags(path, tags[, mode])	Set tags for given existing key
read_block(fn, offset, length[, delimiter])	Read a block of bytes from an S3 file
rm(path[, recursive])	Remove keys and/or bucket.
rmdir(path, **kwargs)	Remove empty key or bucket
setxattr(path[, copy_kwargs])	Set metadata.
tail(path[, size])	Return last bytes of file
touch(path[, acl])	Create empty key
url(path[, expires])	Generate presigned URL to access path by HTTP
walk(path[, refresh, directories])	Return all real keys below path

copy	
copy_managed	
invalidate_cache	
object_version_info	

bulk_delete(*pathlist*, ***kwargs*)
Remove multiple keys with one call

Parameters

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

cat(*path*, ***kwargs*)
Returns contents of file

chmod(*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(*refresh=False*)
Establish S3 connection object.

Parameters

refresh [bool (True)] Whether to use cached filelists, if already read

copy_basic(*path1*, *path2*, ***kwargs*)
Copy file between locations on S3

classmethod current()
Return the most recently created S3FileSystem

If no S3FileSystem has been created, then create one

du(*path*, *total=False*, *deep=False*, ***kwargs*)
Bytes in keys at path

exists (*path*)
Does such a file/directory exist?

get (*path*, *filename*, ***kwargs*)
Stream data from file at path to local filename

get_delegated_s3pars (*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 (*path*)
Retrieve tag key/values for the given path

Returns

{str: str}

getxattr (*path*, *attr_name*, ***kwargs*)
Get an attribute from the metadata.

Examples

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

glob (*path*)
Find files by glob-matching.

Note that the bucket part of the path must not contain a "*"

head (*path*, *size=1024*, ***kwargs*)
Return first bytes of file

info (*path*, *version_id=None*, *refresh=False*, ***kwargs*)
Detail on the specific file pointed to by path.

Gets details only for a specific key, directories/buckets cannot be used with info.

Parameters

version_id [str, optional] version of the key to perform the head_object on

refresh [bool] If true, don't look in the info cache

ls (*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 (*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 (*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 (*path*, *acl=''*, ***kwargs*)
Make new bucket or empty key

Parameters

acl: str ACL to set when creating

region_name [str] region in which the bucket should be created

mv (*path1*, *path2*, ***kwargs*)
Move file between locations on S3

open (*path*, *mode='rb'*, *block_size=None*, *acl=''*, *version_id=None*, *fill_cache=None*, *encoding=None*, ***kwargs*)
Open a file for reading or writing

Parameters

path: string Path of file on S3

mode: string One of 'r', 'w', 'a', 'rb', 'wb', or 'ab'. These have the same meaning as they do for the built-in *open* function.

block_size: int Size of data-node blocks if reading

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 set when writing

version_id [str] Explicit version of the object to open. This requires that the s3 filesystem is version aware and bucket versioning is enabled on the relevant bucket.

encoding [str] The encoding to use if opening the file in text mode. The platform's default text encoding is used if not given.

kwargs: dict-like Additional parameters used for s3 methods. Typically used for ServerSideEncryption.

put (*filename*, *path*, ***kwargs*)
Stream data from local filename to file at path

put_tags (*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.

read_block (*fn*, *offset*, *length*, *delimiter=None*, ***kwargs*)
Read a block of bytes from an S3 file

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 on S3

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:

`distributed.utils.read_block`

Examples

```
>>> s3.read_block('data/file.csv', 0, 13)  # doctest: +SKIP
b'Alice, 100\nBo'
>>> s3.read_block('data/file.csv', 0, 13, delimiter=b'\n')  # doctest: +SKIP
b'Alice, 100\nBob, 200\n'
```

Use `length=None` to read to the end of the file. >>> s3.read_block('data/file.csv', 0, None, delimiter=b'n') # doctest: +SKIP b'Alice, 100nBob, 200nCharlie, 300'

rm (*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 (*path*, ***kwargs*)
Remove empty key or bucket

setxattr (*path*, *copy_kwargs=None*, ***kw_args*)
Set metadata.

Attributes have to be of the form documented in the **'Metadata Reference'_**.

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

tail (*path*, *size=1024*, ***kwargs*)
Return last bytes of file

touch (*path*, *acl=''*, ***kwargs*)
Create empty key

If path is a bucket only, attempt to create bucket.

url (*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 (*path*, *refresh=False*, *directories=False*)
Return all real keys below path

class s3fs.core.**S3File**(*s3*, *path*, *mode='rb'*, *block_size=5242880*, *acl=''*, *version_id=None*, *fill_cache=True*, *s3_additional_kwargs=None*)
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.

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
```

Methods

close()	Close file
flush([force, retries])	Write buffered data to S3.
getxattr(xattr_name, **kwargs)	Get an attribute from the metadata.
info(**kwargs)	File information about this path
metadata([refresh])	Return metadata of file.
read([length])	Return data from cache, or fetch pieces as necessary
readable()	Return whether the S3File was opened for reading
readline([length])	Read and return a line from the stream.
readlines()	Return all lines in a file as a list
seek(loc[, whence])	Set current file location
seekable()	Return whether the S3File is seekable (only in read mode)
setxattr([copy_kwargs])	Set metadata.
tell()	Current file location
url(**kwargs)	HTTP URL to read this file (if it already exists)
writable()	Return whether the S3File was opened for writing
write(data)	Write data to buffer.

detach	
next	
read1	
readinto	
readinto1	

close()
Close file

If in write mode, key is only finalized upon close, and key will then be available to other processes.

flush(*force=False*, *retries=10*)
Write buffered data to S3.

Uploads the current buffer, if it is larger than the block-size. If the buffer is smaller than the block-size, this is a no-op.

Due to S3 multi-upload policy, you can only safely force flush to S3 when you are finished writing.

Parameters

force [bool] When closing, write the last block even if it is smaller than blocks are allowed to be.

retries: int

getxattr(*xattr_name*, ***kwargs*)
Get an attribute from the metadata. See `getxattr()`.

Examples

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

info(***kwargs*)
File information about this path

metadata(*refresh=False*, ***kwargs*)
Return metadata of file. See `metadata()`.

Metadata is cached unless *refresh=True*.

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()
Return whether the S3File was opened for reading

readline(*length=-1*)
Read and return a line from the stream.

If length is specified, at most size bytes will be read.

readlines()
Return all lines in a file as a list

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()
Return whether the S3File is seekable (only in read mode)

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

Examples

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

tell()
Current file location

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

writable()
Return whether the S3File was opened for writing

write(*data*)
Write data to buffer.

Buffer only sent to S3 on close() or if buffer is greater than or equal to blocksize.

Parameters

data [bytes] Set of bytes to be written.

class s3fs.mapping.**S3Map**(*root*, *s3=None*, *check=False*, *create=False*)
Wrap an S3FileSystem 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 (perhaps just a bucket name)

s3 [S3FileSystem]

check [bool (=True)] performs a touch at the location, to check writeability.

Examples

```
>>> s3 = s3fs.S3FileSystem() # doctest: +SKIP
>>> d = MapWrapping('mybucket/mapstore/', s3=s3) # doctest: +SKIP
>>> d['loc1'] = b'Hello World' # doctest: +SKIP
>>> list(d.keys()) # doctest: +SKIP
['loc1']
>>> d['loc1'] # doctest: +SKIP
b'Hello World'
```

Methods

clear()	Remove all keys below root - empties out mapping
get(k[,d])	
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])	
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()	

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

filter_dict	

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

Methods

to_kwargs	

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

U

W