# azure-datalake-store Documentation

***Release 0.0.1***

**TBA**

November 29, 2016

Contents

A pure-python interface to the Azure Data-lake Storage system, providing pythonic file-system and file objects, seamless transition between Windows and POSIX remote paths, high-performance up- and down-loader and CLI commands.

This software is under active development and not yet recommended for general use.

CHAPTER 1

# Installation

Using `pip`:

```
pip install azure-datalake-store
```

Manually (bleeding edge):

- Download the repo from https://github.com/Azure/azure-data-lake-store-python
- checkout the `dev` branch
- install the requirememnts (`pip install -r dev_requirements.txt`)
- install in develop mode (`python setup.py develop`)
- optionally: build the documentation (including this page) by running `make html` in the docs directory.

CHAPTER 2

# Auth

Although users can generate and supply their own tokens to the base file-system class, and there is a password-based function in the `lib` module for generating tokens, the most convenient way to supply credentials is via environment parameters. This latter method is the one used by default in both library and CLI usage. The following variables are required:

- azure_tenant_id
- azure_username
- azure_password
- azure_store_name
- azure_url_suffix (optional)

CHAPTER 3

# Pythonic Filesystem

The `AzureDLFileSystem` object is the main API for library usage of this package. It provides typical file-system operations on the remote azure store

```
token = lib.auth(tenant_id, username, password)
adl = core.AzureDLFileSystem(store_name, token)
# alternatively, adl = core.AzureDLFileSystem()
# uses environment variables

print(adl.ls())  # list files in the root directory
for item in adl.ls(detail=True):
    print(item)  # same, but with file details as dictionaries
print(adl.walk(''))  # list all files at any directory depth
print('Usage:', adl.du('', deep=True, total=True))  # total bytes usage
adl.mkdir('newdir')  # create directory
adl.touch('newdir/newfile') # create empty file
adl.put('remotefile', '/home/myuser/localfile') # upload a local file
```

In addition, the file-system generates file objects that are compatible with the python file interface, ensuring compatibility with libraries that work on python files. The recommended way to use this is with a context manager (otherwise, be sure to call `close()` on the file object).

```
with adl.open('newfile', 'wb') as f:
    f.write(b'index,a,b\n')
    f.tell()   # now at position 9
    f.flush()  # forces data upstream
    f.write(b'0,1,True')

with adl.open('newfile', 'rb') as f:
    print(f.readlines())

with adl.open('newfile', 'rb') as f:
    df = pd.read_csv(f) # read into pandas.
```

To seamlessly handle remote path representations across all supported platforms, the main API will take in numerous path types: string, Path/PurePath, and AzureDLPath. On Windows in particular, you can pass in paths separated by either forward slashes or backslashes.

```
import pathlib  # only >= Python 3.4
from pathlib2 import pathlib  # only <= Python 3.3

from azure.datalake.store.core import AzureDLPath

# possible remote paths to use on API
```

```
p1 = '\\foo\\bar'
p2 = '/foo/bar'
p3 = pathlib.PurePath('\\foo\\bar')
p4 = pathlib.PureWindowsPath('\\foo\\bar')
p5 = pathlib.PurePath('/foo/bar')
p6 = AzureDLPath('\\foo\\bar')
p7 = AzureDLPath('/foo/bar')

# p1, p3, and p6 only work on Windows
for p in [p1, p2, p3, p4, p5, p6, p7]:
  with adl.open(p, 'rb') as f:
      print(f.readlines())
```

CHAPTER 4

# Performant up-/down-loading

Classes `ADLUploader` and `ADLDownloader` will chunk large files and send many files to/from azure using multiple threads. A whole directory tree can be transferred, files matching a specific glob-pattern or any particular file.

```
# download the whole directory structure using 5 threads, 16MB chunks
ADLDownloader(adl, '', 'my_temp_dir', 5, 2**24)
```

CHAPTER 5

# Command Line Usage

The package provides the above functionality also from the command line (bash, powershell, etc.). Two principle modes are supported: execution of one particular file-system operation; and interactive mode in which multiple operations can be executed in series.

```
python cli.py ls -l
```

Execute the program without arguments to access documentation.

CHAPTER 6

# Contents

## 6.1 API

| | |
|---|---|
| *AzureDLFileSystem*([token]) | Access Azure DataLake Store as if it were a file-system |
| *AzureDLFileSystem.cat*(path) | Returns contents of file |
| *AzureDLFileSystem.du*(path[, total, deep]) | Bytes in keys at path |
| *AzureDLFileSystem.exists*(path) | Does such a file/directory exist? |
| *AzureDLFileSystem.get*(path, filename) | Stream data from file at path to local filename |
| *AzureDLFileSystem.glob*(path) | Find files (not directories) by glob-matching. |
| *AzureDLFileSystem.info*(path) | File information |
| *AzureDLFileSystem.ls*([path, detail]) | List single directory with or without details |
| *AzureDLFileSystem.mkdir*(path) | Make new directory |
| *AzureDLFileSystem.mv*(path1, path2) | Move file between locations on ADL |
| *AzureDLFileSystem.open*(path[, mode, ...]) | Open a file for reading or writing |
| *AzureDLFileSystem.put*(filename, path[, ...]) | Stream data from local filename to file at path |
| *AzureDLFileSystem.read_block*(fn, offset, length) | Read a block of bytes from an ADL file |
| *AzureDLFileSystem.rm*(path[, recursive]) | Remove a file. |
| *AzureDLFileSystem.tail*(path[, size]) | Return last bytes of file |
| *AzureDLFileSystem.touch*(path) | Create empty file |

| | |
|---|---|
| AzureDLFile(azure, path[, mode, blocksize, ...]) | Open ADL key as a file. |
| AzureDLFile.close() | Close file |
| AzureDLFile.flush([force]) | Write buffered data to ADL. |
| AzureDLFile.info() | File information about this path |
| AzureDLFile.read([length]) | Return data from cache, or fetch pieces as necessary |
| AzureDLFile.seek(loc[, whence]) | Set current file location |
| AzureDLFile.tell() | Current file location |
| AzureDLFile.write(data) | Write data to buffer. |

| | |
|---|---|
| *ADLUploader*(adlfs, rpath, lpath[, nthreads, ...]) | Upload local file(s) using chunks and threads |
| *ADLDownloader*(adlfs, rpath, lpath[, ...]) | Download remote file(s) using chunks and threads |

**class** azure.datalake.store.core.**AzureDLFileSystem**(*token=None*, ***kwargs*)

Access Azure DataLake Store as if it were a file-system

**Parameters** **store_name** : str (“”)

Store name to connect to

**token** : dict

When setting up a new connection, this contains the authorization credentials (see *lib.auth()*).

**url_suffix: str (None)**

Domain to send REST requests to. The end-point URL is constructed using this and the store_name. If None, use default.

**kwargs: optional key/values**

See `lib.auth()`; full list: tenant_id, username, password, client_id, client_secret, resource

### Methods

| | |
|---|---|
| *access*(path) | Does such a file/directory exist? |
| *cat*(path) | Returns contents of file |
| *chmod*(path, mod) | Change access mode of path |
| *chown*(path[, owner, group]) | Change owner and/or owning group |
| *concat*(outfile, filelist[, delete_source]) | Concatenate a list of files into one new file |
| *connect*() | Establish connection object. |
| *cp*(path1, path2) | Copy file between locations on ADL |
| *current*() | Return the most recently created AzureDLFileSystem |
| *df*(path) | Resource summary of path |
| *du*(path[, total, deep]) | Bytes in keys at path |
| *exists*(path) | Does such a file/directory exist? |
| *get*(path, filename) | Stream data from file at path to local filename |
| *glob*(path) | Find files (not directories) by glob-matching. |
| *head*(path[, size]) | Return first bytes of file |
| *info*(path) | File information |
| *invalidate_cache*([path]) | Remove entry from object file-cache |
| *listdir*([path, detail]) | List single directory with or without details |
| *ls*([path, detail]) | List single directory with or without details |
| *merge*(outfile, filelist[, delete_source]) | Concatenate a list of files into one new file |
| *mkdir*(path) | Make new directory |
| *mv*(path1, path2) | Move file between locations on ADL |
| *open*(path[, mode, blocksize, delimiter]) | Open a file for reading or writing |
| *put*(filename, path[, delimiter]) | Stream data from local filename to file at path |
| *read_block*(fn, offset, length[, delimiter]) | Read a block of bytes from an ADL file |
| *remove*(path[, recursive]) | Remove a file. |
| *rename*(path1, path2) | Move file between locations on ADL |
| *rm*(path[, recursive]) | Remove a file. |
| *rmdir*(path) | Remove empty directory |
| *stat*(path) | File information |
| *tail*(path[, size]) | Return last bytes of file |
| *touch*(path) | Create empty file |
| *unlink*(path[, recursive]) | Remove a file. |
| *walk*([path]) | Get all files below given path |

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

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

**chmod**(*path*, *mod*)
Change access mode of path

Note this is not recursive.

**Parameters** **path: str**

Location to change

**mod: str**

Octal representation of access, e.g., "0777" for public read/write. See [docs](http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/WebHDFS.html#Permission)

**chown**(*path*, *owner=None*, *group=None*)
Change owner and/or owning group

Note this is not recursive.

**Parameters** **path: str**

Location to change

**owner: str**

UUID of owning entity

**group: str**

UUID of group

**concat**(*outfile*, *filelist*, *delete_source=False*)
Concatenate a list of files into one new file

**Parameters** **outfile** : path

The file which will be concatenated to. If it already exists, the extra pieces will be appended.

**filelist** : list of paths

Existing adl files to concatenate, in order

**delete_source** : bool (False)

If True, assume that the paths to concatenate exist alone in a directory, and delete that whole directory when done.

**connect**()
Establish connection object.

**cp**(*path1*, *path2*)
Copy file between locations on ADL

**classmethod** **current**()
Return the most recently created AzureDLFileSystem

**df**(*path*)
Resource summary of path

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

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

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

**glob**(*path*)
Find files (not directories) by glob-matching.

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

**info**(*path*)
File information

**invalidate_cache**(*path=None*)
Remove entry from object file-cache

**listdir**(*path=''*, *detail=False*)
List single directory with or without details

**ls**(*path=''*, *detail=False*)
List single directory with or without details

**merge**(*outfile*, *filelist*, *delete_source=False*)
Concatenate a list of files into one new file

**Parameters** **outfile** : path

The file which will be concatenated to. If it already exists, the extra pieces will be appended.

**filelist** : list of paths

Existing adl files to concatenate, in order

**delete_source** : bool (False)

If True, assume that the paths to concatenate exist alone in a directory, and delete that whole directory when done.

**mkdir**(*path*)
Make new directory

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

**open**(*path*, *mode='rb'*, *blocksize=33554432*, *delimiter=None*)
Open a file for reading or writing

**Parameters** **path: string**

Path of file on ADL

**mode: string**

One of 'rb' or 'wb'

**blocksize: int**

Size of data-node blocks if reading

**delimiter: byte(s) or None**

For writing delimiter-ended blocks

**put** (*filename*, *path*, *delimiter=None*)
Stream data from local filename to file at path

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

**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

```
>>> adl.read_block('data/file.csv', 0, 13)
b'Alice, 100\nBo'
>>> adl.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. >>> adl.read_block('data/file.csv', 0, None, delimiter=b'n') # doctest: +SKIP b'Alice, 100nBob, 200nCharlie, 300'

**remove** (*path*, *recursive=False*)
Remove a file.

**Parameters** **path** : string

The location to remove.

**recursive** : bool (True)

Whether to remove also all entries below, i.e., which are returned by *walk()*.

**rename** (*path1*, *path2*)
Move file between locations on ADL

**rm** (*path*, *recursive=False*)
Remove a file.

**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*)
Remove empty directory

**stat**(*path*)
File information

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

**touch**(*path*)
Create empty file

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

**unlink**(*path*, *recursive=False*)
Remove a file.

**Parameters path** : string

The location to remove.

**recursive** : bool (True)

Whether to remove also all entries below, i.e., which are returned by *walk()*.

**walk**(*path=''*)
Get all files below given path

**class** azure.datalake.store.multithread.**ADLUploader**(*adlfs*, *rpath*, *lpath*, *nthreads=None*, *chunksize=268435456*, *buffersize=4194304*, *blocksize=4194304*, *client=None*, *run=True*, *delimiter=None*, *overwrite=False*, *verbose=True*)
Upload local file(s) using chunks and threads

Launches multiple threads for efficient uploading, with *chunksize* assigned to each. The path can be a single file, a directory of files or a glob pattern.

**Parameters adlfs: ADL filesystem instance**

**rpath: str**

remote path to upload to; if multiple files, this is the dircetory root to write within

**lpath: str**

local path. Can be single file, directory (in which case, upload recursively) or glob pattern. Recursive glob patterns using ** are not supported.

**nthreads: int [None]**

Number of threads to use. If None, uses the number of cores.

**chunksize: int [2**28]**

Number of bytes for a chunk. Large files are split into chunks. Files smaller than this number will always be transferred in a single thread.

**buffersize: int [2**22]**

Number of bytes for internal buffer. This block cannot be bigger than a chunk and cannot be smaller than a block.

**blocksize: int [2**22]**

Number of bytes for a block. Within each chunk, we write a smaller block for each API call. This block cannot be bigger than a chunk.

**client: ADLTransferClient [None]**

Set an instance of ADLTransferClient when finer-grained control over transfer parameters is needed. Ignores *nthreads*, *chunksize*, and *delimiter* set by constructor.

**run: bool [True]**

Whether to begin executing immediately.

**delimiter: byte(s) or None**

If set, will write blocks using delimiters in the backend, as well as split files for uploading on that delimiter.

**overwrite: bool [False]**

Whether to forcibly overwrite existing files/directories. If False and remote path is a directory, will quit regardless if any files would be overwritten or not. If True, only matching filenames are actually overwritten.

**See also:**

`azure.datalake.store.transfer.ADLTransferClient`

#### Attributes

| |
|---|
| `hash` |

#### Methods

| | |
|---|---|
| *active*() | Return whether the uploader is active |
| *clear_saved*() | Remove references to all persisted uploads. |
| *load*() | Load list of persisted transfers from disk, for possible resumption. |
| *run*([nthreads, monitor]) | Populate transfer queue and execute downloads |
| *save*([keep]) | Persist this upload |
| *successful*() | Return whether the uploader completed successfully. |

**active**()
Return whether the uploader is active

static **clear_saved**()
Remove references to all persisted uploads.

static **load**()
Load list of persisted transfers from disk, for possible resumption.

**Returns** A dictionary of upload instances. The hashes are auto-

generated unique. The state of the chunks completed, errored, etc., can be seen in the status attribute. Instances can be resumed with `run()`.

**run**(*nthreads=None*, *monitor=True*)
Populate transfer queue and execute downloads

**Parameters** **nthreads: int [None]**

Override default nthreads, if given

**monitor: bool [True]**

To watch and wait (block) until completion.

**save**(*keep=True*)

Persist this upload

Saves a copy of this transfer process in its current state to disk. This is done automatically for a running transfer, so that as a chunk is completed, this is reflected. Thus, if a transfer is interrupted, e.g., by user action, the transfer can be restarted at another time. All chunks that were not already completed will be restarted at that time.

See methods `load` to retrieved saved transfers and `run` to resume a stopped transfer.

**Parameters** **keep: bool (True)**

If True, transfer will be saved if some chunks remain to be completed; the transfer will be sure to be removed otherwise.

**successful**()

Return whether the uploader completed successfully.

It will raise AssertionError if the uploader is active.

**class** azure.datalake.store.multithread.**ADLDownloader**(*adlfs*, *rpath*, *lpath*, *nthreads=None*, *chunksize=268435456*, *buffersize=4194304*, *blocksize=4194304*, *client=None*, *run=True*, *overwrite=False*, *verbose=True*)

Download remote file(s) using chunks and threads

Launches multiple threads for efficient downloading, with *chunksize* assigned to each. The remote path can be a single file, a directory of files or a glob pattern.

**Parameters** **adlfs: ADL filesystem instance**

**rpath: str**

remote path/globstring to use to find remote files. Recursive glob patterns using ** are not supported.

**lpath: str**

local path. If downloading a single file, will write to this specific file, unless it is an existing directory, in which case a file is created within it. If downloading multiple files, this is the root directory to write within. Will create directories as required.

**nthreads: int [None]**

Number of threads to use. If None, uses the number of cores.

**chunksize: int [2**28]**

Number of bytes for a chunk. Large files are split into chunks. Files smaller than this number will always be transferred in a single thread.

**buffersize: int [2**22]**

Number of bytes for internal buffer. This block cannot be bigger than a chunk and cannot be smaller than a block.

**blocksize: int [2**22]**

Number of bytes for a block. Within each chunk, we write a smaller block for each API call. This block cannot be bigger than a chunk.

**client: ADLTransferClient [None]**

Set an instance of ADLTransferClient when finer-grained control over transfer parameters is needed. Ignores *nthreads* and *chunksize* set by constructor.

**run: bool [True]**

Whether to begin executing immediately.

**overwrite: bool [False]**

Whether to forcibly overwrite existing files/directories. If False and local path is a directory, will quit regardless if any files would be overwritten or not. If True, only matching filenames are actually overwritten.

**See also:**

`azure.datalake.store.transfer.ADLTransferClient`

#### Attributes

| |
|---|
| `hash` |

#### Methods

| | |
|---|---|
| *active*() | Return whether the downloader is active |
| *clear_saved*() | Remove references to all persisted downloads. |
| *load*() | Load list of persisted transfers from disk, for possible resumption. |
| *run*([nthreads, monitor]) | Populate transfer queue and execute downloads |
| *save*([keep]) | Persist this download |
| *successful*() | Return whether the downloader completed successfully. |

**active**()
Return whether the downloader is active

static **clear_saved**()
Remove references to all persisted downloads.

static **load**()
Load list of persisted transfers from disk, for possible resumption.

**Returns** A dictionary of download instances. The hashes are auto-

generated unique. The state of the chunks completed, errored, etc., can be seen in the status attribute. Instances can be resumed with `run()`.

**run**(*nthreads=None*, *monitor=True*)
Populate transfer queue and execute downloads

**Parameters nthreads: int [None]**

Override default nthreads, if given

**monitor: bool [True]**

To watch and wait (block) until completion.

**save**(*keep=True*)
Persist this download

Saves a copy of this transfer process in its current state to disk. This is done automatically for a running transfer, so that as a chunk is completed, this is reflected. Thus, if a transfer is interrupted, e.g., by user action, the transfer can be restarted at another time. All chunks that were not already completed will be restarted at that time.

See methods `load` to retrieved saved transfers and `run` to resume a stopped transfer.

**Parameters keep: bool (True)**

If True, transfer will be saved if some chunks remain to be completed; the transfer will be sure to be removed otherwise.

**successful**()
Return whether the downloader completed successfully.

It will raise AssertionError if the downloader is active.

CHAPTER 7

# Indices and tables

- genindex
- modindex
- search

## P



## R



## S



## T



## U



## W