MINIOFS Documentation

Release 1.1.2

Max Klein

Mar 19, 2020

Contents

MINIOFS is a PyFilesystem interface to Amazon S3 cloud storage.

As a PyFilesystem concrete class, MINIOFS allows you to work with MinIO-flavored S3 in the same as any other supported filesystem.

CHAPTER 1

Installing

MINIOFS may be installed from pip with the following command:

```
pip install fs-miniofs
```

This will install the most recent stable version.

Alternatively, if you want the cutting edge code, you can check out the GitHub repos at https://github.com/pyfilesystem/miniofs

CHAPTER 2

Opening a MinIO-flavored S3 Filesystem

There are two options for constructing a miniofs instance. The simplest way is with an *opener*, which is a simple URL like syntax. Here is an example:

```
from fs import open_fs
miniofs = open_fs('minio://mybucket/')
```

For more granular control, you may import the MINIOFS class and construct it explicitly:

```
from fs_miniofs import MINIOFS
miniofs = MINIOFS('mybucket')
```

2.1 MINIOFS Constructor

class fs_miniofs.**MINIOFS**(*bucket_name*, *dir_path='/'*, *aws_access_key_id=None*, *aws_secret_access_key=None*, *aws_session_token=None*, *endpoint_url=None*, *region=None*, *delimiter='/'*, *strict=False*, *cache_control=None*, *acl=None*, *upload_args=None*, *download_args=None*)

Construct an Amazon S3 filesystem for PyFilesystem

Parameters

- **bucket_name** (*str*) – The S3 bucket name.
- **dir_path** (*str*) – The root directory within the S3 Bucket. Defaults to `"/"`
- **aws_access_key_id** (*str*) – The access key, or `None` to read the key from standard configuration files.
- **aws_secret_access_key** (*str*) – The secret key, or `None` to read the key from standard configuration files.
- **endpoint_url** (*str*) – Alternative endpoint url (`None` to use default).

- **aws_session_token** (*str*) –
- **region** (*str*) – Optional S3 region.
- **delimiter** (*str*) – The delimiter to separate folders, defaults to a forward slash.
- **strict** (*bool*) – Set to `False` (default) to disable validation of destination paths. Currently this is need to work with the way that MinIO automatically flattens all directories. When `True` (default) MINIOFS will follow the PyFilesystem specification exactly.
- **cache_control** (*str*) – Sets the 'Cache-Control' header for uploads.
- **acl** (*str*) – Sets the Access Control List header for uploads.
- **upload_args** (*dict*) – A dictionary for additional upload arguments. See https://boto3.readthedocs.io/en/latest/reference/services/s3.html#S3.Object.put for details.
- **download_args** (*dict*) – Dictionary of extra arguments passed to the S3 client.

copy(*src_path*, *dst_path*, *overwrite=False*)

Copy file contents from `src_path` to `dst_path`.

Arguments: src_path (str): Path of source file. dst_path (str): Path to destination file. overwrite (bool): If *True*, overwrite the destination file

if it exists (defaults to *False*).

Raises:

fs.errors.DestinationExists: If `dst_path` exists, and `overwrite` is *False*.

fs.errors.ResourceNotFound: If a parent directory of `dst_path` does not exist.

download(*path*, *file*, *chunk_size=None*, ***options*)

Copies a file from the filesystem to a file-like object.

This may be more efficient that opening and copying files manually if the filesystem supplies an optimized method.

Arguments: path (str): Path to a resource. file (file-like): A file-like object open for writing in

binary mode.

chunk_size (int, optional): Number of bytes to read at a time, if a simple copy is used, or *None* to use sensible default.

****options: Implementation specific options required to open** the source file.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

Example:

```
>>> with open('starwars.mov', 'wb') as write_file:
...     my_fs.download('/movies/starwars.mov', write_file)
```

exists(*path*)

Check if a path maps to a resource.

Arguments: path (str): Path to a resource.

Returns: bool: *True* if a resource exists at the given path.

getinfo(*path*, *namespaces=None*, *check_parent=False*)

Get information about a resource on a filesystem.

Arguments: path (str): A path to a resource on the filesystem. namespaces (list, optional): Info namespaces to query

(defaults to *[basic]*).

Returns: ~fs.info.Info: resource information object.

For more information regarding resource information, see info.

geturl (*path*, *purpose='download'*)
Get the URL to a given resource.

Parameters: path (str): A path on the filesystem purpose (str): A short string that indicates which URL

to retrieve for the given path (if there is more than one). The default is `'download'`, which should return a URL that serves the file. Other filesystems may support other values for `purpose`.

Returns: str: a URL.

Raises: fs.errors.NoURL: If the path does not map to a URL.

isdir (*path*)
Check if a path maps to an existing directory.

Parameters: path (str): A path on the filesystem.

Returns: bool: *True* if `path` maps to a directory.

isempty (*path*)
Check if a directory is empty.

A directory is considered empty when it does not contain any file or any directory.

Parameters: path (str): A path to a directory on the filesystem.

Returns: bool: *True* if the directory is empty.

Raises: errors.DirectoryExpected: If `path` is not a directory. errors.ResourceNotFound: If `path` does not exist.

listdir (*path*)
Get a list of the resource names in a directory.

This method will return a list of the resources in a directory. A *resource* is a file, directory, or one of the other types defined in *~fs.enums.ResourceType*.

Arguments: path (str): A path to a directory on the filesystem

Returns: list: list of names, relative to `path`.

Raises: fs.errors.DirectoryExpected: If `path` is not a directory. fs.errors.ResourceNotFound: If `path` does not exist.

makedir (*path*, *permissions=None*, *recreate=False*)
Make a directory.

Arguments: path (str): Path to directory from root. permissions (~fs.permissions.Permissions, optional): a

Permissions instance, or *None* to use default.

recreate (bool): Set to *True* to avoid raising an error if the directory already exists (defaults to *False*).

Returns: ~fs.subfs.SubFS: a filesystem whose root is the new directory.

Raises: fs.errors.DirectoryExists: If the path already exists. fs.errors.ResourceNotFound: If the path is not found.

move (*src_path*, *dst_path*, *overwrite=False*)
Move a file from `src_path` to `dst_path`.

Arguments: src_path (str): A path on the filesystem to move. dst_path (str): A path on the filesystem where the source

file will be written to.

overwrite (bool): If *True*, destination path will be overwritten if it exists.

Raises:

fs.errors.FileExpected: If `src_path` maps to a directory instead of a file.

fs.errors.DestinationExists: If `dst_path` exists, and `overwrite` is *False*.

fs.errors.ResourceNotFound: If a parent directory of `dst_path` does not exist.

openbin (*path*, *mode='r'*, *buffering=-1*, ***options*)
Open a binary file-like object.

Arguments: path (str): A path on the filesystem. mode (str): Mode to open file (must be a valid non-text mode,

defaults to *r*). Since this method only opens binary files, the `b` in the mode string is implied.

buffering (int): Buffering policy (-1 to use default buffering, 0 to disable buffering, or any positive integer to indicate a buffer size).

****options: keyword arguments for any additional information** required by the filesystem (if any).

Returns: io.IOBase: a *file-like* object.

Raises: fs.errors.FileExpected: If the path is not a file. fs.errors.FileExists: If the file exists, and *exclusive mode*

is specified (`x` in the mode).

fs.errors.ResourceNotFound: If the path does not exist.

readbytes (*path*)
Get the contents of a file as bytes.

Arguments: path (str): A path to a readable file on the filesystem.

Returns: bytes: the file contents.

Raises: fs.errors.ResourceNotFound: if `path` does not exist.

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

Arguments: path (str): Path of the file to remove.

Raises: fs.errors.FileExpected: If the path is a directory. fs.errors.ResourceNotFound: If the path does not exist.

removedir (*path*)
Remove a directory from the filesystem.

Arguments: path (str): Path of the directory to remove.

Raises:

fs.errors.DirectoryNotEmpty: If the directory is not empty (see *~fs.base.FS.removetree* for a way to remove the directory contents.).

fs.errors.DirectoryExpected: If the path does not refer to a directory.

fs.errors.ResourceNotFound: If no resource exists at the given path.

fs.errors.RemoveRootError: If an attempt is made to remove the root directory (i.e. `'/'`)

`scandir`(*path*, *namespaces=None*, *page=None*)
Get an iterator of resource info.

Arguments: path (str): A path to a directory on the filesystem. namespaces (list, optional): A list of namespaces to include

in the resource information, e.g. `['basic', 'access']`.

page (tuple, optional): May be a tuple of `(<start>, <end>)` indexes to return an iterator of a subset of the resource info, or *None* to iterate over the entire directory. Paging a directory scan may be necessary for very large directories.

Returns: ~collections.abc.Iterator: an iterator of *Info* objects.

Raises: fs.errors.DirectoryExpected: If `path` is not a directory. fs.errors.ResourceNotFound: If `path` does not exist.

`setinfo`(*path*, *info*)
Set info on a resource.

This method is the complement to *~fs.base.FS.getinfo* and is used to set info values on a resource.

Arguments: path (str): Path to a resource on the filesystem. info (dict): Dictionary of resource info.

Raises:

fs.errors.ResourceNotFound: If `path` does not exist on the filesystem

The `info` dict should be in the same format as the raw info returned by `getinfo(file).raw`.

Example:

```
>>> details_info = {"details": {
...     "modified": time.time()
... }}
>>> my_fs.setinfo('file.txt', details_info)
```

`upload`(*path*, *file*, *chunk_size=None*, ***options*)
Set a file to the contents of a binary file object.

This method copies bytes from an open binary file to a file on the filesystem. If the destination exists, it will first be truncated.

Arguments: path (str): A path on the filesystem. file (io.IOBase): a file object open for reading in

binary mode.

chunk_size (int, optional): Number of bytes to read at a time, if a simple copy is used, or *None* to use sensible default.

****options: Implementation specific options required to open** the source file.

Note that the file object `file` will *not* be closed by this method. Take care to close it after this method completes (ideally with a context manager).

Example:

```
>>> with open('~/movies/starwars.mov', 'rb') as read_file:
...     my_fs.upload('starwars.mov', read_file)
```

writebytes(*path*, *contents*)
Copy binary data to a file.

Arguments: path (str): Destination path on the filesystem. contents (bytes): Data to be written.

Raises: TypeError: if contents is not bytes.

CHAPTER 3

Limitations

Amazon S3 isn't strictly speaking a *filesystem*, in that it contains files, but doesn't offer true *directories*. MINIOFS follows the convention of simulating directories by creating an object that ends in a forward slash. For instance, if you create a file called *"foo/bar"*, MINIOFS will create an S3 object for the file called *"foo/bar" and* an empty object called *"foo/"* which stores that fact that the *"foo"* directory exists.

If you create all your files and directories with MINIOFS, then you can forget about how things are stored under the hood. Everything will work as you expect. You *may* run in to problems if your data has been uploaded without the use of MINIOFS. For instance, if you create a *"foo/bar"* object without a *"foo/"* object. If this occurs, then MINIOFS may give errors about directories not existing, where you would expect them to be. The solution is to create an empty object for all directories and subdirectories. Fortunately most tools will do this for you, and it is probably only required of you upload your files manually.

CHAPTER 4

Authentication

If you don't supply any credentials, then MINIOFS will use the access key and secret key configured on your system. You may also specify when creating the filesystem instance. Here's how you would do that with an opener:

```
miniofs = open_fs('minio://<access key>:<secret key>@mybucket')
```

Here's how you specify credentials with the constructor:

```
miniofs = MINIOFS(
    'mybucket'
    aws_access_key_id=<access key>,
    aws_secret_access_key=<secret key>
)
```

Note: Amazon recommends against specifying credentials explicitly like this in production.

CHAPTER 5

S3 Info

You can retrieve S3 info via the `s3` namespace. Here's an example:

```
>>> info = s.getinfo('foo', namespaces=['s3'])
>>> info.raw['s3']
{'metadata': {}, 'delete_marker': None, 'version_id': None, 'parts_count': None,
↪'accept_ranges': 'bytes', 'last_modified': 1501935315, 'content_length': 3,
↪'content_encoding': None, 'request_charged': None, 'replication_status': None,
↪'server_side_encryption': None, 'expires': None, 'restore': None, 'content_type':
↪'binary/octet-stream', 'sse_customer_key_md5': None, 'content_disposition': None,
↪'storage_class': None, 'expiration': None, 'missing_meta': None, 'content_language
↪': None, 'ssekms_key_id': None, 'sse_customer_algorithm': None, 'e_tag': '
↪"37b51d194a7513e45b56f6524f2d51f2"', 'website_redirect_location': None, 'cache_
↪control': None}
```

CHAPTER 6

URLs

You can use the `geturl` method to generate an externally accessible URL from an S3 object. Here's an example:

```
>>> miniofs.geturl('foo')
'https://fsexample.s3.amazonaws.com//foo?AWSAccessKeyId=AKIAIEZZDQU72WQP3JUA&
↪Expires=1501939084&Signature=4rfDuqVgmvILjtTeYOJvyIXRMvs%3D'
```

CHAPTER 7

More Information

See the PyFilesystem Docs for documentation on the rest of the PyFilesystem interface.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index