Pydap Documentation

Release 3.2

Roberto De Almeida

May 24, 2017

Contents

Pydap is a pure Python library implementing the Data Access Protocol, also known as **DODS** or **OPeNDAP**. You can use Pydap as a client to access hundreds of scientific datasets in a transparent and efficient way through the internet; or as a server to easily distribute your data from a variety of formats.

CHAPTER 1

Quickstart

You can install the latest version (3.2) using pip. After installing pip you can install Pydap with this command:

```
$ pip install Pydap
```

This will install Pydap together with all the required dependencies. You can now open any remotely served dataset, and Pydap will download the accessed data on-the-fly as needed:

```
>>> from pydap.client import open_url
>>> dataset = open_url('http://test.opendap.org/dap/data/nc/coads_climatology.nc')
>>> var = dataset['SST']
>>> var.shape
(12, 90, 180)
>>> var.dtype
dtype('>f4')
>>> data = var[0,10:14,10:14]  # this will download data from the server
>>> data
<GridType with array 'SST' and maps 'TIME', 'COADSY', 'COADSX'>
>>> print(data.data)
[array([[[ -1.26285708e+00,  -9.99999979e+33,  -9.99999979e+33,
           -9.99999979e+33],
         [ -7.69166648e-01,  -7.79999971e-01,  -6.75454497e-01,
           -5.95714271e-01],
         [  1.28333330e-01,  -5.00000156e-02,  -6.36363626e-02,
           -1.41666666e-01],
         [  6.38000011e-01,   8.95384610e-01,   7.21666634e-01,
            8.10000002e-01]]], dtype=float32), array([ 366.]), array([-69., -67., -65.,
↪ -63.]), array([ 41.,  43.,  45.,  47.])]
```

For more information, please check the documentation on using Pydap as a client. Pydap also comes with a simple server, implemented as a WSGI application. To use it, you first need to install Pydap with the server extras dependencies. If you want to serve netCDF files, install Pydap with the `handlers.netcdf` extra:

```
$ pip install Pydap[server,handlers.netcdf]
```

More handlers for different formats are available, if necessary. To run a simple standalone server just issue the command:

```
$ pydap --data ./myserver/data/ --port 8001
```

This will start a standalone server running on http://localhost:8001/, serving netCDF files from `./myserver/data/`, similar to the test server at http://test.pydap.org/. Since the server uses the WSGI standard, it can easily be run behind Apache. The server documentation has more information on how to better deploy Pydap.

CHAPTER 2

Help

If you need any help with Pydap, please feel free to send an email to the mailing list.

CHAPTER 3

Documentation

Using the client

Pydap can be used as a client to inspect and retrieve data from any of the hundreds of scientific datasets available on the internet on OPeNDAP servers. This way, it's possible to instrospect and manipulate a dataset as if it were stored locally, with data being downloaded on-the-fly as necessary.

Accessing gridded data

Let's start accessing gridded data, i.e., data that is stored as a regular multidimensional array. Here's a simple example where we access the COADS climatology from the official OPeNDAP server:

```
>>> from pydap.client import open_url
>>> dataset = open_url('http://test.opendap.org/dap/data/nc/coads_climatology.nc')
>>> type(dataset)
<class 'pydap.model.DatasetType'>
```

Here we use the `pydap.client.open_url` function to open an URL specifying the location of the dataset; this URL should be stripped of the extensions commonly used for OPeNDAP datasets, like *.dds* or *.das*. When we access the remote dataset the function returns a `DatasetType` object, which is a *Structure* – a fancy dictionary that stores other variables. We can check the names of the store variables like we would do with a Python dictionary:

```
>>> list(dataset.keys())
['COADSX', 'COADSY', 'TIME', 'SST', 'AIRT', 'UWND', 'VWND']
```

Let's work with the `SST` variable; we can reference it using the usual dictionary syntax of `dataset['SST']`, or using the "lazy" syntax `dataset.SST`:

```
>>> sst = dataset['SST']  # or dataset.SST
>>> type(sst)
<class 'pydap.model.GridType'>
```

Note that the variable is of type `GridType`, a multidimensional array with specific axes defining each of its dimensions:

```
>>> sst.dimensions
('TIME', 'COADSY', 'COADSX')
>>> sst.maps
OrderedDict([('TIME', <BaseType with data BaseProxy('http://test.opendap.org/dap/data/
↪nc/coads_climatology.nc', 'SST.TIME', dtype('>f8'), (12,), (slice(None, None, None),
↪))>), ('COADSY', <BaseType with data BaseProxy('http://test.opendap.org/dap/data/nc/
↪coads_climatology.nc', 'SST.COADSY', dtype('>f8'), (90,), (slice(None, None, None),
↪))>), ('COADSX', <BaseType with data BaseProxy('http://test.opendap.org/dap/data/nc/
↪coads_climatology.nc', 'SST.COADSX', dtype('>f8'), (180,), (slice(None, None, None),
↪))>)])
```

Each map is also, in turn, a variable that can be accessed using the same syntax used for Structures:

```
>>> sst.TIME
<BaseType with data BaseProxy('http://test.opendap.org/dap/data/nc/coads_climatology.
↪nc', 'SST.TIME', dtype('>f8'), (12,), (slice(None, None, None),))>
```

The axes are all of type `BaseType`. This is the OPeNDAP equivalent of a multidimensional array, with a specific shape and type. Even though no data have been downloaded up to this point, we can introspect these attributes from the axes or from the Grid itself:

```
>>> sst.shape
(12, 90, 180)
>>> sst.dtype
dtype('>f4')
>>> sst.TIME.shape
(12,)
>>> sst.TIME.dtype
dtype('>f8')
```

We can also introspect the variable attributes; they are stored in an attribute appropriately called `attributes`, and they can also be accessed with a "lazy" syntax:

```
>>> import pprint
>>> pprint.pprint(sst.attributes)
{'_FillValue': -9.99999979e+33,
 'history': 'From coads_climatology',
 'long_name': 'SEA SURFACE TEMPERATURE',
 'missing_value': -9.99999979e+33,
 'units': 'Deg C'}
>>> sst.units
'Deg C'
```

Finally, we can also download some data. To download data we simply access it like we would access a Numpy array, and the data for the corresponding subset will be dowloaded on the fly from the server:

```
>>> sst.shape
(12, 90, 180)
>>> grid = sst[0,10:14,10:14]  # this will download data from the server
>>> grid
<GridType with array 'SST' and maps 'TIME', 'COADSY', 'COADSX'>
```

The data itself can be accessed in the `array` attribute of the Grid, and also on the individual axes:

```
>>> grid.array[:]
<BaseType with data array([[[ -1.26285708e+00,  -9.99999979e+33,  -9.99999979e+33,
          -9.99999979e+33],
        [ -7.69166648e-01,  -7.79999971e-01,  -6.75454497e-01,
          -5.95714271e-01],
        [  1.28333330e-01,  -5.00000156e-02,  -6.36363626e-02,
          -1.41666666e-01],
        [  6.38000011e-01,   8.95384610e-01,   7.21666634e-01,
           8.10000002e-01]]], dtype=float32)>
>>> print(grid.array[:].data)
[[[ -1.26285708e+00  -9.99999979e+33  -9.99999979e+33  -9.99999979e+33]
  [ -7.69166648e-01  -7.79999971e-01  -6.75454497e-01  -5.95714271e-01]
  [  1.28333330e-01  -5.00000156e-02  -6.36363626e-02  -1.41666666e-01]
  [  6.38000011e-01   8.95384610e-01   7.21666634e-01   8.10000002e-01]]]
>>> grid.COADSX[:]
<BaseType with data array([ 41.,  43.,  45.,  47.])>
>>> print(grid.COADSX[:].data)
[ 41.  43.  45.  47.]
```

Alternatively, we could have dowloaded the data directly, skipping the axes:

```
>>> print(sst.array[0,10:14,10:14].data)
[[[ -1.26285708e+00  -9.99999979e+33  -9.99999979e+33  -9.99999979e+33]
  [ -7.69166648e-01  -7.79999971e-01  -6.75454497e-01  -5.95714271e-01]
  [  1.28333330e-01  -5.00000156e-02  -6.36363626e-02  -1.41666666e-01]
  [  6.38000011e-01   8.95384610e-01   7.21666634e-01   8.10000002e-01]]]
```

Older Servers

Some servers using a very old OPeNDAP application might run of of memory when attempting to retrieve both the data and the coordinate axes of a variable. The work around is to simply disable the retrieval of coordinate axes by using the `output_grid` option to open url:

```
>>> from pydap.client import open_url
>>> dataset = open_url('http://test.opendap.org/dap/data/nc/coads_climatology.nc',␣
↪output_grid=False)
>>> grid = sst[0,10:14,10:14]  # this will download data from the server
>>> grid
<GridType with array 'SST' and maps 'TIME', 'COADSY', 'COADSX'>
```

Accessing sequential data

Now let's see an example of accessing sequential data. Sequential data consists of one or more records of related variables, such as a simultaneous measurements of temperature and wind velocity, for example. In this example we're going to access data from the Argo project, consisting of profiles made by autonomous buoys drifting on the ocean:

```
>>> from pydap.client import open_url
>>> dataset = open_url('http://dapper.pmel.noaa.gov/dapper/argo/argo_all.cdp')
```

This dataset is fairly complex, with several variables representing heterogeneous 4D data. The layout of the dataset follows the Dapper in-situ conventions, consisting of two nested sequences: the outer sequence contains, in this case, a latitude, longitude and time variable, while the inner sequence contains measurements along a z axis.

The first thing we'd like to do is limit our region; let's work with a small region in the Tropical Atlantic:

```
>>> type(dataset.location)
<class 'pydap.model.SequenceType'>
>>> dataset.location.keys()
['LATITUDE', 'JULD', 'LONGITUDE', '_id', 'profile', 'attributes', 'variable_attributes
↪']
>>> my_location = dataset.location[
...          (dataset.location.LATITUDE > -2) &
...          (dataset.location.LATITUDE < 2) &
...          (dataset.location.LONGITUDE > 320) &
...          (dataset.location.LONGITUDE < 330)]
```

Note that the variable `dataset.location` is of type `SequenceType` – also a Structure that holds other variables. Here we're limiting the sequence `dataset.location` to measurements between given latitude and longitude boundaries. Let's access the identification number of the first 10-or-so profiles:

```
>>> for i, id_ in enumerate(my_location['_id'].iterdata()):
...     print(id_)
...     if i == 10:
...         print('...')
...         break
1125393
835304
839894
875344
110975
864748
832685
887712
962673
881368
1127922
...
>>> len(my_location['_id'].iterdata())
623
```

Note that calculating the length of a sequence takes some time, since the client has to download all the data and do the calculation locally. This is why you should use `len(my_location['_id'])` instead of `len(my_location)`. Both should give the same result (unless the dataset changes between requests), but the former retrieves only data for the `_id` variable, while the later retrives data for all variables.

We can explicitly select just the first 5 profiles from our sequence:

```
>>> my_location = my_location[:5]
>>> len(my_location['_id'].iterdata())
5
```

And we can print the temperature profiles at each location. We're going to use the coards module to convert the time to a Python `datetime` object:

```
>>> from coards import from_udunits
>>> for position in my_location.iterdata():
...     date = from_udunits(position.JULD.data, position.JULD.units.replace('GMT',
↪'+0:00'))
...     print(position.LATITUDE.data, position.LONGITUDE.data, date)
...     print('=' * 40)
...     i = 0
...     for pressure, temperature in zip(position.profile.PRES, position.profile.
↪TEMP):
```

```
...         print(pressure, temperature)
...         if i == 10:
...             print('...')
...             break
...         i += 1
-1.01 320.019 2009-05-03 11:42:34+00:00
========================================
5.0 28.59
10.0 28.788
15.0 28.867
20.0 28.916
25.0 28.94
30.0 28.846
35.0 28.566
40.0 28.345
45.0 28.05
50.0 27.595
55.0 27.061
...
-0.675 320.027 2006-12-25 13:24:11+00:00
========================================
5.0 27.675
10.0 27.638
15.0 27.63
20.0 27.616
25.0 27.617
30.0 27.615
35.0 27.612
40.0 27.612
45.0 27.605
50.0 27.577
55.0 27.536
...
-0.303 320.078 2007-01-12 11:30:31.001000+00:00
========================================
5.0 27.727
10.0 27.722
15.0 27.734
20.0 27.739
25.0 27.736
30.0 27.718
35.0 27.694
40.0 27.697
45.0 27.698
50.0 27.699
55.0 27.703
...
-1.229 320.095 2007-04-22 13:03:35.002000+00:00
========================================
5.0 28.634
10.0 28.71
15.0 28.746
20.0 28.758
25.0 28.755
30.0 28.747
35.0 28.741
40.0 28.737
45.0 28.739
```

```
50.0 28.748
55.0 28.806
...
-1.82 320.131 2003-04-09 13:20:03+00:00
========================================
5.1 28.618
9.1 28.621
19.4 28.637
29.7 28.662
39.6 28.641
49.6 28.615
59.7 27.6
69.5 26.956
79.5 26.133
89.7 23.937
99.2 22.029
...
```

These profiles could be easily plotted using matplotlib:

```
>>> for position in my_location.iterdata():
...     plot(position.profile.TEMP, position.profile.PRES)
>>> show()
```

You can also access the deep variables directly. When you iterate over these variables the client will download the data as nested lists:

```
>>> for value in my_location.profile.PRES.iterdata():
...     print(value[:10])
[5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]
[5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]
[5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]
[5.0, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0]
[5.0999999, 9.1000004, 19.4, 29.700001, 39.599998, 49.599998, 59.700001, 69.5, 79.5,␣
↪89.699997]
```

Pydap 3.0 has been rewritten to make it easier to work with Dapper datasets like this one, and it should be intuitive[1] to work with these variables.

Authentication

To use Basic and Digest authentication, simply add your username and password to the dataset URL. Keep in mind that if the server only supports Basic authentication your credentials will be sent as plaintext, and could be sniffed on the network.

```
>>> from pydap.client import open_url
>>> dataset = open_url('http://username:password@server.example.com/path/to/dataset')
```

The Central Authentication Service (CAS) is a single sign-on protocol for the web, usually involving a web browser and cookies. Nevertheless it's possible to use Pydap with an OPeNDAP server behind a CAS. The function `install_cas_client` below replaces Pydap's default HTTP function with a new version able to submit authentication data to an HTML form and store credentials in cookies. (In this particular case, the server uses Javascript to redirect the browser to a new location, so the client has to parse the location from the Javascript code; other CAS would require a tweaked function.)

[1] But please check this quote.

To use it, just attach a web browsing `session` with authentication cookies:

```
>>> from pydap.client import open_url
>>> from pydap.cas.get_cookies import setup_session
>>> session = setup_session(authentication_url, username, password)
>>> dataset = open_url('http://server.example.com/path/to/dataset', session=session)
```

This method could work but each CAS is slightly different and might require a specifically designed `setup_session` instance. Two CAS are however explicitly supported by `pydap`:

URS NASA EARTHDATA

Authentication is done through a `username` and a `password`:

```
>>> from pydap.client import open_url
>>> from pydap.cas.urs import setup_session
>>> dataset_url = 'http://server.example.com/path/to/dataset'
>>> session = setup_session(username, password, check_url=dataset_url)
>>> dataset = open_url(dataset_url, session=session)
```

Earth System Grid Federation (ESGF)

Authentication is done through an `openid` and a `password`:

```
>>> from pydap.client import open_url
>>> from pydap.cas.esgf import setup_session
>>> dataset_url = 'http://server.example.com/path/to/dataset'
>>> session = setup_session(openid, password, check_url=dataset_url)
>>> dataset = open_url(dataset_url, session=session)
```

If your `openid` contains contains the string `ceda.ac.uk` authentication requires an additional `username` argument:

```
>>> from pydap.client import open_url
>>> from pydap.cas.esgf import setup_session
>>> session = setup_session(openid, password, check_url=dataset_url,␣
↪username=username)
>>> dataset = open_url(dataset_url, session=session)
```

Advanced features

When you open a remote dataset, the `DatasetType` object has a special attribute named `functions` that can be used to invoke any server-side functions. Here's an example of using the `geogrid` function from Hyrax:

```
>>> dataset = open_url('http://test.opendap.org/dap/data/nc/coads_climatology.nc')
>>> new_dataset = dataset.functions.geogrid(dataset.SST, 10, 20, -10, 60)
>>> new_dataset.SST.shape
(12, 12, 21)
>>> new_dataset.SST.COADSY[:]
[-11.  -9.  -7.  -5.  -3.  -1.   1.   3.   5.   7.   9.  11.]
>>> new_dataset.SST.COADSX[:]
[ 21.  23.  25.  27.  29.  31.  33.  35.  37.  39.  41.  43.  45.  47.  49.
  51.  53.  55.  57.  59.  61.]
```

Unfortunately, there's currently no standard mechanism to discover which functions the server support. The `function` attribute will accept any function name the user specifies, and will try to pass the call to the remote server.

You can pass any URL to the `open_url` function, together with any valid constraint expression. Here's an example of restricting values for the months of January, April, July and October:

```
>>> dataset = open_url('http://test.opendap.org/dap/data/nc/coads_climatology.nc?
↪SST[0:3:11][0:1:89][0:1:179]')
>>> dataset.SST.shape
(4, 90, 180)
```

This can be extremely useful for server side-processing; for example, we can create and access a new variable `A` in this dataset, equal to twice `SSH`:

```
>>> dataset = open_url('http://hycom.coaps.fsu.edu:8080/thredds/dodsC/las/dynamic/
↪data_A5CDC5CAF9D810618C39646350F727FF.jnl_expr_%7B%7D%7Blet%20A=SSH*2%7D?A')
>>> dataset.keys()
['A']
```

In this case, we're using the Ferret syntax `let A=SSH*2` to define the new variable, since the data is stored in an F-TDS server. Server-side processing is useful when you want to reduce the data before downloading it, to calculate a global average, for example.

The client module has a special function called `open_dods`, used to access raw data from a DODS response:

```
>>> from pydap.client import open_dods
>>> dataset = open_dods(
...     'http://test.opendap.org/dap/data/nc/coads_climatology.nc.dods?
↪SST[0:3:11][0:1:89][0:1:179]')
```

This function allows you to access raw data from any URL, including appending expressions to F-TDS and GDS servers or calling server-side functions directly. By default this method downloads the data directly, and skips metadata from the DAS response; if you want to investigate and introspect datasets you should set the `get_metadata` parameter to true:

```
>>> dataset = open_dods(
...     'http://test.opendap.org/dap/data/nc/coads_climatology.nc.dods?
↪SST[0:3:11][0:1:89][0:1:179]',
...      get_metadata=True)
>>> dataset.attributes['NC_GLOBAL']['history']
FERRET V4.30 (debug/no GUI) 15-Aug-96
```

You can specify a cache directory in the `pydap.lib.CACHE` global variable. If this value is different than `None`, the client will try (if the server headers don't prohibit) to cache the result, so repeated requests will be read from disk instead of the network:

```
>>> import pydap.lib
>>> pydap.lib.CACHE = "/tmp/pydap-cache/"
```

To specify a timeout for the client, just set the desired number of seconds using the `timeout` option to `open_url(...)` or `open_dods(...)`. For example, the following commands would timeout after 30 seconds without receiving a response from the server:

```
>>> dataset = open_url('http://test.opendap.org/dap/data/nc/coads_climatology.nc,␣
↪timeout=30)
>>> dataset = open_dods('http://test.opendap.org/dap/data/nc/coads_climatology.nc.
↪dods, timeout=30)
```

It's possible to configure Pydap to access the network through a proxy server. Here's an example for an HTTP proxy running on `localhost` listening on port 8000:

```
>>> import httplib2
>>> from pydap.util import socks
>>> import pydap.lib
>>> pydap.lib.PROXY = httplib2.ProxyInfo(
...         socks.PROXY_TYPE_HTTP, 'localhost', 8000)
```

This way, all further calls to `pydap.client.open_url` will be routed through the proxy server. You can also authenticate to the proxy:

```
>>> pydap.lib.PROXY = httplib2.ProxyInfo(
...         socks.PROXY_TYPE_HTTP, 'localhost', 8000,
...         proxy_user=USERNAME, proxy_pass=PASSWORD)
```

A user has reported that `httplib2` has problems authenticating against a NTLM proxy server. In this case, a simple solution is to change the `pydap.http.request` function to use `urllib2` instead of `httplib2`, monkeypatching the code like in the *CAS authentication example above*:

```
import urllib2
import logging

def install_urllib2_client():
    def new_request(url):
        log = logging.getLogger('pydap')
        log.INFO('Opening %s' % url)

        f = urllib2.urlopen(url.rstrip('?&'))
        headers = dict(f.info().items())
        body = f.read()
        return headers, body

    from pydap.util import http
    http.request = new_request
```

The function `install_urllib2_client` should then be called before doing any requests.

Running a server

Pydap comes with a lightweight and scalable OPeNDAP server, implemented as a WSGI application. Being a WSGI application, Pydap can run on a variety of servers, and frameworks including Apache, Nginx, IIS, uWSGI, Flask or as a standalone Python process. It can also be seamless combined with different middleware for authentication/authorization, GZip compression, and much more.

There is no one right way to run Pydap server; your application requirements and software stack will inform your deployment decisions. In this chapter we provide a few examples to try and get you on the right track.

In order to distribute your data first you need to install a proper handler, that will convert the data format to the Pydap data model.

Running standalone

If you just want to quickly test the Pydap server, you can run it as a standalone Python application using the server that comes with Python Paste and gunicorn. To run the server, first make sure that you have installed Pydap with the

server extras dependencies:

```
$ pip install Pydap[server]
```

and then just run the `pydap` script that pip installs into your bin directory:

```
$ pydap --data /path/to/my/data/files --port 8080
```

To change the default directory listing, the help page and the HTML form, you can point a switch to your template directory

```
$ pydap --data /path/to/my/data/files --templates /path/to/my/templates.
```

The HTML form template is fairly complex, since it contain some application logic and some Javascript code, so be careful to not break anything.

WSGI Application

Pydap follows the WSGI specification, and most web servers gateways simply require a WSGI callable and a small amount of boiler plate code. Pydap provides the `DapServer` class which is a WSGI callable located in the `pydap.wsgi.app` module. A simple WSGI application script file would be something like this:

```
from pydap.wsgi.app import DapServer
application = DapServer('/path/to/my/data/files')
```

Flask

The Flask framework simply requires a couple more function calls to spin up an application server. A simple server would looks something like this (your mileage may vary):

```
from flask import Flask
from pydap.wsgi.app import DapServer

application = DapServer('/path/to/my/data/files')
app = Flask(__name__)
app.wsgi_app = application
app.run('0.0.0.0', 8000)
```

Apache

For a robust deployment you can run Pydap with Apache, using mod_wsgi. After installing mod_wsgi, create a sandbox in a directory *outside* your DocumentRoot, say `/var/www/pydap/`, using a virtual environment:

```
$ mkdir /var/www/pydap
$ python3 -m venv /var/www/pydap/env
```

If you want the sandbox to use your system installed packages (like Numpy, e.g.) you can use the `--system-site-packages` flag:

```
$ python3 -m venv --system-site-packages /var/www/pydap/env
```

Now let's activate the sandbox and install Pydap – this way the module and its dependencies can be isolated from the system libraries:

```
$ source /var/www/pydap/env/bin/activate.sh
(env)$ pip install Pydap
```

Create a WSGI script file somewhere convenient (e.g. /var/www/pydap/server/apache/pydap.wsgi) that reads something like this:

```
import site
# force mod_wsgi to use the Python modules from the sandbox
site.addsitedir('/var/www/pydap/env/lib/pythonX.Y/site-packages')

from pydap.wsgi.app import DapServer
application = DapServer('/path/to/my/data/files')
```

Now create an entry in your Apache configuration pointing to the `pydap.wsgi` file you just edited. To mount the server on the URL `/pydap`, for example, you should configure it like this:

```
WSGIScriptAlias /pydap /var/www/pydap/server/apache/pydap.wsgi

<Directory /var/www/pydap/server/apache>
    Order allow,deny
    Allow from all
</Directory>
```

This is the file I use for the test.pydap.org virtualhost:

```
<VirtualHost *:80>
    ServerAdmin rob@pydap.org
    ServerName test.pydap.org

    DocumentRoot /var/www/sites/test.pydap.org/server/data

    <Directory /var/www/sites/test.pydap.org/server/data>
        Order allow,deny
        Allow from all
    </Directory>

    WSGIScriptAlias / /var/www/sites/test.pydap.org/server/apache/pydap.wsgi

    <Directory /var/www/sites/test.pydap.org/server/apache>
        Order allow,deny
        Allow from all
    </Directory>

    ErrorLog /var/log/apache2/test.pydap.org.error.log

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog /var/log/apache2/test.pydap.org.access.log combined
    ServerSignature On
</VirtualHost>
```

You can find more information on the mod_wsgi configuration guide. Just remember that Pydap is a WSGI application like any other else, so any information on WSGI applications applies to it as well.

uWSGI

uWSGI is a "fast, self-healing and developer/sysadmin-friendly application container server coded in pure C" that can run Pydap. This is the recommended way to run Pydap if you don't have to integrate it with other web applications. Simply install uWSGI, follow the instructions in the last section in order to create a virtualenv and Pydap installation:

```
$ mkdir /var/www/pydap
$ python virtualenv.py /var/www/pydap/env
$ source /var/www/pydap/env/bin/activate.sh
(env)$ pip install Pydap uWSGI
(env)$ cd /var/www/pydap
```

Create a WSGI application file myapp.wsgi *as above*

Now create a file in `/etc/init/pydap.conf` with the content:

```
description "uWSGI server for Pydap"

start on runlevel [2345]
stop on runlevel [!2345]

respawn

exec /var/www/pydap/env/bin/uwsgi \
    --http-socket 0.0.0.0:80 \
    -H /var/www/pydap/env \
    --master --processes 4 \
    --wsgi-file /var/www/pydap/myapp.wsgi
```

In order to make it run automatically during boot on Linux you can type:

```
$ sudo initctl reload-configuration
```

Docker

Users have reported success deploying Pydap with a docker image built with nginx + uWSGI + Flask (based on https://hub.docker.com/r/tiangolo/uwsgi-nginx-flask/. A full configuration is somewhat beyond the scope of this documentation (since it will depend on your requirements and your software stack), but it is certainly possible.

Handlers

Handlers are special Python modules that convert between a given data format and the data model used by Pydap (defined in the `pydap.model` module). They are necessary in order to Pydap be able to actually serve a dataset. There are handlers for NetCDF, HDF 4 & 5, Matlab, relational databases, Grib 1 & 2, CSV, Seabird CTD files, and a few more.

Installing data handlers

NetCDF

NetCDF is a format commonly used in oceanography, meteorology and climate science to store data in a machine-independent format. You can install the NetCDF handler using pip:

```
$ pip install Pydap[handlers.netcdf]
```

This will take care of the necessary dependencies. You don't even need to have to NetCDF libraries installed, since the handler will use a pure Python NetCDF library from scipy.io.netcdf.

The NetCDF handler uses a buffered reader that access the data in contiguous blocks from disk, avoiding reading everything into memory at once. You can configure the size of the buffer by specifying a key in the `server.ini` file:

```
[app:main]
use = egg:pydap#server
root = %(here)s/data
templates = %(here)s/templates
x-wsgiorg.throw_errors = 0
pydap.handlers.netcdf.buf_size = 10000
```

In this example, the handler will read 10 thousand values at a time, converting the data and sending to the client before reading more blocks.

NCA

The `pydap.handlers.nca` is a simple handler for NetCDF aggregation (hence the name). The configuration is extremely simple. As an example, to aggregate model output in different files (say, `output1.nc`, `output2.nc`, etc.) along a new axis "ensemble" just create an INI file with the extension `.nca`:

```
; output.nca
[dataset]
match = /path/to/output*.nc
axis = ensemble
; below optional metadata:
history = Test for NetCDF aggregator

[ensemble]
values = 1, 2, ...
long_name = Ensemble members
```

This will assign the values 1, 2, and so on to each ensemble member. The new, aggregated dataset, will be accessed at the location of the INI file:

```
http://server.example.com/output.nca
```

Another example: suppose we have monthly data in files `data01.nc`, `data02.nc`, ..., `data12.nc`, and we want to aggregate along the time axis:

```
[dataset]
match = /path/to/data*.nc
axis = TIME  # existing axis
```

The handler only works with NetCDF files for now, but in the future it should be changed to work with any other Pydap-supported data format. As all handlers, it can be installed using pip:

```
$ pip install pydap.handlers.nca
```

CDMS

This is a handler that uses the `cdms2.open` function from CDAT/CdatLite to read files in any of the self-describing formats netCDF, HDF, GrADS/GRIB (GRIB with a GrADS control file), or PCMDI DRS. It can be installed using pip:

```
$ pip install pydap.handlers.cdms
```

The handler will automatically install `CdatLite`, which requires the NetCDF libraries to be installed on the system.

SQL

The SQL handler reads data from a relation database, as the name suggests. It works by reading a file with the extension `.sql`, defining the connection to the database and other metadata using either YAML or INI syntax. Below is an example that reads data from a SQLite database:

```
# please read http://groups.google.com/group/pydap/browse_thread/thread/
↪c7f5c569d661f7f9 before
# setting your password on the DSN
database:
    dsn: 'sqlite://simple.db'
    table: test

dataset:
    NC_GLOBAL:
        history: Created by the Pydap SQL handler
        dataType: Station
        Conventions: GrADS

    contact: roberto@dealmeida.net
    name: test_dataset
    owner: Roberto De Almeida
    version: 1.0
    last_modified: !Query 'SELECT time FROM test ORDER BY time DESC LIMIT 1;'

sequence:
    name: simple
    items: !Query 'SELECT COUNT(id) FROM test'

_id:
    col: id
    long_name: sequence id
    missing_value: -9999

lon:
    col: lon
    axis: X
    grads_dim: x
    long_name: longitude
    units: degrees_east
    missing_value: -9999
    type: Float32
    global_range: [-180, 180]
    valid_range: !Query 'SELECT min(lon), max(lon) FROM test'

lat:
    col: lat
```

```
    axis: Y
    grads_dim: y
    long_name: latitude
    units: degrees_north
    missing_value: -9999
    type: Float32
    global_range: [-90, 90]
    valid_range: !Query 'SELECT min(lat), max(lat) FROM test'

time:
    col: time
    axis: T
    grads_dim: t
    long_name: time
    missing_value: -9999
    type: String

depth:
    axis: Z
    col: depth
    long_name: depth
    missing_value: -9999
    type: Float32
    units: m

temp:
    col: temp
    long_name: temperature
    missing_value: -9999
    type: Float32
    units: degc
```

The handler works with SQLite, MySQL, PostgreSQL, Oracle, MSSQL and ODBC databases. To install the handler use pip; you should also install the dependencies according to the database used:

```
$ pip install pydap.handlers.sql
$ pip install "pydap.handlers.sql[oracle]"
$ pip install "pydap.handlers.sql[postgresql]"
$ pip install "pydap.handlers.sql[mysql]"
$ pip install "pydap.handlers.sql[mssql]"
```

Proxy

This is a simple handler intended to serve remote datasets locally. For example, suppose you want to serve this dataset on your Pydap server. The URL of the dataset is:

```
http://test.opendap.org:8080/dods/dts/D1
```

So we create an INI file called, say, `D1.url`:

```
[dataset]
url = http://test.opendap.org:8080/dods/dts/D1
pass = dds, das, dods
```

The file specifies that requests for the DDS, DAS and DODS responses should be passed directly to the server (so that the data is downloaded directly from the remote server). Other requests, like for the HTML form or a WMS image are

built by Pydap; in this case Pydap acts as an Opendap client, connecting to the remote server and downloading data to fulfill the request.

CSV

This is a handler for files with comma separated values. The first column should contain the variable names, and subsequent lines the data. Metadata is not supported. The handler is used mostly as a reference for building handlers for sequential data. You can install it with:

```
$ pip install pydap.handlers.csv
```

HDF5

A handler for HDF5 files, based on h5py. In order to install it:

```
$ pip install pydap.handlers.hdf5
```

SQLite

This is a handler very similar to the SQL handler. The major difference is that data and metadata are all contained in a single `.db` SQLite file. Metadata is stored as JSON in a table called `attributes`, while data goes into a table `data`.

The handler was created as a way to move sequential data from one server to another. It comes with a script called `freeze` which will take an Opendap dataset with sequential data and create a `.db` file that can be served using this handler. For example:

```
$ freeze http://opendap.ccst.inpe.br/Observations/PIRATA/pirata_stations.sql
```

This will creata file called `pirata_stations.db` that can be served using the SQLite handler.

Responses

Like handlers, responses are special Python modules that convert between the Pydap data model and an external representation. For instance, to access a given dataset an Opendap client request two diferent representations of the dataset: a *Dataset Attribute Structure* (DAS) response, describing the attributes of the dataset, and a *Dataset Descriptor Structure* (DDS), describing its structure (shape, type, hierarchy). These responses are returned by appending the extension `.das` and `.dds` to the dataset URL, respectively.

Other common responses include the ASCII (`.asc` or `.ascii`) response, which returns an ASCII representation of the data; and an HTML form for data request using the browser, at the `.html` extension. And perhaps the most important response is the `.dods` response, which actually carries the data in binary format, and is used when clients request data from the server. All these responses are standard and come with Pydap.

There are other extension responses available for Pydap; these are not defined in the DAP specification, but improve the user experience by allowing data to be accessed in different formats.

Installing additional responses

Web Map Service

This response enables Pydap to act like a Web Map Service 1.1.1 server, returning images (maps) of the available data. These maps can be visualized in any WMS client like Openlayers or Google Earth.

You can install the WMS response using pip:

```
$ pip install pydap.responses.wms
```

This will take care of the necessary dependencies, which include Matplotlib and Pydap itself. Once the response is installed you can introspect the available layers at the URL:

```
http://server.example.com/dataset.wms?REQUEST=GetCapabilities
```

The response will create valid layers from any COARDS compliant datasets, ie, grids with properly defined latitude and longitude axes. If the data is not two-dimensional it will be averaged along each axis except for the last two, so the map represents a time and/or level average of the data. Keep in mind that Opendap constraint expressions apply before the map is generated, so it's possible to create a map of a specific level (or time) by constraining the variable on the URL:

```
http://server.example.com/dataset.wms?var3d[0]&REQUEST=GetCapabilities
```

You can specify the default colormap and the DPI resolution in the `server.ini` file:

```
[app:main]
use = egg:pydap#server
root = %(here)s/data
templates = %(here)s/templates
x-wsgiorg.throw_errors = 0
pydap.responses.wms.dpi = 80
pydap.responses.wms.cmap = jet
```

Google Earth

This response converts a Pydap dataset to a KML representation, allowing the data to be visualized using Google Earth as a client. Simply install it with pip:

```
$ pip install pydap.responses.kml
```

And open a URL by appending the `.kml` extension to the dataset, say:

```
http://server.example.com/dataset.kml
```

For now, the KML response will only return datasets that have a valid WMS representation. These datasets can be overlayed as layers on top of Google Earth, and are presented with a nice colorbar. In the future, Dapper-compliant datasets should be supported too.

NetCDF

This response allows data to be downloaded as a NetCDF file; it works better with gridded data, but sequential data will be converted into 1D variables. To install it, just type:

```
$ pip install pydap.responses.netcdf
```

And try to append the extension `.nc` to a request. The data will be converted on-the-fly to a NetCDF file.

Matlab

The Matlab response returns data in a Matlab v5 file. It is returned when the file is requested with the extension `.mat`, and can be installed by with:

```
$ pip install pydap.responses.matlab
```

Excel spreadsheet

This response returns sequential data as an Excel spreadsheet when `.xls` is appended to the URL. Install with:

```
$ pip install pydap.responses.xls
```

Developer documentation

This documentation is intended for other developers that would like to contribute with the development of Pydap, or extend it in new ways. It assumes that you have a basic knowledge of Python and HTTP, and understands how data is stored in different formats. It also assumes some familiarity with the Data Access Protocol, though a lot of its inner workings will be explained in detail here.

The DAP data model

The DAP is a protocol designed for the efficient transmission of scientific data over the internet. In order to transmit data from the server to a client, both must agree on a way to represent data: *is it an array of integers?*, *a multi-dimensional grid?* In order to do this, the specification defines a *data model* that, in theory, should be able to represent any existing dataset.

Metadata

Pydap has a series of classes in the `pydap.model` module, representing the DAP data model. The most fundamental data type is called `BaseType`, and it represents a value or an array of values. Here an example of creating one of these objects:

Note: Prior to Pydap 3.2, the name argument was optional for all date types. Since Pydap 3.2, it is mandatory.

```
>>> from pydap.model import *
>>> import numpy as np
>>> a = BaseType(
...         name='a',
...         data=np.array([1]),
...         attributes={'long_name': 'variable a'})
```

All Pydap types have five attributes in common. The first one is the `name` of the variable; in this case, our variable is called “a”:

```
>>> a.name
'a'
```

Note that there's a difference between the variable name (the local name `a`) and its attribute `name`; in this example they are equal, but we could reference our object using any other name:

```
>>> b = a  # b now points to a
>>> b.name
'a'
```

We can use special characters for the variable names; they will be quoted accordingly:

```
>>> c = BaseType(name='long & complicated')
>>> c.name
'long%20%26%20complicated'
```

The second attribute is called `id`. In the examples we've seen so far, `id` and `name` are equal:

```
>>> a.name
'a'
>>> a.id
'a'
>>> c.name
'long%20%26%20complicated'
>>> c.id
'long%20%26%20complicated'
```

This is because the `id` is used to show the position of the variable in a given dataset, and in these examples the variables do not belong to any datasets. First let's store our variables in a container object called `StructureType`. A `StructureType` is a special type of ordered dictionary that holds other Pydap types:

```
>>> s = StructureType('s')
>>> s['a'] = a
>>> s['c'] = c
Traceback (most recent call last):
    ...
KeyError: 'Key "c" is different from variable name "long%20%26%20complicated"!'
```

Note that the variable name has to be used as its key on the `StructureType`. This can be easily remedied:

```
>>> s[c.name] = c
```

There is a special derivative of the `StructureType` called `DatasetType`, which represent the dataset. The difference between the two is that there should be only one `DatasetType`, but it may contain any number of `StructureType` objects, which can be deeply nested. Let's create our dataset object:

```
>>> dataset = DatasetType(name='example')
>>> dataset['s'] = s
>>> dataset.id
'example'
>>> dataset['s'].id
's'
>>> dataset['s']['a'].id
's.a'
```

Note that for objects on the first level of the dataset, like `s`, the id is identical to the name. Deeper objects, like `a` which is stored in `s`, have their id calculated by joining the names of the variables with a period. One detail is that we can access variables stored in a structure using a "lazy" syntax like this:

```
>>> dataset.s.a.id
's.a'
```

The third common attribute that variables share is called `attributes`, which hold most of its metadata. This attribute is a dictionary of keys and values, and the values themselves can also be dictionaries. For our variable `a` we have:

```
>>> a.attributes
{'long_name': 'variable a'}
```

These attributes can be accessed lazily directly from the variable:

```
>>> a.long_name
'variable a'
```

But if you want to create a new attribute you'll have to insert it directly into `attributes`:

```
>>> a.history = 'Created by me'
>>> a.attributes
{'long_name': 'variable a'}
>>> a.attributes['history'] = 'Created by me'
>>> sorted(a.attributes.items())
[('history', 'Created by me'),
('long_name', 'variable a')]
```

It's always better to use the correct syntax instead of the lazy one when writing code. Use the lazy syntax only when introspecting a dataset on the Python interpreter, to save a few keystrokes.

The fourth attribute is called `data`, and it holds a representation of the actual data. We'll take a detailed look of this attribute in the next subsection.

Note: Prior to Pydap 3.2, all variables had also an attribute called `_nesting_level`. This attribute had value 1 if the variable was inside a `SequenceType` object, 0 if it's outside, and >1 if it's inside a nested sequence. Since Pydap 3.2, the `_nesting_level` has been deprecated and there is no intrinsic way of finding the where in a deep object a variable is located.

Data

As we saw on the last subsection, all Pydap objects have a `data` attribute that holds a representation of the variable data. This representation will vary depending on the variable type.

BaseType

For the simple `BaseType` objects the `data` attributes is usually a Numpy array, though we can also use a Numpy scalar or Python number:

```
>>> a = BaseType(name='a', data=np.array(1))
>>> a.data
array(1)

>>> b = BaseType(name='b', data=np.arange(4))
>>> b.data
array([0, 1, 2, 3])
```

Note that starting from Pydap 3.2 the datatype is inferred from the input data:

```
>>> a.dtype
dtype('int64')
>>> b.dtype
dtype('int64')
```

When you *slice* a `BaseType` array, the slice is simply passed onto the data attribute. So we may have:

```
>>> b[-1]
<BaseType with data array(3)>
>>> b[-1].data
array(3)
>>> b[:2]
<BaseType with data array([0, 1])>
>>> b[:2].data
array([0, 1])
```

You can think of a `BaseType` object as a thin layer around Numpy arrays, until you realize that the `data` attribute can be *any* object implementing the array interface! This is how the DAP client works – instead of assigning an array with data directly to the attribute, we assign a special object which behaves like an array and acts as a *proxy* to a remote dataset.

Here's an example:

```
>>> from pydap.handlers.dap import BaseProxy
>>> pseudo_array = BaseProxy(
...         'http://test.opendap.org/dap/data/nc/coads_climatology.nc',
...         'SST.SST',
...         np.float64,
...         (12, 90, 180))
>>> print(pseudo_array[0, 10:14, 10:14])  # download the corresponding data
[[[ -1.26285708e+00  -9.99999979e+33  -9.99999979e+33  -9.99999979e+33]
  [ -7.69166648e-01  -7.79999971e-01  -6.75454497e-01  -5.95714271e-01]
  [  1.28333330e-01  -5.00000156e-02  -6.36363626e-02  -1.41666666e-01]
  [  6.38000011e-01   8.95384610e-01   7.21666634e-01   8.10000002e-01]]]
```

In the example above, the data is only downloaded in the last line, when the pseudo array is sliced. The object will construct the appropriate DAP URL, request the data, unpack it and return a Numpy array.

StructureType

A `StructureType` holds no data; instead, its `data` attribute is a property that collects data from the children variables:

```
>>> s = StructureType(name='s')
>>> s[a.name] = a
>>> s[b.name] = b
>>> a.data
array(1)
>>> b.data
array([0, 1, 2, 3])
>>> print(s.data)
[array(1), array([0, 1, 2, 3])]
```

The opposite is also true; it's possible to specify the structure data and have it propagated to the children:

```
>>> s.data = (1, 2)
>>> print(s.a.data)
1
>>> print(s.b.data)
2
```

The same is true for objects of `DatasetType`, since the dataset is simply the root structure.

SequenceType

A `SequenceType` object is a special kind of `StructureType` holding sequential data. Here's an example of a sequence holding the variables `a` and `c` that we created before:

```
>>> s = SequenceType(name='s')
>>> s[a.name] = a
>>> s[c.name] = c
```

Let's add some data to our sequence. This can be done by setting a structured numpy array to the data attribute:

```
>>> print(s)
<SequenceType with children 'a', 'long%20%26%20complicated'>
>>> test_data = np.array([
... (1, 10),
... (2, 20),
... (3, 30)],
... dtype=np.dtype([
... ('a', np.int32), ('long%20%26%20complicated', np.int16)]))
>>> s.data = test_data
>>> print(s.data)
[(1, 10) (2, 20) (3, 30)]
```

Note that the data for the sequence is an aggregation of the children data, similar to Python's `zip()` builtin. This will be more complicated when encountering nested sequences, but for flat sequences they behave the same.

We can also iterate over the `SequenceType`. In this case, it will return a series of tuples with the data:

```
>>> for record in s.iterdata():
...     print(record)
(1, 10)
(2, 20)
(3, 30)
```

Prior to Pydap 3.2.2, this approach was not possible and one had to iterate directly over `SequenceType`:

```
>>> for record in s:
...     print(record)
(1, 10)
(2, 20)
(3, 30)
```

This approach will be deprecated in Pydap 3.4.

The `SequenceType` behaves pretty much like record arrays from Numpy, since we can reference them by column (`s['a']`) or by index:

```
>>> s[1].data
(2, 20)
```

```
>>> s[ s.a < 3 ].data
array([(1, 10), (2, 20)],
      dtype=[('a', '<i4'), ('long%20%26%20complicated', '<i2')])
```

Note that these objects are also `SequenceType` themselves. The basic rules when working with sequence data are:

1. When a `SequenceType` is sliced with a string the corresponding children is returned. For example: `s['a']` will return child `a`;
2. When a `SequenceType` is iterated over (using `.iterdata()` after Pydap 3.2.2) it will return a series of tuples, each one containing the data for a record;
3. When a `SequenceType` is sliced with an integer, a comparison or a `slice()` a new `SequenceType` will be returned;
4. When a `SequenceType` is sliced with a tuple of strings a new `SequenceType` will be returned, containing only the children defined in the tuple in the new order. For example, `s[('c', 'a')]` will return a sequence `s` with the children `c` and `a`, in that order.

Note that except for rule 4 `SequenceType` mimics the behavior of Numpy record arrays.

Now imagine that we want to add to a `SequenceType` data pulled from a relational database. The easy way would be to fetch the data in the correct column order, and insert it into the sequence. But what if we don't want to store the data in memory, and instead we would like to stream it directly from the database? In this case we can create an object that behaves like a record array, similar to the proxy object that implements the array interface. Pydap defines a "protocol" called `IterData`, which is simply any object that:

1. Returns data when iterated over.
2. Returns a new `IterData` when sliced such that:
   (a) if the slice is a string the new `IterData` contains data only for that children;
   (b) if the slice is a tuple of strings the object contains only those children, in that order;
   (c) if the slice is an integer, a `slice()` or a comparison, the data is filter accordingly.

The base implementation works by wrapping data from a basic Numpy array. And here is an example of how we would use it:

```
>>> from pydap.handlers.lib import IterData
>>> s.data = IterData(np.array([(1, 2), (10, 20)]), s)
>>> print(s)
<SequenceType with children 'a', 'long%20%26%20complicated'>
>>> s2 = s.data[ s['a'] > 1 ]
>>> print(s2)
<IterData to stream array([[ 1,  2],
       [10, 20]])>
>>> for record in s2.iterdata():
...     print(record)
(10, 20)
```

One can also iterate directly over the `IterData` object to obtain the data:

```
>>> for record in s2:
...     print(record)
(10, 20)
```

This approach will not be deprecated in Pydap 3.4.

There are many implementations of classes derived from `IterData`: `pydap.handlers.dap.SequenceProxy` is a proxy to sequential data on Opendap servers, `pydap.handlers.csv.CSVProxy` wraps a CSV file, and `pydap.handlers.sql.SQLProxy` works as a stream to a relational database.

GridType

A `GridType` is a special kind of object that behaves like an array and a `StructureType`. The class is derived from `StructureType`; the major difference is that the first defined variable is a multidimensional array, while subsequent children are vector maps that define the axes of the array. This way, the `data` attribute on a `GridType` returns the data of all its children: the n-dimensional array followed by *n* maps.

Here is a simple example:

```
>>> g = GridType(name='g')
>>> data = np.arange(6)
>>> data.shape = (2, 3)
>>> g['a'] = BaseType(name='a', data=data, shape=data.shape, type=np.int32,␣
↪dimensions=('x', 'y'))
>>> g['x'] = BaseType(name='x', data=np.arange(2), shape=(2,), type=np.int32)
>>> g['y'] = BaseType(name='y', data=np.arange(3), shape=(3,), type=np.int32)
>>> g.data
[array([[0, 1, 2],
           [3, 4, 5]]), array([0, 1]), array([0, 1, 2])]
```

Grid behave like arrays in that they can be sliced. When this happens, a new `GridType` is returned with the proper data and axes:

```
>>> print(g)
<GridType with array 'a' and maps 'x', 'y'>
>>> print(g[0])
<GridType with array 'a' and maps 'x', 'y'>
>>> print(g[0].data)
[array([0, 1, 2]), array(0), array([0, 1, 2])]
```

It is possible to disable this feature (some older servers might not handle it nicely):

```
>>> g = GridType(name='g')
>>> g.set_output_grid(False)
>>> data = np.arange(6)
>>> data.shape = (2, 3)
>>> g['a'] = BaseType(name='a', data=data, shape=data.shape, type=np.int32,␣
↪dimensions=('x', 'y'))
>>> g['x'] = BaseType(name='x', data=np.arange(2), shape=(2,), type=np.int32)
>>> g['y'] = BaseType(name='y', data=np.arange(3), shape=(3,), type=np.int32)
>>> g.data
[array([[0, 1, 2],
       [3, 4, 5]]), array([0, 1]), array([0, 1, 2])]
>>> print(g)
<GridType with array 'a' and maps 'x', 'y'>
>>> print(g[0])
<BaseType with data array([0, 1, 2])>
>>> print(g[0].name)
a
>>> print(g[0].data)
[0  1  2]
```

Handlers

Now that we saw the Pydap data model we can understand handlers: handlers are simply classes that convert data into the Pydap data model. The NetCDF handler, for example, reads a NetCDF file and builds a `DatasetType` object. The SQL handler reads a file describing the variables and maps them to a given table on a relational database. Pydap uses entry points in order to find handlers that are installed in the system. This means that handlers can be developed and installed separately from Pydap. Handlers are mapped to files by a regular expression that usually matches the extension of the file.

Here is the minimal configuration for a handler that serves `.npz` files from Numpy:

```
import os
import re

import numpy

from pydap.model import *
from pydap.handlers.lib import BaseHandler
from pydap.handlers.helper import constrain

class Handler(BaseHandler):

    extensions = re.compile(r'^.*\.npz$', re.IGNORECASE)

    def __init__(self, filepath):
        self.filename = os.path.split(filepath)[1]
        self.f = numpy.load(filepath)

    def parse_constraints(self, environ):
        dataset = DatasetType(name=self.filename)

        for name in self.f.files:
            data = self.f[name][:]
            dataset[name] = BaseType(name=name, data=data, shape=data.shape,␣
↪type=data.dtype.char)

        return constrain(dataset, environ.get('QUERY_STRING', ''))

if __name__ == '__main__':
    import sys
    from paste.httpserver import serve

    application = Handler(sys.argv[1])
    serve(application, port=8001)
```

So let's go over the code. Our handler has a single class called `Handler` that should be configured as an entry point in the `setup.py` file for the handler:

```
[pydap.handler]
npz = path.to.my.handler:Handler
```

Here the name of our handler ("npz") can be anything, as long as it points to the correct class. In order for Pydap to be able to find the handler, it must be installed in the system with either a `python setup.py install` or, even better, `python setup.py develop`.

The class-level attribute `extensions` defines a regular expression that matches the files supported by the handler. In this case, the handler will match all files ending with the `.npz` extension.

When the handler is instantiated the complete filepath to the data file is passed in `__init__`. With this information,

our handler extracts the filename of the data file and opens it using the `load()` function from Numpy. The handler will be initialized for every request, and immediately its `parse_constraints` method is called.

The `parse_constraints` method is responsible for building a dataset object based on information for the request available on `environ`. In this simple handler we simply built a `DatasetType` object with the entirety of our dataset, i.e., we added *all data from all variables* that were available on the `.npz` file. Some requests will ask for only a few variables, and only a subset of their data. The easy way parsing the request is simply passing the complete dataset together with the `QUERY_STRING` to the `contrain()` function:

```
return constrain(dataset, environ.get('QUERY_STRING', ''))
```

This will take care of filtering our dataset according to the request, although it may not be very efficient. For this reason, handlers usually implement their own parsing of the Opendap constraint expression. The SQL handler, for example, will translate the query string into an SQL expression that filters the data on the database, and not on Python.

Finally, note that the handler is a WSGI application: we can initialize it with a filepath and pass it to a WSGI server. This enables us to quickly test the handler, by checking the different responses at `http://localhost:8001/.dds`, for example. It also means that it is very easy to dinamically serve datasets by plugging them to a route dispatcher.

Responses

If handlers are responsible for converting data into the Pydap data model, responses to the opposite: the convert from the data model to different representations. The Opendap specification defines a series of standard responses, that allow clients to introspect a dataset by downloading metadata, and later download data for the subset of interest. These standard responses are the DDS (Dataset Descriptor Structure), the DAS (Dataset Attribute Structure) and the DODS response.

Apart from these, there are additional non-standard responses that add functionality to the server. The ASCII response, for example, formats the data as ASCII for quick visualization on the browser; the HTML response builds a form from which the user can select a subset of the data.

Here is an example of a minimal Pydap response that returns the attributes of the dataset as JSON:

```
from simplejson import dumps

from pydap.responses.lib import BaseResponse
from pydap.lib import walk

class JSONResponse(BaseResponse):
    def __init__(self, dataset):
        BaseResponse.__init__(self, dataset)
        self.headers.append(('Content-type', 'application/json'))

    @staticmethod
    def serialize(dataset):
        attributes = {}
        for child in walk(dataset):
            attributes[child.id] = child.attributes

        if hasattr(dataset, 'close'):
            dataset.close()

        return [dumps(attributes)]
```

This response is mapped to a specific extension defined in its entry point:

```
[pydap.response]
json = path.to.my.response:JSONResponse
```

In this example the response will be called when the `.json` extension is appended to any dataset.

The most important method in the response is the `serialize` method, which is responsible for serializing the dataset into the external format. The method should be a generator or return a list of strings, like in this example. Note that the method is responsible for calling `dataset.close()`, if available, since some handlers use this for closing file handlers or database connections.

One important thing about the responses is that, like handlers, they are also WSGI applications. WSGI applications should return an iterable when called; usually this is a list of strings corresponding to the output from the application. The `BaseResponse` application, however, returns a special iterable that contains both the `DatasetType` object and its serialization function. This means that WSGI middleware that manipulate the response have direct access to the dataset, avoiding the need for deserialization/serialization of the dataset in order to change it.

Here is a simple example of a middleware that adds an attribute to a given dataset:

```
from webob import Request

from pydap.model import *
from pydap.handlers.lib import BaseHandler

class AttributeMiddleware(object):

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        # say that we want the parsed response
        environ['x-wsgiorg.want_parsed_response'] = True

        # call the app with the original request and get the response
        req = Request(environ)
        res = req.get_response(self.app)

        # get the dataset from the response and set attribute
        dataset = res.app_iter.x_wsgiorg_parsed_response(DatasetType)
        dataset.attributes['foo'] = 'bar'

        # return original response
        response = BaseHandler.response_map[ environ['pydap.response'] ]
        responder = response(dataset)
        return responder(environ, start_response)
```

The code should actually do more bookkeeping, like checking if the dataset can be retrieved from the response or updating the `Content-Length` header, but I wanted to keep it simple. Pydap comes with a WSGI middleware for handling server-side functions (`pydap.wsgi.ssf`) that makes heavy use of this feature. It works by removing function calls from the request, fetching the dataset from the modified request, applying the function calls and returning a new dataset.

Templating

Pydap uses an experimental backend-agnostic templating API for rendering HTML and XML by responses. Since the API is neutral Pydap can use any templating engine, like Mako or Genshi, and templates can be loaded from disk, memory or a database. The server that comes with Pydap, for example, defines a `renderer` object that loads Genshi templates from a directory `templatedir`:

```
from pydap.util.template import FileLoader, GenshiRenderer

class FileServer(object):
    def __init__(self, templatedir, ...):
        loader = FileLoader(templatedir)
        self.renderer = GenshiRenderer(options={}, loader=loader)
        ...

    def __call__(self, environ, start_response):
        environ.setdefault('pydap.renderer', self.renderer)
        ...
```

And here is how the HTML response uses the renderer: the response requests a template called `html.html`, that is loaded from the directory by the `FileLoader` object, and renders it by passing the context:

```
def serialize(dataset):
    ...
    renderer = environ['pydap.renderer']
    template = renderer.loader('html.html')
    output = renderer.render(template, context, output_format='text/html')
    return [output]
```

(This is acutally a simplification; if you look at the code you'll notice that there's also code to fallback to a default renderer if one is not found in the `environ`.)

License

Copyright (c) 2003–2010 Roberto De Almeida

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search