Raven Documentation

Release 0.18.2

David Huard

Apr 16, 2024

CONTENTS

Raven is a server providing access to hydrological modeling and analysis through the Web Processing Service (WPS) standard. It was made to help hydrologists work with climate data and automate some of the tedious work required to calibrate and run hydrological models. Its modeling engine is the Raven hydrological modeling framework, which can emulate a variety of lumped and distributed models.

Raven is part of the birdhouse, a community building a suite of WPS servers supporting climate services and sustainable development goals. The idea is that instead of downloading large volumes of data locally and then analyzing it, part of the analysis can be done remotely by servers, close to the source data. Instead of users sending a plain download request, users send a request for pre-processed data. Work with low-added value can be delegated to a server, and the real research is performed on reduced datasets on local machines.

In this model, scientists need to interact closely with a server to submit requests and poll the server for its response once the job is complete. Because the boilerplate code and formats used to communicate with a server can detract from the science, we've built a generic WPS client interface (see birdy) that hides the WPS protocol behind a native looking python interface. Remote WPS processes can be called just like a python function, returning an asynchronous response whose progress can be easily monitored.

CHAPTER

ONE

USER DOCUMENTATION

1.1 Getting started

- Install birdy with *pip install birdhouse-birdy*
- Connect to a Raven WPS server. You may deploy your own version or use the server hosted at Ouranos.

```
from birdy import WPSClient

url = "https://pavics.ouranos.ca/twitcher/ows/proxy/raven"
wps = WPSClient(url)
```

The `wps` object behaves as a module, holding functions that, instead of being executed on your machine, call a remote process on the server. See the notebook tutorials for examples.

If you don't want to install anything and just try it, go to https://pavics.ouranos.ca/jupyter and login with the *public* account and the *public* password. Note that your notebooks will be publicly visible and modifiable, so don't leave anything valuable there. Also, from time to time we'll reset the public folders, so make sure you keep a local copy of your work.

1.2 Installation

1.2.1 Install from Conda-Forge (suggested)

Create an Anaconda environment named *ravenwps-env*:

```
$ conda env create -n ravenwps-env python=3.7
$ source activate ravenwps-env
```

This should now prepend the environment to your shell commands (ie: *(ravenwps-env) $*). Now install directly from *conda-forge*:

```
(ravenwps-env) $ conda install -c conda-forge raven-wps
```

1.2.2 Install from GitHub

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

```
$ git clone https://github.com/Ouranosinc/raven.git
$ cd raven
```

Environment Setup with Anaconda (macOS/Linux)

Create Conda environment named *raven*:

```
$ conda env create -n ravenwps-env -f environment.yml
# or alternatively,
$ make conda_env
```

The environment can then be activated with:

```
$ source activate ravenwps-env
```

This should now prepend the environment to your shell commands (ie: *(ravenwps-env) $*).

Environment Setup using System Libraries and Sources (Linux)

Warning: This approach is not formally supported, but is presently working for the time being. It is up to the user to install the *raven* model and *ostrich* model optimization binaries. Those can be downloaded from source via the following links:

- RAVEN: http://raven.uwaterloo.ca/Downloads.html
- OSTRICH: https://github.com/usbr/ostrich/

Note: While the following shows how to install *raven* for an Deb-based Linux, if the OS-equivalent dependencies are available to Python, *raven* should be able to run on any modern operating system (macOS/Windows/*nix).

First we need to install several system libraries that RavenWPS and RavenPy depend upon and make a virtual environment:

```
$ sudo apt-get install libhdf5-dev netcdf-bin libnetcdf-dev libgdal-dev libproj-dev␣
↪libgeos-dev libspatialindex-dev python3-dev
$ pip3 install virtualenv
$ virtualenv ravenwps-env
$ . ravenwps-env/bin/activate
```

We then need to install the *ravenpy* python library from sources:

```
(ravenwps-env) $ git clone https://github.com/CSHS-CWRA/RavenPy/
(ravenwps-env) $ pip install RavenPy/.[gis]
(ravenwps-env) $ pip install RavenPy/. --verbose --install-option="--with-binaries"
```

If we want to perform an interactive/editable installation for dev purposes, substitute the following for the final installation command:

```
(ravenwps-env) $ pip install -e RavenPy/. --verbose --install-option="--with-binaries"
```

1.2.3 Installing and Launching RavenWPS

Now we can install the raven-wps service:

```
(ravenwps-env) $ pip install -e .
# or alternatively,
(ravenwps-env) $ make install
```

For development you can use this command:

```
(ravenwps-env) $ pip install -e .[dev]
# or alternatively,
(ravenwps-env) $ make develop
```

Then clone the Raven Test Data repo somewhere on your disk:

```
(ravenwps-env) $ git clone https://github.com/Ouranosinc/raven-testdata.git
```

You can then run the test suite by doing:

```
(ravenwps-env) $ export RAVENPY_TESTDATA_PATH=/path/to/raven-testdata
(ravenwps-env) $ pytest
```

Start Raven PyWPS service

After successful installation you can start the service using the `raven` command-line:

```
(ravenwps-env) $ raven-wps --help # show help
(ravenwps-env) $ raven-wps start  # start service with default configuration
# or alternatively,
(ravenwps-env) $ raven-wps start --daemon # start service as daemon
loading configuration
forked process id: 42
```

The deployed WPS service is by default available on:

http://localhost:9099/wps?service=WPS&version=1.0.0&request=GetCapabilities.

You can find which process uses a given port using the following command (here for port 5000)::

```
$ netstat -nlp | grep :5000
```

Check the log files for errors:

```
$ tail -f  pywps.log
```

... or do it the lazy way

You can also use the `Makefile` to start and stop the service:

```
(ravenwps-env) $ make start
(ravenwps-env) $ make status
(ravenwps-env) $ tail -f pywps.log
(ravenwps-env) $ make stop
```

You can also run Raven as a Docker container.

1.2.4 Use Ansible to deploy Raven on your System

Use the Ansible playbook for PyWPS to deploy Raven on your system.

1.3 Configuration

1.3.1 Command-line options

You can overwrite the default PyWPS configuration by using command-line options. See the Raven help which options are available:

```
$ raven start --help
--hostname HOSTNAME        hostname in PyWPS configuration.
--port PORT                port in PyWPS configuration.
```

Start service with different hostname and port:

```
$ raven start --hostname localhost --port 5001
```

1.3.2 Use a custom configuration file

You can overwrite the default PyWPS configuration by providing your own PyWPS configuration file (just modifiy the options you want to change). Use one of the existing `sample-*.cfg` files as example and copy them to `etc/custom.cfg`.

For example change the hostname (*demo.org*) and logging level:

```
$ cd raven
$ vim etc/custom.cfg
$ cat etc/custom.cfg
[server]
url = http://demo.org:9099/wps
outputurl = http://demo.org:9099/outputs

[logging]
level = DEBUG
```

Start the service with your custom configuration:

```
# start the service with this configuration
$ raven start -c etc/custom.cfg
```

1.4 Notebooks

Most notebooks have now been migrated to the RavenPy repository available on GitHub.

Those that remain are related to geoprocessing:

1.4.1 Geoprocessing

Region Selection and Map Preview with Ipyleaflet

[1]:
```
# Import the necessary libraries to format, send, and parse our returned results
import os

import birdy
import geopandas as gpd
import ipyleaflet
import ipywidgets
```

If your `notebook` is version prior to `5.3`, you might need to run this command `jupyter nbextension enable --py --sys-prefix ipyleaflet`. For more information see https://ipyleaflet.readthedocs.io/en/latest/installation.html.

[2]:
```
# Create WPS instances
# Set environment variable WPS_URL to "http://localhost:9099" to run on the default␣
↪local server
pavics_url = "https://pavics.ouranos.ca"
raven_url = os.environ.get("WPS_URL", f"{pavics_url}/twitcher/ows/proxy/raven/wps")

raven = birdy.WPSClient(raven_url)
```

[3]:
```
# Build an interactive map with ipyleaflet

initial_lat_lon = (48.63, -74.71)

leaflet_map = ipyleaflet.Map(
    center=initial_lat_lon,
    basemap=ipyleaflet.basemaps.OpenTopoMap,
)

# Add a custom zoom slider
zoom_slider = ipywidgets.IntSlider(description="Zoom level:", min=1, max=10, value=6)
ipywidgets.jslink((zoom_slider, "value"), (leaflet_map, "zoom"))
widget_control1 = ipyleaflet.WidgetControl(widget=zoom_slider, position="topright")
leaflet_map.add_control(widget_control1)

# Add a marker to the map
marker = ipyleaflet.Marker(location=initial_lat_lon, draggable=True)
leaflet_map.add_layer(marker)
```

[4]:
```
# Add an overlay widget

html = ipywidgets.HTML("""Hover over a feature!""")
html.layout.margin = "0px 10px 10px 10px"

control = ipyleaflet.WidgetControl(widget=html, position="bottomleft")
leaflet_map.add_control(control)


def update_html(feature, **kwargs):
    html.value = """
        <h2><b>USGS HydroBASINS</b></h2>
        <h4>ID: {}</h4>
        <h4>Upstream Area: {} sq. km.</h4>
        <h4>Sub-basin Area: {} sq. km.</h4>
    """.format(
        feature["properties"]["id"],
        feature["properties"]["UP_AREA"],
        feature["properties"]["SUB_AREA"],
    )
```

[5]:
```
# Load the map in the notebook
leaflet_map
```

[5]:
```
Map(center=[48.63, -74.71], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪'zoom_in_title', 'zoom_...
```

Before continuing!

Try dragging and placing the marker at the mouth of a river, over a large lake such as Lac Saint Jean (next to Alma, east of the initial marker position), or anywhere else within North America.

[6]:
```
user_lonlat = list(reversed(marker.location))
user_lonlat
```

[6]:
```
[-74.71, 48.63]
```

[7]:
```
# Get the shape of the watershed contributing to flow at the selected location.
resp = raven.hydrobasins_select(location=str(user_lonlat), aggregate_upstream=True)
```

[8]:
```
# Before continuing, wait for the process above to finish.

# Extract the URL of the resulting GeoJSON feature
feat = resp.get(asobj=False).feature
gdf = gpd.read_file(feat)
gdf
```

[8]:
```
                                    id    HYBAS_ID   NEXT_DOWN   NEXT_SINK  \
0  USGS_HydroBASINS_lake_na_lev12.99410  7129001061  7120323452  7120034520

   DIST_SINK  DIST_MAIN  SUB_AREA  UP_AREA       PFAF_ID SIDE  LAKE  ENDO  \
0      517.6      517.6    8470.5   8773.1  725209301000    L    56     0

   COAST  ORDER   SORT                                           geometry
0      0      2  99410  POLYGON ((-75.18990 47.99540, -75.18190 47.994...
```

[9]:
```
# Add this GeoJSON to the map above!
# Scroll back up after executing this cell to see the watershed displayed on the map.
user_geojson = ipyleaflet.GeoData(
    geo_dataframe=gdf,
    style={
        "color": "blue",
        "opacity": 1,
        "weight": 1.9,
        "fillOpacity": 0.5,
    },
    hover_style={"fillColor": "#b08a3e", "fillOpacity": 0.9},
)

leaflet_map.add_layer(user_geojson)
user_geojson.on_hover(update_html)
```

Subsetting climate variables over a watershed

Hydrological models are driven by precipitation, temperature and a number of other variables depending on the processes that are simulated. These variables are typically provided by networks of weather stations. For practicality, these point values are often interpolated over space to create *gridded products*, that is, arrays of climate variables over regular coordinates of time and space.

Global hydrological models however require time series of average climate variables over the entire watersheds. When the watershed includes multiples stations, or covers multiple grid cells, we first need to average these multiple stations or grids to yield a single value per time step. The Raven modeling framework can work directly with gridded datasets, provided the configuration includes the weights to apply to the array. For example, all grid cells outside the watershed could be given weights of 0, while all grid cells inside given a weight proportional to the area of the grid that is inside the watershed.

While there are now utilities to work with grid weights, it is usually more computationally efficient to feed Raven sub-watershed averages. Here we fetch a watershed outline from a geospatial data server, then use the Finch server to

compute the watershed average.

[1]:
```
# Import the necessary libraries.
import datetime as dt
import os

import birdy
import geopandas as gpd
```

[2]:
```
# Set the links to the servers.
# Note that if Finch is a remote server, Raven needs to be accessible on the next␣
↪because some cells
# below use the output from Raven processes to feed into Finch.

url_finch = os.environ.get(
    "FINCH_WPS_URL", "https://pavics.ouranos.ca/twitcher/ows/proxy/finch/wps"
)

url_raven = os.environ.get(
    "WPS_URL", "https://pavics.ouranos.ca/twitcher/ows/proxy/raven/wps"
)

# Establish client connexions to the remote servers
finch = birdy.WPSClient(url_finch)
raven = birdy.WPSClient(url_raven)
```

Extracting the watershed contour and sub-basins identifiers

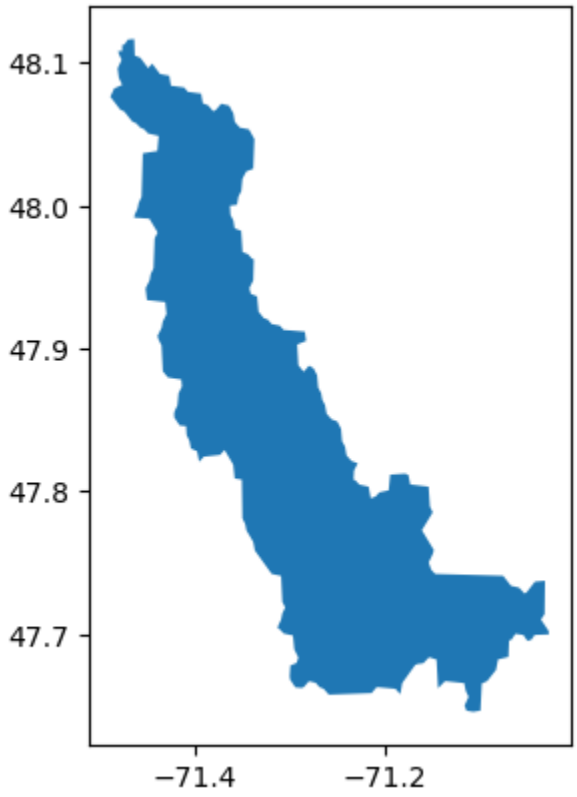
Let's try to identify a sub-basin feeding into Lake Kénogami. We'll start by launching a process with Raven to find the upstream watersheds. The process, called `hydrobasins_select`, takes as an input geographical point coordinates, finds the HydroSheds sub-basin including this point, then looks up into the HydroSheds database to find all upstream sub-basins. It returns the polygon of the watershed contour as a GeoJSON file, as well as a list of all the sub-basins IDs within the watershed.

[3]:
```
# Send a request to the server and get the response.
hydrobasin_resp = raven.hydrobasins_select(
    location="-71.41, 47.96", aggregate_upstream=True
)

# Wait for the process to complete before continuing with calculations.
```

[4]:
```
# Collecting the response: the watershed contour and the sub-basins ids
feature_url, sb_ids = hydrobasin_resp.get()
feature, subbasin_ids = hydrobasin_resp.get(asobj=True)
```

[5]:
```
# Plot our vector shapefile
df = gpd.GeoDataFrame.from_file(feature_url)
df.plot()

display(f"Number of subbasins: {len(subbasin_ids)}")
```

```
'Number of subbasins: 4'
```



Subsetting a gridded climate dataset

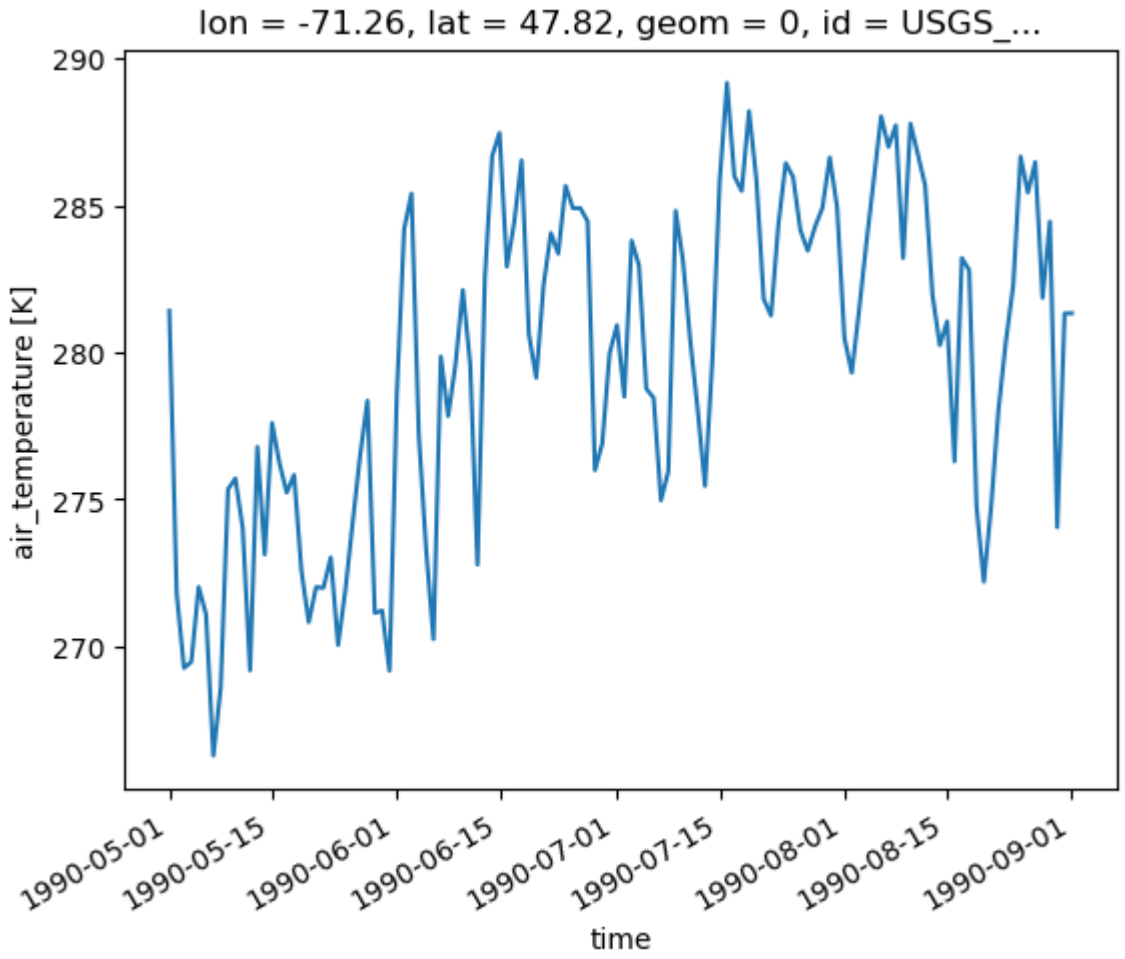
We can then use this watershed outline to average climate data. The watershed shape is given as a GeoJSON file to the `average_polygon` process, along with the gridded file to average.

[6]:
```
# Compute the watershed temperature average.
nc_file = "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/dodsC/birdhouse/testdata/
↪xclim/NRCANdaily/nrcan_canada_daily_tasmin_1990.nc"

resp = finch.average_polygon(
    resource=str(nc_file),
    shape=feature_url,
    tolerance=0.1,
    start_date=dt.datetime(1990, 5, 1),
    end_date=dt.datetime(1990, 9, 1),
)
```

[7]:
```
# Get the output files and plot the temperature time series averaged over the sub-basins.
ds, meta = resp.get(asobj=True)
ds.tasmin.plot()
```

```
Downloading to /tmp/tmp23yzts38/nrcan_canada_daily_tasmin_1990_avg.nc.
```

[7]: [<matplotlib.lines.Line2D at 0x7f51d1ffd050>]



1.5 Developer Guide

Warning: To create new processes look at examples in Emu.

1.5.1 Re-create a fresh environment

```
$ make stop  # in case you previously did 'make start'
$ conda deactivate  # exit the current 'raven' conda env so we can destroy it
$ conda env remove -n raven  # destroy the current conda env to recreate one from scratch
$ conda env create -f environment.yml
$ conda activate raven
$ make develop  # install raven-wps and additional dev tools
```

1.5.2 Building the docs

First install dependencies for the documentation:

```
$ make develop
```

Run the Sphinx docs generator:

```
$ make docs
```

1.5.3 Running tests

Run tests using pytest.

First activate the `raven` Conda environment and install `pytest`.

```
$ source activate raven
$ pip install -r requirements_dev.txt  # if not already installed
OR
$ make develop
```

Run quick tests (skip slow and online):

```
$ pytest -m 'not slow and not online'"
```

Run all tests:

```
$ pytest
```

Check PEP8:

```
$ flake8
```

1.5.4 Run tests the lazy way

Do the same as above using the `Makefile`.

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

1.5.5 Running notebooks tests

Assuming that the `raven` conda env has already been created and is up-to-date and raven-wps has been installed with `make develop`:

```
   # start local raven-wps server to test against
   $ make start  # remember to make stop once done

   # to test all notebooks
   $ make test-notebooks

Or::

   # to test a single notebook (note the .run at the end of the notebook path)
   $ make docs/source/notebooks/Subset_climate_data_over_watershed.ipynb.run
```

The notebooks may also require other WPS services (Finch and Flyingpigeon). By default these are from the production server but we can point the notebooks to local servers if needed for development purposes:

```
# to test all notebooks
$ make FLYINGPIGEON_WPS_URL=http://localhost:8093 FINCH_WPS_URL=http://localhost:5000␣
↪test-notebooks
```

Or:

```
# to test a single notebook (note the .run at the end of the notebook path)
$ make FLYINGPIGEON_WPS_URL=http://localhost:8093 FINCH_WPS_URL=http://localhost:5000␣
↪docs/source/notebooks/Subset_climate_data_over_watershed.ipynb.run
```

If instead we want to run the notebooks against the production raven-wps server or any other raven-wps servers:

```
# to test all notebooks
$ make WPS_URL=https://pavics.ouranos.ca/twitcher/ows/proxy/raven/wps test-notebooks
```

Or:

```
# to test juste 1 notebook (note the .run at the end of the notebook path)
$ make WPS_URL=https://pavics.ouranos.ca/twitcher/ows/proxy/raven/wps docs/source/
↪notebooks/Subset_climate_data_over_watershed.ipynb.run
```

We can also override all three of the server variables (WPS_URL, FINCH_WPS_URL, FLYINGPIGEON_WPS_URL) to pick and choose any servers/services from anywhere we want.

1.5.6 Starting local Jupyter server to edit/develop notebooks

Assuming that the `raven` conda env has already been created and is up-to-date and raven-wps has been installed with `make develop`:

```
# start local raven-wps server to test against
$ make start  # remember to make stop once done

# to start local jupyter notebook server listing all current notebooks
$ make notebook  # Control-C to terminate once done

# Can also override all three WPS_URL, FINCH_WPS_URL and FLYINGPIGEON_WPS_URL here as␣
↪well,
# just like 'make test-notebooks' to be able to pick and choose any servers anywhere we␣
↪want.

# By overriding these variables at the 'make notebook' step, we will not need to
# override them one by one in each notebook as each notebook will also look
# for those variables as environment variables.
```

1.5.7 Bulk refresh all notebooks output

This automated refresh only works for notebooks that passed `make test-notebooks` above. For those that failed, manually starting a local Jupyter server and refresh them manually.

Assuming that the `raven` conda env has already been created and is up-to-date and raven-wps has been installed with `make develop`:

```
# start local raven-wps server to test against
$ make start  # remember to make stop once done

# to refresh all notebooks
$ make refresh-notebooks
```

Or:

```
# to refresh a single notebook (note the .refresh at the end of the notebook path)
$ make docs/source/notebooks/Assess_probabilistic_flood_risk.ipynb.refresh

# Can also override all three of the server variables (WPS_URL, FINCH_WPS_URL and␣
↪FLYINGPIGEON_WPS_URL) here as well,
# just like 'make test-notebooks' to be able to pick and choose any servers/services␣
↪from anywhere we want.
```

1.5.8 Prepare a release

Update the Conda specification file to build identical environments on a specific OS.

Note: You should run this on your target OS, in our case Linux.

```
$ conda env create -f environment.yml
$ source activate raven
$ make clean
$ make install
$ conda list -n raven --explicit > spec-file.txt
```

1.5.9 Bump a new version

Make a new version of Raven 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 0.8.1 patch`
- Do it: `bumpversion --new-version 0.8.1 patch`
- ... or: `bumpversion --new-version 0.9.0 minor`
- Push it: `git push`
- Push tag: `git push --tags`

See the bumpversion documentation for details.

1.6 Processes

1.7 Changes

1.7.1 0.18.3 (unreleased)

- Added a GitHub Workflow to test the Dockerfile recipe configuration for RavenWPS.
- Cleaned up the Dockerfile recipe configuration for RavenWPS, now using Gunicorn for service management.

1.7.2 0.18.2 (2023-07-06)

- Removed pin on *owslib* below v0.29 and pin on *fiona* below v2.0.
- Added a GitHub Workflow to test against macOS builds.
- Adapted zonal statistics processes to support the latest *fiona* and *zonalstats* API changes.

1.7.3 0.18.1 (2023-05-23)

- Removed obsolete components related to HPC interfaces (#477)
- Added Python3.11 to supported versions with a build for Python3.11 in CI (#477)
- Adjusted ReadMe to reflect recent significant changes (#477)
- Updated deprecated GitHub Actions (#477)

1.7.4 0.18.0 (2023-05-23)

Major Changes

- *singularity* components have been removed from `raven` (#470)
- Removed Raven WPS capabilities for hydrological modelling, graphing and forecasting (moved to RavenPy) (#464)
- Removed notebooks and migrated to Ravenpy. Adapted them to the new Ravenpy configuration (#464)
- Removed all tests related to Raven WPS modelling (#464)
- Raise error message if shape area for NALCMS zonal stats raster is above 100,000 km2. (#473)

1.7.5 0.17.1 (2023-04-04)

Internal Changes

- Dockerfile configuration now uses Python3.8 and *condaforge/mambaforge* base image (#466)
- *pandas* is temporarily pinned below v2.0 (#466)

1.7.6 0.17.0 (2023-02-28)

Major Changes

- Updated testing ensemble to use *pytest-xdist* (#448)
- Updated *RavenPy* to v0.11.0, *raven-hydro* to v3.6, and *fiona* to v1.9 (#461)
- Modified several geospatial processes to adapt to new APIs (#461)
- Datetime signatures for some models used in notebooks have been adjusted/fixed (#453)

Internal Changes

- Makefile updates to better perform notebook refresh actions (#459)
- Pre-commit style updates (#446, #447, #449, #461)
- Use *provision-with-micromamba* GitHub Action in CI workflows (#462)

1.7.7 0.16.0 (2022-11-01)

Major Changes

- Added data assimilation workbook (#421)
- Overhaul of all existing notebooks within documentation (#424)
- Added notebooks for case-study paper (#435)
- Update to RavenPy 0.8.1 (#439)
- Dropped support for Python3.7 (#439)

Internal Changes

- Added pre-commit.ci to workflows and updated black formatting (#428 and #429)
- Adjust documentation to remove sphinx-autoapi artefact files and set ReadTheDocs to fail_on_warning (#439)
- **Set pre-commit to run new correction and verification hooks (#439):**
  - pyupgrade: Ensure that coding style uses Python3.8+ conventions
  - pygrep: Checks for bare *noqa* comments and malformed code blocks in documentation
  - nbqa: Black, Isort, PyUpgrade now runs over notebooks
  - check-manifest: Ensure relevant modules and data are explicitly installed
  - black + blackdoc + yamllint: Clean up code, code examples within documentation and reformat yaml files for readability
  - check-jsonschema: Verify that GitHub and ReadTheDocs workflows are valid
- Added a Zenodo/DOI configuration

1.7.8 0.15.1 (2022-01-14)

- Modified handling for GDAL to better support conda-build configuration
- Update to RavenPy 0.7.8
- Upgrade to PyWPS 4.5.1

1.7.9 0.15.0 (2021-12-22)

- Update to RavenPy 0.7.7
- Update required Python consistently to v3.7+
- Set development status to Beta.
- Replace pip-installed packages with conda-forge equivalents.

1.7.10 0.14.2 (2021-09-03)

- Update to RavenPy 0.7.4 (pin climpred below version 2.1.6)
- Fixed a process-breaking bug in *wps_hydrobasins_shape_selection*

1.7.11 0.14.1 (2021-08-31)

- Update to RavenPy 0.7.3 (pin xclim version 0.28.1)

1.7.12 0.14.0 (2021-08-30)

- Update to RavenPy 0.7.2
- Use new OWSlib WFS topological filters
- More informative install documentation
- Upgrade to PyWPS 4.4.5

1.7.13 0.13.0 (2021-05-14)

- Update RavenPy to 0.5.1
- Remove the `name` (watershed name) from the WPS interface for Raven processes
- Add `random_numbers` WPS param to pass optional `OstRandomNumbers.txt` file to Ostrich processes
- Add error handlers for regionalisation and climatology processes

1.7.14 0.12.1 (2021-04-16)

- Fix bug where the name of configuration files was used, while the client transmission of data does not carry the file name.
- Update notebooks
- Move draft notebooks to sandbox

1.7.15 0.12.0 (2021-04-14)

- Update RavenPy to 0.4.2
- Migrate utilities to RavenPy
- Add notebook for advanced forecasting
- Add notebook for probabilistic flood assessment
- Option to skip slow tests
- Add climpred verification WPS service
- Pre-commit hooks
- Install from conda Raven and Ostrich libraries
- Support passing HRUs

- Use scale/offset instead of linear_transform
- Enable GitHub CI
- Fix broken notebooks
- Improve error reporting by including stack trace in error messages.

1.7.16 0.11.x (2021-02-01)

- Add processes to run hydrological simulations on ECCC GEPS forecasts/hindcasts
- Add process to create forecast graphic
- Add first basic data assimilation utilities
- Factor out extra project RavenPy (at version 0.2.2), using Raven 3.0.1
- Upgrade to xclim +0.23.0
- Upgrade to xarray +0.16.2
- Add configuration options: `deaccumulate`
- Clean notebooks
- Pin RavenPy to 0.3.0
- Pin owslib to 0.21
- Fix RavenC binaries installation for deployment
- Move some tests to RavenPy
- Regionalization data is now bundled with RavenPy
- Upgrade and pin PyWPS to 4.4.1
- Factor out most GIS functions to RavenPy (0.3.0)
- Add `nalcms-zonal-stats-raster` process using `pymetalink`
- Simplify documentation build environment.

1.7.17 0.10.x (2020-03-09) Oxford

- `suppress_ouput` also triggers `:DontWriteWatershedStorage`
- Added support for ERA5 (hourly), NRCan and CANOPEX datasets
- Support linear transforms (unit changes)
- Calibration now uses :SuppressOutput by default
- Added options for rain_snow_fraction, evaporation and ow_evaporation
- Updated Raven version to 295
- Support passing shapes as zip files

1.7.18 0.9.x (2019-11-11)

- Return configuration files used to run model in a zip archive

1.7.19 0.8.x (2019-10-22)

- Added more documentation for users
- Fixed reprojection errors in GIS utilities
- Specified HydroBASINS in lieu of HydroSHEDS in processes
- Optimized memory usage in ReadTheDocs builds when using Sphinx autodoc by employing mock
- Cleaner GeoJSON outputs for many subsetting processes
- Employed ipyleaflets for notebook-based web-maps
- Run py.test on notebooks from local or remote server

1.7.20 0.7.x (2019-06-25)

- Regionalization database
- Graphics for frequency analysis
- Many new notebook tutorials
- Bug fixes

1.7.21 0.6.x (2019-06-05)

- Regionalization process allowing the estimation of parameters of ungauged watersheds
- Added time series analysis processes, including frequential analysis
- Added processes creating graphics
- GIS processes now use GeoServer capabilities
- Docker configuration

1.7.22 0.5.0 (2019-04-12)

- Added watershed geospatial analysis processes - Hydroshed basin selection (with upstream contributors) - Watershed properties - DEM property analysis - Land-use property analysis
- Added multi-parameter parallel simulations
- Added multi-model parallel simulations
- Added multi-bassin parallel simulations

1.7.23 0.4.0 (2019-03-12)

- Added model calibration processes using Ostrich
- Added support for launching a singularity image
- Added library functions for model regionalization

1.7.24 0.3.0 (2019-01-24)

- Adds process for MOHYSE emulator
- Adds process for HBV-EC emulator

1.7.25 0.2.0 (2018-11-29) Washington

- Provides generic RAVEN framework configuration
- Process for GR4J-Cemaneige emulator
- Process for HMETS emulator

CHAPTER

TWO

CREDITS

This project was funded by the CANARIE research software program.

Hydrological models are based on the Raven modeling framework.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search