DataFS Data Management System Documentation

Release 0.7.0

Climate Impact Lab

Apr 10, 2017

Contents

DataFS provides an abstraction layer for data storage systems. You can combine versioning, metadata management, team permissions and logging, and file storage all in a simple, intuitive interface.

Our goal is to make reading and writing large numbers of files in a team in a distributed, performance-critical computing environment feel as easy and intuitive as working with a handful of files on a local machine.

Contents:

CHAPTER 1

DataFS Data Management System

DataFS is a package manager for data. It manages file versions, dependencies, and metadata for individual use or large organizations.

Configure and connect to a metadata Manager and multiple data Services using a specification file and you'll be sharing, tracking, and using your data in seconds.

- Free software: MIT license
- Documentation: https://datafs.readthedocs.io.

Features

- Explicit version and metadata management for teams
- Unified read/write interface across file systems
- Easily create out-of-the-box configuration files for users
- Track data dependencies and usage logs
- Use datafs from python or from the command line
- Permissions handled by managers & services, giving you control over user access

Usage

First, configure an API. Don't worry. It's not too bad. Check out the quickstart to follow along.

We'll assume we already have an API object created and attached to a service called "local". Once you have this, you can start using DataFS to create and use archives.

```
$ datafs create my_new_data_archive --description "a test archive"
created versioned archive <DataArchive local://my_new_data_archive>

$ echo "initial file contents" > my_file.txt

$ datafs update my_new_data_archive my_file.txt

$ datafs cat my_new_data_archive
initial file contents
```

Versions are tracked explicitly. Bump versions on write, and read old versions if desired.

```
$ echo "updated contents" > my_file.txt

$ datafs update my_new_data_archive my_file.txt --bumpversion minor
uploaded data to <DataArchive local://my_new_data_archive>. version bumped 0.0.1 -->␣
↪0.1.

$ datafs cat my_new_data_archive
updated contents

$ datafs cat my_new_data_archive --version 0.0.1
initial file contents
```

Pin versions using a requirements file to set the default version

```
$ echo "my_new_data_archive==0.0.1" > requirements_data.txt

$ datafs cat my_new_data_archive
initial file contents
```

All of these features are available from (and faster in) python:

```
>>> import datafs
>>> api = datafs.get_api()
>>> archive = api.get_archive('my_new_data_archive')
>>> with archive.open('r', version='latest') as f:
...     print(f.read())
...
updated contents
```

If you have permission to delete archives, it's easy to do. See administrative tools for tips on setting permissions.

```
$ datafs delete my_new_data_archive
deleted archive <DataArchive local://my_new_data_archive>
```

See examples for more extensive use cases.

Installation

`pip install datafs`

Additionally, you'll need a manager and services:

Managers:

- MongoDB: `pip install pymongo`

- DynamoDB: `pip install boto3`

Services:

- Ready out-of-the-box:
  - local
  - shared
  - mounted
  - zip
  - ftp
  - http/https
  - in-memory
- Requiring additional packages:
  - AWS/S3: `pip install boto`
  - SFTP: `pip install paramiko`
  - XMLRPC: `pip install xmlrpclib`

Requirements

For now, DataFS requires python 2.7. We're working on 3x support.

Todo

See issues to see and add to our todos.

Credits

This package was created by Justin Simcock and Michael Delgado of the Climate Impact Lab. Check us out on github.

Major kudos to the folks at PyFilesystem. Thanks also to audreyr for the wonderful cookiecutter package, and to Pyup, a constant source of inspiration and our silent third contributor.

CHAPTER 2

Quickstart

Setup

- Create a local working data directory (e.g. `mkdir ~/datafs`)
- download `this file`
- copy it into the file opened with `datafs configure --edit`, editing the data directory `~/datafs` to match the one you created.
- install and configure a local manager

Using the command line

Create an archive

```
$ datafs create new_archive
```

Use the archive

```
$ echo "initial contents" | datafs update new_archive --string

$ datafs cat new_archive
initial contents

$ echo "new contents" | datafs update new_archive --string

$ datafs cat new_archive
new contents

$ datafs cat new_archive --version 0.0.1
initial contents
```

See the full *command line documentation*

Using python

```
>>> import datafs

>>> api = datafs.get_api()

>>> archive = api.get_archive('new_archive')

>>> with archive.open('r') as f:
...     print(f.read())
new contents

>>> with archive.open('w+', bumpversion='major') as f:
...     f.write(u'first release')
...

>>> archive.get_versions()
['0.0.1', '0.0.2', '1.0']
```

Use other packages:

```
>>> import pandas as pd, numpy as np

>>> with archive.open('w+b') as f:
...     pd.DataFrame(np.random.random(4,5)).to_csv(f)

>>> with archive.open('w+') as f:
...     f.write(u'')

>>> with archive.open('r', version='1.0.1') as f:
...     print(pd.read_csv(f))
...
          0         1         2         3         4
0  0.064340  0.266545  0.739165  0.892549  0.576971
1  0.586370  0.903017  0.874171  0.046859  0.747309
2  0.349005  0.628717  0.638336  0.670759  0.493050
3  0.323830  0.697789  0.006091  0.629318  0.039715
```

See the full *python api documentation*

CHAPTER 3

Installation

To use DataFS, you'll need to install DataFS and a working combination of *Managers* and *Services*.

Installing DataFS

Stable release

To install DataFS Data Management System, run this command in your terminal:

```
$ pip install datafs
```

This is the preferred method to install DataFS Data Management System, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

From sources

The sources for DataFS Data Management System can be downloaded from the Github repo.

You can either clone the public repository:

Or download the tarball:

Once you have a copy of the source, you can install it with:

Additional dependencies

In addition to DataFS, you'll need to install and configure a manager and any services you'd like to use.

See the *Examples* to see managers and services in use.

Managers

You'll need to connect to a MongoDB or DynamoDB database to use DataFS. These can be local installs for demos/testing/personal use or full installations for teams.

You can download the local installations here:

- MongoDB local
- DynamoDB local

To connect to these databases, you'll need to install the python interfaces to them:

- MongoDB: `pip install pymongo`
- DynamoDB: `pip install boto3`

To configure a manager for datafs, you'll need to be able to connect to the database using these interfaces. For help, check out their documentation:

- MongoDB Tutorial
- DynamoDB Quickstart

Services

Similar to *Managers*, you'll need data at least one storage service to use DataFS. For local/testing/personal use, a local directory can be useful, and is the easiest to set up.

- Ready out-of-the-box:
  - local
  - shared
  - mounted
  - zip
  - ftp
  - http/https
  - in-memory
- Requiring additional packages:
  - AWS/S3: `pip install boto`
  - SFTP: `pip install paramiko`
  - XMLRPC: `pip install xmlrpclib`

CHAPTER 4

Configuring DataFS

Specifying API objects

Specifying API objects manually from within python

```
>>> from datafs.managers.manager_mongo import MongoDBManager
>>> from datafs import DataAPI
>>> from fs.osfs import OSFS
>>> import tempfile
>>>
>>> api = DataAPI(
...       username='My Name',
...       contact = 'my.email@example.com')
...
>>> manager = MongoDBManager(
...      database_name = 'MyDatabase',
...      table_name = 'DataFiles')
...
>>> manager.create_archive_table('DataFiles')
>>>
>>> api.attach_manager(manager)
>>>
>>> local = OSFS('~/datafs/my_data/')
>>>
>>> api.attach_authority('local', local)
```

Specifying an API object with a specifcation file

Alternatively, you can do the other thing.

Configuring a Data Manager

Adding Data Stores

Caching Files Locally

Configuration with Specifcation Files

CHAPTER 5

Using the Python API

Creating Data Archives

Archives are the basic unit of a DataFS filesystem. They are essentially files, metadata, history, versions, and dependencies wrapped into a single object.

You can create archives from within python or using the *command line interface*.

View the source for the code samples on this page in *Python API: Creating Archives*.

Naming Archives

Archives can be named anything, as long as the data service you use can handle the name. If create an archive with a name illegal for the corresponding data service, you will receive an error on write (rather than on archive creation). Since this is an error specific to the storate service, we do not catch this error on creation.

Create an archive using the *datafs.DataAPI.create()* command.

```
>>> archive = api.create('my_archive_name')
```

Specifying an Authority

If you have more than one authority, you will need to specify an authority on archive creation:

```
>>> archive = api.create('my_archive_name')
Traceback (most recent call last):
...
ValueError: Authority ambiguous. Set authority or DefaultAuthorityName.
```

This can be done using the `authority_name` argument:

```
>>> archive = api.create(
...     'my_archive_name',
...     authority_name='my_authority')
...
```

Alternatively, you can set the *DefaultAuthorityName* attribute:

```
>>> api.DefaultAuthorityName = 'my_authority'
>>> archive = api.create('my_archive_name')
```

Adding Metadata

Arbitrary metadata can be added using the `metadata` dictionary argument:

```
>>> archive = api.create(
...     'my_archive_name',
...     metadata={
...         'description': 'my test archive',
...         'source': 'Burke et al (2015)',
...         'doi': '10.1038/nature15725'})
...
```

Required Metadata

Administrators can set up metadata requirements using the manager's *Administrative Tools* tools. If these required fields are not provided, an error will be raised on archive creation.

For example, when connected to a manager requiring the *'description'* field:

```
>>> archive = api.create(
...     'my_archive_name',
...     metadata = {
...         'source': 'Burke et al (2015)',
...         'doi': '10.1038/nature15725'})
...
Traceback (most recent call last):
...
AssertionError: Required value "description" not found. Use helper=True or
the --helper flag for assistance.
```

Trying again with a "description" field will work as expected.

Using the Helper

Instead of providing all fields in the `create` call, you can optionally use the `helper` argument. Setting `helper=True` will start an interactive prompt, requesting each required item of metadata:

```
>>> archive = api.create(
...     'my_archive_name',
...     metadata={
...         'source': 'Burke et al (2015)',
...         'doi': '10.1038/nature15725'},
...         helper=True)
```

```
...
Enter a description:
```

Tagging Archives

You can tag archives from within python or using the *command line interface*.

View the source for the code samples on this page in *Python API: Tagging*.

Tagging on archive creation

When creating archives, you can specify tags using the `tags` argument. You can specify as many as you would like as elements in a list:

```
>>> archive1 = api.create(
...       'archive1',
...       tags=["foo", "bar"],
...       metadata={'description': 'tag test 1 has bar'})
...
>>> archive2 = api.create(
...       'archive2',
...       tags=["foo", "baz"],
...       metadata={'description': 'tag test 1 has baz'})
...
```

You can then search for archives that have these tags using the *`search()`* method:

```
>>> list(api.search('bar'))
['archive1']

>>> list(api.search('baz'))
['archive2']

>>> list(api.search('foo'))
['archive1', 'archive2']
```

Searching for multiple tags yields the set of archives that match all of the criteria:

```
>>> archive3 = api.create(
...       'archive3',
...       tags=["foo", "bar", "baz"],
...       metadata={'description': 'tag test 3 has all the tags!'})
...

>>> list(api.search('bar', 'foo'))
['archive1', 'archive3']

>>> list(api.search('bar', 'foo', 'baz'))
['archive3']
```

Searches that include a set of tags not jointly found in any archives yield no results:

```
>>> list(api.search('qux'))
[]
```

```
>>> list(api.search('foo', 'qux'))
[]
```

Viewing and modifying tags on existing archives

Tags can be listed using the *get_tags()* method:

```
>>> archive1.get_tags()
['foo', 'bar']
```

You can add tags to an archive using the *add_tags()* method:

```
>>> archive1.add_tags('qux')
>>>
>>> list(api.search('foo', 'qux'))
['archive1']
```

Removing tags from an archive

Specific tags can be removed from an archive using the *delete_tags()* method:

```
>>> archive1.delete_tags('foo', 'bar')
>>>
>>> list(api.search('foo', 'bar'))
['archive3']
```

Reading and Writing Files

View the source for the code samples on this page in *Python API: Reading and Writing*.

Writing Archive Files

To write to a file in DataFS you'll first need to create the archive. We'll assume you've already set up your api with manager, and authorities. We'll use *create()*.

```
>>> api.create(
...     'sample_archive',
...     metadata={'description': 'metadata for your archive'})
...
<DataArchive local://sample_archive>
```

DataFS needs a file path in order to update files. So now that we have an archive on our manager, we can create a file and put it up in our filesystem. So let's first create a file called `sample.txt`.

```
>>> with open('sample.txt', 'w+') as f:
...     f.write('this is a sample archive')
...
```

Now that we have a file can we can use the `update()` method to upload our file. Depending on size of file and network speeds, there may be some latency when calling `update()`.

```
>>> sample_var = api.get_archive('sample_archive')
>>> sample_var.update('sample.txt')
```

Now that our archive is up let's say we want to now pull it down and read it. Reading an archive file is an interface the python users will recognize. We initialize a context manager using the `with` statement. The actual call happens through *`open()`*

```
>>> with sample_var.open('r') as f:
...     print(f.read())
...
this is a sample archive
```

This is really great. Let's see what happens when we want to make some updates to the file.

```
>>> with open('sample.txt', 'w+') as f:
...     f.write('this is a sample archive with some more information')
...
>>> sample_var.update('sample.txt')
```

Now let's open and read to see.

```
>>> with sample_var.open('r') as f:
...     print(f.read())
...
this is a sample archive with some more information
```

Looks good!

Since DataFS simply needs a filepath, you can simply provide a filepath and it will upload and write the file. If you have a file locally that you want managed by DataFS you can create an archive and put it on your filesystem.

```
>>> sample_var.update('sample.txt')
```

Downloading

If you want to download the latest version of an archive all you need to do is provide a path and set `version='latest'`. This will download the latest version to the filepath specified. We'll use `get_archive()` to get the archive and then use *`download()`*

```
>>> sample_archive_local = api.get_archive('sample_archive')
>>> sample_archive_local.download('path_to_sample.txt', version='latest')
```

Let's just double check that we indeed have our file

```
>>> with open('path_to_sample.txt', 'r') as f:
...     print(f.read())
...
Local file to update to our FS
```

Writing Streaming Objects

If you are working with certain packages like pandas, or xarray that need a filepath, the interaction is slightly modified from typical file objects. Let's first create the dataset we want to write to. The method we'll use for this operation is `datafs.core.DataArchive.get_local_path()` and xarray's open_dataset method

```
>>> import numpy as np
>>> import pandas as pd
>>> import xarray as xr
>>>
>>> np.random.seed(123)
>>>
>>> times = pd.date_range('2000-01-01', '2001-12-31', name='time')
>>> annual_cycle = np.sin(2 * np.pi * (times.dayofyear / 365.25 - 0.28))
>>>
>>> base = 10 + 15 * annual_cycle.reshape(-1, 1)
>>> tmin_values = base + 3 * np.random.randn(annual_cycle.size, 3)
>>> tmax_values = base + 10 + 3 * np.random.randn(annual_cycle.size, 3)
>>>
>>> ds = xr.Dataset({'tmin': (('time', 'location'), tmin_values),
...                  'tmax': (('time', 'location'), tmax_values)},
...                 {'time': times, 'location': ['IA', 'IN', 'IL']})
>>>
>>>
>>>
>>> streaming_archive = api.create(
...     'streaming_archive',
...     metadata={'description': 'metadata description for your archive'})
...
>>> with streaming_archive.get_local_path() as f:
...       ds.to_netcdf(f)
...
>>>
>>>
```

Downloading Streaming Objects

Reading a streaming object is similar to reading a regular file object but we generate a file path that is then passed to the package you are using for reading and writing. In this case we are using xarray so we'll use our *get_local_path()* and xarray's open_dataset method

```
>>> with streaming_archive.get_local_path() as f:
...     with xr.open_dataset(f) as ds:
...         print(ds)
...
<xarray.Dataset>
Dimensions:   (location: 3, time: 731)
Coordinates:
  * location  (location) |S2 'IA' 'IN' 'IL'
  * time      (time) datetime64[ns] 2000-01-01 2000-01-02 2000-01-03 ...
Data variables:
    tmax      (time, location) float64 12.98 3.31 6.779 0.4479 6.373 ...
    tmin      (time, location) float64 -8.037 -1.788 -3.932 -9.341 ...
```

Check out *Examples* for more information on how to write and read files DataFS on different filesystems

Managing Metadata

Metadata management is one of the core components of DataFS. Metadata is managed at the Archive level and the version level. This documentation will refer to the Archive level metadata.

View the source for the code samples on this page in *Python API: Managing Metadata*

We'll assume that your manager and authority have been set up already. Depending on usage, metadata requirements can be enforced and archive creation will be possible only with inclusion of the required metadata.

```
>>> sample_archive = api.create(
...     'sample_archive',
...      metadata = {
...          'oneline_description': 'tas by admin region',
...          'source': 'NASA BCSD',
...          'notes': 'important note'})
...
```

Let's check to see if we have any other archives. It looks like we only have our `sample_archive`.

```
>>> for archive_name in api.filter():
...     print(archive_name)
...
sample_archive
```

Nope just this one. So let's have a look at the metadata using *`get_metadata()`*.

```
>>> sample_archive.get_metadata()
{u'notes': u'important note', u'oneline_description': u'tas by admin region
', u'source': u'NASA BCSD'}
```

Let's say you later on realize that you want to add another field to the archive metadata and that you also want to update on of the required fields. The archive metadata is modified in place and so you'll be able to do both at the same time with *`update_metadata()`*

```
>>> sample_archive.update_metadata(dict(
...     source='NOAAs better temp data',
...     related_links='http://wwww.noaa.gov'))
...
```

As you can see the source was updated and the `related_links` key and value were added.

```
>>> sample_archive.get_metadata()
{u'notes': u'important note', u'oneline_description': u'tas by admin region
', u'source': u'NOAAs better temp data', u'related_links': u'http://wwww.no
aa.gov'}
```

As long as a particular metadata key-value pair is not a required field, you can remove it. If you want to remove a particular key-value pair from the archive metadata, set the value to None:

```
>>> sample_archive.update_metadata(dict(related_links=None))
>>>
>>> sample_archive.get_metadata()
{u'notes': u'important note', u'oneline_description': u'tas by admin region
', u'source': u'NOAAs better temp data'}
```

Now our `related_links` key-value pair has been removed. To edit the required metadata fields, please see *Administrative Tools*.

Tracking Versions

One of the main features of DataFS is its ability to manage versions of archives. In the *Managing Metadata* section we worked with Archive level metadata. In this section we will work with versioning and demonstrate some of its metadata properties.

We'll assume you have an api created and configured with a manager and authority. On `create()`, versioning is true by default. Should you want to turn it off just set `versioned=False`.

View the source for the code samples on this page in *Python API: Versioning Data*.

Setting a Version

Let's write to a file that we'll upload to our `sample_archive` and see how DataFS manages versioning. Once we've created our arhive we'll use `update()` to start tracking versions. This time we'll set our version as `prerelease` and set it to `alpha`. `beta` is also an option.

```
>>> with open('sample_archive.txt', 'w+', ) as f:
...     f.write(u'this is a sample archive')
...
>>> sample_archive = api.create(
...     'sample_archive',
...     metadata = {'description': 'metadata description'})
...
>>> sample_archive.update('sample_archive.txt', prerelease='alpha')
```

Getting Versions

Now let's use `get_versions()` to see which versions we are tracking

```
>>> sample_archive.get_versions()
[BumpableVersion ('0.0.1a1')]
```

That's pretty cool.

Bumping Versions

Now let's make some changes to our archive files. When we call `update()` we'll specify `bumpversion='minor'`. The `bumpversion` param takes values of `major` , `minor` or `patch`.

```
>>> with open('sample_archive.txt', 'w+', ) as f:
...     f.write(u'Sample archive with more text so we can bumpversion')
...
>>>
>>> sample_archive.update('sample_archive.txt', bumpversion='minor')
>>> sample_archive.get_versions()
[BumpableVersion ('0.0.1a1'), BumpableVersion ('0.1')]
```

Getting the latest version

What if I want to see what the latest version is? You can use the `get_latest_version()`

```
>>> sample_archive.get_latest_version()
BumpableVersion ('0.1')
```

Getting the latest hash

So we can see that it will return the latest version which in this case is the `minor` bump that we just did. How does it know about this? DataFS hashes the file contents of every version and creates a unique hash for every file. Each time an update is made to the file contents a hash is made and saved. You can access this value with *`get_latest_hash()`*

```
>>> sample_archive.get_latest_hash()
u'510d0e2eadd19514788e8fdf91e3bd5c'
```

Getting a specific version

Let's say we want to get an older version. We can do this by specifying `version` in *`get_archive()`*

```
>>> sample_archive1 = api.get_archive(
...     'sample_archive',
...     default_version='0.0.1a1')
...
>>> with sample_archive1.open('r') as f:
...    print(f.read())
...
Sample archive with more text so we can bumpversion
```

We can see that this is our first version that saved as a prerelease alpha.

To see more information on versioning check out `BumpableVersion`.

Managing Data Dependencies

Dependency graphs can be tracked explicitly in datafs, and each version can have its own dependencies.

You specify dependencies from within python or using the *command line interface*.

Note: Dependencies are not currently validated in any way, so entering a dependency that is not a valid archive name or version will not raise an error.

View the source for the code samples on this page in *Python API: Dependencies*.

Specifying Dependencies

On write

Dependencies can be set when using the `dependencies` argument to *`DataArchive`*'s *`update()`*, *`open()`*, or *`get_local_path()`* methods.

`dependencies` must be a dictionary containing archive names as keys and version numbers as values. A value of `None` is also a valid dependency specification, where the version is treated as unpinned and is always interpreted as the dependency's latest version.

For example:

```
>>> my_archive = api.create('my_archive')
>>> with my_archive.open('w+',
...     dependencies={'archive2': '1.1', 'archive3': None}) as f:
...
...     res = f.write(u'contents depend on archive 2 v1.1')
...
>>> my_archive.get_dependencies()
{'archive2': '1.1', 'archive3': None}
```

After write

Dependencies can also be added to the latest version of an archive using the *set_dependencies()* method:

```
>>> with my_archive.open('w+') as f:
...
...     res = f.write(u'contents depend on archive 2 v1.2')
...
>>> my_archive.set_dependencies({'archive2': '1.2'})
>>> my_archive.get_dependencies()
{'archive2': '1.2'}
```

Using a requirements file

If a requirements file is present at api creation, all archives written with that api object will have the specified dependencies by default.

For example, with the following requirements file as `requirements_data.txt`:

```
1 dep1==1.0
2 dep2==0.4.1a3
```

Archives written while in this working directory will have these requirements:

```
>>> api = datafs.get_api(
...     requirements = 'requirements_data.txt')
...
>>>
>>> my_archive = api.get_archive('my_archive')
>>> with my_archive.open('w+') as f:
...     res = f.write(u'depends on dep1 and dep2')
...
>>> my_archive.get_dependencies()
{'dep1': '1.0', 'dep2': '0.4.1a3'}
```

Using Dependencies

Retrieve dependencies with *DataArchive*'s *get_dependencies()* method:

```
>>> my_archive.get_dependencies()
{'dep1': '1.0', 'dep2': '0.4.1a3'}
```

Get dependencies for older versions using the `version` argument:

```
>>> my_archive.get_dependencies(version='0.0.1')
{'archive2': '1.1', 'archive3': None}
```

Searching and Finding Archives

DataFS allows you to search and locate archives with the following methods: *listdir()*, *filter()*, and *search()*. Let's look at each method to see how they work.

Using `listdir()`

listdir() works just like typlical unix style `ls` in the sense that it returns all objects subordinate to the specified directory. If your team has used `/` to organize archive naming then you can explore the archive namespace just as you would explore a directory in a filesystem.

For example if we provide `impactlab/conflict/global` as an argument to `listdir` we get the following:

```
>>> api.listdir('impactlab/conflict/global')
[u'conflict_glob_day.csv']
```

It looks like we only have one file `conflict_global_daily.csv` in our directory.

Let's see what kind of archives we have in our system.

```
>>> api.listdir('')
[u'impactlab']
```

It looks like our top level directory is `impactlab`.

Then if we use `impactlab` as an argument we see that we have several directory-like groupings below this.

```
>>> api.listdir('impactlab')
[u'labor', u'climate', u'conflict', u'mortality']
```

Let's explore `conflict` to see what kind of namespace groupings we have in there.

```
>>> api.listdir('impactlab/conflict')
[u'global']
```

OK. Just one. Now let's have a look inside the `impactlab/conflict/global` namespace.

```
>>> api.listdir('impactlab/conflict/global')
[u'conflict_glob_day.csv']
>>> api.listdir('impactlab/conflict/global/conflict_glob_day.csv')
[u'0.0.1']
```

We see that if we give a full path with a file extension that we get version numbers of our archives.

Using `filter()`

DataFS also lets you filter so you can limit the search space on archive names. With *filter()* you can use the `prefix`, `path`, `str`, and `regex` pattern options to filter archives. Let's look at using the prefix `project1_variable1_` which corresponds to the `prefix` option, the beginning string of a set of archive names. Let's also see how many archives we have in total by filtering without arguments.

```
>>> len(list(api.filter()))
125
>>> filtered_list1 = api.filter(prefix='project1_variable1_')
>>> list(filtered_list1)
[u'project1_variable1_scenario1.nc', u'project1_variable1_scenario2.nc',
u'project1_variable1_scenario3.nc', u'project1_variable1_scenario4.nc',
u'project1_variable1_scenario5.nc']
```

We see there are 125. By filtering with our prefix we can significantly reduce the number of archives we are looking at.

We can also filter on `path`. In this case we want to filter all NetCDF files that match a specific pattern. We need to set our `engine` value to `path` and put in our search pattern.

```
>>> filtered_list2 = api.filter(pattern='*_variable4_scenario4.nc',
...     engine='path')
>>> list(filtered_list2)
[u'project1_variable4_scenario4.nc', u'project2_variable4_scenario4.nc',
u'project3_variable4_scenario4.nc', u'project4_variable4_scenario4.nc',
u'project5_variable4_scenario4.nc']
```

We can also filter archives with archive names containing a specific string by setting `engine` to `str`. In this example we want all archives with the string `variable2`. The filtering query returns 25 items. Let's look at the first few.

```
>>> filtered_list3 = list(api.filter(pattern='variable2', engine='str'))
>>> len(filtered_list3)
25
>>> filtered_list3[:4]
[u'project1_variable2_scenario1.nc', u'project1_variable2_scenario2.nc',
u'project1_variable2_scenario3.nc', u'project1_variable2_scenario4.nc']
```

Using `search()`

DataFS *`search()`* capabilites are enabled via tagging of archives. The arguments of the *`search()`* method are tags associated with a given archive. If archives are not tagged, they cannot be searched with the *`search()`* method. See *Tagging Archives* for info on how to tag archives.

If we use *`search()`* without arguments, it is the same implementation as *`filter()`* without arguments.

Let's see this in action.

```
>>> archives_search = list(api.search())
>>> archives_filter = list(api.filter())
>>> len(archives_search)
125
>>> len(archives_filter)
125
```

Our archives have been tagged with `team1`, `team2`, or `team3` Let's search for some archives with tag `team3`. It brings back 41 archives. So we'll just look at a few.

```
>>> tagged_search = list(api.search('team3'))
>>> len(tagged_search)
41
>>> tagged_search[:4]
[u'project1_variable1_scenario2.nc', u'project1_variable2_scenario1.nc',
u'project1_variable2_scenario3.nc', u'project1_variable3_scenario2.nc']
```

And lets look at the some of these archives to see what their tags are. We'll use *get_tags()*

```
>>> tags = []
>>> for arch in tagged_search[:4]:
...     tags.append(api.manager.get_tags(arch)[0])
>>> tags
[u'team3', u'team3', u'team3', u'team3']
```

```
>>> tagged_search_team1 = list(api.search('team1'))
>>> len(tagged_search_team1)
42
>>> tagged_search_team1[:4]
[u'project1_variable1_scenario1.nc', u'project1_variable1_scenario4.nc',
u'project1_variable2_scenario2.nc', u'project1_variable2_scenario5.nc']
```

And how about with tag `team1`. We see that there are 42 archives with `team1` tag.

```
>>> tagged_search_team1 = list(api.search('team1'))
>>> len(tagged_search_team1)
42
>>> tagged_search_team1[:4]
[u'project1_variable1_scenario1.nc', u'project1_variable1_scenario4.nc',
u'project1_variable2_scenario2.nc', u'project1_variable2_scenario5.nc']
```

And let's use *get_tags()* to confirm the tags are `team1`

```
>>> tags = []
>>> for arch in tagged_search_team1[:4]:
...     tags.append(api.manager.get_tags(arch)[0])
>>> tags
[u'team1', u'team1', u'team1', u'team1']
```

We want your feedback. If you have improvements or suggestions for the documentation please consider making contributions.

CHAPTER 6

Using the Command-Line Interface

Creating Data Archives

Archives are the basic unit of a DataFS filesystem. They are essentially files, metadata, history, versions, and dependencies wrapped into a single object.

You can create archives from the command line interface or from *python*.

Create an archive using the `create` command:

```
$ datafs create my_archive
created versioned archive <DataArchive local://my_archive>
```

Naming Archives

Archives can be named anything, as long as the data service you use can handle the name.

For example, Amazon's S3 storage cannot handle underscores in object names. If you create an archive with underscores in the name, you will receive an error on write (rather than on archive creation). Since this is an error specific to the storage service, we do not catch this error on creation.

Specifying an Authority

If you have more than one authority, you will need to specify an authority on archive creation:

```
$ datafs create my_archive --authority_name "my_authority"
created versioned archive <DataArchive my_authority://my_archive>
```

Adding Metadata

Arbitrary metadata can be added as keyword arguments:

```
$ datafs create my_archive --description 'my test archive'
created versioned archive <DataArchive local://my_archive>
```

Required Metadata

Administrators can set up metadata requirements using the manager's *Administrative Tools* tools. If these required fields are not provided, an error will be raised on archive creation.

For example, when connected to a manager requiring the *'description'* field:

```
$ datafs create my_archive --doi '10.1038/nature15725' \
>     --author "burke" # doctest: +ELLIPSIS +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
AssertionError: Required value "description" not found. Use helper=True or
the --helper flag for assistance.
```

Trying again with a `--description "[desc]"` argument will work as expected.

Using the Helper

Instead of providing all fields in the `create` call, you can optionally use the `helper` flag. Using the flag `--helper` will start an interactive prompt, requesting each required item of metadata:

```
$ datafs create my_archive --helper
created versioned archive <DataArchive local://my_archive>
```

Tagging Archives

You can tag archives from the command line interface or from *python*.

View the source for the code samples on this page in *Command Line Interface: Tagging*.

Tagging on archive creation

When creating archives, you can specify tags using the `--tag` argument. You can specify as many as you would like:

```
$ datafs create archive1 --tag "foo" --tag "bar" --description \
>     "tag test 1 has bar"
created versioned archive <DataArchive local://archive1>

$ datafs create archive2 --tag "foo" --tag "baz" --description \
>     "tag test 2 has baz"
created versioned archive <DataArchive local://archive2>
```

You can then search for archives that have these tags using the `search` command:

```
$ datafs search bar
archive1

$ datafs search baz
archive2
```

```
$ datafs search foo
archive1
archive2
```

Searching for multiple tags yields the set of archives that match all of the criteria:

```
$ datafs create archive3 --tag "foo" --tag "bar" --tag "baz" \
>     --description 'tag test 3 has all the tags!'
created versioned archive <DataArchive local://archive3>

$ datafs search bar foo
archive3
archive1


$ datafs search bar foo baz
archive3
```

Searches that include a set of tags not jointly found in any archives yield no results:

```
$ datafs search qux

$ datafs search foo qux
```

Viewing and modifying tags on existing archives

Tags can be listed using the `get_tags` command:

```
$ datafs get_tags archive1
foo bar
```

You can add tags to an archive using the `add_tags` command:

```
$ datafs add_tags archive1 qux

$ datafs search foo qux
archive1
```

Removing tags from an archive

Specific tags can be removed from an archive using the `delete_tags` command:

```
$ datafs delete_tags archive1 foo bar

$ datafs search foo bar
archive3
```

Reading and Writing Files

Reading from and writing to files is straight forward in DataFS. In this section we'll cover the command-line implementation of this. The *python* implementation is also available.

We'll assume you have your api configured with a manager and an authority. Check the *configure* documentation for more information on how to set up DataFS.

Listing Archives

If I want to first check to see if I have any archives I can use the *filter* command. Here we see we don't currently have any archives

```
$ datafs filter
```

So let's create an archive so we have something to work with.

```
$ datafs create my_archive
created versioned archive <DataArchive local://my_archive>
```

Now when we list we see our archive. Great!

```
$ datafs filter
my_archive
```

Writing to Archives

This time we will simply demonstrate how you can

```
$ datafs update my_archive \
>   --string 'barba crescit caput nescit' # doctest: +NORMALIZE_WHITESPACE
uploaded data to <DataArchive local://my_archive>. new version 0.0.1
created.
```

Great!

Reading from Archives

```
$ datafs download my_archive 'my_archive.txt'
downloaded v0.0.1 to my_archive.txt
```

Now let's read this to make sure we got what we want

```
$ cat my_archive.txt
barba crescit caput nescit
```

Writing to Archives with Filepaths

Let's say we made some major edits to *my_archive* locally and we want to update them in the manager and at our authority. We can update the same as before but this time we'll add the filepath that points to our file.

```
$ datafs update my_archive my_archive.txt # doctest: +NORMALIZE_WHITESPACE
uploaded data to <DataArchive local://my_archive>. version bumped 0.0.1 -->
0.0.2.
```

And now to read this file, let's download to a different spot and read from there.

```
$ datafs download my_archive my_archive_placeholder.txt
downloaded v0.0.2 to my_archive_placeholder.txt

$ cat my_archive_placeholder.txt # doctest: +NORMALIZE_WHITESPACE
barba crescit caput nescit
luctuat nec mergitur
```

We can see that our updates have been added and that they are reflected in a new version number.

Managing Metadata

In this section we'll take a look at how to access archive metadata from the command line. There is also a *python* version.

Viewing Metadata

We'll keep working with `my_archive` that we created earlier. Right now we'll take a look at our metadata.

```
$ datafs metadata my_archive # doctest: +SKIP
{u'description': u'my example archive',
 u'source': u'Burke et al. (2016)',
 u'doi': u'10.1038/nature15725',
 u'author': u'fBurke'}
```

Updating Metadata

```
$ datafs update_metadata my_archive \
>     --description 'Spatial impact meta-analysis' \
>     --method 'downscaled Burke et al (2015) data'
```

We'll need to read the metadata again to check to see if we succeeded

```
$ datafs metadata my_archive # doctest: +SKIP
{u'description': u'Spatial impact meta-analysis',
 u'source': u'Burke et al. (2016)',
 u'doi': u'10.1038/nature15725',
 u'author': u'Burke',
 u'method': u'downscaled Burke et al (2015) data'}
```

Great!

It should be noted that the command line tool does not deal with whitespaces well so you'll need to wrap text in quotes if it refers to a single entry.

Tracking Versions

In this section we'll have a look at the archive versioning options available through the command line.

We'll assume we have our api configured and that our manager and authority is already set-up. We go ahead and creat our sample archive again to demonstrate how versions are managed.

```
$ datafs create my_archive \
>   --my_metadata_field 'useful metadata'
created versioned archive <DataArchive local://my_archive>
```

So now we have our archive being tracked by manager.

```
$ datafs update my_archive --string \
>     'barba crescit caput nescit' # doctest: +NORMALIZE_WHITESPACE
uploaded data to <DataArchive local://my_archive>. new version 0.0.1
created.
```

Explicit Versioning

As we learned in our section on writing and reading archives, the version is set to 0.0.1 on creation by default. If you wanted to specify a prerelease or a minor release you would do either of the following

```
$ datafs update my_archive --bumpversion patch --string \
>     'Aliquando et insanire iucundum est' # doctest: +NORMALIZE_WHITESPACE
uploaded data to <DataArchive local://my_archive>. version bumped 0.0.1 -->
0.0.2.

$ datafs update my_archive --bumpversion minor --string \
>     'animum debes mutare non caelum' # doctest: +NORMALIZE_WHITESPACE
uploaded data to <DataArchive local://my_archive>. version bumped 0.0.2 -->
0.1.
```

Get Version History

What if we want to view our versions?

```
$ datafs versions my_archive
['0.0.1', '0.0.2', '0.1']
```

Downloading Specific Versions

How can we get a specific version?

```
$ datafs download my_archive my_archive_versioned.txt --version 0.0.2
downloaded v0.0.2 to my_archive_versioned.txt
```

Managing Data Dependencies

Dependency graphs can be tracked explicitly in datafs, and each version can have its own dependencies.

You specify dependencies from the command line interface or from within *python*.

Note: Dependencies are not currently validated in any way, so entering a dependency that is not a valid archive name or version will not raise an error.

Specifying Dependencies

On write

Dependencies can be set when using the `--dependency` option to the `update` command. To specify several dependencies, use multiple `--dependency archive[==version]` arguments.

Each `--dependency` value should have the syntax `archive_name==version`. Supplying only the archive name will result in a value of `None`. A value of `None` is a valid dependency specification, where the version is treated as unpinned and is always interpreted as the dependency's latest version.

For example:

```
$ datafs create my_archive

$ echo "contents depend on archive 2 v1.1" >> arch.txt

$ datafs update my_archive arch.txt  --dependency "archive2==1.1" --dependency
↪"archive3"

$ datafs get_dependencies my_archive
{'archive2': '1.1', 'archive3': None}
```

After write

Dependencies can also be added to the latest version of an archive using the `set_dependencies` command:

```
$ datafs set_dependencies my_archive --dependency archive2==1.2

$ datafs get_dependencies my_archive
{'archive2': '1.2'}
```

Using a requirements file

If a requirements file is present at api creation, all archives written with that api object will have the specified dependencies by default.

For example, with the following requirements file as `requirements_data.txt`:

```
1 dep1==1.0
2 dep2==0.4.1a3
```

Archives written while in this working directory will have these requirements:

```
$ echo "depends on dep1 and dep2" > arch.txt

$ datafs update my_archive arch.txt --requirements_file 'requirements_data.txt'

$ datafs get_dependencies my_archive
{'dep1': '1.0', 'dep2': '0.4.1a3'}
```

Using Dependencies

Retrieve dependencies with the `dependencies` command:

```
$ datafs get_dependencies my_archive
{'dep1': '1.0', 'dep2': '0.4.1a3'}
```

Get dependencies for older versions using the `--version` argument:

```
$ datafs get_dependencies my_archive --version 0.0.1
{'archive2': '1.1', 'archive3': None}
```

Finding Archives

In this section we'll take a look at finding archives via the command line.

You can find archives from the command line interface or from *python*. This documentation mirrors the python documentation.

Using `listdir`

In our database we have many archives. We know that `impactlab` is a top-level directory-like namespace in our database. Let's have a look.

```
$ datafs listdir impactlab
labor
climate
conflict
mortality
```

Ok. We see that `labor`, `climate`, `mortality` and `conflict` are all directory-like namespaces groupings below `impactlab`. Lets have a look at `conflict`.

```
$ datafs listdir impactlab/conflict
global
```

Let's see what is in `impactlab/conflict/global`.

```
$ datafs listdir impactlab/conflict/global
conflict_global_daily.csv
$ datafs listdir impactlab/conflict/global/conflict_global_daily.csv
0.0.1
```

We can see that there is currently only version `0.0.1` of `conflict_global_daily.csv`

Using `filter`

DataFS lets you filter so you can limit the search space on archive names. At the command line, you can use the `prefix`, `path`, `str`, and `regex` pattern options to filter archives. Let's look at using the `prefix` `project1_variable1_` which corresponds to the `prefix` option, the beginning string of a set of archive names.

```
$ datafs filter --prefix project1_variable1_ # doctest: +SKIP
project1_variable1_scenario5.nc
project1_variable1_scenario1.nc
project1_variable1_scenario4.nc
project1_variable1_scenario2.nc
project1_variable1_scenario3.nc
```

We can also filter on `path`. In this case we want to filter all NetCDF files that match a specific pattern. We need to set our `engine` value to `path` and put in our search pattern.

```
$ datafs filter --pattern *_variable4_scenario4.nc --engine path # doctest: +SKIP
project1_variable4_scenario4.nc
project2_variable4_scenario4.nc
project3_variable4_scenario4.nc
project5_variable4_scenario4.nc
project4_variable4_scenario4.nc
```

We can also filter archives with archive names containing a specific string by setting `engine` to `str`. In this example we want all archives with the string `variable2`.

```
$ datafs filter --pattern variable2 --engine str # doctest: +ELLIPSIS +SKIP
project1_variable2_scenario1.nc
project1_variable2_scenario2.nc
project1_variable2_scenario3.nc
...
project5_variable2_scenario3.nc
project5_variable2_scenario4.nc
project5_variable2_scenario5.nc
```

Using `search`

DataFS `search` capabilites are enabled via tagging of archives. The arguments of the `search` command are tags associated with a given archive. If archives are not tagged, they cannot be searched. Please see *this* for a reference on how to tag archives.

Our archives have been tagged with `team1`, `team2`, or `team3` Let's search for some archives with tag `team3`.

```
$ datafs search team3 # doctest: +ELLIPSIS +SKIP
project2_variable2_scenario2.nc
project5_variable4_scenario1.nc
project1_variable5_scenario4.nc
project3_variable2_scenario1.nc
project2_variable1_scenario1.nc
...
project5_variable1_scenario2.nc
project2_variable5_scenario5.nc
project5_variable2_scenario5.nc
project3_variable2_scenario5.nc
```

Let's use `get_tags` to have a look at one of our archives' tags.

```
$ datafs get_tags project2_variable2_scenario2.nc
team3
```

We can see that indeed it has been tagged with `team3`.

For completeness, let's have a look at archives with tag of `team1`.

```
$ datafs search team1 # doctest: +ELLIPSIS +SKIP
project1_variable1_scenario4.nc
project1_variable2_scenario2.nc
project1_variable2_scenario5.nc
project1_variable3_scenario3.nc
project1_variable4_scenario1.nc
```

```
project1_variable4_scenario4.nc
...
project5_variable3_scenario2.nc
project5_variable3_scenario5.nc
project5_variable4_scenario3.nc
project5_variable5_scenario1.nc
project5_variable5_scenario4.nc
```

And now let's have a look at one of them to see what tags are associated with it.

```
$ datafs get_tags project2_variable5_scenario1.nc
team1
```

We can see clearly that our archive has been tagged with `team1`.

We want your feedback. If you find bugs or have suggestions to improve this documentation, please consider contributing.

CHAPTER 7

Administrative Tools

CHAPTER 8

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/ClimateImpactLab/datafs/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

DataFS could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

To test the documentation you write, run the command:

```
$ sphinx-build -W -b html -d docs/_build/doctrees docs/. docs/_build/html
```

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ClimateImpactLab/datafs/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Great! There are a couple steps to follow when contributing code.

Setting up your development environment

Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv datafs
$ cd datafs/
$ python setup.py develop
```

Developing your feature

When making any changes to the DataFS codebase, follow the following steps:

1. Check for issues on our issues page. If no issue exists for the feature you would like to add, add one! Make sure the scope of the issue is limited and precise, so anyone can understand the behaviour/feature you would like to see.
2. Fork the *datafs* repo on GitHub.
3. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/datafs.git
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. Write tests for your feature first. Think through all the use cases and make sure your tests cover all the ways your code might be used. Include the issue number you are addressing in the docstring of your tests:

```
def test_my_new_feature():
    ''' Test my_new_feature. Addresses :issue:`1234` '''

    # test code
```

6. Implement your feature, writing as little code as is required to satisfy the tests you just wrote. Run tests frequently to make sure you are maintaining compatibility with the rest of the package:

```
$ python setup.py test
$ flake8 datafs tests examples docs
```

   You can run only the tests you wrote using pytest'sexpression matching syntax, e.g.:

```
$ pytest -k test_my_new_feature
```

7. When you are passing all of your tests, run the full test suite.
8. Make changes to the docs describing your new feature if necessary.
9. Add an entry to the latest whatsnew document describing your changes. Make sure to reference the issue number in your entry.
6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Happy hunting!

CHAPTER 9

What's New

These are new features and improvements of note in each release.

v0.7.1 (Latest Build)

New features

- Archive names are normalized in DataAPI methods. See *normalizing archive names* (GH #220 & GH #235).

Normalizing archive names

DataAPI methods *create()*, *get_archive()*, *batch_get_archive()* and *listdir()*, and the *default_versions()* property, are normalized using `DataAPI._normalize_archive_name()`. This allows users to create and get archives using leading slashes and authority names interchangably. For example, the following are all equivalent:

```
>>> api.create('my/sample/archive.txt')
>>> api.create('/my/sample/archive.txt')
>>> api.create('authority://my/sample/archive.txt')
```

Furthermore, they can all be found using similarly flexible searches. The following will all return the archive_name or archive created in the above examples:

```
>>> api.get_archive('my/sample/archive.txt')
>>> api.get_archive('/my/sample/archive.txt')
>>> api.get_archive('authority://my/sample/archive.txt')
>>> api.batch_get_archive(['authority://my/sample/archive.txt'])
>>> api.search(prefix='my/samp')
>>> api.search(prefix='/my/samp')
>>> api.search(pattern='my/samp*')
>>> api.search(pattern='*my/samp*')
>>> api.search(pattern='/my/samp*')
```

Search patterns do not accept authority names:

```
>>> api.search(prefix='authority://my') # no results
```

Archive name checking

The normalization process catches illegal archive names:

```
>>> api.create('!!!\\\\~')
Traceback (most recent call last):
...
fs.errors.InvalidCharsInPathError: Path contains invalid characters: !!!\\~
```

This error checking is done by `fs`, using the implementations of `validatepath()` on the relevant authority. Currently, both `fs.osfs.OSFS.validatepath()` and the method on whatever filesystem is used by the authority are both checked. This dual restriction is used because checking against OSFS restrictions is useful to prevent errors when using a cache.

Backwards incompatible API changes

- Authority names are now limited to names that match `r'[\w\-]+'`. This regex value is set by the module parameter `_VALID_AUTHORITY_PATTERNS` in `datafs/core/data_api.py` (GH #186).
- Introduces a new property *`datafs.DataAPI.default_versions()`*, which does archive name coersion/alignment. `datafs.DataAPI._default_versions()` should no longer be accessed under any circumstances (GH #220 and GH #235).

Performance Improvements

Bug Fixes

- Messages are now coerced to strings, and `log()` and the CLI `log` command no longer fail when used on archives with non-string messages (GH #232)
- *delete* and *api.delete_archive* now implement *removedir* which removes the archive level name space from the filesystem. Previous implementations of these methods removed the archive from the manager and only removed versions from the file system leaving top level name spaces as file system objects. (GH #212)

Under the hood

- Use `:issue:` and `:pull:` directives to reference a github issue or pull request (GH #209)
- The sphinx build is now tested on travis. Run the tests locally with the command `sphinx-build -W -b html -d docs/_build/doctrees docs/. docs/_build/html` (GH #211)
- The docs structure has been reorganized
- DynamoDB support added to command line tests

Documentation additions

- Additional documentation on *tagging files* and *searching and finding files*

See the issue tracker on GitHub for a complete list.

v0.7.0 (March 9, 2017)

New features

- `listdir()` method allowing listing archive path components given a prefix
- new batch get archive method `batch_get_archive()`

Using the listdir search method

List archive path components given the path of the "directory" to search

---

Note: When using listdir on versioned archives, listdir will provide the version numbers when a full archive path is supplied as the location argument. This is because DataFS stores the archive path as a directory and the versions as the actual files when versioning is on.

Usage

```
>>> api.listdir('s3://ACP/climate/')
['gcm-modelweights.csv', 'hddcdd', 'smme']
```

```
$ datafs listdir s3://ACP/climate/
gcm-modelweights.csv
hddcdd
smme
```

Bulk archive retrieval with *batch_get_archive*

Batch version of `get_archive()`. Accepts an iterable of archive names to retrieve and optional default versions.

Returns a dict of `DataArchive` objects, indexed by archive name. If an archive is not found, it is omitted (*batch_get_archive* does not raise a `KeyError` on invalid archive names).

Example

```
>>> api.batch_get_archive(api.search())
{arch1: <DataArchive s3://arch1>, arch2: <DataArchive s3://arch2>, ...}
```

`batch_get_archive` has no equivalent on the Command Line Interface.

See the issue tracker on GitHub for a complete list.

v0.6.9 (February 21, 2017)

New features

- archive pattern constraints (GH #168)
- set tags from command line
- add tagging and searching documentation

Archive pattern constraints

List of regex patterns that must match archive_name before archive creation is allowed

Create an archive pattern using manager.set_required_archive_patterns. e.g. require only w, ., or / characters:

```
>>> api = get_api()
>>> api.manager.set_required_archive_patterns([r'^[\w\/\.]+$'])
```

Now, archives that do not match this will not be supported:

Tagging from CLI

Added three new commands which reflect their DataArchive counterparts:

```
datafs add_tags
datafs get_tags
datafs delete_tags
```

Additionally, a `--tag` option was added to datafs create so that tags could be supplied on archive creation:

```
datafs create my_archive --description "my description" --tag tag1 \
    --tag tag2 --source "where it's from" --tag tag3
```

Backwards incompatible API changes

- stop coercing underscores to slashes in archive_path
- drop archive_path argument from DataAPI.create

See the issue tracker on GitHub for a complete list.

v0.6.8 (February 7, 2017)

This is a patch release primarily improving the documentation and testing of DataFS. There are no backward incompatible changes in v0.6.8

New features

- Add command line docstrings (GH #115)
- Add tests for Python API documentation snippets (GH #108)
- Integrate clatter - checks to make sure CLI documentation is accurate

Bug Fixes

- More robust `*args`, `**kwargs` handling in CLI, with better error messages
- Fix click round-trip compatibility issue - print results on CLI using `\n` instead of `\r\n` on windows
- Raises error when loading a non-existant profile from config (GH #135)

See the issue tracker on GitHub for a complete list.

v0.6.7 (February 1, 2017)

New features

- Allow tag specification on create

Performance Improvements

- Restructure conftest.py: api_with_diverse_archives to be session-scoped

Under the hood

- Consolidate manager._create_archive_table and _create_spec_table into one function
- Move archive document creation to separate method in manager (allows batch write in tests)
- Add tests for search and filter queries on very large manager tables

See the issue tracker on GitHub for a complete list.

v0.6.6 (January 20, 2017)

New features

- Introduces search features in command line:

```
datafs search
```

  and the API:

```
api.search(query)
```

See the issue tracker on GitHub for a complete list.

v0.6.5 (January 13, 2017)

New features

- regex and unix-style filter searches

Backwards incompatible API changes

- Prevent update/write of empty files

See the issue tracker on GitHub for a complete list.

v0.6.4 (January 12, 2017)

Under the hood

- Test/fix handling of multiple read/write of large netCDF datasets

See the issue tracker on GitHub for a complete list.

v0.6.3 (January 11, 2017)

New features

- dependency handling

Backwards incompatible API changes

- raise error when passing non-None versions to unversioned archive methods
- change API method name: `create_archive` –> *`create()`*
- change CLI subcommand name: `upload` –> `update`

Under the hood

- improve test coverage

Bug Fixes

- prevent users from deleting required metadata elements

See the issue tracker on GitHub for a complete list.

v0.6.2 (January 9, 2017)

New features

- New template in docs for AWS configuration (GH #73)

Backwards incompatible API changes

- Drop DataArchive properties that access manager (GH #72)
- Manager archive listing attribute `versions` changed to `version_history`

Manager-calling properties converted to methods

- `latest_version` -> *`get_latest_version()`*
- `versions` -> *`get_versions()`*
- `latest_hash` -> *`get_latest_hash()`*
- `history` -> *`get_history()`*
- `metadata` -> *`get_metadata()`*

See the issue tracker on GitHub for a complete list.

v0.6.1 (January 6, 2017)

See the issue tracker on GitHub for a complete list.

v0.6.0 (January 4, 2017)

New features

- Explicit versioning & version pinning (GH #62)
- Explicit dependency tracking (GH #63)
- Update metadata from the command line
- Support for version tracking & management with requirements files (GH #70)
- Configure archive specification on manager table

Set dependencies from Python API on write

DataArchive.update

```
def update(
    self,
    filepath,
    cache=False,
    remove=False,
    bumpversion='patch',
    prerelease=None,
    dependencies=None,
    **kwargs):

    ...

    self._update_manager(
        checksum,
        kwargs,
        version=next_version,
        dependencies=dependencies)
```

DataArchive.open

```
def open(
    self,
    mode='r',
    version=None,
    bumpversion='patch',
    prerelease=None,
    dependencies=None,
    *args,
    **kwargs):
    ...

    updater = lambda *args, **kwargs: self._update_manager(
            *args,
            version=next_version,
            dependencies=dependencies,
            **kwargs)
    ...
```

DataArchive.get_local_path

similar to *DataArchive.open*

DataArchive._update_manager

```
def _update_manager(
        self,
        checksum,
        metadata={},
        version=None,
        dependencies=None):

    # by default, dependencies is the last version of dependencies
```

```
    if dependencies is None:
        history = self.history
        if len(history) == 0:
            dependencies = []
        else:
            dependencies = history[-1]['dependencies']

    ....
```

Under the hood

- Table schemas have been moved from the dynamo and mongo modules to the BaseDataManager.
- versions attr is now version_history in table schema and DataArchive method `get_versions` is now `get_version_history()`

See the issue tracker on GitHub for a complete list.

v0.5.0 (December 21, 2016)

New features

- command line `download` feature

See the issue tracker on GitHub for a complete list.

v0.4.0 (December 15, 2016)

New features

- create API object from config file

See the issue tracker on GitHub for a complete list.

v0.3.0 (December 14, 2016)

New features

- cached read/write

See the issue tracker on GitHub for a complete list.

v0.2.0 (December 12, 2016)

See the issue tracker on GitHub for a complete list.

v0.1.0 (November 18, 2017)

See the issue tracker on GitHub for a complete list.

CHAPTER 10

DataFS API

Subpackages

datafs.core package

Submodules

datafs.core.data_api module

class datafs.core.data_api.**DataAPI**(*default_versions=None*, ***kwargs*)

Bases: object

DefaultAuthorityName = None

attach_authority(*service_name*, *service*)

attach_cache(*service*)

attach_manager(*manager*)

batch_get_archive(*archive_names*, *default_versions=None*)

Batch version of *get_archive()*

Parameters

- **archive_names** (*list*) – Iterable of archive names to retrieve
- **default_versions** (*str, object, or dict*) – Default versions to assign to each returned archive. If default_versions is a dict, each archive_name must be a key in default_versions and the value must be a valid version. Versions must be a strict version number, a StrictVersion, or a BumpableVersion object.

Returns **archives** – List of *DataArchive* objects. If an archive is not found, it is omitted (*batch_get_archive* does not raise a KeyError on invalid archive names).

Return type list

cache

close()

create(*archive_name*, *authority_name=None*, *versioned=True*, *raise_on_err=True*, *metadata=None*, *tags=None*, *helper=False*)
Create a DataFS archive

Parameters

- **archive_name** (*str*) – Name of the archive
- **authority_name** (*str*) – Name of the data service to use as the archive's data authority
- **versioned** (*bool*) – If true, store all versions with explicit version numbers (defualt)
- **raise_on_err** (*bool*) – Raise an error if the archive already exists (default True)
- **metadata** (*dict*) – Dictionary of additional archive metadata
- **helper** (*bool*) – If true, interactively prompt for required metadata (default False)

default_authority

default_authority_name

default_versions

delete_archive(*archive_name*)
Delete an archive

Parameters **archive_name** (*str*) – Name of the archive to delete

filter(*pattern=None*, *engine='path'*, *prefix=None*)
Performs a filtered search on entire universe of archives according to pattern or prefix.

Parameters

- **prefix** (*str*) – string matching beginning characters of the archive or set of archives you are filtering. Note that authority prefixes, e.g. `local://my/archive.txt` are not supported in prefix searches.
- **pattern** (*str*) – string matching the characters within the archive or set of archives you are filtering on. Note that authority prefixes, e.g. `local://my/archive.txt` are not supported in pattern searches.
- **engine** (*str*) – string of value 'str', 'path', or 'regex'. That indicates the type of pattern you are filtering on

Returns

Return type generator

get_archive(*archive_name*, *default_version=None*)
Retrieve a data archive

Parameters

- **archive_name** (*str*) – Name of the archive to retrieve
- **default_version** (*version*) – str or `StrictVersion` giving the default version number to be used on read operations

Returns **archive** – New *DataArchive* object

Return type object

Raises KeyError: – A KeyError is raised when the `archive_name` is not found

static hash_file(*f*)
Utility function for hashing file contents

Overload this function to change the file equality checking algorithm

Parameters **f** (*file-like*) – File-like object or file path from which to compute checksum value

Returns **checksum** – dictionary with {'algorithm': 'md5', 'checksum': hexdigest}

Return type dict

listdir(*location*, *authority_name=None*)
List archive path components at a given location

Note: When using listdir on versioned archives, listdir will provide the version numbers when a full archive path is supplied as the location argument. This is because DataFS stores the archive path as a directory and the versions as the actual files when versioning is on.

Parameters

- **location** (*str*) – Path of the "directory" to search

  location can be a path relative to the authority root (e.g */MyFiles/Data*) or can include authority as a protocol (e.g. *my_auth://MyFiles/Data*). If the authority is specified as a protocol, the *authority_name* argument is ignored.

- **authority_name** (*str*) – Name of the authority to search (optional)

  If no authority is specified, the default authority is used (if only one authority is attached or if *DefaultAuthorityName* is assigned).

Returns Archive path components that exist at the given "directory" location on the specified authority

Return type list

Raises `ValueError` – A ValueError is raised if the authority is ambiguous or invalid

lock_authorities()

lock_manager()

manager

remove(*archive_name*, *authority_name=None*)
Removes file system objects from a file system

Parameters **archive_name** (*str*) –

remove_dir(*path*, *authority_name=None*, *recursive=False*, *force=False*, *cache=False*)
removes the directory level file system objects a ssociated with an archive_name.

Deleting archive_name name space will erase all data associated with file system object For help setting user permissions, see *Administrative Tools*

Parameters

- **path** (*str*) – name-space to be removed from file system
- **authority_name** (*str*) – file system authority for the archive

- **recursive** (*bool*) – If True, will remove empty file system objects subordinate to the the name space provided
- **force** (*bool*) – If True, will remove file system objects wether or not they are empty
- **cache** (*bool*) – If True, will remove name space from cache file system

Returns

Return type None

search(**query*, ***kwargs*)
Searches based on tags specified by users

Parameters

- **query** (*str*) – tags to search on. If multiple terms, provided in comma delimited string format
- **prefix** (*str*) – start of archive name. Providing a start string improves search speed.

exception datafs.core.data_api.**PermissionError**
Bases: exceptions.NameError

datafs.core.data_archive module

class datafs.core.data_archive.**DataArchive**(*api*, *archive_name*, *authority_name*, *archive_path*, *versioned=True*, *default_version=None*)
Bases: object

add_tags(**tags*)
Set tags for a given archive

archive_path

authority

authority_name

cache(*version=None*)

delete()
Deletes the archive from the manager and the filesystem.

> **Warning:** Deleting an archive will erase all data and metadata permanently. For help setting user permissions, see *Administrative Tools*

delete_tags(**tags*)
Deletes tags for a given archive

desc(*version=None*, **args*, ***kwargs*)
Return a short descriptive text regarding a path

download(*filepath*, *version=None*)
Downloads a file from authority to local path

1. First checks in cache to check if file is there and if it is, is it up to date
2. If it is not up to date, it will download the file to cache

exists(*version=None*, **args*, ***kwargs*)
Check whether a path exists as file or directory

get_default_version()

get_dependencies(*version=None*)

Parameters **version** (*str*) – string representing version number whose dependencies you are looking up

get_history()

get_latest_hash()

get_latest_version()

get_local_path(**args*, ***kwds*)
Returns a local path for read/write

Parameters

- **version** (*str*) – Version number of the file to retrieve (default latest)
- **bumpversion** (*str*) – Version component to update on write if archive is versioned. Valid bumpversion values are 'major', 'minor', and 'patch', representing the three components of the strict version numbering system (e.g. "1.2.3"). If bumpversion is None the version number is not updated on write. Either bumpversion or prerelease (or both) must be a non-None value. If the archive is not versioned, bumpversion is ignored.
- **prerelease** (*str*) – Prerelease component of archive version to update on write if archive is versioned. Valid prerelease values are 'alpha' and 'beta'. Either bumpversion or prerelease (or both) must be a non-None value. If the archive is not versioned, prerelease is ignored.
- **metadata** (*dict*) – Updates to archive metadata. Pass {key: None} to remove a key from the archive's metadata.

get_metadata()

get_tags()
Returns a list of tags for the archive

get_version_hash(*version=None*)

get_version_path(*version=None*)
Returns a storage path for the archive and version

If the archive is versioned, the version number is used as the file path and the archive path is the directory. If not, the archive path is used as the file path.

Parameters **version** (*str or object*) – Version number to use as file name on versioned archives (default latest unless `default_version` set)

Examples

```
>>> arch = DataArchive(None, 'arch', None, 'a1', versioned=False)
>>> print(arch.get_version_path())
a1
>>>
>>> ver = DataArchive(None, 'ver', None, 'a2', versioned=True)
>>> print(ver.get_version_path('0.0.0'))
a2/0.0
```

```
>>>
>>> print(ver.get_version_path('0.0.1a1'))
a2/0.0.1a1
>>>
>>> print(ver.get_version_path('latest'))
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'manager'
```

get_versions()

getinfo(*version=None*, **args*, ***kwargs*)
Return information about the path e.g. size, mtime

getmeta(*version=None*, **args*, ***kwargs*)
Get the value of a filesystem meta value, if it exists

hasmeta(*version=None*, **args*, ***kwargs*)
Check if a filesystem meta value exists

is_cached(*version=None*)
Set the cache property to start/stop file caching for this archive

isfile(*version=None*, **args*, ***kwargs*)
Check whether the path exists and is a file

log()

open(**args*, ***kwds*)
Opens a file for read/write

Parameters

- **mode** (*str*) – Specifies the mode in which the file is opened (default 'r')
- **version** (*str*) – Version number of the file to open (default latest)
- **bumpversion** (*str*) – Version component to update on write if archive is versioned. Valid bumpversion values are 'major', 'minor', and 'patch', representing the three components of the strict version numbering system (e.g. "1.2.3"). If bumpversion is None the version number is not updated on write. Either bumpversion or prerelease (or both) must be a non-None value. If the archive is not versioned, bumpversion is ignored.
- **prerelease** (*str*) – Prerelease component of archive version to update on write if archive is versioned. Valid prerelease values are 'alpha' and 'beta'. Either bumpversion or prerelease (or both) must be a non-None value. If the archive is not versioned, prerelease is ignored.
- **metadata** (*dict*) – Updates to archive metadata. Pass {key: None} to remove a key from the archive's metadata.

args, kwargs sent to file system opener

remove_from_cache(*version=None*)

set_dependencies(*dependencies=None*)

update(*filepath*, *cache=False*, *remove=False*, *bumpversion=None*, *prerelease=None*, *dependencies=None*, *metadata=None*, *message=None*)
Enter a new version to a DataArchive

Parameters

- **filepath** (*str*) – The path to the file on your local file system

- **cache** (*bool*) – Turn on caching for this archive if not already on before update
- **remove** (*bool*) – removes a file from your local directory
- **bumpversion** (*str*) – Version component to update on write if archive is versioned. Valid bumpversion values are 'major', 'minor', and 'patch', representing the three components of the strict version numbering system (e.g. "1.2.3"). If bumpversion is None the version number is not updated on write. Either bumpversion or prerelease (or both) must be a non-None value. If the archive is not versioned, bumpversion is ignored.
- **prerelease** (*str*) – Prerelease component of archive version to update on write if archive is versioned. Valid prerelease values are 'alpha' and 'beta'. Either bumpversion or prerelease (or both) must be a non-None value. If the archive is not versioned, prerelease is ignored.
- **metadata** (*dict*) – Updates to archive metadata. Pass {key: None} to remove a key from the archive's metadata.

update_metadata(*metadata*)

versioned

datafs.core.data_file module

datafs.core.data_file.**get_local_path**(**args*, ***kwds*)
Context manager for retrieving a system path for I/O and updating on change

Parameters

- **authority** (*object*) – `pyFilesystem` filesystem object to use as the authoritative, up-to-date source for the archive
- **cache** (*object*) – `pyFilesystem` filesystem object to use as the cache. Default `None`.
- **use_cache** (*bool*) – update, service_path, version_check, **kwargs

datafs.core.data_file.**open_file**(**args*, ***kwds*)
Context manager for reading/writing an archive and uploading on changes

Parameters

- **authority** (*object*) – `pyFilesystem` filesystem object to use as the authoritative, up-to-date source for the archive
- **cache** (*object*) – `pyFilesystem` filesystem object to use as the cache. Default `None`.
- **use_cache** (*bool*) – update, service_path, version_check, **kwargs

Module contents

datafs.config package

Submodules

datafs.config.config_file module

class datafs.config.config_file.**ConfigFile**(*config_file=None*, *default_profile='default'*)
Bases: `object`

edit_config_file()

get_config_from_api(*api*, *profile=None*)

get_profile_config(*profile=None*)

parse_configfile_contents(*config*)

read_config()

write_config(*fp=None*)

write_config_from_api(*api*, *config_file=None*, *profile=None*)

Create/update the config file from a DataAPI object

Parameters

- **api** (*object*) – The *datafs.DataAPI* object from which to create the config profile
- **profile** (*str*) – Name of the profile to use in the config file (default "default-profile")
- **config_file** (*str or file*) – Path or file in which to write config (default is your OS's default datafs application directory)

Examples

Create a simple API and then write the config to a buffer:

```
>>> from datafs import DataAPI
>>> from datafs.managers.manager_mongo import MongoDBManager
>>> from fs.osfs import OSFS
>>> from fs.tempfs import TempFS
>>> import os
>>> import tempfile
>>> import shutil
>>>
>>> api = DataAPI(
...      username='My Name',
...      contact = 'me@demo.com')
>>>
>>> manager = MongoDBManager(
...     database_name = 'MyDatabase',
...     table_name = 'DataFiles')
>>>
>>> manager.create_archive_table(
...     'DataFiles',
...     raise_on_err=False)
>>>
>>> api.attach_manager(manager)
>>>
>>> tmpdir = tempfile.mkdtemp()
>>> local = OSFS(tmpdir)
>>>
>>> api.attach_authority('local', local)
>>>
>>> # Create a StringIO object for the config file
...
>>> try:
...   from StringIO import StringIO
... except ImportError:
...   from io import StringIO
...
```

```
>>> conf = StringIO()
>>>
>>> config_file = ConfigFile(default_profile='my-api')
>>> config_file.write_config_from_api(
...     api,
...     profile='my-api',
...     config_file=conf)
>>>
>>> print(conf.getvalue())
default-profile: my-api
profiles:
  my-api:
    api:
      user_config: {contact: me@demo.com, username: My Name}
    authorities:
      local:
        args: [...]
        service: OSFS
        kwargs: {}
    manager:
      args: []
      class: MongoDBManager
      kwargs:
        client_kwargs: {}
        database_name: MyDatabase
        table_name: DataFiles

>>> conf.close()
>>> local.close()
>>> shutil.rmtree(tmpdir)
```

At this point, we can retrieve the api object from the configuration file:

```
>>> try:
...   from StringIO import StringIO
... except ImportError:
...   from io import StringIO
...
>>> conf = StringIO("""
... default-profile: my-api
... profiles:
...   my-api:
...     api:
...       user_config: {contact: me@demo.com, username: My Name}
...     authorities:
...       local:
...         args: []
...         service: TempFS
...         kwargs: {}
...     manager:
...       args: []
...       class: MongoDBManager
...       kwargs:
...         client_kwargs: {}
...         database_name: MyDatabase
...         table_name: DataFiles
... """)
>>>
```

```
>>> import datafs
>>> from fs.tempfs import TempFS
>>> api = datafs.get_api(profile='my-api', config_file=conf)
>>>
>>> cache = TempFS()
>>> api.attach_cache(cache)
>>>
>>> conf2 = StringIO()
>>>
>>> config_file = ConfigFile(default_profile='my-api')
>>> config_file.write_config_from_api(
...     api,
...     profile='my-api',
...     config_file=conf2)
>>>
>>> print(conf2.getvalue())
default-profile: my-api
profiles:
  my-api:
    api:
      user_config: {contact: me@demo.com, username: My Name}
    authorities:
      local:
        args: []
        service: TempFS
        kwargs: {}
    cache:
        args: []
        service: TempFS
        kwargs: {}
    manager:
      args: []
      class: MongoDBManager
      kwargs:
        client_kwargs: {}
        database_name: MyDatabase
        table_name: DataFiles
```

exception datafs.config.config_file.**ProfileNotFoundError**
Bases: exceptions.KeyError

datafs.config.constructor module

class datafs.config.constructor.**APIConstructor**
Bases: object

classmethod attach_cache_from_config(*api*, *config*)

classmethod attach_manager_from_config(*api*, *config*)

classmethod attach_services_from_config(*api*, *config*)

static generate_api_from_config(*config*)

datafs.config.helpers module

datafs.config.helpers.**check_requirements**(*to_populate*, *prompts*, *helper=False*)

datafs.config.helpers.**get_api**(*profile=None*, *config_file=None*, *requirements=None*)

Generate a datafs.DataAPI object from a config profile

`get_api` generates a DataAPI object based on a pre-configured datafs profile specified in your datafs config file.

To create a datafs config file, use the command line tool `datafs configure --helper` or export an existing DataAPI object with `datafs.ConfigFile.write_config_from_api()`

Parameters

- **profile** (*str*) – (optional) name of a profile in your datafs config file. If profile is not provided, the default profile specified in the file will be used.
- **config_file** (*str or file*) – (optional) path to your datafs configuration file. By default, get_api uses your OS's default datafs application directory.

Examples

The following specifies a simple API with a MongoDB manager and a temporary storage service:

```
>>> try:
...   from StringIO import StringIO
... except ImportError:
...   from io import StringIO
...
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
>>>
>>> config_file = StringIO("""
... default-profile: my-data
... profiles:
...     my-data:
...         manager:
...             class: MongoDBManager
...             kwargs:
...                 database_name: 'MyDatabase'
...                 table_name: 'DataFiles'
...
...         authorities:
...             local:
...                 service: OSFS
...                 args: ['{}']
... """.format(tempdir))
>>>
>>> # This file can be read in using the datafs.get_api helper function
...
>>>
>>> api = get_api(profile='my-data', config_file=config_file)
>>> api.manager.create_archive_table(
...     'DataFiles',
...     raise_on_err=False)
>>>
>>> archive = api.create(
...     'my_first_archive',
...     metadata = dict(description = 'My test data archive'),
...     raise_on_err=False)
>>>
>>> with archive.open('w+') as f:
```

```
...     res = f.write(u'hello!')
...
>>> with archive.open('r') as f:
...     print(f.read())
...
hello!
>>>
>>> # clean up
...
>>> archive.delete()
>>> import shutil
>>> shutil.rmtree(tempdir)
```

datafs.config.helpers.**to_config_file**(*api*, *config_file=None*, *profile='default'*)

Module contents

datafs.managers package

Submodules

datafs.managers.manager module

class datafs.managers.manager.**BaseDataManager**(*table_name*)

Bases: object

Base class for DataManager metadata store objects

Should be subclassed. Not intended to be used directly.

TimestampFormat = '%Y%m%d-%H%M%S'

add_tags(*archive_name*, *tags*)

Add tags to an archive

Parameters

- **archive_name** (*str*) – Name of archive
- **tags** (*list or tuple of strings*) – tags to add to the archive

batch_get_archive(*archive_names*)

Batched version of `_get_archive_listing()`

Returns a list of full archive listings from an iterable of archive names

Note: Invalid archive names will simply not be returned, so the response may not be the same length as the supplied *archive_names*.

Parameters **archive_names** (*list*) – List of archive names

Returns **archive_listings** – List of archive listings

Return type list

create_archive(*archive_name*, *authority_name*, *archive_path*, *versioned*, *raise_on_err=True*, *metadata=None*, *user_config=None*, *tags=None*, *helper=False*)
Create a new data archive

Returns **archive** – new *DataArchive* object

Return type object

create_archive_table(*table_name*, *raise_on_err=True*)

Parameters

- **table_name** (*str*) –
- **a table to store archives for your project** (*Creates*) –
- **creates and populates a table with basic spec for user and** (*Also*) –
- **config** (*metadata*) –

Returns

Return type None

classmethod create_timestamp()
Utility function for formatting timestamps

Overload this function to change timestamp formats

delete_archive_record(*archive_name*)
Deletes an archive from the database

Parameters **archive_name** (*str*) – name of the archive to delete

delete_table(*table_name=None*, *raise_on_err=True*)

delete_tags(*archive_name*, *tags*)
Delete tags from an archive

Parameters

- **archive_name** (*s tr*) – Name of archive
- **tags** (*list or tuple of strings*) – tags to delete from the archive

get_archive(*archive_name*)
Get a data archive given an archive name

Returns **archive_specification** – archive_name: name of the archive to be retrieved authority: name of the archive's authority archive_path: service path of archive

Return type dict

get_latest_hash(*archive_name*)
Retrieve the file hash for a given archive

Parameters **archive_name** (*str*) – name of the archive for which to retrieve the hash

Returns **hashval** – hash value for the latest version of archive_name

Return type str

get_metadata(*archive_name*)
Retrieve the metadata for a given archive

Parameters **archive_name** (*str*) – name of the archive to be retrieved

Returns **metadata** – current archive metadata

Return type dict

get_tags(*archive_name*)
Returns the list of tags associated with an archive

get_version_history(*archive_name*)

required_archive_metadata

required_archive_patterns

required_user_config

search(*search_terms*, *begins_with=None*)

Parameters **search_terms** (*str*) – strings of terms to search for

If called as *api.manager.search()*, *search_terms* should be a list or a tuple of strings

set_required_archive_metadata(*metadata_config*)
Sets required archive metatdata for all users

Parameters **metadata_config** (*dict*) – Dictionary of required archive metada and metadata field descriptions. All archives created on this manager table will be required to have these keys in their archive's metadata.

If the archive metadata does not contain these keys, an error will be raised with the description in the value associated with the key.

set_required_archive_patterns(*required_archive_patterns*)
Sets archive_name regex patterns for the enforcement of naming conventions on archive creation

Parameters **required_archive_patterns** (*strings of args*) –

set_required_user_config(*user_config*)
Sets required user metadata for all users

Parameters **user_config** (*dict*) – Dictionary of required user metadata and metadata field descriptions. All archive creation and update actions will be required to have these keys in the user_config metadata.

If the archive or version metadata does not contain these keys, an error will be raised with the descrtiption in the value associated with the key.

table_names

update(*archive_name*, *version_metadata*)
Register a new version for archive `archive_name`

Note: need to implement hash checking to prevent duplicate writes

---

update_metadata(*archive_name*, *archive_metadata*)
Update metadata for archive `archive_name`

update_spec_config(*document_name*, *spec*)
Set the contents of a specification document by name

This method should not be used directly. Instead, use *set_required_user_config()* or *set_required_archive_metadata()*.

Parameters: document_name : str

Name of a specification document's key

spec : dict

Dictionary metadata specification

datafs.managers.manager_dynamo module

class datafs.managers.manager_dynamo.**DynamoDBManager**(*table_name*, *session_args=None*, *resource_args=None*)

Bases: *datafs.managers.manager.BaseDataManager*

Parameters

- **table_name** (*str*) – Name of the data archive table
- **session_args** (*dict*) – Keyword arguments used in initializing a `boto3.Session` object
- **resource_args** (*dict*) – Keyword arguments used in initializing a dynamodb `ServiceResource` object

config

datafs.managers.manager_mongo module

class datafs.managers.manager_mongo.**MongoDBManager**(*database_name*, *table_name*, *client_kwargs=None*)

Bases: *datafs.managers.manager.BaseDataManager*

Parameters

- **database_name** (*str*) – Name of the database containing the DataFS tables
- **table_name** (*str*) – Name of the data archive table
- **client_kwargs** (*dict*) – Keyword arguments used in initializing a `pymongo.MongoClient` object

collection

config

database_name

db

spec_collection

table_name

Module contents

datafs.services package

Submodules

datafs.services.service module

class datafs.services.service.**DataService**(*fs*)

Bases: `object`

upload(*filepath*, *service_path*, *remove=False*)
"Upload" a file to a service

This copies a file from the local filesystem into the `DataService`'s filesystem. If `remove==True`, the file is moved rather than copied.

If `filepath` and `service_path` paths are the same, `upload` deletes the file if `remove==True` and returns.

Parameters

- **filepath** (*str*) – Relative or absolute path to the file to be uploaded on the user's filesystem
- **service_path** (*str*) – Path to the destination for the file on the `DataService`'s filesystem
- **remove** (*bool*) – If true, the file is moved rather than copied

Module contents

Submodules

datafs.datafs module

Module contents

class datafs.**DataAPI**(*default_versions=None*, ***kwargs*)
Bases: object

DefaultAuthorityName = None

attach_authority(*service_name*, *service*)

attach_cache(*service*)

attach_manager(*manager*)

batch_get_archive(*archive_names*, *default_versions=None*)
Batch version of *get_archive()*

Parameters

- **archive_names** (*list*) – Iterable of archive names to retrieve
- **default_versions** (*str, object, or dict*) – Default versions to assign to each returned archive. If `default_versions` is a dict, each `archive_name` must be a key in `default_versions` and the value must be a valid version. Versions must be a strict version number, a `StrictVersion`, or a `BumpableVersion` object.

Returns **archives** – List of *DataArchive* objects. If an archive is not found, it is omitted (*batch_get_archive* does not raise a `KeyError` on invalid archive names).

Return type list

cache

close()

create(*archive_name*, *authority_name=None*, *versioned=True*, *raise_on_err=True*, *metadata=None*, *tags=None*, *helper=False*)
Create a DataFS archive

Parameters

- **archive_name** (*str*) – Name of the archive
- **authority_name** (*str*) – Name of the data service to use as the archive's data authority
- **versioned** (*bool*) – If true, store all versions with explicit version numbers (defualt)
- **raise_on_err** (*bool*) – Raise an error if the archive already exists (default True)
- **metadata** (*dict*) – Dictionary of additional archive metadata
- **helper** (*bool*) – If true, interactively prompt for required metadata (default False)

default_authority

default_authority_name

default_versions

delete_archive(*archive_name*)
Delete an archive

Parameters **archive_name** (*str*) – Name of the archive to delete

filter(*pattern=None*, *engine='path'*, *prefix=None*)
Performs a filtered search on entire universe of archives according to pattern or prefix.

Parameters

- **prefix** (*str*) – string matching beginning characters of the archive or set of archives you are filtering. Note that authority prefixes, e.g. `local://my/archive.txt` are not supported in prefix searches.
- **pattern** (*str*) – string matching the characters within the archive or set of archives you are filtering on. Note that authority prefixes, e.g. `local://my/archive.txt` are not supported in pattern searches.
- **engine** (*str*) – string of value 'str', 'path', or 'regex'. That indicates the type of pattern you are filtering on

Returns

Return type generator

get_archive(*archive_name*, *default_version=None*)
Retrieve a data archive

Parameters

- **archive_name** (*str*) – Name of the archive to retrieve
- **default_version** (*version*) – str or `StrictVersion` giving the default version number to be used on read operations

Returns **archive** – New *DataArchive* object

Return type object

Raises KeyError: – A KeyError is raised when the `archive_name` is not found

static hash_file(*f*)
Utility function for hashing file contents

Overload this function to change the file equality checking algorithm

Parameters **f** (*file-like*) – File-like object or file path from which to compute checksum value

Returns **checksum** – dictionary with {'algorithm': 'md5', 'checksum': hexdigest}

Return type dict

listdir(*location*, *authority_name=None*)
List archive path components at a given location

Note: When using listdir on versioned archives, listdir will provide the version numbers when a full archive path is supplied as the location argument. This is because DataFS stores the archive path as a directory and the versions as the actual files when versioning is on.

Parameters

- **location** (*str*) – Path of the "directory" to search

  location can be a path relative to the authority root (e.g */MyFiles/Data*) or can include authority as a protocol (e.g. *my_auth://MyFiles/Data*). If the authority is specified as a protocol, the *authority_name* argument is ignored.

- **authority_name** (*str*) – Name of the authority to search (optional)

  If no authority is specified, the default authority is used (if only one authority is attached or if *DefaultAuthorityName* is assigned).

Returns Archive path components that exist at the given "directory" location on the specified authority

Return type list

Raises ValueError – A ValueError is raised if the authority is ambiguous or invalid

lock_authorities()

lock_manager()

manager

remove(*archive_name*, *authority_name=None*)
Removes file system objects from a file system

Parameters **archive_name** (*str*) –

remove_dir(*path*, *authority_name=None*, *recursive=False*, *force=False*, *cache=False*)
removes the directory level file system objects a ssociated with an archive_name.

Deleting archive_name name space will erase all data associated with file system object For help setting user permissions, see *Administrative Tools*

Parameters

- **path** (*str*) – name-space to be removed from file system
- **authority_name** (*str*) – file system authority for the archive

- **recursive** (*bool*) – If True, will remove empty file system objects subordinate to the the name space provided
- **force** (*bool*) – If True, will remove file system objects wether or not they are empty
- **cache** (*bool*) – If True, will remove name space from cache file system

Returns

Return type None

search(**query*, ***kwargs*)
Searches based on tags specified by users

Parameters

- **query** (*str*) – tags to search on. If multiple terms, provided in comma delimited string format
- **prefix** (*str*) – start of archive name. Providing a start string improves search speed.

datafs.**get_api**(*profile=None*, *config_file=None*, *requirements=None*)
Generate a datafs.DataAPI object from a config profile

`get_api` generates a DataAPI object based on a pre-configured datafs profile specified in your datafs config file.

To create a datafs config file, use the command line tool `datafs configure --helper` or export an existing DataAPI object with `datafs.ConfigFile.write_config_from_api()`

Parameters

- **profile** (*str*) – (optional) name of a profile in your datafs config file. If profile is not provided, the default profile specified in the file will be used.
- **config_file** (*str or file*) – (optional) path to your datafs configuration file. By default, get_api uses your OS's default datafs application directory.

Examples

The following specifies a simple API with a MongoDB manager and a temporary storage service:

```
>>> try:
...   from StringIO import StringIO
... except ImportError:
...   from io import StringIO
...
>>> import tempfile
>>> tempdir = tempfile.mkdtemp()
>>>
>>> config_file = StringIO("""
... default-profile: my-data
... profiles:
...     my-data:
...         manager:
...             class: MongoDBManager
...             kwargs:
...                 database_name: 'MyDatabase'
...                 table_name: 'DataFiles'
...
...         authorities:
```

```
...             local:
...                 service: OSFS
...                 args: ['{}']
... """.format(tempdir))
>>>
>>> # This file can be read in using the datafs.get_api helper function
...
>>>
>>> api = get_api(profile='my-data', config_file=config_file)
>>> api.manager.create_archive_table(
...     'DataFiles',
...     raise_on_err=False)
>>>
>>> archive = api.create(
...     'my_first_archive',
...     metadata = dict(description = 'My test data archive'),
...     raise_on_err=False)
>>>
>>> with archive.open('w+') as f:
...     res = f.write(u'hello!')
...
>>> with archive.open('r') as f:
...     print(f.read())
...
hello!
>>>
>>> # clean up
...
>>> archive.delete()
>>> import shutil
>>> shutil.rmtree(tempdir)
```

datafs.**to_config_file**(*api*, *config_file=None*, *profile='default'*)

CHAPTER 11

Examples

Using DataFS Locally

This tutorial builds a DataFS server with MongoDB and the local filesystem

Running this example

To run this example:

1. Create a MongoDB server by following the MongoDB's Tutorial installation and startup instructions.
2. Start the MongoDB server (e.g. *mongod –dbpath . –nojournal*)
3. Follow the steps below

Set up the workspace

We need a few things for this example:

```
>>> from datafs.managers.manager_mongo import MongoDBManager
>>> from datafs import DataAPI
>>> from fs.osfs import OSFS
>>> import os
>>> import tempfile
>>> import shutil
>>>
>>> # overload unicode for python 3 compatability:
>>>
>>> try:
...     unicode = unicode
... except NameError:
...     unicode = str
```

Additionally, you'll need MongoDB and pymongo installed and a MongoDB instance running.

Create an API

Begin by creating an API instance:

```
>>> api = DataAPI(
...       username='My Name',
...       contact = 'my.email@example.com')
```

Attach Manager

Next, we'll choose an archive manager. DataFS currently supports MongoDB and DynamoDB managers. In this example we'll use a MongoDB manager. Make sure you have a MongoDB server running, then create a MongoDBManager instance:

```
>>> manager = MongoDBManager(
...      database_name = 'MyDatabase',
...      table_name = 'DataFiles')
```

If this is the first time you've set up this database, you'll need to create a table:

```
>>> manager.create_archive_table('DataFiles', raise_on_err=False)
```

All set. Now we can attach the manager to our DataAPI object:

```
>>> api.attach_manager(manager)
```

Attach Service

Now we need a storage service. DataFS is designed to be used with remote storage (S3, FTP, etc), but it can also be run on your local filesystem. In this tutorial we'll use a local service.

First, let's create a temporary filesystem to use for this example:

```
>>> temp = tempfile.mkdtemp()
>>> local = OSFS(temp)
```

We attach this file to the api and give it a name:

```
>>> api.attach_authority('local', local)
>>> api.default_authority
<DataService:OSFS object at ...>
```

Create archives

Next we'll create our first archive. An archive must have an archive_name. In addition, you can supply any additional keyword arguments, which will be stored as metadata. To suppress errors on re-creation, use the `raise_on_err=False` flag.

```
>>> api.create(
...      'my_first_archive',
...      metadata = dict(description = 'My test data archive'))
<DataArchive local://my_first_archive>
```

Retrieve archive metadata

Now that we have created an archive, we can retrieve it from anywhere as long as we have access to the correct service. When we retrieve the archive, we can see the metadata that was created when it was initialized.

```
>>> var = api.get_archive('my_first_archive')
```

We can access the metadata for this archive through the archive's `get_metadata()` method:

```
>>> print(var.get_metadata()['description'])
My test data archive
```

Add a file to the archive

An archive is simply a versioned history of data files. So let's get started adding data!

First, we'll create a local file, `test.txt`, and put some data in it:

```
>>> with open('test.txt', 'w+') as f:
...     f.write('this is a test')
```

Now we can add this file to the archive:

```
>>> var.update('test.txt', remove=True)
```

This file just got sent into our archive! And we deleted the local copy:

```
>>> os.path.isfile('test.txt')
False
```

Reading from the archive

Next we'll read from the archive. That file object returned by `var.open()` can be read just like a regular file

```
>>> with var.open('r') as f:
...     print(f.read())
...
this is a test
```

Updating the archive

Open the archive and write to the file:

```
>>> with var.open('w+') as f:
...     res = f.write(unicode('this is the next test'))
...
```

Retrieving the latest version

Now let's make sure we're getting the latest version:

```
>>> with var.open('r') as f:
...     print(f.read())
...
this is the next test
```

Looks good!

Cleaning up

```
>>> var.delete()
>>> api.manager.delete_table('DataFiles')
>>> shutil.rmtree(temp)
```

Next steps

The *next tutorial* describes setting up DataFS for remote obejct stores, such as with AWS storage.

Using Remote Files Locally

This tutorial demonstrates reading and writing files to remote archives using on-disk I/O operations.

To demonstrate this, we make use of the `xarray` module, which cannot read from a streaming object.

Set up the workspace

```
>>> from datafs import DataAPI
>>> from datafs.managers.manager_mongo import MongoDBManager
>>> from fs.s3fs import S3FS
>>> from fs.s3fs import RemoveRootError, ResourceNotFoundError
```

Initialize the API

```
>>> api = DataAPI(
...      username='My Name',
...      contact = 'my.email@example.com')
>>>
>>> manager = MongoDBManager(
...     database_name = 'MyDatabase',
...     table_name = 'DataFiles')
>>>
>>> manager.create_archive_table('DataFiles', raise_on_err=False)
>>>
>>> api.attach_manager(manager)
```

Attach a remote service

In this example we'll use a remote file system, in this case AWS S3. This filesystem returns streaming objects returned by `boto` or `request` calls.

```
>>> s3 = S3FS(
...     'test-bucket',
...     aws_access_key='MY_KEY',
...     aws_secret_key='MY_SECRET_KEY')
>>>
>>> api.attach_authority('aws', s3)
>>>
>>> var = api.create(
...     'streaming_archive',
...     metadata = dict(description = 'My test data archive'))
>>>
```

Create sample data

Create a sample dataset (from the xarray docs):

```
>>> import xarray as xr
>>> import numpy as np
>>> import pandas as pd
>>>
>>> np.random.seed(123)
>>>
>>> times = pd.date_range('2000-01-01', '2001-12-31', name='time')
>>> annual_cycle = np.sin(2 * np.pi * (times.dayofyear / 365.25 - 0.28))
>>>
>>> base = 10 + 15 * annual_cycle.reshape(-1, 1)
>>> tmin_values = base + 3 * np.random.randn(annual_cycle.size, 3)
>>> tmax_values = base + 10 + 3 * np.random.randn(annual_cycle.size, 3)
>>>
>>> ds = xr.Dataset({'tmin': (('time', 'location'), tmin_values),
...                  'tmax': (('time', 'location'), tmax_values)},
...                 {'time': times, 'location': ['IA', 'IN', 'IL']})
>>>
>>> ds.attrs['version'] = 'version 1'
```

Upload the dataset to the archive

```
>>> with var.get_local_path() as f:
...     ds.to_netcdf(f)
...
```

Read and write to disk

NetCDF files cannot be read from a streaming object:

```
>>> with var.open() as f:
...     print(type(f))
...
<type '_io.TextIOWrapper'>
```

```
>>> with var.open() as f:
...     with xr.open_dataset(f) as ds:
...         print(ds)
Traceback (most recent call last):
```

```
...
UnicodeDecodeError: 'utf8' codec can't decode byte 0x89 in position 0: \
invalid start byte
```

Instead, we can get a local path to open:

```
>>> with var.get_local_path() as f:
...     with xr.open_dataset(f) as ds:
...         print(ds)
...
<xarray.Dataset>
Dimensions:   (location: 3, time: 731)
Coordinates:
  * location  (location) |S2 'IA' 'IN' 'IL'
  * time      (time) datetime64[ns] 2000-01-01 2000-01-02 2000-01-03 ...
Data variables:
    tmax      (time, location) float64 12.98 3.31 6.779 0.4479 6.373 ...
    tmin      (time, location) float64 -8.037 -1.788 -3.932 -9.341 ...
Attributes:
    version: version 1
```

We can update file in the same way:

```
>>> with var.get_local_path() as f:
...     with xr.open_dataset(f) as ds:
...
...         # Load the dataset fully into memory and then close the file
...
...         dsmem = ds.load()
...         ds.close()
...
...     # Update the version and save the file
...
...     dsmem.attrs['version'] = 'version 2'
...     dsmem.to_netcdf(f)
...
```

Now let's open the file and see if our change was saved:

```
>>> # Acquire the file from the archive and print the version
... with var.get_local_path() as f:
...     with xr.open_dataset(f) as ds:
...         print(ds)
...
<xarray.Dataset>
Dimensions:   (location: 3, time: 731)
Coordinates:
  * location  (location) |S2 'IA' 'IN' 'IL'
  * time      (time) datetime64[ns] 2000-01-01 2000-01-02 2000-01-03 ...
Data variables:
    tmax      (time, location) float64 12.98 3.31 6.779 0.4479 6.373 ...
    tmin      (time, location) float64 -8.037 -1.788 -3.932 -9.341 ...
Attributes:
    version: version 2
```

Clean-Up

```
>>> api.delete_archive('streaming_archive')
```

Using DataFS with AWS's S3

Use this tutorial to build a DataFS server system using MongoDB and a Simple Storage Service such as AWS's S3.

Running this example

To run this example:

1. Create a MongoDB server by following the MongoDB's Tutorial installation and startup instructions.
2. Start the MongoDB server (e.g. `mongod --dbpath . --nojournal`)
3. Follow the steps below

Set up the workspace

We need a few things for this example:

```
>>> from datafs.managers.manager_mongo import MongoDBManager
>>> from datafs import DataAPI
>>> from fs.tempfs import TempFS
>>> from fs.errors import RemoveRootError
>>> import os
>>> import tempfile
>>> import shutil
>>>
>>> # overload unicode for python 3 compatability:
>>>
>>> try:
...     unicode = unicode
... except NameError:
...     unicode = str
```

This time, we'll import PyFilesystem's S3 Filesystem abstraction:

```
>>> from fs.s3fs import S3FS
```

Additionally, you'll need MongoDB and pymongo installed and a MongoDB instance running.

Create an API

Begin by creating an API instance:

```
>>> api = DataAPI(
...      username='My Name',
...      contact = 'my.email@example.com')
```

Attach Manager

Next, we'll choose an archive manager. DataFS currently supports MongoDB and DynamoDB managers. In this example we'll use a local MongoDB manager. Make sure you have a MongoDB server running, then create a MongoDBManager instance:

```
>>> manager = MongoDBManager(
...     database_name = 'MyDatabase',
...     table_name = 'DataFiles')
```

If this is the first time you've set up this database, you'll need to create a table:

```
>>> manager.create_archive_table('DataFiles', raise_on_err=False)
```

All set. Now we can attach the manager to our DataAPI object:

```
>>> api.attach_manager(manager)
```

Attach Service

Now we need a storage service. Let's attach the S3FS filesystem we imported:

```
>>> s3 = S3FS(
...     'test-bucket',
...     aws_access_key='MY_KEY',
...     aws_secret_key='MY_SECRET_KEY')
>>>
>>> api.attach_authority('aws', s3)
```

Create archives

Now we can create our first archive. An archive must have an archive_name. In addition, you can supply any additional keyword arguments, which will be stored as metadata. To suppress errors on re-creation, use the `raise_on_err=False` flag.

```
>>> api.create(
...     'my_remote_archive',
...     metadata = dict(description = 'My test data archive'))
<DataArchive aws://my_remote_archive>
```

View all available archives

Let's see what archives we have available to us.

```
>>> print(next(api.filter()))
my_remote_archive
```

Retrieve archive metadata

Now that we have created an archive, we can retrieve it from anywhere as long as we have access to the correct service. When we retrieve the archive, we can see the metadata that was created when it was initialized.

```
>>> var = api.get_archive('my_remote_archive')
```

We can access the metadata for this archive through the archive's `get_metadata()` method:

```
>>> print(var.get_metadata()['description'])
My test data archive
```

Add a file to the archive

An archive is simply a versioned history of data files. So let's get started adding data!

First, we'll create a local file, `test.txt`, and put some data in it:

```
>>> with open('test.txt', 'w+') as f:
...     f.write('this is a test')
```

Now we can add this file to the archive:

```
>>> var.update('test.txt')
```

This file just got sent into our archive! Now we can delete the local copy:

```
>>> os.remove('test.txt')
```

Reading from the archive

Next we'll read from the archive. That file object returned by `var.open()` can be read just like a regular file

```
>>> with var.open('r') as f:
...     print(f.read())
...
this is a test
```

Updating the archive

Open the archive and write to the file:

```
>>> with var.open('w+') as f:
...     res = f.write(unicode('this is the next test'))
```

Retrieving the latest version

Now let's make sure we're getting the latest version:

```
>>> with var.open() as f:
...     print(f.read())
...
this is the next test
```

Looks good!

Cleaning up

```
>>> api.delete_archive('my_remote_archive')
```

Next steps

Using Other Services describes setting up DataFS for other filesystems, such as sftp or http.

Using Other Services

Because the file operations in DataFS are backed by pyFilesystem, DataFS supports any of the following remote storage services:

FTP An interface to FTP servers. See `fs.ftpfs`

Memory A filesystem that exists entirely in memory. See `fs.memoryfs`

OS An interface to the OS Filesystem. See `fs.osfs`

RPCFS An interface to a file-system served over XML RPC, See `fs.rpcfs` and `fs.expose.xmlrpc`

SFTP A secure FTP filesystem. See `fs.sftpfs`

S3 A filesystem to access an Amazon S3 service. See `fs.s3fs`

Temporary Creates a temporary filesystem in an OS provided location. See `fs.tempfs`

Zip An interface to zip files. See `fs.zipfs`

Caching Remote Results Locally

Setup

```
>>> from datafs.managers.manager_mongo import MongoDBManager
>>> from datafs import DataAPI
>>> from fs.tempfs import TempFS
>>> from fs.s3fs import S3FS
>>> from fs.errors import ResourceNotFoundError, RemoveRootError
>>> import os
>>> import tempfile
>>> import shutil
```

Create an API and attach a manager

```
>>> api = DataAPI(
...       username='My Name',
...       contact = 'my.email@example.com')
>>>
>>> manager = MongoDBManager(
...      database_name = 'MyDatabase',
...      table_name = 'DataFiles')
>>>
>>> manager.create_archive_table('DataFiles', raise_on_err=False)
```

```
>>> api.attach_manager(manager)
>>>
```

For this example we'll use an AWS S3 store. Any filesystem will work, though:

```
>>> s3 = S3FS(
...     'test-bucket',
...     aws_access_key='MY_KEY',
...     aws_secret_key='MY_SECRET_KEY')
>>>
>>> api.attach_authority('aws', s3)
```

Create an archive

```
>>> var = api.create(
...     'caching/archive.txt',
...     metadata = dict(description = 'My cached remote archive'),
...     authority_name='aws')
>>>
>>> with var.open('w+') as f:
...     res = f.write(u'hello')
...
>>>
>>> with var.open('r') as f:
...     print(f.read())
hello
```

Let's peek under the hood to see where this data is stored:

```
>>> url = var.authority.fs.getpathurl(var.get_version_path())
>>> print(url)
https://test-bucket.s3.amazonaws.com/caching/...AWSAccessKeyId=MY_KEY
```

Now let's set up a cache. This would typically be a local or networked directory but we'll use a temporary filesystem for this example:

```
>>> cache = TempFS()
>>> api.attach_cache(cache)
```

Now we can activate caching for our archive:

```
>>> var.cache()
```

When we read the data from the cache, it downloads the file for future use:

```
>>> with var.open('r') as f:
...     print(f.read())
hello
```

Cleanup

```
>>> var.delete()
```

Configuring DataFS for your Team

This tutorial walks through the process of creating the specification files and setting up resources for use on a large team. It assumes a basic level of familiarity with the purpose of DataFS, and also requires administrative access to any resources you'd like to use, such as AWS.

Set up a connection to AWS

To use AWS resources, you'll need credentials. These are most easily specified in a credentials file.

We've provided a sample file `here`:

```
[aws-test]
aws_access_key_id=MY_AWS_ACCESS_KEY_ID
aws_secret_access_key=MY_AWS_SECRET_ACCESS_KEY
```

This file is located `~/.aws/credentials` by default, but we'll tell AWS how to find it locally using an environment variable for the purpose of this example:

```
>>> import os
>>>
>>> # Change this to wherever your credentials file is:
... credentials_file_path = os.path.join(
...     os.path.dirname(__file__),
...     'credentials')
...
>>> os.environ['AWS_SHARED_CREDENTIALS_FILE'] = credentials_file_path
```

Configure DataFS for your organization/use

Now that you have a connection to AWS, you can specify how you want DataFS to work. DataFS borrows the idea of profiles, allowing you to have multiple pre-configured file managers at once.

We'll set up a test profile called "example" `here`:

```
# Specify a default profile
default-profile: example

# Configure your profiles here
profiles:

    # Everything under this key specifies the example profile
    example:

        api:

            # Enter user data for each user
            user_config:
                contact: me@email.com
                username: My Name


        # Add multiple data filesystems to use as
        # the authoritative source for an archive
        authorities:
```

```
            # The authority "local" is an OSFS
            # (local) filesystem, and has the relative
            # path "example_data_dir" as it's root.
            local:
                service: OSFS
                args: [example_data_dir]

            # The authority "remote" is an AWS S3FS
            # filesystem, and uses the "aws-test"
            # profile in the aws config file to
            # connect to resources on Amazon's us-east-1
            remote:
                service: S3FS
                args: ['test-bucket']
                kwargs:
                    region_name: us-east-1
                    profile_name: 'aws-test'


        # Add one manager per profile
        manager:

            # This manager accesses the table
            # 'my-test-data' in a local instance
            # of AWS's DynamoDB. To use a live
            # DynamoDB, remove the endpoint_url
            # specification.
            class: DynamoDBManager
            kwargs:
                resource_args:
                    endpoint_url: 'http://localhost:8000/'
                    region_name: 'us-east-1'

                session_args:
                    profile_name: aws-test

                table_name: my-test-data
```

Set up team managers and services

Make sure that the directories, buckets, etc. that your services are connecting to exist:

```
>>> if not os.path.isdir('example_data_dir'):
...     os.makedirs('example_data_dir')
```

Now, boot up an API and create the archive table on your manager that corresponds to the one specified in your

```
>>> import datafs
>>> api = datafs.get_api(
...     profile='example',
...     config_file='examples/preconfigured/.datafs.yml')
>>>
>>> api.manager.create_archive_table('my-test-data')
```

Finally, we'll set some basic reporting requirements that will be enforced when users interact with the data.

We can require user information when writing/updating an archive. `set_required_user_config()` allows administrators to set user configuration requirements and provide a prompt to help users:

```
>>> api.manager.set_required_user_config({
...     'username': 'your full name',
...     'contact': 'your email address'})
```

Similarly, `set_required_archive_metadata()` sets the metadata that is required for each archive:

```
>>> api.manager.set_required_archive_metadata({
...     'description': 'a long description of the archive'})
```

Attempts by users to create/update archives without these attributes will now fail.

Using the API

At this point, any users with properly specified credentials and config files can use the data api.

From within python:

```
>>> import datafs
>>> api = datafs.get_api(
...     profile='example',
...     config_file='examples/preconfigured/.datafs.yml')
>>>
>>> archive = api.create(
...     'archive1',
...     authority_name='local',
...     metadata = {'description': 'my new archive'})
```

Note that the metadata requirements you specified are enforced. If a user tries to skip the description, an error is raised and the archive is not created:

```
>>> archive = api.create(
...     'archive2',
...     authority_name='local')
...
Traceback (most recent call last):
...
AssertionError: Required value "description" not found. Use helper=True or
the --helper flag for assistance.
>>>
>>> print(next(api.filter()))
archive1
```

Setting User Permissions

Users can be managed using policies on AWS's admin console. An example policy allowing users to create, update, and find archives without allowing them to delete archives or to modify the required metadata specification is provided `here`:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:DescribeTable",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/my-test-data"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:DescribeTable",
                "dynamodb:GetItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": [
                "arn:aws:dynamodb:*:*:table/my-test-data.spec"
            ]
        }
    ]
}
```

A user with AWS access keys using this policy will see an AccessDeniedException when attempting to take restricted actions:

```
>>> import datafs
>>> api = datafs.get_api(profile='user')
>>>
>>> archive = api.get_archive('archive1')
>>>
>>> archive.delete()
Traceback (most recent call last):
...
botocore.exceptions.ClientError: An error occurred (AccessDeniedException)
when calling the DeleteItem operation: ...
```

Teardown

A user with full privileges can completely remove archives and manager tables:

```
>>> api.delete_archive('archive1')
>>> api.manager.delete_table('my-test-data')
```

CHAPTER 12

Documentation Code Source

Command Line Interface: Tagging

This is the tested source code for the snippets used in *Tagging Archives*. The config file we're using in this example can be downloaded `here`.

Example 1

Displayed example 1 code:

```
$ datafs create archive1 --tag "foo" --tag "bar" --description \
>     "tag test 1 has bar"
created versioned archive <DataArchive local://archive1>

$ datafs create archive2 --tag "foo" --tag "baz" --description \
>     "tag test 2 has baz"
created versioned archive <DataArchive local://archive2>
```

Example 2

```
$ datafs search bar
archive1

$ datafs search baz
archive2

$ datafs search foo
archive1
archive2
```

Example 3

```
$ datafs create archive3 --tag "foo" --tag "bar" --tag "baz" \
>     --description 'tag test 3 has all the tags!'
created versioned archive <DataArchive local://archive3>

$ datafs search bar foo
archive3
archive1


$ datafs search bar foo baz
archive3
```

Example 4

```
$ datafs search qux

$ datafs search foo qux
```

Example 5

```
$ datafs get_tags archive1
foo bar
```

Example 6

```
$ datafs add_tags archive1 qux

$ datafs search foo qux
archive1
```

Example 7

```
$ datafs delete_tags archive1 foo bar

$ datafs search foo bar
archive3
```

Teardown

```
$ datafs delete archive1
deleted archive <DataArchive local://archive1>

$ datafs delete archive2
deleted archive <DataArchive local://archive2>
```

```
$ datafs delete archive3
deleted archive <DataArchive local://archive3>
```

Python API: Creating Archives

This is the tested source code for the snippets used in *Creating Data Archives*. The config file we're using in this example can be downloaded `here`.

Setup

```
>>> import datafs
>>> from fs.tempfs import TempFS
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have a config file at the above location. The config file we're using in this example can be downloaded `here`.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('my_archive_name')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Example 1

Displayed example 1 code:

```
>>> archive = api.create('my_archive_name')
```

Example 1 cleanup:

```
>>> api.delete_archive('my_archive_name')
```

Example 2

Example 2 setup

```
>>> api.attach_authority('my_authority', TempFS())
```

Displayed example 2 code

```
>>> archive = api.create('my_archive_name')
Traceback (most recent call last):
...
ValueError: Authority ambiguous. Set authority or DefaultAuthorityName.
```

Example 3

```
>>> archive = api.create(
...     'my_archive_name',
...     authority_name='my_authority')
...
```

Example 3 cleanup:

```
>>> try:
...     api.delete_archive('my_archive_name')
... except KeyError:
...     pass
...
```

Example 4

```
>>> api.DefaultAuthorityName = 'my_authority'
>>> archive = api.create('my_archive_name')
```

Example 4 cleanup:

```
>>> try:
...     api.delete_archive('my_archive_name')
... except KeyError:
...     pass
...
```

Example 5

```
>>> archive = api.create(
...     'my_archive_name',
...     metadata={
...         'description': 'my test archive',
...         'source': 'Burke et al (2015)',
...         'doi': '10.1038/nature15725'})
...
```

Example 5 cleanup:

```
>>> try:
...     api.delete_archive('my_archive_name')
... except KeyError:
...     pass
...
```

Example 6

Example 6 setup:

```
>>> api.manager.set_required_archive_metadata(
...     {'description': 'Enter a description'})
...
```

Displayed example:

```
>>> archive = api.create(
...     'my_archive_name',
...     metadata = {
...         'source': 'Burke et al (2015)',
...         'doi': '10.1038/nature15725'})
...
Traceback (most recent call last):
...
AssertionError: Required value "description" not found. Use helper=True or
the --helper flag for assistance.
```

Example 6 cleanup:

```
>>> try:
...     api.delete_archive('my_archive_name')
... except KeyError:
...     pass
...
```

Example 7

```
>>> archive = api.create(
...     'my_archive_name',
...     metadata={
...         'source': 'Burke et al (2015)',
...         'doi': '10.1038/nature15725'},
...         helper=True)
...
Enter a description:
```

Teardown

```
>>> try:
...     api.delete_archive('my_archive_name')
... except KeyError:
...     pass
```

```
...

>>> api.manager.delete_table('DataFiles')
```

Python API: Tagging

This is the tested source code for the snippets used in *Tagging Archives*. The config file we're using in this example can be downloaded `here`.

Setup

```
>>> import datafs
>>> from fs.tempfs import TempFS
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have a config file at the above location. The config file we're using in this example can be downloaded `here`.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('archive1')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.delete_archive('archive2')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.delete_archive('archive3')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Example 1

Displayed example 1 code:

```
>>> archive1 = api.create(
...       'archive1',
...       tags=["foo", "bar"],
...       metadata={'description': 'tag test 1 has bar'})
...
>>> archive2 = api.create(
...       'archive2',
...       tags=["foo", "baz"],
...       metadata={'description': 'tag test 1 has baz'})
...
```

Example 2

```
>>> list(api.search('bar'))
['archive1']

>>> list(api.search('baz'))
['archive2']

>>> list(api.search('foo'))
['archive1', 'archive2']
```

Actual Example Block 2 Test

```
>>> assert set(api.search('bar')) == {'archive1'}
>>> assert set(api.search('baz')) == {'archive2'}
>>> assert set(api.search('foo')) == {'archive1', 'archive2'}
```

Example 3

```
>>> archive3 = api.create(
...       'archive3',
...       tags=["foo", "bar", "baz"],
...       metadata={'description': 'tag test 3 has all the tags!'})
...

>>> list(api.search('bar', 'foo'))
['archive1', 'archive3']

>>> list(api.search('bar', 'foo', 'baz'))
['archive3']
```

Actual example block 3 search test:

```
>>> assert set(api.search('bar', 'foo')) == {'archive1', 'archive3'}
>>> assert set(api.search('bar', 'foo', 'baz')) == {'archive3'}
```

Example 4

```
>>> list(api.search('qux'))
[]
```

```
>>> list(api.search('foo', 'qux'))
[]
```

Actual example block 4 test

```
>>> assert set(api.search('qux')) == set([])
```

Example 5

```
>>> archive1.get_tags()
['foo', 'bar']
```

Actual example block 5 test

```
>>> assert set(archive1.get_tags()) == {'foo', 'bar'}
```

Example 6

```
>>> archive1.add_tags('qux')
>>>
>>> list(api.search('foo', 'qux'))
['archive1']
```

Actual example block 6 search test

```
>>> assert set(api.search('foo', 'qux')) == {'archive1'}
```

Example 7

```
>>> archive1.delete_tags('foo', 'bar')
>>>
>>> list(api.search('foo', 'bar'))
['archive3']
```

Actual example block 7 search test

```
>>> assert set(api.search('foo', 'bar')) == {'archive3'}
```

Teardown

```
>>> archive1.delete()
>>> archive2.delete()
>>> archive3.delete()
```

Python API: Reading and Writing

This is the tested source code for the snippets used in *Reading and Writing Files*. The config file we're using in this example can be downloaded `here`.

Setup

```
>>> import datafs
>>> from fs.tempfs import TempFS
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have a config file at the above location. The config file we're using in this example can be downloaded `here`.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('sample_archive')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Example 1

Displayed example 1 code:

```
>>> api.create(
...     'sample_archive',
...     metadata={'description': 'metadata for your archive'})
...
<DataArchive local://sample_archive>
```

Example 2

```
>>> with open('sample.txt', 'w+') as f:
...     f.write('this is a sample archive')
...
```

Example 3

```
>>> sample_var = api.get_archive('sample_archive')
>>> sample_var.update('sample.txt')
```

Example 4

```
>>> with sample_var.open('r') as f:
...     print(f.read())
...
this is a sample archive
```

Example 5

```
>>> with open('sample.txt', 'w+') as f:
...     f.write('this is a sample archive with some more information')
...
>>> sample_var.update('sample.txt')
```

Example 6

```
>>> with sample_var.open('r') as f:
...     print(f.read())
...
this is a sample archive with some more information
```

Example 7

```
>>> import os
>>> with open('sample.txt', 'w+') as f:
...         f.write(u'Local file to update to our FS')
...
```

```
>>> sample_var.update('sample.txt')
```

```
>>> sample_archive_local = api.get_archive('sample_archive')
>>> sample_archive_local.download('path_to_sample.txt', version='latest')
```

Example 9

```
>>> with open('path_to_sample.txt', 'r') as f:
...     print(f.read())
...
Local file to update to our FS
```

Teardown

```
>>> try:
...     api.delete_archive('sample_archive')
... except (KeyError, OSError):
...     pass
...
```

```
>>> os.remove('path_to_sample.txt')
>>> os.remove('sample.txt')
```

```
>>> import numpy as np
>>> import pandas as pd
>>> import xarray as xr
>>>
>>> np.random.seed(123)
>>>
>>> times = pd.date_range('2000-01-01', '2001-12-31', name='time')
>>> annual_cycle = np.sin(2 * np.pi * (times.dayofyear / 365.25 - 0.28))
>>>
>>> base = 10 + 15 * annual_cycle.reshape(-1, 1)
>>> tmin_values = base + 3 * np.random.randn(annual_cycle.size, 3)
>>> tmax_values = base + 10 + 3 * np.random.randn(annual_cycle.size, 3)
>>>
>>> ds = xr.Dataset({'tmin': (('time', 'location'), tmin_values),
...                  'tmax': (('time', 'location'), tmax_values)},
...                 {'time': times, 'location': ['IA', 'IN', 'IL']})
>>>
>>>
>>>
>>> streaming_archive = api.create(
...     'streaming_archive',
...     metadata={'description': 'metadata description for your archive'})
...
>>> with streaming_archive.get_local_path() as f:
...         ds.to_netcdf(f)
...
>>>
>>>
```

```
>>> with streaming_archive.get_local_path() as f:
...     with xr.open_dataset(f) as ds:
...         print(ds)
...
<xarray.Dataset>
Dimensions:   (location: 3, time: 731)
Coordinates:
  * location  (location) |S2 'IA' 'IN' 'IL'
  * time      (time) datetime64[ns] 2000-01-01 2000-01-02 2000-01-03 ...
Data variables:
    tmax      (time, location) float64 12.98 3.31 6.779 0.4479 6.373 ...
    tmin      (time, location) float64 -8.037 -1.788 -3.932 -9.341 ...
```

Teardown

```
>>> streaming_archive.delete()
```

Python API: Managing Metadata

This is the tested source code for the snippets used in *Managing Metadata*. The config file we're using in this example can be downloaded `here`.

Setup

```
>>> import datafs
>>> from fs.tempfs import TempFS
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have a config file at the above location. The config file we're using in this example can be downloaded here.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('sample_archive')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Example 1

Displayed example 1 code

```
>>> sample_archive = api.create(
...    'sample_archive',
...     metadata = {
...         'oneline_description': 'tas by admin region',
...         'source': 'NASA BCSD',
...         'notes': 'important note'})
...
```

Example 2

Displayed example 2 code

```
>>> for archive_name in api.filter():
...     print(archive_name)
...
sample_archive
```

Example 3

Displayed example 3 code

```
>>> sample_archive.get_metadata()
{u'notes': u'important note', u'oneline_description': u'tas by admin region
', u'source': u'NASA BCSD'}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> sample_archive.get_metadata() == {
...     u'notes': u'important note',
...     u'oneline_description': u'tas by admin region',
...     u'source': u'NASA BCSD'}
...
True
```

Example 4

Displayed example 4 code

```
>>> sample_archive.update_metadata(dict(
...     source='NOAAs better temp data',
...     related_links='http://wwww.noaa.gov'))
...
```

Example 5

Displayed example 5 code

```
>>> sample_archive.get_metadata()
{u'notes': u'important note', u'oneline_description': u'tas by admin region
', u'source': u'NOAAs better temp data', u'related_links': u'http://wwww.no
aa.gov'}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> sample_archive.get_metadata() == {
...     u'notes': u'important note',
...     u'oneline_description': u'tas by admin region',
...     u'source': u'NOAAs better temp data',
...     u'related_links': u'http://wwww.noaa.gov'}
...
True
```

Example 6

Displayed example 6 code

```
>>> sample_archive.update_metadata(dict(related_links=None))
>>>
>>> sample_archive.get_metadata()
{u'notes': u'important note', u'oneline_description': u'tas by admin region
', u'source': u'NOAAs better temp data'}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> sample_archive.get_metadata() == {
...     u'notes': u'important note',
...     u'oneline_description': u'tas by admin region',
...     u'source': u'NOAAs better temp data'}
...
True
```

Teardown

```
>>> try:
...     api.delete_archive('sample_archive')
... except KeyError:
...     pass
...

>>> api.manager.delete_table('DataFiles')
```

Python API: Versioning Data

This is the tested source code for the snippets used in *Tracking Versions*. The config file we're using in this example can be downloaded `here`.

SetUp

```
>>> import datafs
>>> from fs.tempfs import TempFS
>>> import os
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have a config file at the above location. The config file we're using in this example can be downloaded `here`.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('sample_archive')
... except (KeyError, OSError):
...     pass
```

```
...
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Example 1

```
>>> with open('sample_archive.txt', 'w+', ) as f:
...     f.write(u'this is a sample archive')
...
>>> sample_archive = api.create(
...     'sample_archive',
...     metadata = {'description': 'metadata description'})
...
>>> sample_archive.update('sample_archive.txt', prerelease='alpha')
```

cleanup:

```
>>> os.remove('sample_archive.txt')
```

Example 2

```
>>> sample_archive.get_versions()
[BumpableVersion ('0.0.1a1')]
```

Example 3

```
>>> with open('sample_archive.txt', 'w+', ) as f:
...     f.write(u'Sample archive with more text so we can bumpversion')
...
>>>
>>> sample_archive.update('sample_archive.txt', bumpversion='minor')
>>> sample_archive.get_versions()
[BumpableVersion ('0.0.1a1'), BumpableVersion ('0.1')]
```

cleanup:

```
>>> os.remove('sample_archive.txt')
```

Example 4

```
>>> sample_archive.get_latest_version()
BumpableVersion ('0.1')
```

Example 5

```
>>> sample_archive.get_latest_hash()
u'510d0e2eadd19514788e8fdf91e3bd5c'
```

Example 6

```
>>> sample_archive1 = api.get_archive(
...     'sample_archive',
...     default_version='0.0.1a1')
...
>>> with sample_archive1.open('r') as f:
...    print(f.read())
...
Sample archive with more text so we can bumpversion
```

Teardown

```
>>> try:
...     api.delete_archive('sample_archive')
... except (KeyError, OSError):
...     pass
...
```

Python API: Dependencies

This is the tested source code for the snippets used in *Managing Data Dependencies*. The config file we're using in this example can be downloaded `here`. Later on in the script, we use a requirements file. That file can be downloaded `here`.

Setup

```
>>> import datafs
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have the provided config file at the above location.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('my_archive')
... except (KeyError, OSError):
...     pass
...
>>> try:
```

```
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Example 1

Displayed example 1 code

```
>>> my_archive = api.create('my_archive')
>>> with my_archive.open('w+',
...     dependencies={'archive2': '1.1', 'archive3': None}) as f:
...
...     res = f.write(u'contents depend on archive 2 v1.1')
...
>>> my_archive.get_dependencies()
{'archive2': '1.1', 'archive3': None}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> my_archive.get_dependencies() == {
...     'archive2': '1.1', 'archive3': None}
...
True
```

Example 2

Displayed example 2 code

```
>>> with my_archive.open('w+') as f:
...
...     res = f.write(u'contents depend on archive 2 v1.2')
...
>>> my_archive.set_dependencies({'archive2': '1.2'})
>>> my_archive.get_dependencies()
{'archive2': '1.2'}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> my_archive.get_dependencies() == {'archive2': '1.2'}
True
```

Example 3

The following text is displayed to demonstrate the file contents:

```
dep1==1.0
dep2==0.4.1a3
```

The following code creates the actual file contents

Example 4

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml',
...     requirements='examples/snippets/resources/requirements_data.txt')
...
```

Example:

```
>>> api = datafs.get_api(
...     requirements = 'requirements_data.txt')
...
>>>
>>> my_archive = api.get_archive('my_archive')
>>> with my_archive.open('w+') as f:
...     res = f.write(u'depends on dep1 and dep2')
...
>>> my_archive.get_dependencies()
{'dep1': '1.0', 'dep2': '0.4.1a3'}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> my_archive.get_dependencies() == {'dep1': '1.0', 'dep2': '0.4.1a3'}
True
```

Example 5

```
>>> my_archive.get_dependencies()
{'dep1': '1.0', 'dep2': '0.4.1a3'}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> my_archive.get_dependencies() == {'dep1': '1.0', 'dep2': '0.4.1a3'}
True
```

Example 6

```
>>> my_archive.get_dependencies(version='0.0.1')
{'archive2': '1.1', 'archive3': None}
```

The last line of this test cannot be tested directly (exact dictionary formatting is unstable), so is tested in a second block:

```
>>> my_archive.get_dependencies(version='0.0.1') == {
...     'archive2': '1.1', 'archive3': None}
True
```

Teardown

```
>>> try:
...     api.delete_archive('my_archive')
... except KeyError:
...     pass
...

>>> api.manager.delete_table('DataFiles')
```

Python API: Searching and Finding Archives

This is the tested source code for the snippets used in *Searching and Finding Archives*. The config file we're using in this example can be downloaded `here`.

Setup

```
>>> import datafs
>>> from fs.tempfs import TempFS
>>> import os
>>> import itertools
```

We test with the following setup:

```
>>> api = datafs.get_api(
...     config_file='examples/snippets/resources/datafs_mongo.yml')
...
```

This assumes that you have a config file at the above location. The config file we're using in this example can be downloaded `here`.

clean up any previous test failures

```
>>> try:
...     api.delete_archive('my_archive')
...     api.delete_archive('streaming_archive')
...     api.delete_archive('sample_archive')
... except (KeyError, OSError):
...     pass
...
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
...
```

Add a fresh manager table:

```
>>> api.manager.create_archive_table('DataFiles')
```

Set up some archives to search

```
>>> with open('test.txt', 'w') as f:
...     f.write('test test')
...
>>> tas_archive = api.create('impactlab/climate/tas/tas_day_us.csv')
>>> tas_archive.update('test.txt')
>>> precip_archive = api.create('impactlab/climate/pr/pr_day_us.csv')
>>> precip_archive.update('test.txt')
>>> socio = api.create('impactlab/mortality/global/mortality_glob_day.csv')
>>> socio.update('test.txt')
>>> socio1 = api.create('impactlab/conflict/global/conflict_glob_day.csv')
>>> socio1.update('test.txt')
>>> socio2 = api.create('impactlab/labor/global/labor_glob_day.csv')
>>> socio2.update('test.txt')
```

Example 1

Displayed example 1 code

```
>>> api.listdir('impactlab/conflict/global')
[u'conflict_glob_day.csv']
```

Example 2

Displayed example 2 code

```
>>> api.listdir('')
[u'impactlab']
```

Example 3

Displayed example 3 code

```
>>> api.listdir('impactlab')
[u'labor', u'climate', u'conflict', u'mortality']
```

Example 4

Displayed example 4 code

```
>>> api.listdir('impactlab/conflict')
[u'global']
```

Example 5

Displayed example 5 code

```
>>> api.listdir('impactlab/conflict/global')
[u'conflict_glob_day.csv']
>>> api.listdir('impactlab/conflict/global/conflict_glob_day.csv')
[u'0.0.1']
```

Teardown

```
>>> try:
...     tas_archive.delete()
...     precip_archive.delete()
...     socio.delete()
...     socio1.delete()
...     socio2.delete()
...     os.remove('test.txt')
... except KeyError:
...     pass
>>> try:
...     api.manager.delete_table('DataFiles')
... except KeyError:
...     pass
```

Setup

```

```

```
>>> api.manager.create_archive_table('DataFiles')
```

Filter example setup

```
>>> archive_names = []
>>> for indices in itertools.product(*(range(1, 6) for _ in range(3))):
...     archive_name = (
...     'project{}_variable{}_scenario{}.nc'.format(*indices))
...     archive_names.append(archive_name)
>>>
>>> for i, name in enumerate(archive_names):
...     if i % 3  == 0:
...         api.create(name, tags=['team1'])
...     elif i % 2 == 0:
...         api.create(name, tags=['team2'])
...     else:
...         api.create(name, tags=['team3'])
<DataArchive local://project1_variable1_scenario1.nc>
<DataArchive local://project1_variable1_scenario2.nc>
<DataArchive local://project1_variable1_scenario3.nc>
...
<DataArchive local://project5_variable5_scenario3.nc>
<DataArchive local://project5_variable5_scenario4.nc>
<DataArchive local://project5_variable5_scenario5.nc>
```

Example 6

Displayed example 6 code

```
>>> len(list(api.filter()))
125
>>> filtered_list1 = api.filter(prefix='project1_variable1_')
>>> list(filtered_list1)
[u'project1_variable1_scenario1.nc', u'project1_variable1_scenario2.nc',
u'project1_variable1_scenario3.nc', u'project1_variable1_scenario4.nc',
u'project1_variable1_scenario5.nc']
```

Example 7

Displayed example 7 code

```
>>> filtered_list2 = api.filter(pattern='*_variable4_scenario4.nc',
...     engine='path')
>>> list(filtered_list2)
[u'project1_variable4_scenario4.nc', u'project2_variable4_scenario4.nc',
u'project3_variable4_scenario4.nc', u'project4_variable4_scenario4.nc',
u'project5_variable4_scenario4.nc']
```

Example 8

Displayed example 8 code

```
>>> filtered_list3 = list(api.filter(pattern='variable2', engine='str'))
>>> len(filtered_list3)
25
>>> filtered_list3[:4]
[u'project1_variable2_scenario1.nc', u'project1_variable2_scenario2.nc',
u'project1_variable2_scenario3.nc', u'project1_variable2_scenario4.nc']
```

Example 9

Displayed example 9 code

```
>>> archives_search = list(api.search())
>>> archives_filter = list(api.filter())
>>> len(archives_search)
125
>>> len(archives_filter)
125
```

Example 10

Displayed example 10 code

```
>>> tagged_search = list(api.search('team3'))
>>> len(tagged_search)
41
>>> tagged_search[:4]
[u'project1_variable1_scenario2.nc', u'project1_variable2_scenario1.nc',
u'project1_variable2_scenario3.nc', u'project1_variable3_scenario2.nc']
```

Example 11

Displayed example 11 code

```
>>> tags = []
>>> for arch in tagged_search[:4]:
...     tags.append(api.manager.get_tags(arch)[0])
>>> tags
[u'team3', u'team3', u'team3', u'team3']
```

Example 12

Displayed example 12 code

```
>>> tagged_search_team1 = list(api.search('team1'))
>>> len(tagged_search_team1)
42
>>> tagged_search_team1[:4]
[u'project1_variable1_scenario1.nc', u'project1_variable1_scenario4.nc',
u'project1_variable2_scenario2.nc', u'project1_variable2_scenario5.nc']
```

Example 13

Displayed example 13 code

```
>>> tags = []
>>> for arch in tagged_search_team1[:4]:
...     tags.append(api.manager.get_tags(arch)[0])
>>> tags
[u'team1', u'team1', u'team1', u'team1']
```

Teardown

```
>>> api.manager.delete_table('DataFiles')
```

CHAPTER 13

Indices and tables

- genindex
- modindex

CHAPTER 14

License

DataFS is available under the open source MIT license.

CHAPTER 15

History

DataFS was developed for use by the Climate Impact Lab. Check us out on github.

Python Module Index

d

Index

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W