opendap-protocol Documentation

Release 1.1.1.dev3+g5d8b126

Philipp Falke

Sep 27, 2021

Contents:

A pure Python implementation of the OPeNDAP server protocol.

This module allows you to serve arbitrary data structures through the web framework of your choice as OPeNDAP data objects. It implements just the bare minimum of the DAP 2.0 protocol: DDS, DAS, and DODS responses and slicing. Array data needs to be supplied as *numpy.ndarray*.

CHAPTER 1

Installation

1.1 Stable release

To install opendap-protocol, run this command in your terminal:

```
$ pip install opendap-protocol
```

This is the preferred method to install opendap-protocol, 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.

1.2 From source

The sources for opendap-protocol can be downloaded from the GitHub repo.

You can either clone the public repository:

```
$ git clone git@github.com:MeteoSwiss/opendap-protocol.git
```

Or download the ZIP-file:

```
$ curl  -OL https://github.com/MeteoSwiss/opendap-protocol/archive/master.zip
```

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

```
$ python setup.py install
```

CHAPTER 2

Usage

For more information on the DAP 2.0 protocol, visit the specification page.

2.1 Create a DDS, DAS and DODS responses from arbitrary *numpy* arrays

Basic imports:

```
import numpy as np
import opendap_protocol as dap
```

Define dimensions:

```
x = dap.Array(name='x', data=np.array([0, 1, 2]), dtype=dap.Int16)
y = dap.Array(name='y', data=np.array([10, 11, 12]), dtype=dap.Int16)
z = dap.Array(name='z', data=np.array([20, 21, 22]), dtype=dap.Int16)
```

Define an array holding our data:

```
data_array = dap.Grid(name='data',
                      data=np.random.rand(3, 3, 3),
                      dtype=dap.Float64,
                      dimensions=[x, y, z])
```

Define attributes:

```
attrs = [
    dap.Attribute(name='units', value='m s-1', dtype=dap.String),
    dap.Attribute(name='version', value=2, dtype=dap.Int16),
]
```

and attach them to the data array:

```
data_array.append(*attrs)
```

Glue it all together by creating the *Dataset* and appending to it:

```
dataset = dap.Dataset(name='Example')
dataset.append(x, y, z, data_array)
```

Each of the DAP 2.0 responses returns a *generator iterator*. Such a generator has generally a low memory footprint, since the serialized data set does not need to be held in memory at once. Rather, serialization takes place as a consumer iterates through the data set.

For testing purposes, each response can be printed as string:

```
print(''.join(dataset.dds()))

# Dataset {
# Int16 x[x = 3];
# Int16 y[y = 3];
# Int16 z[z = 3];
# Grid {
#   Array:
#     Float64 data[x = 3][y = 3][z = 3];
#   Maps:
#     Int16 x[x = 3];
#     Int16 y[y = 3];
#     Int16 z[z = 3];
#   } data;
# } Example;

print(''.join(dataset.das()))

# Attributes {
#     x {
#     }
#     y {
#     }
#     z {
#     }
#     data {
#         String units "m s-1";
#         Int16 version 2;
#     }
# }

print(b''.join(dataset.dods()))

# See for yourself ;-)
```

2.2 Serving data through a web service using Flask

Note: We assume, that the `dataset` created above is still available as a variable.

Basic setup:

```
import urllib
from flask import Flask, Response, request

app = Flask(__name__)
```

Define the web service endpoints needed by the DAP protocol:

```
@app.route('/dataset.dds', methods=['GET'])
def dds_response():
    # Retrieve constraints from the request to handle slicing, etc.
    constraint = urllib.parse.urlsplit(request.url)[3]
    return Response(
        dataset.dds(constraint=constraint),
        mimetype='text/plain')

@app.route('/dataset.das', methods=['GET'])
def das_response():
    constraint = urllib.parse.urlsplit(request.url)[3]
    return Response(
        dataset.das(constraint=constraint),
        mimetype='text/plain')

@app.route('/dataset.dods', methods=['GET'])
def dods_response():
    constraint = urllib.parse.urlsplit(request.url)[3]
    return Response(
        dataset.dods(constraint=constraint),
        mimetype='application/octet-stream')

app.run(debug=True)
```

Data can then be loaded from any Python terminal using `xarray` or `netCDF4`.

Note: Please be aware, that for opening a dataset the suffix (`.dds`, `.das` or `.dods`) needs to be omitted. The netCDF library figures out on its own which endpoint it has to call in what order.

`xarray`:

```
import xarray as xr

data = import xr.open_dataset('http://localhost:5000/dataset')
data.load()

# <xarray.Dataset>
# Dimensions:  (x: 3, y: 3, z: 3)
# Coordinates:
#   * x        (x) int16 0 1 2
#   * y        (y) int16 10 11 12
#   * z        (z) int16 20 21 22
# Data variables:
#     data     (x, y, z) float64 0.7793 0.3464 0.1331 ... 0.2244 0.4277 0.1545
```

`netCDF4`:

```
import netCDF4 as nc
```

(continues on next page)

(continued from previous page)

```
data = nc.Dataset('http://localhost:5000/dataset')
data

# <class 'netCDF4._netCDF4.Dataset'>
# root group (NETCDF3_CLASSIC data model, file format DAP2):
#     dimensions(sizes): x(3), y(3), z(3)
#     variables(dimensions): int16 x(x), int16 y(y), int16 z(z), float64 data(x,y,z)
#     groups:
```

CHAPTER 3

API Documentation

A pure Python implementation of the OPeNDAP server protocol.

This module allows you to serve arbitrary data structures through the web framework of your choice as OPeNDAP data objects. It implements just the bare minimum of the DAP 2.0 protocol: DDS, DAS, and DODS responses and slicing. Array data needs to be supplied as `numpy.ndarray`.

The classes defined here allow the user to construct a data model in a flexible way, by describing the data hierarchy using data types defined by DAP.

This library only implements the server side encoding. It is tested to serve clients using the netCDF4 library. PyDAP client libraries are not supported.

class opendap_protocol.protocol.**Array**(*name=''*, *parent=None*, **args*, ***kwargs*)

Bases: *opendap_protocol.protocol.DAPDataObject*

dds(*constraint=''*, *slicing=None*)

class opendap_protocol.protocol.**Attribute**(*value=None*, *name=None*, *dtype=None*)

Bases: *opendap_protocol.protocol.DAPObject*

das(*constraint=''*)

dds(**args*, ***kwargs*)

class opendap_protocol.protocol.**Byte**(*value=None*, *name=None*, *parent=None*)

Bases: *opendap_protocol.protocol.DAPAtom*

dtype

alias of `numpy.uint8`

str = 'B'

class opendap_protocol.protocol.**Config**(*DASK_ENCODE_CHUNK_SIZE: int = 20000000.0*)

Bases: `object`

DASK_ENCODE_CHUNK_SIZE = 20000000.0

class opendap_protocol.protocol.**DAPAtom**(*value=None*, *name=None*, *parent=None*)
Bases: *opendap_protocol.protocol.DAPObject*

A class for handling DAP atomic variables.

classmethod byteorder()

das(*constraint=''*)

dds(*constraint=''*)

dods_data(*constraint=''*)

str = None

classmethod subclasses()
Return a list of subclasses.

classmethod type_from_np(*nptype*)
Return the appropriate DAP type for a given numpy dtype.

Parameters **nptype** – A `numpy.dtpye` object

Returns A subclass of *DAPAtom*

class opendap_protocol.protocol.**DAPDataObject**(*name=''*, *parent=None*, **args*, ***kwargs*)
Bases: *opendap_protocol.protocol.DAPObject*

A generic class for typed non-atomic objects holding actual data (i.e. Array and Grid).

dods_data(*constraint=''*)

exception opendap_protocol.protocol.**DAPError**
Bases: `Exception`

class opendap_protocol.protocol.**DAPObject**(*name=''*, *parent=None*, **args*, ***kwargs*)
Bases: `object`

A generic DAP object class.

append(**obj*)

das(*constraint=''*)

dashead()

dastail()

data_path

dds(*constraint=''*)

ddshead()

ddstail()

dods(*constraint=''*)

dods_data(*constraint=''*)

indent

class opendap_protocol.protocol.**Dataset**(*name=''*, *parent=None*, **args*, ***kwargs*)
Bases: *opendap_protocol.protocol.Structure*

Class representing a DAP dataset.

dods_data(*constraint=''*)

class opendap_protocol.protocol.**Float32**(*value=None*, *name=None*, *parent=None*)
Bases: *opendap_protocol.protocol.DAPAtom*

dtype
alias of numpy.float32

str = '>f4'

class opendap_protocol.protocol.**Float64**(*value=None*, *name=None*, *parent=None*)
Bases: *opendap_protocol.protocol.DAPAtom*

dtype
alias of numpy.float64

str = '>f8'

class opendap_protocol.protocol.**Grid**(*name=''*, *parent=None*, **args*, ***kwargs*)
Bases: *opendap_protocol.protocol.DAPDataObject*

dds(*constraint=''*)

class opendap_protocol.protocol.**Int16**(*value=None*, *name=None*, *parent=None*)
Bases: *opendap_protocol.protocol.DAPAtom*

dtype
alias of numpy.int16

str = '>i4'

class opendap_protocol.protocol.**Int32**(*value=None*, *name=None*, *parent=None*)
Bases: *opendap_protocol.protocol.DAPAtom*

dtype
alias of numpy.int32

str = '>i4'

class opendap_protocol.protocol.**Sequence**(**args*, ***kwargs*)
Bases: *opendap_protocol.protocol.DAPObject*

Class representing a DAP sequence.

add_schema(*schema*)

append(**item*)

das(*constraint=''*)

dds(*constraint=''*)

dods_data(*constraint=''*)

end_of_seq = b'\xa5\x00\x00\x00'

start_of_inst = b'Z\x00\x00\x00'

class opendap_protocol.protocol.**SequenceInstance**(*name=''*, *parent=None*, **args*, ***kwargs*)
Bases: *opendap_protocol.protocol.DAPObject*

Class representing a data item that will be added to a sequence.

data_path

dods_data(*constraint=''*)

validates(*schema*)
Validate the sequence instance against a sequence schema

Parameters **schema** – A *SequenceSchema* instance.

class opendap_protocol.protocol.**SequenceSchema**(*name=''*, *parent=None*, **args*, ***kwargs*)

Bases: *opendap_protocol.protocol.DAPObject*

Class holding a schema against which SequenceItems are validated.

class opendap_protocol.protocol.**String**(*value=None*, *name=None*, *parent=None*)

Bases: *opendap_protocol.protocol.DAPAtom*

dods_data(*constraint=''*)

dtype

alias of `numpy.str_`

str = 'S'

class opendap_protocol.protocol.**Structure**(*name=''*, *parent=None*, **args*, ***kwargs*)

Bases: *opendap_protocol.protocol.DAPObject*

Class representing a DAP structure.

class opendap_protocol.protocol.**UInt16**(*value=None*, *name=None*, *parent=None*)

Bases: *opendap_protocol.protocol.DAPAtom*

dtype

alias of `numpy.uint16`

str = '>u4'

class opendap_protocol.protocol.**UInt32**(*value=None*, *name=None*, *parent=None*)

Bases: *opendap_protocol.protocol.DAPAtom*

dtype

alias of `numpy.uint32`

str = '>u4'

class opendap_protocol.protocol.**URL**(*value=None*, *name=None*, *parent=None*)

Bases: *opendap_protocol.protocol.String*

dtype

alias of `numpy.str_`

str = 'S'

opendap_protocol.protocol.**dods_encode**(*data*, *dtype*)

This is the fast XDR conversion. A 100x100 array takes around 40 micro- seconds. This is a speedup of factor 100.

opendap_protocol.protocol.**meets_constraint**(*constraint_expr*, *data_path*)

Parse the constraint expression and check if data_path meets the criteria.

Parameters

- **constraint_expr** – (string) A DAP constraint string
- **data_path** – (string) Path of a DAP object within the dataset

Returns a boolean

opendap_protocol.protocol.**parse_slice**(*token*)

Parse a single slice string

Parameters **token** – A string containing a number [3], a range [3:7] or a colon [:]

Returns An integer for simple numbers, or a slice object

opendap_protocol.protocol.**parse_slice_constraint**(*constraint*)
Parses the slicing part of a constraint expression.

Parameters **constraint** – A complete constraint string as received through DAP request.

Returns A tuple of slices that can be used for accessing a subdomain of a dataset.

opendap_protocol.protocol.**set_dask_encoding_chunk_size**(*chunk_size: int*)
Set the maximum chunk size used to encode ‘‘dask.Array‘‘s to XDR.

Parameters **chunk_size** – (int) Encoding chunk size in Bytes

Returns None

CHAPTER 4

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:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/MeteoSwiss/opendap-protocol/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.

4.1.2 Fix Bugs

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

4.1.3 Implement Features

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

4.1.4 Write Documentation

We 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.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/MeteoSwiss/opendap-protocol/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 :)

4.2 Get Started!

Ready to contribute? Here's how to set up *opendap_protocol* for local development.

1. Clone *opendap_protocol* repo from GitHub.

   $ git clone git@github.com:MeteoSwiss/opendap-protocol.git

2. Create a virtualenv and install dependencies:

```
$ cd opendap_protocol
$ virtualenv venv
$ source venv/bin/activate
$ python setup.py develop
```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes, check that your changes pass pylint and the tests, including testing other Python versions with tox:

```
$ python setup.py test
$ pylint opendap_protocol
$ tox
```

5. Commit your changes and push your branch to GitHub:

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

6. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6 (or greater). Check https://github.com/MeteoSwiss/opendap-protocol/pulls and make sure that the tests pass for all supported Python versions.

CHAPTER 5

Credits

5.1 Development Lead

- Philipp Falke <philipp.falke@meteoswiss.ch>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.1.0 (2020-03-27)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

E

F

G

I

M

O

P

S

T

U

V