Birdy Documentation

Release 0.8.7

Birdhouse

May 08, 2024

CONTENTS

Birdy (the bird)

Birdy is not a bird but likes to play with them.

Birdy is a Python library to work with Web Processing Services (WPS). It is using *OWSLib* from the *GeoPython* project.

You can try Birdy online using Binder (just click on the binder link below), or view the notebooks on NBViewer.

Birdy is part of the Birdhouse project.

Full documentation is on ReadTheDocs.

CHAPTER

ONE

INSTALLATION

1.1 Install from PyPI

```
$ pip install birdhouse-birdy
```

1.2 Install from Anaconda

```
$ conda install -c conda-forge birdy
```

1.3 Install from GitHub

Check out code from the birdy GitHub repo and start the installation:

```
$ git clone https://github.com/bird-house/birdy.git
$ cd birdy
$ conda env create -f environment.yml
$ python setup.py install
```

CHAPTER

TWO

EXAMPLES

You can try these notebook online using Binder, or view the notebooks on NBViewer.

2.1 Basic Usage

2.1.1 Birdy WPSClient example with Emu WPS

[ ]:
```
from birdy import WPSClient
```

Use Emu WPS

https://github.com/bird-house/emu

[ ]:
```
emu = WPSClient(url="http://localhost:5000/wps")
emu_i = WPSClient(url="http://localhost:5000/wps", progress=True)
```

Get Infos about `hello`

[ ]:
```
emu.hello?
```

Run `hello`

[ ]:
```
emu.hello(name="Birdy").get()[0]
```

Run a long running process

```
[ ]: result = emu_i.sleep(delay="1.0")
```

```
[ ]: result.get()[0]
```

Run a process returning a reference to a text document

```
[ ]: emu.chomsky(times="5").get()[0]
```

Pass a local file to a remote process

The client can look up local files on this machine and embed their content in the WPS request to the server. Just set the path to the file or an opened file-like object.

```
[ ]: fn = "/tmp/text.txt"
with open(fn, "w") as f:
    f.write("Just an example")
emu.wordcounter(text=fn).get(asobj=True)
```

Automatically convert the output to a Python object

The client is able to convert input objects into strings to create requests, and also convert output strings into python objects. This can be demonstrated with the `inout` process, which simply takes a variety of `LiteralInputs` of different data types and directs them to the output without any change.

```
[ ]: emu.inout?
```

```
[ ]: import datetime as dt

result = emu.inout(
    string="test",
    int=1,
    float=5.6,
    boolean=True,
    time="15:45",
    datetime=dt.datetime(2018, 12, 12),
    text=None,
    dataset=None,
)
```

Get result as object

```
[ ]: result.get(asobj=True).text
```

Example with multiple_outputs

Similarly, the `multiple_outputs` function returns a `text/plain` file. The converter will automatically convert the text file into a string.

[ ]:
```
out = emu.multiple_outputs(1).get(asobj=True)[0]
print(out)
```

... or use the metalink library on the referenced metalink file:

[ ]:
```
out = emu.multiple_outputs(1).get(asobj=False)[0]
print(out)
```

[ ]:
```
from metalink import download

download.get(out, path="/tmp", segmented=False)
```

2.1.2 Interactive usage of Birdy WPSClient with widgets

[ ]:
```
import birdy
from birdy import WPSClient
from birdy.client import gui

emu = WPSClient(url="http://localhost:5000/wps")
raven = WPSClient(url="http://127.0.0.1:9099/")
```

[ ]:
```
resp = gui(emu.binaryoperatorfornumbers)
```

[ ]:
```
resp.get()
```

[ ]:
```
resp = gui(emu.non_py_id)
```

[ ]:
```
resp = gui(emu.output_formats)
```

[ ]:
```
resp.get()
```

2.1.3 OWSLib versus Birdy

This notebook shows a side-by-side comparison of `owslib.wps.WebProcessingService` and `birdy.WPSClient`.

[ ]:
```
from owslib.wps import WebProcessingService

from birdy import WPSClient

url = "https://bovec.dkrz.de/ows/proxy/emu?Service=WPS&Request=GetCapabilities&Version=1.
↪0.0"

wps = WebProcessingService(url)
cli = WPSClient(url=url)
```

Displaying available processes

With `owslib`, `wps.processes` is the list of processes offered by the server. With `birdy`, the client is like a module with functions. So you just write `cli.` and press `Tab` to display a drop-down menu of processes.

[ ]:
```
wps.processes
```

Documentation about a process

With `owslib`, the process title and abstract can be obtained simply by looking at these attributes. For the process inputs, we need to iterate on the inputs and access their individual attributes. To facilitate this, `owslib.wps` provides the `printInputOuput` function.

With `birdy`, just type `help(cli.hello)` and the docstring will show up in your console. With the IPython console or a Jupyter Notebook, `cli.hello?` would do as well. The docstring follows the NumPy convention.

[ ]:
```
from owslib.wps import printInputOutput

p = wps.describeprocess("hello")
print("Title: ", p.title)
print("Abstract: ", p.abstract)

for inpt in p.dataInputs:
    printInputOutput(inpt)
```

[ ]:
```
help(cli.hello)
```

Launching a process and retrieving literal outputs

With `owslib`, processes are launched using the `execute` method. Inputs are an an argument to `execute` and defined by a list of key-value tuples. These keys are the input names, and the values are string representations. The `execute` method returns a `WPSExecution` object, which defines a number of methods and attributes, including `isComplete` and `isSucceeded`. The process outputs are stored in the `processOutputs` list, whose content is stored in the `data` attribute. Note that this `data` is a list of strings, so we may have to convert it to a `float` to use it.

[ ]:
```
resp = wps.execute(
    "binaryoperatorfornumbers",
    inputs=[("inputa", "1.0"), ("inputb", "2.0"), ("operator", "add")],
)
if resp.isSucceeded:
    (output,) = resp.processOutputs
    print(output.data)
```

With `birdy`, inputs are just typical keyword arguments, and outputs are already converted into python objects. Since some processes may have multiple outputs, processes always return a `namedtuple`, even in the case where there is only a single output.

[ ]:
```
z = cli.binaryoperatorfornumbers(1, 2, operator="add").get()[0]
z
```

[ ]:
```
out = cli.inout().get()
out.date
```

Retrieving outputs by references

For `ComplexData` objects, WPS servers often return a reference to the output (an http link) instead of the actual data. This is useful if that output is to serve as an input to another process, so as to avoid passing back and forth large files for nothing.

With `owslib`, that means that the `data` attribute of the output is empty, and we instead access the `reference` attribute. The referenced file can be written to the local disk using the `writeToDisk` method.

With `birdy`, the outputs are by default the references themselves, but it's also possible to download these references in the background and convert them into python objects. To trigger this automatic conversion, set `convert_objects` to `True` when instantating the client `WPSClient(url, convert_objects=True)`. Ini the example below, the first output is a plain text file, and the second output is a json file. The text file is converted into a string, and the json file into a dictionary.

[ ]:
```
resp = wps.execute("multiple_outputs", inputs=[("count", "1")])
output, ref = resp.processOutputs
print(output.reference)
print(ref.reference)
output.writeToDisk("/tmp/output.txt")
```

[ ]:
```
output = cli.multiple_outputs(1).get()[0]
print(output)
# as reference
output = cli.multiple_outputs(1).get(asobj=True)[0]
print(output)
```

2.2 Demo

2.2.1 AGU 2018 Demo

This notebook shows how to use `birdy`'s high-level interface to WPS processes.

Here we access a test server called `Emu` offering a dozen or so dummy processes.

The shell interface

[ ]:
```
%%bash
export WPS_SERVICE="http://localhost:5000/wps?Service=WPS&Request=GetCapabilities&
↪Version=1.0.0"
birdy -h
```

[ ]:
```
%%bash
export WPS_SERVICE="http://localhost:5000/wps?Service=WPS&Request=GetCapabilities&
↪Version=1.0.0"
birdy hello -h
```

[ ]:
```
%%bash
export WPS_SERVICE="http://localhost:5000/wps?Service=WPS&Request=GetCapabilities&
↪Version=1.0.0"
birdy hello --name stranger
```

The python interface

The `WPSClient` function creates a *mock* python module whose functions actually call a remote WPS process. The docstring and signature of the function are dynamically created from the remote's process description. If you type `wps.` and then press `Tab`, you should see a drop-down list of available processes. Simply call `help` on each process of type `?` after the process to print the docstring for that process.

[ ]:
```
from birdy import WPSClient

url = "http://localhost:5000/wps?Service=WPS&Request=GetCapabilities&Version=1.0.0"
wps = WPSClient(url, verify=False)
help(wps.binaryoperatorfornumbers)
```

Type `wps.` and the press `Tab`, you should see a drop-down list of available processes.

[ ]:
```
# wps.
```

Process execution

Processes are executed by calling the function. Each process instantaneoulsy returns a `WPSExecute` object. The actual output values of the process are obtained by calling the `get` method. This `get` method returns a namedtuple storing the process outputs as native python objects.

[ ]:
```
resp = wps.binaryoperatorfornumbers(1, 2, operator="add")
print(resp)
resp.get()
```

For instance, the `inout` function returns a wide variety of data types (float, integers, dates, etc) all of which are converted into a corresponding python type.

[ ]:
```
wps.inout().get()
```

Retrieving outputs by references

For `ComplexData` objects, WPS servers often return a reference to the output (an http link) instead of the actual data. This is useful if that output is to serve as an input to another process, so as to avoid passing back and forth large files for nothing.

With `birdy`, the outputs are by default return values are the references themselves, but it's also possible to download these references in the background and convert them into python objects. To trigger this automatic conversion, set `asobj` to `True` when calling the `get` method. In the example below, we're using a dummy process called `output_formats`, whose first output is a netCDF file, and second output is a json file. With `asobj=True`, the netCDF file is opened and returned as a `netcdf4.Dataset` instance, and the json file into a dictionary.

[ ]:
```
# NBVAL_SKIP
# This cell is failing due to an unautheticated SSL certificate
out = wps.output_formats()
nc, json = out.get()
print(out.get())
ds, json = out.get(asobj=True)
print(json)
ds
```

Progress bar

It's possible to display a progress bar when calling a process. The interface to do so at the moment goes like this. Note that the cancel button does not do much here, as the WPS server does not support interruption requests.

[ ]:
```
wps = WPSClient("http://localhost:5000/wps", progress=True)
resp = wps.sleep()
```

CHAPTER

THREE

DEVELOPMENT

3.1 Get Started!

Check out code from the birdy GitHub repo and start the installation:

```
$ git clone https://github.com/bird-house/birdy.git
$ cd birdy
$ conda env create -f environment.yml
$ pip install --editable .
```

Install additional dependencies:

```
$ pip install -r requirements_dev.txt
```

When you're done making changes, check that your changes pass *black*, *flake8* and the tests:

```
$ flake8 birdy tests
$ black --check --target-version py39 birdy tests
$ pytest -v tests
```

Or use the Makefile:

```
$ make lint
$ make test
$ make test-all
```

3.1.1 Add pre-commit hooks

Before committing your changes, we ask that you install *pre-commit* in your environment. *Pre-commit* runs git hooks that ensure that your code resembles that of the project and catches and corrects any small errors or inconsistencies when you *git commit*:

```
$ conda install -c conda-forge pre-commit
$ pre-commit install
```

3.2 Write Documentation

You can find the documentation in the *docs/source* folder. To generate the Sphinx documentation locally you can use the *Makefile*:

```
$ make docs
```

3.3 Bump a new version

Make a new version of Birdy in the following steps:

- Make sure everything is commit to GitHub.
- Update `CHANGES.rst` with the next version.
- Dry Run: `bumpversion --dry-run --verbose --new-version 0.3.1 patch`
- Do it: `bumpversion --new-version 0.3.1 patch`
- ... or: `bumpversion --new-version 0.4.0 minor`
- Push it: `git push --tags`

See the bumpversion documentation for details.

3.4 Build a source distribution and wheel

To build a source distribution (*.sdist*) and wheel (*.whl*) locally, run the following command:

```
$ python -m build
```

This will create a *dist* folder with the source distribution and wheel.

See the build documentation for details.

3.5 Release a new version

Leveraging GitHub Workflows, maintainers can release new versions of Birdy automatically:

- Ensure that the changelog and version on the main development branch have been updated to reflect the new version.
- **Create a tag (*vX.Y.Z*) of the main development branch and push to the GitHub repository.**
  - This will trigger a workflow that will attempt to build Birdy and publish it to TestPyPI.
  - When this actions succeeds, be sure to verify on TestPyPI that the package reflects changes.
- **On GitHub, a maintainer can then publish a new version using the newly created tag.**
  - This will trigger a workflow that will attempt to build Birdy and publish it to PyPI.
  - Be warned that once published to PyPI, a version number can never be overwritten! Bad versions may only be yanked.

CHAPTER

FOUR

API REFERENCE

4.1 Using the command line

4.1.1 Birdy CLI module

Birdy has a command line interface to interact with a Web Processing Service.

Example

Here is an example with Emu WPS service:

```
$ birdy -h
$ birdy hello -h
$ birdy hello --name stranger
'Hello Stranger'
```

Configure WPS service URL

By default Birdy talks to a WPS service on URL http://localhost:5000/wps. You can change this URL by setting the enivronment variable `WPS_SERVICE`:

```
$ export WPS_SERVICE=http://localhost:5000/wps
```

Configure SSL verification for HTTPS

In case you have a WPS serive using HTTPS with a self-signed certificate you need to configure the environment variable `WPS_SSL_VERIFY`:

```
# deactivate SSL server validation for a self-signed certificate.
$ export WPS_SSL_VERIFY=false
```

You can also set the path of the service certificate. Read the requests documentation.

Use an OAuth2 access token

If the WPS service is secured by an OAuth2 access tokens then you can provide an access token with the `--token` option:

```
$ birdy --token abc123 hello --name stranger
```

Use client certificate to access WPS service

If the WPS service is secured by x509 certificates you can add a certificate with the `--cert` option to a request:

```
# run hello with certificate
$ birdy --cert cert.pem hello --name stranger
```

Using the output_formats option for a process

Each process also has a default option named *output_formats*. It can be used to override a process' output format's default values.

This option takes three parameters;

The format identifier: the name given to it

The `as reference` parameter: if the output is returned as a link of not. Can be True, False, or None (which uses the process' default value)

The MIME type: of which MIME type is the output. Unless the process has multiple supported mime types, this can be left to None.

Looking at the emu process *output_formats*, the JSON output's default's the `as reference` parameter to False and returns the content directly:

```
$ birdy output_formats
  Output:
  netcdf=http://localhost:5000/outputs/d9abfdc4-08d6-11eb-9334-0800274cd70c/dummy.nc
  json=['{"testing": [1, 2]}']
```

We can then use the output_formats option to redefine it:

```
$ birdy output_formats --output_formats json True None
  Output:
  netcdf=http://localhost:5000/outputs/38e9aefe-08db-11eb-9334-0800274cd70c/dummy.nc
  json=http://localhost:5000/outputs/38e9aefe-08db-11eb-9334-0800274cd70c/dummy.json
```

4.2 Using the Python library

4.2.1 WPSClient Class

The `WPSClient` class aims to make working with WPS servers easy, even without any prior knowledge of WPS.

Calling the `WPSClient` class creates an instance whose methods call WPS processes. These methods are generated at runtime based on the process description provided by the WPS server. Calling a function sends an *execute* request to the server. The server response is parsed and returned as a `WPSExecution` instance, which includes information about the job status, the progress percentage, the starting time, etc. The actual output from the process are obtained by calling the *get* method.

The output is parsed to convert the outputs in native Python whenever possible. *LiteralOutput* objects (string, float, integer, boolean) are automatically converted to their native format. For *ComplexOutput*, the module can either return a link to the output files stored on the server, or try to convert the outputs to a Python object based on their mime type. This conversion will occur with *get(asobj=True)*. So for example, if the mime type is 'application/json', the output would be a *dict*.

Inputs to processes can be native Python types (string, float, int, date, datetime), http links or local files. Local files can be transferred to a remote server by including their content into the WPS request. Simply set the input to a valid path or file object and the client will take care of reading and converting the file.

Example

If a WPS server with a simple *hello* process is running on the local host on port 5000:

```
>>> from birdy import WPSClient
>>> emu = WPSClient('http://localhost:5000/')
>>> emu.hello
<bound method hello of <birdy.client.base.WPSClient object>>
>>> print(emu.hello.__doc__)

# Just says a friendly Hello. Returns a literal string output with Hello plus the␣
↪inputed name.

# Parameters
# ----------
# name : string
#     Please enter your name.
#
# Returns
# -------
# output : string
#     A friendly Hello from us.
#
# ""
#
# # Call the function. The output is a namedtuple
# >>> emu.hello('stranger')
# hello(output='Hello stranger')
```

Authentication

If you want to connect to a server that requires authentication, the `WPSClient` class accepts an *auth* argument that behaves exactly like in the popular *requests* module (see requests Authentication)

The simplest form of authentication is HTTP Basic Auth. Although wps processes are not commonly protected by this authentication method, here is a simple example of how to use it:

```
>>> from birdy import WPSClient
>>> from requests.auth import HTTPBasicAuth
>>> auth = HTTPBasicAuth('user', 'pass')
>>> wps = WPSClient('http://www.example.com/wps', auth=auth)
```

Because any *requests*-compatible class is accepted, custom authentication methods are implemented the same way as in *requests*.

For example, to connect to a magpie protected wps, you can use the requests-magpie module:

```
>>> from birdy import WPSClient
>>> from requests_magpie import MagpieAuth
>>> auth = MagpieAuth('https://www.example.com/magpie', 'user', 'pass')
>>> wps = WPSClient('http://www.example.com/wps', auth=auth)
```

Output format

Birdy automatically manages process output to reflect its default values or Birdy's own defaults.

However, it's possible to customize the output of a process. Each process has an input named `output_formats`, that takes a dictionary as a parameter:

```
# example format = {
#     'output_identifier': {
#         'as_ref': <True, False or None>
#         'mimetype': <MIME type as a string or None>,
#     },
# }

# A dictionary defining netcdf and json outputs
>>> custom_format = {
>>>     'netcdf': {
>>>         'as_ref': True,
>>>         'mimetype': 'application/json',
>>>     },
>>>     'json': {
>>>         'as_ref': False,
>>>         'mimetype': None
>>>     }
>>> }
```

Utility functions can also be used to create this dictionary:

```
>>> custom_format = create_output_dictionary('netcdf', True, 'application/json')
>>> add_output_format(custom_format, 'json', False, None)
```

The created dictionary can then be used with a process:

```
>>> cli = WPSClient("http://localhost:5000")
>>> z = cli.output_formats(output_formats=custom_format).get()
>>> z
```

CHAPTER

FIVE

CHANGE HISTORY

5.1 v0.8.7 (2024-05-07)

- Fix regression, where loading TIFF files would return a Dataset instead of a DataArray, the behavior prior to 0.8.5. Loading a multi-band TIFF file will now return a DataArray with the bands as dimensions.

5.2 v0.8.6 (2024-03-18)

Changes:

- Restructure the package so that the wheel does not install the testing and docs as non-importable packages.
- Ensure that data required to run tests and build docs is present in the source distribution (via *Manifest.in* changes).
- Documentation now includes a *sphinx-apidoc*-based listing of all installed modules and functions
- Add *sphinx-copybutton* and *sphinx-codeautolink* in order to increase the usefulness of code-blocks in the example documentation (copying of code blocks and ability to click on *birdy* objects and go straight to the documentation entry for the object).
- All documentation build warnings have been addressed.
- Add the *birdy[extra]* pip install recipe to be able to install all extras needed more directly.
- Raise the minimum Python required to 3.9 in the setup block.
- Remove the Python package for *pandoc* (unmaintained).
- Add a documentation entry on using *build* to build the documentation.

5.3 0.8.5 (2024-03-14)

Changes:

- Update how TIFF files are converted to xarray datasets because *open_rasterio* is deprecated. See issue *239*.
- Remove *GeotiffRasterioConverter*.
- Remove Python 3.7 and 3.8 from CI test suite.
- Now using Trusted Publisher for TestPyPI/PyPI releases.
- Update *black* to v24.2.0 and code formatting conventions to Python3.9+.

5.4 0.8.4 (2023-05-24)

Changes:

- Fix docstring creation error occurring when the server identification abstract is None. See issue *228*.
- Handle case where the server *describeProcess* does not understand "ALL" as the process identifier. See issue *229*.

5.5 0.8.3 (2023-05-03)

Changes:

- Added the *packaging* library to the list of requirements.

5.6 0.8.2 (2023-04-28)

Changes:

- Relax dependency check on GeoTiff rioxarray and rasterio converters due to some mysterious gdal error.
- Remove tests with live 52North WPS server since it seems offline.
- Remove Python 3.6 from test matrix and add 3.10.
- Handle the removal of the *verbose* argument in *OWSLib.WebProcessingService* 0.29.0.

5.7 0.8.1 (2021-12-01)

Changes:

- Before trying to open a netCDF dataset, determine whether link is a valid OPeNDAP endpoint to avoid unnecessarily raising the cryptic `syntax error, unexpected WORD_WORD, expecting SCAN_ATTR or SCAN_DATASET or SCAN_ERROR`.

5.8 0.8.0 (2021-05-25)

Changes:

- Added a converter for loading GeoTIFF using xarray/rioxarray (#193).
- Update notebook process forms. See *client.gui* function.
- Add support for Path objects in *utils.guess_type*.
- Support multiple mimetypes in converters.
- Removed geojson mimetypes from BINARY_MIMETYPES so it's embedded as a string rather than bytes.

API changes:

- *mimetype* (str) replaced by *mimetypes* (tuple) in *client.converters.BaseConverter*.

5.9 0.7.0 (2021-01-15)

Changes:

- Added multiple language support (#164).
- Added an Ipyleaflet wrapper for WFS support (#179).
- Updated GeoJSON mimetype (#181).
- Added ability to specify output format for process execution (#182).
- Fixed tests (#184).
- Use GitHub Actions for CI build instead of Travis CI (#185).
- Use black formatting (#186, #187).

5.10 0.6.9 (2020-03-10)

Changes:

- Fixed passing Path objects (#169)
- Trying to guess mime type of inputs rather than taking the first value (#171)

5.11 0.6.6 (2020-03-03)

Changes:

- Fixed the docs (#150).
- Added outputs to execute in CLI (#151).
- Updated tests (#152).
- Added offline tests (#153).
- Updated conda links (#155).
- Handle Python keywords (#158)
- Fix emu (#159).
- Updated demo notebook tests (#160).
- Added ECMWF demo notebook (#162).
- Added roocs wps demo notebook (#165).
- Added missing files in MANIFEST.in for pypi install (#166).

5.12 0.6.5 (2019-08-19)

Changes:

- Fix arguments ordering (#139).
- Fix imports warning (#138).
- Using nbsphinx (#142).
- Fix pip install (#143).
- Add custom authentication methods (#144).
- Use oauth token (#145).
- Skip Python 2.7 (#146).

5.13 0.6.4 (2019-07-03)

Changes:

- Fix default converter to return bytes (#137).

5.14 0.6.3 (2019-06-21)

Changes:

- Disabled segmented metalink downloads (#132).
- Fix nested conversion (#135).

5.15 0.6.2 (2019-06-06)

Changes:

- Added support for passing sequences (list, tuple) as WPS inputs (#128).

5.16 0.6.1 (2019-05-27)

Changes:

- Added verify argument when downloading files to disk (#123).
- Bugfixes: #118, #121

5.17 0.6.0 (2019-04-04)

Changes:

- Added conversion support for nested outputs (metalink, zip) (#114).
- Added support for Metalink (#113).
- Added support for zip converter (#111).
- Added support for ESGF CWT API (#102).
- Speed up by using *DescribeProcess* with *identifier=all* (#98).
- Added support for passing local files to server as raw data (#97).
- Cleaned up notebooks (#107).
- Various Bugfixes: #83, #91, #99

5.18 0.5.1 (2018-12-18)

Changes:

- Added support to launch Jupyter notebooks with birdy examples on binder (#94, #95).

5.19 0.5.0 (2018-12-03)

Changes:

- Renamed pythonic WPS client (#63): `birdy.client.base.WPSClient` and `from birdy import WPSClient`.
- Added *WPSResult* for WPS outputs as *namedtuple* (#84, #64).
- Support for Jupter Notebooks (#40): cancel button (work in progress), progress bar, input widget.
- Updated notebooks with examples for *WPSClient*.

5.20 0.4.2 (2018-09-26)

Changes:

- Fixed WPS default parameter (#52).
- Using `WPS_SSL_VERIFY` environment variable (#50).

5.21 0.4.1 (2018-09-14)

Changes:

- Fixed test-suite (#49).
- Import native client with `import_wps` (#47).
- Fix: using string type when dataType is not provided (#46).
- Updated docs for native client (#43).

5.22 0.4.0 (2018-09-06)

Release for Dar Es Salaam.

Changes:

- Conda support on RTD (#42).
- Fix optional input (#41).

5.23 0.3.3 (2018-07-18)

Changes:

- Added initial native client (#24, #37).

5.24 0.3.2 (2018-06-06)

Changes:

- Fix MANIFEST.in.

5.25 0.3.1 (2018-06-06)

Changes:

- Fix bumpversion.

5.26 0.3.0 (2018-06-05)

Changes:

- Use bumpversion (#29).
- Use click for CLI (#6).
- Using GitHub templates for issues, PRs and contribution guide.

5.27 0.2.2 (2018-05-08)

Fixes:

- Update travis for Python 3.x (#19).
- Fix parsing of WPS capabilities with `%` (#18).

New Features:

- using `mode` for async execution in OWSLib (#22).

5.28 0.2.1 (2018-03-14)

Fixes:

- Fixed Sphinx and updated docs: #15.

New Features:

- Fix #14: added `--cert` option to use x509 certificates.

5.29 0.2.0 (2017-09-25)

- removed buildout ... just using conda.
- cleaned up docs.
- updated travis.
- fixed tests.
- added compat module for python 3.x

5.30 0.1.9 (2017-04-07)

- updated buildout and Makefile.
- updated conda environment.
- fixed tests.
- replaced nose by pytest.
- pep8.
- fixed travis.
- fixed ComplexData input.
- show status message in log.

5.31 0.1.8 (2016-05-02)

- added backward compatibility for owslib.wps without headers and verify parameter.

5.32 0.1.7 (2016-05-02)

- added twitcher token parameter.
- using ssl verify option again.

5.33 0.1.6 (2016-03-22)

- added support for bbox parameters.

5.34 0.1.5 (2016-03-15)

- fixed wps init (using standard owslib).
- update makefile.

5.35 0.1.4 (2015-10-29)

- using ssl verify option of WebProcessingSerivce
- moved python requirements to requirements/deploy.txt

5.36 0.1.3 (2015-08-20)

- more unit tests.
- fixed unicode error in wps description.
- using latest ComplexDataInput from owslib.wps.

5.37 0.1.2 (2015-08-14)

- fixed encoding of input text files.
- more unit tests.

5.38 0.1.1 (2015-08-13)

- allow local file path for complex inputs.
- send complex data inline with requet to remote wps service.

5.39 0.1.0 (2014-12-02)

- Initial Release.

CHAPTER

SIX

BIRDY

6.1 birdy package

6.1.1 Subpackages

birdy.cli package

Birdy CLI module

Birdy has a command line interface to interact with a Web Processing Service.

Example

Here is an example with Emu WPS service:

```
$ birdy -h
$ birdy hello -h
$ birdy hello --name stranger
'Hello Stranger'
```

Configure WPS service URL

By default Birdy talks to a WPS service on URL http://localhost:5000/wps. You can change this URL by setting the enivronment variable `WPS_SERVICE`:

```
$ export WPS_SERVICE=http://localhost:5000/wps
```

Configure SSL verification for HTTPS

In case you have a WPS serive using HTTPS with a self-signed certificate you need to configure the environment variable `WPS_SSL_VERIFY`:

```
# deactivate SSL server validation for a self-signed certificate.
$ export WPS_SSL_VERIFY=false
```

You can also set the path of the service certificate. Read the requests documentation.

Use an OAuth2 access token

If the WPS service is secured by an OAuth2 access tokens then you can provide an access token with the `--token` option:

```
$ birdy --token abc123 hello --name stranger
```

Use client certificate to access WPS service

If the WPS service is secured by x509 certificates you can add a certificate with the `--cert` option to a request:

```
# run hello with certificate
$ birdy --cert cert.pem hello --name stranger
```

Using the output_formats option for a process

Each process also has a default option named *output_formats*. It can be used to override a process' output format's default values.

This option takes three parameters;

The format identifier: the name given to it

The `as reference` parameter: if the output is returned as a link of not. Can be True, False, or None (which uses the process' default value)

The MIME type: of which MIME type is the output. Unless the process has multiple supported mime types, this can be left to None.

Looking at the emu process *output_formats*, the JSON output's default's the `as reference` parameter to False and returns the content directly:

```
$ birdy output_formats
  Output:
  netcdf=http://localhost:5000/outputs/d9abfdc4-08d6-11eb-9334-0800274cd70c/dummy.nc
  json=['{"testing": [1, 2]}']
```

We can then use the output_formats option to redefine it:

```
$ birdy output_formats --output_formats json True None
  Output:
  netcdf=http://localhost:5000/outputs/38e9aefe-08db-11eb-9334-0800274cd70c/dummy.nc
  json=http://localhost:5000/outputs/38e9aefe-08db-11eb-9334-0800274cd70c/dummy.json
```

Submodules

birdy.cli.base module

class birdy.cli.base.**BirdyCLI**(**args: Any*, ***kwargs: Any*)

Bases: MultiCommand

BirdyCLI is an implementation of click.MultiCommand.

Adds each process of a Web Processing Service as command to the command-line interface.

Parameters

- **url** (*str*) – URL of the Web Processing Service.
- **caps_xml** (*str*) – A WPS GetCapabilities response for testing.
- **desc_xml** (*str*) – A WPS DescribeProcess response with "identifier=all" for testing.

_get_command_info(*name*, *ctx*)

_update_commands()

static format_command_help(*process*)

get_command(*ctx*, *name*)

static get_param_default(*param*)

static get_param_type(*param*)

list_commands(*ctx*)

property wps

birdy.cli.misc module

birdy.cli.misc.**get_ssl_verify**()

birdy.cli.misc.**monitor**(*execution*)

birdy.cli.run module

birdy.cli.run.**_set_language**(*ctx*, *param*, *value*)

birdy.cli.run.**_show_languages**(*ctx*, *param*, *value*)

birdy.cli.types module

class birdy.cli.types.**ComplexParamType**(**args: Any*, ***kwargs: Any*)

Bases: ParamType

convert(*value*, *param*, *ctx*)

name = 'complex'

birdy.client package

WPSClient Class

The `WPSClient` class aims to make working with WPS servers easy, even without any prior knowledge of WPS.

Calling the `WPSClient` class creates an instance whose methods call WPS processes. These methods are generated at runtime based on the process description provided by the WPS server. Calling a function sends an *execute* request to the server. The server response is parsed and returned as a `WPSExecution` instance, which includes information about the job status, the progress percentage, the starting time, etc. The actual output from the process are obtained by calling the *get* method.

The output is parsed to convert the outputs in native Python whenever possible. *LiteralOutput* objects (string, float, integer, boolean) are automatically converted to their native format. For *ComplexOutput*, the module can either return a link to the output files stored on the server, or try to convert the outputs to a Python object based on their mime type. This conversion will occur with *get(asobj=True)*. So for example, if the mime type is 'application/json', the output would be a *dict*.

Inputs to processes can be native Python types (string, float, int, date, datetime), http links or local files. Local files can be transferred to a remote server by including their content into the WPS request. Simply set the input to a valid path or file object and the client will take care of reading and converting the file.

Example

If a WPS server with a simple *hello* process is running on the local host on port 5000:

```
>>> from birdy import WPSClient
>>> emu = WPSClient('http://localhost:5000/')
>>> emu.hello
<bound method hello of <birdy.client.base.WPSClient object>>
>>> print(emu.hello.__doc__)

# Just says a friendly Hello. Returns a literal string output with Hello plus the␣
↪inputed name.

# Parameters
# ----------
# name : string
#     Please enter your name.
#
# Returns
# -------
# output : string
#     A friendly Hello from us.
```

(continues on next page)

(continued from previous page)

```
#
# ""
#
# # Call the function. The output is a namedtuple
# >>> emu.hello('stranger')
# hello(output='Hello stranger')
```

Authentication

If you want to connect to a server that requires authentication, the `WPSClient` class accepts an *auth* argument that behaves exactly like in the popular *requests* module (see requests Authentication)

The simplest form of authentication is HTTP Basic Auth. Although wps processes are not commonly protected by this authentication method, here is a simple example of how to use it:

```
>>> from birdy import WPSClient
>>> from requests.auth import HTTPBasicAuth
>>> auth = HTTPBasicAuth('user', 'pass')
>>> wps = WPSClient('http://www.example.com/wps', auth=auth)
```

Because any *requests*-compatible class is accepted, custom authentication methods are implemented the same way as in *requests*.

For example, to connect to a magpie protected wps, you can use the requests-magpie module:

```
>>> from birdy import WPSClient
>>> from requests_magpie import MagpieAuth
>>> auth = MagpieAuth('https://www.example.com/magpie', 'user', 'pass')
>>> wps = WPSClient('http://www.example.com/wps', auth=auth)
```

Output format

Birdy automatically manages process output to reflect its default values or Birdy's own defaults.

However, it's possible to customize the output of a process. Each process has an input named `output_formats`, that takes a dictionary as a parameter:

```
# example format = {
#     'output_identifier': {
#         'as_ref': <True, False or None>
#         'mimetype': <MIME type as a string or None>,
#     },
# }

# A dictionary defining netcdf and json outputs
>>> custom_format = {
>>>     'netcdf': {
>>>         'as_ref': True,
>>>         'mimetype': 'application/json',
>>>     },
>>>     'json': {
```

(continues on next page)

(continued from previous page)

```
>>>         'as_ref': False,
>>>         'mimetype': None
>>>     }
>>> }
```

Utility functions can also be used to create this dictionary:

```
>>> custom_format = create_output_dictionary('netcdf', True, 'application/json')
>>> add_output_format(custom_format, 'json', False, None)
```

The created dictionary can then be used with a process:

```
>>> cli = WPSClient("http://localhost:5000")
>>> z = cli.output_formats(output_formats=custom_format).get()
>>> z
```

Submodules

birdy.client.base module

class birdy.client.base.**WPSClient**(*url*, *processes=None*, *converters=None*, *username=None*, *password=None*, *headers=None*, *auth=None*, *verify=True*, *cert=None*, *progress=False*, *version=owslib.wps.WPS_DEFAULT_VERSION*, *caps_xml=None*, *desc_xml=None*, *language=None*, *lineage=False*, ***kwds*)

Bases: `object`

Returns a class where every public method is a WPS process available at the given url.

Examples

```
>>> emu = WPSClient(url='<server url>')
>>> emu.hello('stranger')
'Hello stranger'
```

_build_inputs(*pid*, ***kwargs*)

Build the input sequence from the function arguments.

_console_monitor(*execution*, *sleep=3*)

Monitor the execution of a process.

Parameters

- **execution** (*WPSExecution instance*) – The execute response to monitor.
- **sleep** (*float*) – Number of seconds to wait before each status check.

_execute(*pid*, ***kwargs*)

Execute the process.

_get_process_description(*processes=None*, *xml=None*)

Return the description for each process.

Sends the server a *describeProcess* request for each process.

Parameters
processes (*str, list, None*) – A process name, a list of process names or None (for all processes).

Returns
A dictionary keyed by the process identifier of process descriptions.

Return type
OrderedDict

_method_factory(*pid*)

Create a custom function signature with docstring, instantiate it and pass it to a wrapper.

The wrapper will call the process on reception.

Parameters
pid (*str*) – Identifier of the WPS process.

Returns
A Python function calling the remote process, complete with docstring and signature.

Return type
func

_parse_output_formats(*outputs*)

Parse an output format dictionary into a list of tuples, as required by wps.execute().

_setup_logging()

property language

property languages

birdy.client.base.nb_form(*wps*, *pid*)

Return a Notebook form to enter input values and launch process.

birdy.client.base.sort_inputs_key(*i*)

Key function for sorting process inputs.

The order is:

- Inputs that have minOccurs >= 1 and no default value
- Inputs that have minOccurs >= 1 and a default value
- Every other input

Parameters
i (*owslib.wps.Input*) – An owslib Input

Notes

The defaultValue for ComplexData is ComplexData instance specifying mimetype, encoding and schema.

birdy.client.converters module

class birdy.client.converters.**BaseConverter**(*output=None*, *path=None*, *verify=True*)

Bases: object

_check_import(*name*, *package=None*)

Check if libraries can be imported.

Parameters

- **name** (*str*) – module name to try to import
- **package** (*str*) – package of the module

check_dependencies()

convert()

To be subclassed.

property data

Return the data from the remote output in memory.

extensions = ()

property file

Return output Path object. Download from server if not found.

mimetypes = ()

nested = False

priority = None

class birdy.client.converters.**GenericConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

convert()

Return raw bytes memory representation.

priority = 0

class birdy.client.converters.**GeoJSONConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['json', 'geojson']

mimetypes = ['application/geo+json', 'application/vnd.geo+json']

priority = 2

class birdy.client.converters.**GeotiffGdalConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['tiff', 'tif']

mimetypes = ['image/tiff; subtype=geotiff']

priority = 1

class birdy.client.converters.**GeotiffRioxarrayConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['tiff', 'tif']

mimetypes = ['image/tiff; subtype=geotiff']

priority = 3

class birdy.client.converters.**ImageConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['png']

mimetypes = ['image/png']

priority = 1

class birdy.client.converters.**JSONConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

convert()

To be subclassed.

extensions = ['json']

mimetypes = ['application/json']

priority = 1

class birdy.client.converters.**MetalinkConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['metalink', 'meta4']

mimetypes = ['application/metalink+xml; version=3.0', 'application/metalink+xml; version=4.0']

nested = True

priority = 1

class birdy.client.converters.Netcdf4Converter(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['nc', 'nc4']

mimetypes = ['application/x-netcdf']

priority = 1

class birdy.client.converters.ShpFionaConverter(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

mimetypes = ['application/x-zipped-shp']

priority = 1

class birdy.client.converters.ShpOgrConverter(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['zip']

mimetypes = ['application/x-zipped-shp']

priority = 2

class birdy.client.converters.TextConverter(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

convert()

Return text content.

extensions = ['txt', 'csv', 'md', 'rst']

mimetypes = ['text/plain']

priority = 1

class birdy.client.converters.**XarrayConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

check_dependencies()

convert()

To be subclassed.

extensions = ['nc', 'nc4']

mimetypes = ['application/x-netcdf']

priority = 2

class birdy.client.converters.**ZipConverter**(*output=None*, *path=None*, *verify=True*)

Bases: *BaseConverter*

convert()

To be subclassed.

extensions = ['zip']

mimetypes = ['application/zip']

nested = True

priority = 1

birdy.client.converters.**_find_converter**(*mimetype=None*, *extension=None*, *converters=()*)

Return a list of compatible converters ordered by priority.

birdy.client.converters.**all_subclasses**(*cls*)

Return all subclasses of a class.

birdy.client.converters.**convert**(*output: owslib.wps.Output | Path | str*, *path: str | Path*, *converters: Sequence[BaseConverter] = None*, *verify: bool = True*)

Convert a file to an object.

Parameters

- **output** (*owslib.wps.Output, Path, str*) – Item to convert to an object.
- **path** (*str, Path*) – Path on disk where temporary files are stored.
- **converters** (*sequence of BaseConverter subclasses*) – Converter classes to search within for a match.
- **verify** (*bool*)

Returns

Python object or file's content as bytes.

Return type

objs

birdy.client.converters.**find_converter**(*obj*, *converters*)

Find converters for a WPS output or a file on disk.

birdy.client.notebook module

class birdy.client.notebook.**Form**(*func*)

Bases: object

Create notebook form to launch WPS process.

build_ui(*input_widgets*, *of_widgets*, *go*)

Create the form.

get(*asobj=False*)

Return the process response outputs.

Parameters

asobj (*bool*) – If True, object_converters will be used.

input_widget_values(*widgets*)

Return values from input widgets.

input_widgets(*inputs*)

Return input parameter widgets.

output_format_widget_values(*widgets*)

Return the *output_formats* dict from output_formats widgets.

output_formats_widgets(*outputs*)

Return output formats parameter widgets for ComplexData outputs that have multiple supported formats.

birdy.client.notebook.**gui**(*func*)

Return a Notebook form to enter input values and launch process.

birdy.client.notebook.**input2widget**(*inpt*)

Return a Notebook widget to enter values for the input.

birdy.client.notebook.**is_notebook**()

Return whether or not this function is executed in a notebook environment.

birdy.client.notebook.**monitor**(*execution*, *sleep=3*)

Monitor the execution of a process using a notebook progress bar widget.

Parameters

- **execution** (*WPSExecution instance*) – The execute response to monitor.
- **sleep** (*float*) – Number of seconds to wait before each status check.

birdy.client.notebook.**output2widget**(*output*)

Return notebook widget based on output mime-type.

birdy.client.outputs module

class birdy.client.outputs.**WPSResult**(**args: Any*, ***kwargs: Any*)

Bases: WPSExecution

_make_output(*convert_objects=False*)

_process_output(*output*, *convert_objects=False*)

Process the output response, whether it is actual data or a URL to a file.

Parameters

- **output** (*owslib.wps.Output*)
- **convert_objects** (*bool*) – If True, object_converters will be used.

attach(*wps_outputs*, *converters=None*)

Attach the outputs according to converters.

Parameters

- **wps_outputs** (*dict*)
- **converters** (*dict*) – Converter dictionary {name: object}

get(*asobj=False*)

Return the process response outputs.

Parameters

asobj (*bool*) – If True, object_converters will be used.

birdy.client.utils module

birdy.client.utils.**add_output_format**(*output_dictionary*, *output_identifier*, *as_ref=None*, *mimetype=None*)

Add an output format to an already existing dictionary.

Parameters

- **output_dictionary** (*dict*) – The dictionary (created with create_output_dictionary()) to which this output format will be added.
- **output_identifier** (*str*) – Identifier of the output.
- **as_ref** (*True, False or None*) – Determines if this output will be returned as a reference or not. None for process default.
- **mimetype** (*str or None*) – If the process supports multiple MIME types, it can be specified with this argument. None for process default.

birdy.client.utils.**build_process_doc**(*process*)

Create docstring from process metadata.

birdy.client.utils.**build_wps_client_doc**(*wps*, *processes*)

Create WPSClient docstring.

Parameters

- **wps** (*owslib.wps.WebProcessingService*)
- **processes** (*Dict[str, owslib.wps.Process]*)

Returns
The formatted docstring for this WPSClient

Return type
str

birdy.client.utils.**create_output_dictionary**(*output_identifier*, *as_ref=None*, *mimetype=None*)

Create an output format dictionary.

Parameters

- **output_identifier** (*str*) – Identifier of the output.
- **as_ref** (*True, False or None*) – Determines if this output will be returned as a reference or not. None for process default.
- **mimetype** (*str or None*) – If the process supports multiple MIME types, it can be specified with this argument. None for process default.

Returns
output_dictionary

Return type
dict

birdy.client.utils.**extend_instance**(*obj*, *cls*)

Apply mixins to a class instance after creation.

birdy.client.utils.**filter_case_insensitive**(*names*, *complete_list*)

Filter a sequence of process names into a *known* and *unknown* list.

birdy.client.utils.**format_type**(*obj*)

Create docstring entry for input parameter from an OWSlib object.

birdy.client.utils.**from_owslib**(*value*, *data_type*)

Convert a string into another data type.

birdy.client.utils.**is_embedded_in_request**(*url*, *value*)

Whether or not to encode the value as raw data content.

Returns True if

- value is a file:/// URI or a local path
- value is a File-like instance
- url is not localhost
- value is a File object
- value is already the string content

birdy.client.utils.**pretty_repr**(*obj*, *linebreaks=True*)

Output pretty repr for an Output.

Parameters

- **obj** (*any type*)
- **linebreaks** (*bool*) – If True, split attributes with linebreaks

birdy.client.utils.**py_type**(*data_type*)

Return the python data type matching the WPS dataType.

birdy.client.utils.**to_owslib**(*value*, *data_type*, *encoding=None*, *mimetype=None*, *schema=None*)

Convert value into OWSlib objects.

birdy.ipyleafletwfs package

IpyleafletWFS Module

This module facilitates the use of a WFS service through the ipyleaflet module in jupyter notebooks. It uses owslib to get a geojson out of a WFS service, and then creates an ipyleaflet GeoJSON layer with it.

Dependencies

Ipyleaflet and Ipywidgets dependencies are included in the requirements_extra.txt, at the root of this repository. To install:

```
$ pip install -r requirements_extra.txt
```

Use

This module is to be used inside a jupyter notebook, either with a standard server or through vscode/pycharm. There are notebook examples which show how to use this module and what can be done with it.

The WFS request is filtered by the extent of the visible map, to make large layers easier to work with. Using the on-map 'Refresh WFS layer' button will make a new request for the current extent.

Warning: WFS requests and GeoJSON layers are costly operations to process and render. Trying to load lake layers at the nationwide extent may take a long time and probably crash. The more dense and complex the layer, the more zoomed-in the map extent should be.

Submodules

birdy.ipyleafletwfs.base module

class birdy.ipyleafletwfs.base.**IpyleafletWFS**(*url*, *wfs_version='2.0.0'*)

Bases: `object`

Create a connection to a WFS service capable of geojson output.

This class is a small wrapper for ipylealet to facilitate the use of a WFS service, as well as provide some automation.

Request to the WFS service is done through the owslib module and requires a geojson output capable WFS. The geojson data is filtered for the map extent and loaded as an ipyleaflet GeoJSON layer.

The automation done through build_layer() supports only a single layer per instance is supported.

For multiple layers, used different instances of IpylealetWFS and Ipyleaflet.Map() or use the create_wfsgeojson_layer() function to build your own custom map and widgets with ipyleaflet.

Parameters

- **url** (*str*) – The url of the WFS service
- **wfs_version** (*str*) – The version of the WFS service to use. Defaults to 2.0.0.

Returns
Instance from which the WFS layers can be created.

Return type
IpyleafletWFS

_create_refresh_widget()

_refresh_layer(*placeholder=None*)

Refresh the wfs layer for the current map extent.

Also updates the existing widgets.

Parameters
placeholder (*string*) – Parameter is only there so that button.on_click() will work properly.

_set_widget(*widget_name*, *feature_property*, *src_map*, *textbox*, *widget_position*)

build_layer(*layer_typename*, *source_map*, *layer_style=None*, *feature_property=None*)

Return an ipyleaflet GeoJSON layer from a geojson wfs request.

Requires the WFS service to be capable of geojson output.

Running this function multiple times will overwrite the previous layer and widgets.

Parameters

- **layer_typename** (*string*) – Typename of the layer to display. Listed as Layer_ID by get_layer_list(). Must include namespace and layer name, separated by a colon.

  ex: public:canada_forest_layer

- **source_map** (*Map instance*) – The map instance on which the layer is to be added.
- **layer_style** (*dictionnary*) – ipyleaflet GeoJSON style format, for example *{ 'color': 'white', 'opacity': 1, 'dashArray': '9', 'fillOpacity': 0.1, 'weight': 1 }*. See ipyleaflet documentation for more information.
- **feature_property** (*string*) – The property key to be used by the widget. Use the property_list() function to get a list of the available properties.

clear_property_widgets()

Remove all property widgets from a map.

This function will remove the property widgets from a given map, without affecting other widgets.

Parameters
src_map (*Map instance*) – The map instance from which the widgets are to be removed.

create_feature_property_widget(*widget_name*, *feature_property=None*, *widget_position='bottomright'*)

Create a visualization widget for a specific feature property.

Will create a widget for the layer and source map. Once the widget is created, click on a map feature to have the information appear in the corresponding box. To replace the default widget that get created by the build_layer() function, set the widget_name parameter to 'main_widget'.

Parameters

- **widget_name** (*string*) – Name of the widget. Must be unique or will overwrite existing widget.
- **feature_property** (*string*) – The property key to be used by the widget. Use the property_list() function to get a list of the available properties. If left empty, it will default to the first property attribute in the list.
- **widget_position** (*string*) – Position on the map for the widget. Choose between 'bottomleft', 'bottomright', 'topleft', or 'topright'.

Notes

Widgets created by this function are unique by their widget_name variable.

create_wfsgeojson_layer(*layer_typename*, *source_map*, *layer_style=None*)

Create a static ipyleaflett GeoJSON layer from a WFS service.

Simple wrapper for a WFS => GeoJSON layer, using owslib.

Will create a GeoJSON layer, filtered by the extent of the source_map parameter. If no source map is given, it will not filter by extent, which can cause problems with large layers.

WFS service need to have geojson output.

Parameters

- **layer_typename** (*string*) – Typename of the layer to display. Listed as Layer_ID by get_layer_list(). Must include namespace and layer name, separated by a colon.

  ex: public:canada_forest_layer
- **source_map** (*Map instance*) – The map instance from which the extent will be used to filter the request.
- **layer_style** (*dictionnary*) – ipyleaflet GeoJSON style format, for example *{ 'color': 'white', 'opacity': 1, 'dashArray': '9', 'fillOpacity': 0.1, 'weight': 1 }*. See ipyleaflet documentation for more information.

Returns
GeoJSON layer

Return type
an instance of an ipyleaflet GeoJSON layer.

feature_properties_by_id(*feature_id*)

Return the properties of a feature.

The id field is usually the first field. Since the name is always different, this is the only assumption that can be made to automate this process. Hence, this will not work if the layer in question does not follow this formatting.

Parameters
feature_id (*int*) – The feature id.

Returns
A dictionary of the layer's properties

Return type
Dict

property geojson

Return the imported geojson data in a python object format.

property layer

property layer_list

Return a simple layer list available to the WFS service.

Returns
A List of the WFS layers available

Return type
List

property property_list

Return a list containing the properties of the first feature.

Retrieves the available properties for use subsequent use by the feature property widget.

Returns
A dictionary of the layer properties.

Return type
Dict

remove_layer()

Remove layer instance and it's widgets from map.

birdy.ipyleafletwfs.base.**_map_extent_to_bbox_filter**(*source_map*)

Return formatted coordinates, from ipylealet format to owslib.wfs format.

This function takes the result of ipyleaflet's Map.bounds() function and formats it so it can be used as a bbox filter in an owslib WFS request.

Parameters
source_map (*Map instance*) – The map instance from which the extent will calculated

Returns
Coordinates formatted to WebFeatureService bounding box filter.

Return type
Tuple

6.1.2 Submodules

birdy.dependencies module

Dependencies Module

Module for managing optional dependencies.

Example usage:

```
>>> from birdy.dependencies import ipywidgets as widgets
```

birdy.exceptions module

class birdy.exceptions.**ConnectionError**(**args: Any*, ***kwargs: Any*)

Bases: ClickException

exception birdy.exceptions.**IPythonWarning**

Bases: UserWarning

exception birdy.exceptions.**ProcessCanceled**

Bases: Exception

exception birdy.exceptions.**ProcessFailed**

Bases: Exception

exception birdy.exceptions.**ProcessIsNotComplete**

Bases: Exception

class birdy.exceptions.**UnauthorizedException**(**args: Any*, ***kwargs: Any*)

Bases: ServiceException

birdy.utils module

birdy.utils.**_encode**(*content*, *mimetype*, *encoding*)

Encode in base64 if mimetype is a binary type.

birdy.utils.**delist**(*data*)

If data is a sequence with a single element, returns this element, otherwise return the sequence.

birdy.utils.**embed**(*value*, *mimetype=None*, *encoding=None*)

Return the content of the file, either as a string or base64 bytes.

Returns
encoded content string and actual encoding

Return type
str

birdy.utils.**fix_url**(*url*)

If url is a local path, add a file:// scheme.

birdy.utils.**guess_type**(*url*, *supported*)

Guess the mime type of the file link.

If the mimetype is not recognized, default to the first supported value.

Parameters

- **url** (*str, Path*) – Path or URL to file.
- **supported** (*list, tuple*) – Supported mimetypes.

Return type
mimetype, encoding

birdy.utils.**is_file**(*path*)

Return True if *path* is a valid file.

birdy.utils.**is_opendap_url**(*url*)

Check if a provided url is an OpenDAP url.

The DAP Standard specifies that a specific tag must be included in the Content-Description header of every request. This tag is one of: “dods-dds” | “dods-das” | “dods-data” | “dods-error”

So we can check if the header starts with *dods*.

Note that this might not work with every DAP server implementation.

birdy.utils.**is_url**(*url*)

Return whether value is a valid URL.

birdy.utils.**sanitize**(*name*)

Lower-case name and replace all non-ascii chars by _.

If name is a Python keyword (like *return*) then add a trailing _.

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

W

X

Z