RavenPy Documentation

Release 0.14.1

David Huard

May 14, 2024

CONTENTS:

CHAPTER

ONE

RAVENPY

A Python wrapper to setup and run the hydrologic modelling framework Raven.

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

RavenPy is a Python wrapper for Raven, accompanied by utility functions that facilitate model configuration, calibration, and evaluation.

Raven is an hydrological modeling framework that lets hydrologists build hydrological models by combining different hydrological processes together. It can also be used to emulate a variety of existing lumped and distributed models. Model structure, parameters, initial conditions and forcing files are configured in text files, which Raven parses to build and run hydrological simulations. A detailed description about modeling capability of Raven can be found in the docs.

RavenPy provides a Python interface to Raven, automating the creation of configuration files and allowing the model to be launched from Python. Results, or errors, are automatically parsed and exposed within the programming environment. This facilitates the launch of parallel simulations, multi-model prediction ensembles, sensitivity analyses and other experiments involving a large number of model runs.

Note that version 0.12 includes major changes compared to the previous 0.11 release, and breaks backward compatibility. The benefits of these changes are a much more intuitive interface for configuring and running the model.

1.1 Features

- Configure, run and parse Raven outputs from Python
- Utility command to create grid weight files
- Extract physiographic information about watersheds
- Algorithms to estimate model parameters from ungauged watersheds
- Exposes outputs (flow, storage) as *xarray.DataArray* objects

1.2 Install

Please see the detailed installation docs.

1.3 Acknowledgements

RavenPy's development has been funded by CANARIE and Ouranos and would be not be possible without the help of Juliane Mai and James Craig.

This package was created with Cookiecutter and the Ouranosinc/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Anaconda Python Installation

For many reasons, we recommend using a Conda environment to work with the full RavenPy installation. This implementation is able to manage the harder-to-install GIS dependencies, like *GDAL*.

Begin by creating an environment:

```
$ conda create -c conda-forge --name ravenpy
```

The newly created environment must then be activated:

```
$ conda activate ravenpy
```

RavenPy can then be installed directly via its *conda-forge* package by running:

```
(ravenpy) $ conda install -c conda-forge ravenpy
```

This approach installs the Raven binary directly to your environment *PATH*, as well as installs all the necessary Python and C libraries supporting GIS functionalities.

2.2 Python Installation (pip)

Warning: In order to compile the Raven model (provided by the *raven-hydro* package, a C++ compiler (*GCC*, *Clang*, *MSVC*, etc.) and either *GNU Make* (Linux/macOS) or *Ninja* (Windows) must be exposed on the *$PATH*.

Warning: The Raven model also requires that NetCDF4 libraries are installed on the system, exposed on the *$PATH*, and discoverable using the *FindNetCDF.cmake* helper script bundled with *raven-hydro*.

On Linux, this can be provided by the *libnetcdf-dev* system library; On macOS by the *netcdf* homebrew package; And on Windows by using UNIDATA's pre-built binaries.

In order to perform this from Ubuntu/Debian:

```
$ sudo apt-get install gcc libnetcdf-dev gdal proj geos
```

Then, from your python environment, run:

```
$ pip install ravenpy[gis]
```

If desired, the core functions of *RavenPy* can be installed without its GIS functionalities as well. This implementation of RavenPy is much lighter on dependencies and can be installed easily with *pip*, without the need for *conda* or *virtualenv*.

```
$ pip install ravenpy
```

2.3 Using A Custom Raven Model Binary

If you wish to install the *Raven* model, either compiling the *Raven* binary from sources for your system or installing the pre-built binary offered by UWaterloo, we encourage you to consult the *Raven* documentation (Raven Downloads).

Once downloaded/compiled, the binary can be pointed to manually (as an absolute path) by setting the environment variable `RAVENPY_RAVEN_BINARY_PATH` in the terminal/command prompt/shell used at runtime.

```
$ export RAVENPY_RAVEN_BINARY_PATH=/path/to/my/custom/raven
```

2.4 Customizing remote service datasets

A number of functions and tests within *RavenPy* are dependent on remote services (THREDDS, GeoServer) for providing climate datasets, hydrological boundaries, and other data. These services are provided by Ouranos through the PAVICS project and may be subject to change in the future.

If for some reason you wish to use alternate services, you can set the following environment variables to point to your own instances of THREDDS and GeoServer:

```
$ export RAVENPY_THREDDS_URL=https://my.domain.org/thredds
$ export RAVENPY_GEOSERVER_URL=https://my.domain.org/geoserver
```

2.5 Development Installation (from sources)

The sources for *RavenPy* can be obtained from the GitHub repo:

```
$ git clone git://github.com/CSHS-CWRA/ravenpy
```

You can then create and activate your Conda environment by doing:

```
$ cd /path/to/ravenpy
$ conda env create -f environment.yml
$ conda activate ravenpy
```

You can then install RavenPy with:

```
# for the python dependencies
(ravenpy) $ pip install --editable ".[dev,gis]"
```

Install the pre-commit hook (to make sure that any code you contribute is properly formatted):

```
(ravenpy-env) $ pre-commit install
```

If everything was properly installed the test suite should run successfully:

```
(ravenpy-env) $ pytest tests
```

CHAPTER

THREE

USAGE

RavenPy is designed to make Raven easier to use from an interactive Python programming environment. It can:

- execute Raven on an existing model configuration;
- expose simulation results from one or more simulations as `xarray.Datasets`;
- create, modify and write model configurations.

In particular, RavenPy includes eight pre-configured model *emulators*.

3.1 Running Raven from existing configuration files

To run Raven using existing configuration files (`.rv*`), simply call the `run` function with the name of the configuration file and the path to the directory storing the RV files:

```
from ravenpy import run

output_path = run(modelname, configdir)
```

`run` simply returns the directory storing the model outputs. If Raven emits warnings, those will be printed in the console. If Raven raises errors, `run` will halt the execution with a `RavenError` and display the error messages in `Raven_errors.txt`.

3.2 Exposing model outputs as Python objects

The model outputs can be read with the `OutputReader` class:

```
from ravenpy import OutputReader

out = OutputReader(run_name, path=output_path)
out.hydrograph.q_sim
out.storage["Ponded Water"]
```

Note that this works only if simulated variables are stored as netCDF files, that is, if the `rvi` file includes a `:WriteNetCDFFormat` command. For more details, see *Accessing simulation results with OutputReader*.

The class `EnsembleReader` does the same for an ensemble of model outputs, concatenating netCDF outputs along a new dimension:

```
from ravenpy import EnsembleReader

out = EnsembleReader(
    run_name,
    paths=[
        output_path,
    ],
    dim="ensemble_dim",
)
out.hydrograph.q_sim
out.storage["Ponded Water"]
```

For more info, see *Accessing results from multiple simulations using EnsembleReader*.

3.3 Configuring emulators

Ravenpy comes packaged with pre-configured emulators, that is, Raven model configurations that can be modified on the fly. These emulators are made out of symbolic expressions, connecting model parameters to properties and coefficients. For example, the code below creates a model configuration for emulated model GR4JCN using the parameters given, as well as a `Gauge` configuration inferred by inspecting the `meteo.nc` file.

```
from ravenpy.config.emulators import GR4JCN
from ravenpy.config.commands import Gauge

gr4jcn = GR4JCN(
    params=[0.5, -3.0, 400, 1.0, 17, 0.9], Gauge=[Gauge.from_nc("meteo.nc")]
)
```

Note that `Gauge.from_nc` will only find the required information if the netCDF file complies with the `CF-Convention`. Otherwise, additional parameters have to be provided to complete the configuration.

Ravenpy includes a suite of eight emulators:

- GR4JCN (GR4J-Cemaneige)
- HMETS
- HBVEC
- Mohyse
- Blended
- CanadianShield
- SACSMA
- HYPR

3.4 Running an emulator

The RV files for the emulator above can be inspected using the `rvi`, `rvh`, `rvp`, `rvc` and `rvt` properties, e.g. `print(gr4jcn.rvt)` will show the `rvt` file as it would be written to disk. Configuration files can then be written to disk using `gr4jcn.write_rv(workdir, modelname)`, and the model launched using the `run` function introduced before.

For convenience, `ravenpy` also proposes the `Emulator` class, designed to streamline the execution of the model and the retrieval of the results.

```
from ravenpy import Emulator

e = Emulator(config=gr4jcn, workdir="/tmp/gr4jcn/run_1")
out = e.run()
out.hydrograph.q_sim
```

If no `workdir` is given, a temporary directory will be created, available from The `Emulator.workdir` property. `Emulator` also has `resume` method that returns a copy of the original configuration with the internal states and start date set to the values stored in the `solution.rvc` file, which can then be used to launch another simulation following the first one. For more on this, see *The Emulator class*.

For more information on model configuration, see *Model Configuration*.

CHAPTER

FOUR

MODEL CONFIGURATION

Raven lets hydrologists customize how models are defined, and provides examples of model configurations that can reproduce outputs from known hydrological models: HBV-EC, HMETS, Sacramento-SMA, etc. RavenPy can be used to create Python objects representing those emulators, facilitating the setup of complex modelling experiments. The following is a walk-through covering the basics of emulator configuration.

4.1 The `Config` class

RavenPy writes configuration files using the `Config` class. `Config` recognizes most Raven commands, and has mechanisms to parse and validate the inputs, and then write them in the appropriate RV file. For example:

```
from ravenpy.config import Config
conf = Config(StartDate="2023-03-31")
conf.rvi
```

`Config` has recognized `StartDate` as a Raven command, and knows it should appear in the `rvi` file as a line starting with `:StartDate` followed by a date in ISO format. `StartDate` could equally have been given as a `datetime.date` or `datetime.datetime` object, and `Config` would have parsed it correctly.

Many other Raven commands are known to `Config` – to see what commands are supported, and the type of inputs they expect, consult the class docstring with `help(Config)`. All these commands are set to a default value of `None`, and won't appear in RV files unless they are given actual values. Note that if you plan on using the `OutputReader`, make sure you set `WriteNetcdfFormat` to True.

Some Raven commands take string values, others floats, integers or dates, but some also expect lists of arguments, such as `:EvaluationMetrics`:

```
Config(EvaluationMetrics=["NASH_SUTCLIFFE", "RMSE"]).rvi
```

Some commands require more complex structures, for example, the configuration for `CustomOutput` is given as a dictionary with arguments `time_per`, `stat`, `variable` and `space_agg`:

```
Config(CustomOutput=[{"time_per": "MONTHLY", "stat": "AVERAGE", "variable": "SNOW",
↪"space_agg": "BY_HRU"}]).rvi
```

All Raven commands with inputs more complex than single values or simple lists are defined in `ravenpy.config.commands`. Their names match exactly with the Raven command names described in the Raven documentation. Consult the docstring to find out how each should be instantiated. Attributes can be given as dictionaries that `Config` will parse, as above, or as `Command` instances:

```
from ravenpy.config import commands as rc

rst = rc.RainSnowTransition(temp=-.5, delta=1)
Config(RainSnowTransition=rst).rvp
```

Another similar example is the `:HRUs` command, which is rendered as a table of HRU properties. Each entry in the table is an `HRU` object. Again, HRUs can be instantiated from dictionaries or from class instances, use the style you prefer:

```
conf = Config(HRUs=
  [dict(hru_id=1, area=1000, elevation=300,  latitude=50,
    longitude=-109),
  rc.HRU(hru_id=2, area=200, elevation=350,  latitude=50.5,
    longitude=-109.2)
    ])
print(conf.rvh)
```

4.2 Gauge, StationForcing, GriddedForcing and ObservationData commands

Meteorological forcing inputs and streamflow observations are specified using the commands `Gauge`, `StationForcing` or `GriddedForcing`, and `ObservationData`. Ravenpy only supports reading and writing time series stored in netCDF. To facilitate the configuration of `rvt` files, each includes a class method `from_nc` that tries to infer the values of configuration options from the file's content. The inference rules are based on the CF-Convention, making the configuration of models using input files that are CF compliant substantially simpler. For example, the lookup for a given forcing type (PRECIP, RAINFALL, SNOWFALL, AVE_TEMP, etc) starts with the CF standard name (`ravenpy.config.conventions.CF_RAVEN`), but if that fails, `from_nc` will try alternative names given in the `alt_names` function argument.

For example, `ObservationData.from_nc(fn, station_idx=1, alt_names=["q_obs"])` opens a netCDF file `fn`, looks for a variable named `water_volume_transport_in_river_channel`, and when that fails, looks for `q_obs`. It returns an `rc.ObservationData` instance, itself holding a `ReadFromNetCDF` command with `FileNameNC`, `VarNameNC`, `DimNamesNC`, `StationIdx`, etc. Any value that should be part of the command but is not correctly extracted can be specified explicitly using extra keyword arguments given to `from_nc`.

Note that in the case of `Gauge.from_nc`, extra keyword arguments for `:Data` and `:ReadFromNetCDF` are set via the `data_kwds` dictionary, keyed by data type. The keyword `"ALL"` signifies that the keywords should be set for all variables. For example, if the forcing file does not include the gauge longitude and latitude, it could be set with `Gauge.from_nc([pr.nc, tas.nc], data_type=["PRECIP", "AVE_TEMP"], data_kwds={"ALL": {"Latitude": 45, "Longitude"=-80}})`. Linear transformations needed to convert units are inferred automatically whenever possible. If those fail, set them explicitly with `data_kwds`, e.g. `data_kwds={"PRECIP": {"Deaccumulate": True, "TimeShift": -0.25, "LinearTransform": {"scale": 1000}}}`. Note that automatic units conversion will fail for precipitation accumulated during the day which need the `:Deaccumulate` option.

4.3 Input validation

Many Raven commands can only take specific values, for example, `:PotentialMeltMethod` can take one of nine values: "POTMELT_DEGREE_DAY", "POTMELT_DATA", "POTMELT_RESTRICTED", etc. Valid options are defined in `ravenpy.config.options.PotentialMeltMethod` as an `enum.Enum` object. If `Enum` objects are not familiar, think of them as a static mapping between keys and values. The full list of options for `PotentialMeltMethod` can be displayed by converting the Enum to a list:

```
from ravenpy.config import options as o

list(o.PotentialMeltMethod)
```

`Config` understands both the option value, and the `Enum` key. For example, both styles below are valid:

```
Config(
       PotentialMeltMethod=o.PotentialMeltMethod.DEGREE_DAY,
       RainSnowFraction="RAINSNOW_DATA"
       ).rvi
```

Of course, setting an option with an invalid value will raise a `ValidationError`. For example, the error message below suggests there is a typo in the routing method, it should read `ROUTE_DIFFUSIVE_WAVE`:

```
Config(Routing="ROUTE_DIFFUSIVE")
```

This validation mechanism is used throughout the configuration, and will catch configuration errors well before executing the model. It relies on pydantic, which compares attribute values to their type annotation. Note that types are not strict, and pydantic tries to cast the values given into the annotated type. A `ValidationError` is raised if this fails. This explains that even though `RunName` has a `str` annotation, it is still possible to pass it an integer, and it'll be converted to a string:

```
Config(RunName=1).rvi
```

4.4 Accessing and modifying configuration options

The configuration options are stored in the `Config` instance as attributes. To respect Python style conventions, all attributes are lower case. The rule we followed is to use underscores to split words, so that in the first example above, the `:StartDate` option is stored as attribute `start_date`:

```
conf.start_date = "2023-04-01"
conf.start_date
```

While modifying configuration in place is sometimes useful, experience suggests it can create confusion and workflows that are harder to understand. We recommend instead to create modified copies of the original configuration using the `duplicate` method:

```
new = conf.duplicate(RunName="new", Duration=10)
print(new.rvi)
```

`duplicate` takes keyword arguments that it uses them to create a modified copy of the original configuration, which is left intact.

4.5 Limitations

Some Raven commands are not yet supported by ravenpy. Trying to give unrecognized configuration attributes will raise a `ValidationError` saying 'extra fields not permitted'. If this happens, please submit a feature request.

4.6 Raven emulators

Raven emulators are pre-configured models meant to reproduce almost exactly the behavior of known hydrological models. Emulators packaged with ravenpy are found in `ravenpy.config.emulators`. Each emulator is just a subclass of `Config`, with default values set for hydrological processes, the soil model, soil profiles, land-use, soil and vegetation classes, etc.

If you look at emulators' attributes, you'll see they are given a pydantic annotation and a default value using `pydantic.Field`, with an alias set to the name of the Raven command, for example:

```
from pydantic import Field

class TestEmulator(Config):
  evaporation: o.Evaporation = Field(default="PET_HARGREAVES", alias="Evaporation")
```

In the example above, the default `evaporation` attribute for `TestEmulator` is set to "PET_HEARGREAVES". It can however be changed when instantiating the model by giving it another value, e.g. `TestEmulator(Evaporation="PET_OUDIN")`. The annotation makes sure that whatever value is given is one of the allowed evaporation values defined in `ravenpy.config.options.Evaporation`.

The `alias` plays two roles:

1. it allows users to define evaporation using either the Raven command name `Evaporation`, in addition to the attribute name `evaporation`;
2. it tells the Config that this attribute is a Raven command that should be rendered as `:Evaporation <value>`.

Note that in general, even once completely defined, emulators cannot be run as is, because: (1) they have no default parameter values, (2) the default HRU has area set to zero, (3) there are no meteorological forcings, and (4) initial conditions may be far off from reasonable values.

4.7 Symbolic expressions in emulator configuration

`ravenpy` supports symbolic expressions in the definition of configuration parameters. That is, it is possible to define the value of Raven commands based on the value of parameters that are still undefined. This is done by:

1. defining a `dataclass` for the parameters, with type annotations allowing for symbolic variables and floats, and with default values set to pymbolic `Variables`;
2. subclassing `Config`, setting the default `params` to an instance of this dataclass;
3. using parameters in symbolic expressions for Raven commands.

For example:

```
from typing import Union
from pydantic.dataclasses import dataclass
from pymbolic.primitives import Variable
from ravenpy.config import Sym
```

(continues on next page)

(continued from previous page)

```
@dataclass(config=dict(arbitrary_types_allowed=True))
class P:
  X01: Union[Variable, float] = Variable("X01")

class MyEmulator(Config):
  params: P = P()
  rain_snow_transition: rc.RainSnowTransition = Field(
  default=rc.RainSnowTransition(temp=P.X01, delta=2),
  alias="RainSnowTransition")
```

This class can be instantiated with `MyEmulator()`, but it cannot be written to disk because parameters have not been set to numerical values. Numerical values for `params` can be set at instantiation, e.g. `MyEmulator(params=[-.5])`, by attribute assignment (e.g. `conf.params=[-.5]`), or using a special-purpose `set_params` method that returns a new configuration object where symbolic expressions have been converted to numerical values.

```
sym = MyEmulator()
num = sym.set_params([-.5])
num
```

Such symbolic model configuration is absolutely essential for model calibration, where the calibration algorithm needs to modify parameters repeatedly. The pre-packaged emulators made available in ravenpy are already setup to perform calibration using a symbolic model configuration.

CHAPTER

FIVE

MODEL EXECUTION

ravenpy provides two mechanisms to execute a model configuration. The first one described is the `ravenpy.run` function, which expects to find a directory with RV files and returns the path to the output directory. The second one is the `ravenpy.Emulator` class which exposes more functionalities and deserves some explanations.

5.1 The Emulator class

The `Emulator` class minimally expects a fully configured `Config` instance, which it immediately writes to disk. Other optional arguments are `workdir`, the path to the work directory where configuration files are written, and `modelname`, the string used to create configuration file names. By default, `workdir` will be set to a temporary directory, so make sure to specify another location if you want to hold on to the results.

When the `run` method of an `Emulator` instance is called, it launches the model which writes simulations results in the `output` folder, then returns an instance of the `OutputReader` class. Beyond the `write_rv` and `run` methods, Emulator has a number of read-only properties:

- `config`: the model configuration `Config` instance;
- `workdir`: the path to the work directory where configuration files are written;
- `output_path`: the path to simulation results;
- `modelname`: the name given to configuration files written to disk;
- `output`: the `OutputReader` instance created out of the simulation results.

The `Emulator` has another method called `resume`, which returns a new `Config` instance where the initial states for HRUs and sub-basins are set from the `solution.rvc` file storing the states at the end of the run. Note that by default, the `StartDate` of this new configuration will be set to the date following the end of the run. Set `timestamp=False` to ignore the time stamp information of the solution file.

CHAPTER

SIX

READING MODEL OUTPUTS

ravenpy includes two relatively simples classes that expose Raven outputs as Python objects: `OutputReader` for single simulation outputs, and `EnsembleReader` to aggregate the results of multiple simulations as would be done in ensemble streamflow forecasting, for example.

6.1 Accessing simulation results with `OutputReader`

Each Raven simulation creates output files in a directory. These outputs can be exposed as Python objects using the `ravenpy.OutputReader` class. `OutputReader` takes a `run_name` and `path` arguments, and returns an object with a number of read-only properties:

- `files`: dictionary of file path keyed by input kind;
- `solution`: dictionary with final model states for HRUs and sub-basins, as well as end date;
- `diagnostics`: dictionary storing the values of evaluation metrics;
- `hydrograph`: `xr.Dataset` of the simulated hydrograph;
- `storage`: `xr.Dataset` of the simulated storage variables;
- `messages`: the content of `Raven_errors.txt`;
- `path`: path to the output directory.

Note that `Emulator.run` returns an `OutputReader` instance, so a typical ravenpy workflow looks like this:

```
from ravenpy.config.emulators import HMETS
from ravenpy import Emulator

# Configure model simulation
conf = HMETS(**kwds)

# Run the model
out = Emulator(conf).run()

# Look at the results
out.hydrograph.q_sim.plot()
```

Note also that the `run_name` parameter should reflect the value of the `:RunName` configuration option. If `:RunName` is not configured, then the `run_name` argument should be left to its default `None` value.

6.2 Accessing results from multiple simulations using EnsembleReader

Together, the `Config` and `Emulator` classes makes it fairly simple to create simulation ensembles. For example, to run the same model with different parameters, you could do something like:

```
# Output directory for all simulations
from pathlib import Path
p = Path("/tmp/ensemble")

# Create base model configuration
conf = HMETS(**kwds)

# Run the model for each parameter set in `params`
runs = [Emulator(conf.set_params(param), workdir=p / f"m{i}").run() for i, param in␣
↪enumerate(params)]
```

Now `runs` stores a list of `OutputReader` instances. The time series stored in `hydrograph` and `storage` can be concatenated together using the `EnsembleReader` class:

```
from ravenpy import EnsembleReader

ens = EnsembleReader(runs=runs, dim="parameters")
ens.hydrograph.q_sim
```

where `q_sim` is a `xarray.DataArray` with dimensions `('time', 'parameters')`. An `EnsembleReader` can also be created from a list of simulation output paths:

```
# Create list of output paths using glob
paths = p.glob("**/output")

ens = EnsembleReader(paths=paths, dim="parameters")
```

CHAPTER

SEVEN

NOTEBOOKS

These notebooks demonstrate a few features of the Ravenpy package when integrated into the PAVICS-Hydro service or run locally.

If you're unfamiliar with notebooks, note that typing *TAB* after an object will display a drop-down menu of the object's attributes and methods, and that you need to either press the "run" button in the top of the JupyterLab window on the PAVICS server, or "hit *CTRL-Enter* to run a *cell*. You can also type *?* after a function or method to display the corresponding help message. See more details in the first Notebook below (00 - Introduction to JupyterLab).

7.1 Getting started - Tutorial

7.1.1 00 - Introduction to JupyterLab

Before going any further:

These notebooks are best visualized by copying them to your writable-workspace on your PAVICS account, as files will be created and written on your writable-workspace that has write access. Please copy the tutorial notebooks (00-12) to that folder before continuing.

Please see the PAVICS-Hydro documentation on the PAVICS website for more details.

JupyterLab

Welcome to this PAVICS-Hydro tutorial, where we will explore the various hydrological modelling and forecasting possibilities offered by PAVICS-Hydro. The platform uses the Raven hydrological modelling framework to emulate different hydrological models, and relies on various scientific Python libraries to analyze the results. This tutorial starts by exploring the JupyterLab environment that this notebook is currently running on.

The file explorer

The file explorer to the left of your screen works in much the same way as any file explorer on Windows, Mac or Linux. Here, we have files and folders that contain notebooks and data that we will want to use in our research or operations. You can:

- Upload files here by using drag-and-drop OR using the button to that effect above the file explorer to send files from computer to the server (e.g. watershed boundaries, model files, input data, streamflow data, etc.) as required;
- Cut, Copy, Paste files from one folder to another in the JupyterLab server;
- Download files by right-clicking the file and saving to your computer locally;

- Open notebook files (*.ipynb*) in the editor to modify and run the codes within;
- Shutting down a running notebook by right-clicking and selecting "Shut Down Kernel".

The file explorer allows users to manage the files and codes. To modify the codes and run them, we need to double-click on a notebook to open it in the file editor.

The file editor

The file editor is what is being used right now to read the contents of this notebook! It is what is on the right side of your screen. If you open multiple notebooks, there will be as many tabs open on the top of the file editor. This is where the magic happens! Once a notebook has been opened, you can see that there are some "Text" cells and some "Code" cells.

The **text cells** give context to what is happening and can be seen as meta-comments on top of the regular code comments, to ensure everything is clear to the users. The cell you are currently reading, for example, is a code cell. If you double-click it, you can modify its contents. To make it appear as text again, press the "play" or "run" button in the button list at the top of the file editor. These texts cells follow the Markdown templating.

The **code cells** will actually perform the work on the PAVICS-Hydro server. For example, here is a simple code cell that will import a python package in our notebook. These code cells are in Python.

```
import xarray as xr
```

The above cell does not do anything unless we tell the notebook that it needs to be run. To run a cell, you need to select it (click on it) and press the "play"/"run" button. This will tell the PAVICS-Hydro server that it needs to run this piece of code. To see if a code is running, the small brackets to the left of **code cells** will briefly turn to an asterisk, and will display a number once the code has finished running. If there is an error, there will be a red box with clear error messages under the executed cell.

We can then see if importing the xarray package has worked by using a quick test. Run the below code. If it displays an error, then the importing has failed and should be run again. Also, there will be an empty space between the brackets. If it has worked, you should see a version in the brackets and the xarray version displayed under the cell!

```
print(xr.__version__)
```

Order of operations and variables in memory

Jupyter Notebooks function in the same way as scripts in most programming languages. That is to say:

- Cells will execute the first line within that cell before the second one, etc.;
- If a cell tries to use a variable that has not been created yet, it will cause an error;
- If a cell creates a variable, then that variable will be available to all other cells from that point on;
- If a cell deletes a variable, then that variable is no longer available to any cell.

To test this, you can try **skipping the next cell and running the following one, first**, which will return an error:

```
# SKIP THIS CELL FOR NOW!

# Run this cell to create variable "b"
b = 5
```

```
# You can also add comments by prefixing a line with a hashtag, like this.
a = b + 3
```

You will get an error saying that "name 'b' is not defined".

This is normal, because the code is expecting that variable "b" exists somewhere in its memory, but it hasn't been created yet! So, let's now create variable "b" by running the cell that we skipped (Note that we are presenting the cells in this order because otherwise errors would break our quality testing checks!)

Now that variable "b" has been created, try and re-run the cell that gave an error previously. It should work, because now "b" exists! So you can see how ordering cells is important. It is also possible to use this to create small tests within a notebook, by changing some variable at key points between larger blocks of code.

Managing files

JupyterLab allows loading and saving files in the file explorer. Let's explore this capability.

First we will create a random array of numbers and save them to a file. We will then read that variable back into memory and compare results.

```
# We will do this with the numpy package:
import numpy as np

c = np.random.rand(100)  # Create 100 random values and store them in variable "c"
np.savetxt("array_c.txt", c)  # Write the array to a file named "array_c.txt"
```

If you look in your workspace, you will see a new file named "array_c.txt". All files you generate will be written to the folder in which your notebook is running from.

Now let's read it back in and verify that the values are the same. We can do this by taking the sum of the absolute differences between each element. If the sum is 0, that means we have succeeded:

```
d = np.loadtxt("array_c.txt")
print(sum(abs(c - d)))
```

As you can see, files can be accessed easily by simply refering to them by their name if they are in the same folder as the notebook. If they aren't you also need to specify the folder path. Example "writable-workspace/array_c.txt". This is also true for all files, even those you upload yourself. You can also access data that can be found online on an accessible server using the URL, but we will get to that in a later notebook.

IMPORTANT: Multiple noteooks running concurrently

Notebooks are independent instances and do not communicate with one another. This means that if you load a package or a variable in memory in one notebook, the information will not be available in your other notebooks. This means you would need to import the data in the different notebooks. This will have the drawback of consuming more memory on the server, which can slow down computations for everyone. Therefore, we ask that you kindly close and shut down the notebooks once you are done with them. This can be done by right-clicking the notebook in the file explorer and selecting "Shut Down Kernel". This will close the instance and free all memory the notebook was taking up. Thanks!

Important closing remarks

JupyerLab environments make using codes easy and repeatable, without having to worry too much about packages, data access and other such elements that can be difficult to work with. However, there are some drawbacks:

- You are running codes on a remote server, so it might be slower than on a high-performance local computer;
- You might require more resources than are available on the remote server;
- You might want to implement major changes that are not compatible with the Python packages available in the PAVICS-Hydro environment.

To add packages, you can simply add a cell and "! pip install **package**" as required, which will add it to your local server. It will need to be re-added every time you close and re-spawn your server. You can also use the Jupyter conda plugin to install via conda (Settings –> Conda Packages Manager).

Otherwise, you can always install the PAVICS-Hydro environment on your local computer following the instructions found [here] (https://pavics-sdi.readthedocs.io/projects/raven/en/latest/index.html). Note that these instructions are for more advanced python users / developers.

In the next notebooks, we will start using your JupyterLab instance to start doing hydrological and hydroclimatological science!

7.1.2 01 - Getting watershed boundaries

Region Selection and Map Preview with Ipyleaflet

In this notebook, you will extract a selected watershed from the HydroSHEDS database (see the reference manual for more information on HydroSHEDS). A GeoJSON with the watershed boundaries will be available for download and usable for other tasks such as extracting meteorological data covered in the next notebooks.

```
# 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 you are running this locally (and not on the PAVICS-Hydro server), and 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.

This next box is all boilerplate, you do not need to understand it or play with it. Simply run it! Many such code snippets are provided throughout the notebooks to make your life easier. You can then modify some options to taylor the code to your needs.

```
# 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)

# Build an interactive map with ipyleaflet
initial_lat_lon = (48.63, -74.71)
```

(continues on next page)

(continued from previous page)

```
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)

# 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"],
    )
```

Using the map to select the outlet of the watershed

When using the “leaflet_map” command, an interative map will be displayed.

Note that a blue marker will be displayed in the middle of the map, which can be dragged by interacting directly with it. 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. This coordinate will be used to find and extract the closest watershed outlet from the Hydrosheds database (see the reference manual for more info on Hydrosheds). The watershed ID and area will be displayed at the bottom left corner of the map.

The user can zoom in and out on the map either by:

- Using the Zoom level on the top right corner;
- Using the + / - icons on the top left corner;
- Double-clicking on the map on the area to zoom in.

```
# Load the map in the notebook
leaflet_map
```

```
# Display the lat/lon coordinates of the marker location.
user_lonlat = list(reversed(marker.location))
print(user_lonlat)
```

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

Before continuing, wait for the process above to finish

This can be monitored when the "[*]:" on the left of the cell is replaced with a number.

```
# Extract the URL of the resulting GeoJSON feature
feat = resp.get(asobj=False).feature
print(
    "This is the geoJSON file that can be used as the watershed contour in other␣
↪toolboxes:"
)
print("")
print(feat)
print("")
```

BEFORE CONTINUING:

- If you are working in the writable-workspace, you will want to download the .geojson file at the link above and deposit it into your workspace on the left of your screen.
- If you are running this in the default workspace after logging in, the workspace is read-only so we will provide files for you, and you can ignore this file for the time being.

```
# Print the properties from the extracted watershed
gdf = gpd.read_file(feat)
gdf
```

Now we will add the extracted watershed to the map above!

Scroll back up after executing the next cell to see the watershed displayed in blue on the map. You may reextract another watershed by moving restarting the kernel or running all the cells from the beginning to reload the map.

```
# Adding the GeoJSON to the map above.
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},
```

(continues on next page)

(continued from previous page)

```
)

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

Congratulations!

You have successfully created a watershed boundary file that can be used in the following notebooks. If you already have the boundaries of your watershed of interest, then you can upload them to your workspace instead of using this notebook to generate them. a geojson file is accepted, as is a shapefile. For shapefiles, provide a zip containing all the shape data (.shp, .shx, .dbf, .prj, etc.).

7.1.3 02 - Extract geographical watershed properties

Extract geographical watershed properties automatically using PAVICS-Hydro's geospatial toolbox

Hydrological models typically need geographical information about watersheds being simulated: latitude and longitude, area, mean altitude, land-use, etc. Raven is no exception. This notebook shows how to obtain this information using remote services that are made available for users in PAVICS-Hydro. These services connect to a digital elevation model (DEM) and a land-use data set to extract relevant information.

The DEM used in the following is the EarthEnv-DEM90, while the land-use dataset is the North American Land Change Monitoring System. Other data sources could be used, given their availability through the Web Coverage Service (WCS) protocol.

Since these computations happen on a specific Geoserver hosted on PAVICS, we need to establish a connection to that service. While the steps are a bit more complex, the good news is that you only need to change a few items in this notebook to tailor results to your needs. For example, this first code snippet is boilerplate and should not be changed.

```
# We need to import a few packages required to do the work
import os

os.environ["USE_PYGEOS"] = "0"
import geopandas as gpd
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import rasterio
import rioxarray as rio
from birdy import WPSClient

from ravenpy.utilities.testdata import get_file

# This is the URL of the Geoserver that will perform the computations for us.
url = os.environ.get(
    "WPS_URL", "https://pavics.ouranos.ca/twitcher/ows/proxy/raven/wps"
)

# Connect to the PAVICS-Hydro Raven WPS server
wps = WPSClient(url)
```

In the previous notebook, we extracted the boundaries of a watershed, which were saved in the "input.geojson" file. We also downloaded the file and re-uploaded it to the workspace, so it should be available now to this workbook, too!

We can now plot the outline of the watershed by loading it into `GeoPandas`.

```
# The contour can be generated using notebook "01_Delineating watersheds, where it would␣
↪be placed
# in the same folder as the notebooks and available in your workspace. The contour could␣
↪then be accessed
# easily by defining it as follows:
"""
feature_url = "input.geojson"
"""
# However, to keep things tidy, we have also prepared a version that can be accessed␣
↪easily for
# demonstration purposes:
feature_url = get_file("notebook_inputs/input.geojson")
df = gpd.read_file(feature_url)
display(df)
df.plot()
```

Generic watershed properties

Now that we have delineated a watershed, lets find the zonal statistics and other properties using the `shape_properties` process. This process requires a `shape` argument defining the watershed contour, the exterior polygon. The polygon can be given either as a link to a geometry file (e.g. a geojson file such as `feature_url`), or as data embeded in a string. For example, if variable `feature` is a `GeoPandas` geometry, `json.dumps(feature)` can be used to convert it to a string and pass it as the `shape` argument.

Typically, we expect users will simply upload a shapefile and use this code to perform the extraction on the region of interest.

```
shape_resp = wps.shape_properties(shape=feature_url)
```

Once the process has completed, we extract the data from the response, as follows. Note that you do not need to change anything here. The code will work and return the desired results.

```
(properties,) = shape_resp.get(asobj=True)
prop = properties[0]
display(prop)

area = prop["area"] / 1000000.0
longitude = prop["centroid"][0]
latitude = prop["centroid"][1]
gravelius = prop["gravelius"]
perimeter = prop["perimeter"]

shape_info = {
    "area": area,
    "longitude": longitude,
    "latitude": latitude,
    "gravelius": gravelius,
    "perimeter": perimeter,
```

(continues on next page)

(continued from previous page)

```
}
display(shape_info)
```

Note that these properties are a mix of the properties of the original file where the shape is stored, and properties computed by the process (area, centroid, perimeter and gravelius). Note also that the computed area is in m^2, while the "SUB_AREA" property is in km^2, and that there are slight differences between the two values due to the precision of HydroSHEDS and the delineation algorithm.

Land-use information

Now we extract the land-use properties of the watershed using the `nalcms_zonal_stats` process. As mentioned, it uses a dataset from the North American Land Change Monitoring System, and retrieve properties over the given region.

With the `nalcms_zonal_stats_raster` process, we also return the raster grid itself. Note that this is a high-resolution dataset, and to avoid taxing the system's resource, requests are limited to areas under 100,000km^2.

```
stats_resp = wps.nalcms_zonal_stats_raster(
    shape=feature_url, select_all_touching=True, band=1, simple_categories=True
)
```

Here we will get the raster data and show it as a grid. Here the `birdy` client automatically transforms the returned `geotiff` file to a `DataArray` using either `gdal`, `rasterio`, or `rioxarray`, depending on what libraries are available in our runtime environment. Note that `pymetalink` needs to be installed for this to work.

```
features, statistics, raster = stats_resp.get(asobj=True)
grid = raster[0]
grid.plot()
```

From there, it's easy to calculate the ratio and percentages of each land-use component. This code should also be left as-is unless you really know what you are doing.

```
lu = statistics[0]
total = sum(lu.values())

land_use = {k: (v / total) for (k, v) in lu.items()}
display("Land use ratios", land_use)

land_use_pct = {k: f"{np.round(v/total*100, 2)} %" for (k, v) in lu.items()}
display("Land use percentages", land_use_pct)
```

Display the land-use statistics

Here we can display the land-use statistics according to the land cover map, as a function of land cover raster pixels over the catchment. Again, this does not need to be modified at all. It can also be simply deleted if the visualization tools are not required for your use-case.

```
unique, counts = np.unique(grid, return_counts=True)
print("The land-use categories available are: " + str(unique))
print("The number of occurrences of each land-use category is: " + str(counts))

# Pixels values at '127' are NaN and can be ignored.
```

(continues on next page)

(continued from previous page)

```
from matplotlib.colors import Normalize

norm = Normalize()
norm.autoscale(unique[:-1])
cm = mpl.colormaps["tab20"]
plt.bar(unique[:-1], counts[:-1], color=cm(norm(unique[:-1])))


# plt.bar(unique[:-1], counts[:-1])
plt.xticks(np.arange(min(unique[:-1]), max(unique[:-1]) + 1, 1.0))
plt.xlabel("Land-use categories")
plt.ylabel("Number of pixels")
plt.show()

grid.where(grid != 127).sel(band=1).plot.imshow(cmap="tab20")
grid.name = "Land-use categories"
plt.show()
```

These values are not very helpful on their own, so the following relationship will be helpful to map the grid to specific land-uses. We can see from this example that we have mostly "Temperate or sub-polar needleaf forest" with some "Sub-polar taiga needleleaf forest" and a bit of "Temperate or sub-polar boardleaf deciduous forest". Exact percentages can be computed from the array of values as extracted and displayed above.

- 0: Ocean
- 1: Temperate or sub-polar needleleaf forest
- 2: Sub-polar taiga needleleaf forest
- 3: Tropical or sub-tropical broadleaf evergreen forest
- 4: Tropical or sub-tropical broadleaf deciduous forest
- 5: Temperate or sub-polar broadleaf deciduous forest
- 6: Mixed forest
- 7: Tropical or sub-tropical shrubland
- 8: Temperate or sub-polar shrubland
- 9: Tropical or sub-tropical grassland
- 10: Temperate or sub-polar grassland
- 11: Sub-polar or polar shrubland-lichen-moss
- 12: Sub-polar or polar grassland-lichen-moss
- 13: Sub-polar or polar barren-lichen-moss
- 14: Wetland
- 15: Cropland
- 16: Barren lands
- 17: Urban
- 18: Water
- 19: Snow and Ice

Since the GeoTiff object was opened as an `xarray.Dataset` with the `.open_rasterio()` method, this makes it very easy to spatially reproject it with the `cartopy` library. Here we provide a sample projection, but this would need to be adapted to your needs.

```
import cartopy.crs as ccrs

# Set a CRS transformation:
crs = ccrs.LambertConformal(
    central_latitude=49, central_longitude=-95, standard_parallels=(49, 77)
)

ax = plt.subplot(projection=crs)
grid.name = "Land-use categories"
grid.where(grid != 127).sel(band=1).plot.imshow(ax=ax, transform=crs, cmap="tab20")
plt.show()
```

Terrain information from the DEM

Here we collect terrain data, such as elevation, slope and aspect, from the DEM. We will do this using the `terrain_analysis` WPS service, which by default uses DEM data from EarthEnv-DEM90.

Note here that while the feature outline is defined above in terms of geographic coordinates (latitude, longitude), the DEM is projected onto a 2D cartesian coordinate system (here NAD83, the Canada Atlas Lambert projection). This is necessary to perform slope calculations. For more information on this, see: https://en.wikipedia.org/wiki/Map_projection

The DEM data returned in the process response here shows `rioxarray`-like access but using the URLs we can open the files however we like.

```
terrain_resp = wps.terrain_analysis(
    shape=feature_url, select_all_touching=True, projected_crs=3978
)
```

```
properties, dem = terrain_resp.get(asobj=True)

elevation = properties[0]["elevation"]
slope = properties[0]["slope"]
aspect = properties[0]["aspect"]

terrain = {"elevation": elevation, "slope": slope, "aspect": aspect}
display(terrain)
```

```
crs = ccrs.LambertConformal(
    central_latitude=49, central_longitude=-95, standard_parallels=(49, 77)
)

dem.name = "Elevation"
dem.attrs["units"] = "m"
ax = plt.subplot(projection=crs)
dem.where(dem != -32768).sel(band=1).plot.imshow(ax=ax, transform=crs, cmap="gnuplot")
plt.show()
```

```
# We can also access the files directly via their URLs:
properties, dem = terrain_resp.get(asobj=False)
display(properties, dem)

# Let's read the data from band=1 as numpy array
display(rasterio.open(dem).read(1))
```

Overview

A synthesis of all watershed properties can be created by merging the various dictionaries created. This allows users to easily access any of these values, and to provide them to Raven as needed.

```
all_properties = {**shape_info, **land_use, **terrain}
display(all_properties)
```

7.1.4 03 - Extracting forcing data

Extracting meteorological data for a selected watershed

Using a GeoJSON file extracted from the HydroSHEDS database or given by the user, meteorological datasets can be extracted inside the watershed's boundaries using the PAVICS-Hydro ERA5 database.

```
import datetime as dt
import tempfile
from pathlib import Path

import fsspec  # noqa
import intake
import numpy as np
import s3fs  # noqa
import xarray as xr
from clisops.core import subset

from ravenpy.utilities.testdata import get_file
```

If we want to extract data for our watershed, we need to know:

- The spatial extent (as defined by the watershed boundaries);
- The temporal extent (as defined by the start and end days of the period of interest).

Let's define those now:

```
# This will be our input section, where we control what we want to extract.
# We know which watershed interests us, it is the input.geojson file that we previously␣
↪generated!

# The contour can be generated using notebook "01_Delineating watersheds, where it would␣
↪be placed
# in the same folder as the notebooks and available in your workspace. The contour could␣
↪then be accessed
# easily by defining it as follows:
```

(continues on next page)

(continued from previous page)

```
"""
basin_contour = "input.geojson"
"""
# However, to keep things tidy, we have also prepared a version that can be accessed␣
↪easily for
# demonstration purposes:
basin_contour = get_file("notebook_inputs/input.geojson")

# Also, we can specify which timeframe we want to extract. Here let's focus on a 10-year␣
↪period
reference_start_day = dt.datetime(1985, 12, 31)
reference_stop_day = dt.datetime(1987, 1, 1)
# Notice we are using one day before and one day after the desired period of 1986-01-01␣
↪to 1986-12-31.
# This is to account for any UTC shifts that might require getting data in a previous or␣
↪later time.
```

We now provide a means to get some data to run our model. Typically, models will require precipitation and temperature data, so let's get that data. We will use a generally reliable dataset that is available everywhere to minimize missing values: the ERA5 Reanalysis.

The code block below gathers the required data automatically. If you need other data or want to use another source, this cell will need to be replaced for your customized needs.

```
# Get the ERA5 data from the Wasabi/Amazon S3 server.
catalog_name = "https://raw.githubusercontent.com/hydrocloudservices/catalogs/main/
↪catalogs/atmosphere.yaml"
cat = intake.open_catalog(catalog_name)
ds = cat.era5_reanalysis_single_levels.to_dask()
```

Get the ERA5 data. We will rechunk it to a single chunk to make it compatible with other codes on the platform, especially bias-correction. We are also taking the daily min and max temperatures as well as the daily total precipitation.

```
# We will add a wrapper to ensure that the following operations will preserve the␣
↪original data attributes, such as units and variable names.
with xr.set_options(keep_attrs=True):
    ERA5_reference = subset.subset_shape(
        ds.sel(time=slice(reference_start_day, reference_stop_day)), basin_contour
    )
    ERA5_tas = ERA5_reference["t2m"].resample(time="1D")
    ERA5_tmin = ERA5_tas.min().chunk(-1, -1, -1)
    ERA5_tmax = ERA5_tas.max().chunk(-1, -1, -1)
    ERA5_pr = ERA5_reference["tp"].resample(time="1D").sum().chunk(-1, -1, -1)

ERA5_pr
```

We can now convert these variables to the desired format and save them to disk in netcdf files to use at a later time (in a future notebook!)

First, we will want to make sure that the units we are working with are compatible with the Raven modelling framework. We will want precipitation to be in mm (per time period, here we are working daily so it will be in mm/day), and temperatures will be in °C. Let's check out the current units:

```
print(f"Tmin units: {ERA5_tmin.units}")
print(f"Tmax units: {ERA5_tmax.units}")
print(f"Precipitation units: {ERA5_pr.units}")
```

We can see that the units are in Kelvin for temperatures and in meters for precipitation. We will want to do some conversions!

Let's start by applying offsets for temperatures and a conversion factor for precipitation:

```
with xr.set_options(keep_attrs=True):
    ERA5_tmin = ERA5_tmin - 273.15  # K to °C
    ERA5_tmin.attrs["units"] = "degC"

    ERA5_tmax = ERA5_tmax - 273.15  # K to °C
    ERA5_tmax.attrs["units"] = "degC"

    ERA5_pr = ERA5_pr * 1000  # m to mm
    ERA5_pr.attrs["units"] = "mm"
```

We can see the changes now by re-inspecting the datasets:

```
print(f"Tmin units: {ERA5_tmin.units}")
print(f"Tmax units: {ERA5_tmax.units}")
print(f"Precipitation units: {ERA5_pr.units}")
```

So let's write them to disk for now. We will use the netcdf format as this is what Raven uses for inputs. It is possible you will get some warnings, this is OK and should not cause any problems. Since our model will run in lumped mode, we will average the spatial dimensions of each variable over the domain.

```
with xr.set_options(keep_attrs=True):
    # Average the variables
    ERA5_tmin = ERA5_tmin.mean({"latitude", "longitude"})
    ERA5_tmax = ERA5_tmax.mean({"latitude", "longitude"})
    ERA5_pr = ERA5_pr.mean({"latitude", "longitude"})

    # Ensure that the precipitation is non-negative, which can happen with some␣
↪reanalysis models.
    ERA5_pr = np.maximum(ERA5_pr, 0)

    # Transform them to a dataset such that they can be written with attributes to netcdf
    ERA5_tmin = ERA5_tmin.to_dataset(name="tmin", promote_attrs=True)
    ERA5_tmax = ERA5_tmax.to_dataset(name="tmax", promote_attrs=True)
    ERA5_pr = ERA5_pr.to_dataset(name="pr", promote_attrs=True)
```

```
# Check and see if the precipitation makes sense:
ERA5_pr.pr.plot()
```

Here we will write the files to disk in a temporary folder since the root folder containing these notebooks is read-only.

You can change the path here to your own preferred path in your writable workspace. Alternatively, if you copy this notebook to your writable-workspace as shown in the introduction documentation, you can save just the filename (no absolute path) and the file will appear "beside" the notebooks, ready to be read by the next series of notebooks.

For this case, you will want to use the second provided option in which all variables are stored in the same netcdf file. This will make the data much easier to find for Raven and prevent some errors, such as if Raven needs to calculate daily average temperature. This will be possible with all variables in the same file, but will cause an error if max and min temperature are in two seperate files.

```
# Option 1, which is not recommended to use in other notebooks but can be really useful␣
↪in various other workflows:
with xr.set_options(keep_attrs=True):
    # Write to disk.
    tmp = Path(tempfile.mkdtemp())
    ERA5_tmin.to_netcdf(tmp / "ERA5_tmin.nc")
    ERA5_tmax.to_netcdf(tmp / "ERA5_tmax.nc")
    ERA5_pr.to_netcdf(tmp / "ERA5_pr.nc")
```

```
# Option 2, which is recommended, in which we prepare a single file that merges all␣
↪three variables into one netcdf file:
with xr.set_options(keep_attrs=True):
    xr.merge([ERA5_tmin, ERA5_tmax, ERA5_pr]).to_netcdf(tmp / "ERA5_weather_data.nc")
```

We now have daily precipitation and minimum/maximum temperatures to drive our Raven Model, which we will do in the next notebook!

Note that our dataset generated here is very short (1 year) but the same dataset for the period 1980-12-31 to 1991-01-01 has been pre-generated and stored on the server for efficiency.

7.1.5 04 - Emulating hydrological models

Using Ravenpy to emulate an existing hydrological model

In this notebook, we will demonstrate the versatility of the Raven modelling framework to emulate one of eight hydrological models that are currently supported. We will walk through the different configuration parameters required to build the model and simulate streamflow on a catchments. We will also show how to import files from a pre-configured Raven configuration that users can inport into Ravenpy instead of using one of the default emulators.

A note on datasets

There are numerous ways to run a Raven model and to pass its required input data. For this introduction to RavenPy, we will use our ERA5 data we generated in the previous notebook and we will configure the Raven model instance on the fly! In the next tutorials, we will see how users can import and use their own datasets to make the entire process flexible and tailored to the user needs.

Using templated model emulators

The first thing we need to run the raven model is… a Raven model! Raven is not a model per se, but a modelling framework that can be used to build hydrological models from their underlying components. For now, PAVICS-Hydro allows building a set of pre-determined models. The Python wrapper offers at present eight model emulators: GR4J-CN, HMETS, MOHYSE, HBV-EC, Canadian Shield, HYPR, Sacramento and Blended. For each of these, templated configuration files are available to facilitate launching the model with options passed by Python at run-time.

In the next cell, we are going to import the possible models, and later, we will configure and run the GR4J-CN model. Please see the documentation for more details on the mandatory vs optional parameters, and what they represent. A small glimpse is provided here.

```
# Import the list of possible model templates.
from ravenpy.config.emulators import (
    GR4JCN,
    HBVEC,
    HMETS,
    HYPR,
    SACSMA,
    Blended,
    CanadianShield,
    Mohyse,
)
```

```
import datetime as dt

# Import other required packages:
import tempfile
from pathlib import Path

from ravenpy.config import commands as rc
from ravenpy.utilities.testdata import get_file
```

In this next step, we will define the hydrological response unit (HRU). For lumped models, there is only one unit so the following structure should be good. However, for distributed modelling, there will be more than one HRU, so we would use another tool to help us build the HRUs in that case. The HRU provides information on the area, elevation, and location of the catchment.

For now, let's provide the basin properties such that Raven can run. These are the minimal values that must always be provided, but some models might require other inputs. Please see the Raven documentation for more information.

```
# Define the hydrological response unit. We can use the information from the tutorial␣
↪notebook #02! Here we are using
# arbitrary data for a test catchment.
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)
```

The next required inputs are the start and end dates for the simulation. The `start_date` and `end_date` arguments indicate when a simulation should start and end. As long as the forcing data covers the simulation period, it should work. If these parameters are not defined, then start and end dates default to the start and end of the driving data.

To keep things simple, we will use a short 5-year period. Note that the dates are python datetime.datetime objects.

```
start_date = dt.datetime(1985, 1, 1)
end_date = dt.datetime(1990, 1, 1)
```

We are now ready to build our first Raven-based hydrological model. the model will be the GR4JCN model. The following code block will show and describe every step. However, more control options are available for users. Please see the documentation for a more detailed explanation on model options.

```
# Import required packages. We already imported the GR4JCN emulator in the first cell,␣
↪but let's keep it here for
# completeness.
from ravenpy.config.emulators import GR4JCN

# Since our meteorological gauge data is all included in a single file, we need to tell␣
↪the model which variables
# we are providing. We will generate the list now and pass it later to Ravenpy as an␣
↪argument to the model.
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]

# Alternative variable names are useful for allowing Raven to read variables from NetCDF␣
↪files even if the variable
# names are not those that are expected by Raven. For example, our ERA5-dernived␣
↪temperature variables are named
# "tmax" and "tmin", whereas Raven expects "TEMP_MAX" and "TEMP_MIN". Therefore, instead␣
↪of forcing users to rename
# their variables, we provide a mechanism to tell Raven which variables in the NetCDF␣
↪files correspond to which
# meteorological variable.
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# As per the Raven model itself, the gauge station elevation, latitude and longitude are␣
↪required to let the model run.
# For lumped models such as the ones we are currently emulating, there is only one
↪"station" that corresponds to the basin-averaged weather.
# In this case, we set the station elevation to that of the mean catchment elevation to␣
↪remove any adiabatic gradient modifications
# to the data. We also provide the catchment centroid latitude and longitude which we␣
↪take from the only HRU defining the entire catchment.
# For multi-gauge basins and semi-distributed models, the latitude and longitude must be␣
↪correctly identified for each station.
data_kwds = {
    "ALL": {  # Use this for all gauges (there is only one gauge, as it is a lumped␣
↪model using basin-averaged weather)
        "elevation": hru[
            "elevation"
        ],  # extract the values directly from the "hru" we previously built
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
```

(continues on next page)

(continued from previous page)

```
}

# Provide a run name used to generate output Raven files.
run_name = "test_NB_04"

# Get the weather data. It can either be to a data file that was already in the same␣
↪folder/workspace as this
# notebook, as we generated in the previous notebook, or it could be a path to a file␣
↪stored elsewhere.
# Example for using the data we just generated in Notebook 03:
"""
ERA5_full = ERA5_weather_data.nc
"""

# In our case, we will prefer to link to existing, pre-computed and locally stored files␣
↪to keep things tidy:
ERA5_full = get_file("notebook_inputs/ERA5_weather_data.nc")


# We need to define some configuration options that all models will need. See each line␣
↪for more details on their use.
default_emulator_config = dict(
    # The HRU as defined earlier must be provided. This is the physical representation␣
↪of the catchment that Raven
    # needs for certain models and processes.
    HRUs=[hru],
    # Model simulation start and end dates.
    StartDate=start_date,
    EndDate=end_date,
    # Name of the simulation. Raven will prefix all .rvX files and model outputs with␣
↪the runName.
    RunName=run_name,
    # Custom outputs allow pre-processing of certain statistics and variables after the␣
↪model runs. Please see the
    # documentation for more details on possible options.
    CustomOutput=[
        rc.CustomOutput(
            time_per="YEARLY",
            stat="AVERAGE",
            variable="PRECIP",
            space_agg="ENTIRE_WATERSHED",
        )
    ],
    # Here we will prepare the weather gauge data to be fed to the model. The data could␣
↪also come from other
    # sources such as the ERA5 reanalysis product.
    Gauge=[
        rc.Gauge.from_nc(
            ERA5_full,  # Path to the ERA5 file containing all three meteorological␣
↪variables
            data_type=data_type,  # Note that this is the list of all the variables
            alt_names=alt_names,  # Note that all variables here are mapped to their␣
```

(continues on next page)

(continued from previous page)

```
↪names in the netcdf file.
            data_kwds=data_kwds,
        )
    ],
)

# Here is where we build the raven configuration. We will first configure the model in␣
↪memory based on the user
# preferences, and then we will write the Raven configuration files to disk such that␣
↪Raven can read them and
# execute a simulation. In this case, we are building a GR4JCN model, but you could␣
↪change this to any of the
# eight models that are preconfigured for direct emulation in Ravenpy.
m = GR4JCN(
    # Raven requires parameters for the GR4JCN model. For now, we provide default values,
↪ but in a later notebook
    # we will show how to calibrate and find new parameters for our model.
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    # GR4JCN needs an extra constant parameter for the catchment, corresponding to the␣
↪G50 parameter in the CEMANEIGE
    # description. Here is how we provide it.
    GlobalParameter={"AVG_ANNUAL_SNOW": 408.480},
    **default_emulator_config,
)

# We are now ready to write this newly-configured model to disk through the use of .rvX␣
↪files that Raven will read.

# In the following code snippet, there is a "workdir" path that must be provided. This␣
↪indicates the path
# where the data used to run the model (RAVEN .RV files) will be made available from the␣
↪PAVICS Jupyter environment.
# The default "workdir" setting puts model outputs in the temporary directory ("/tmp"),
# which is not visible from the Jupyter file explorer. Therefore, You can change the␣
↪last subfolder,
# but '/notebook_dir/writable-workspace/' must be the beginning  of the path when running␣
↪on the PAVICS platform.

workdir = Path(tempfile.mkdtemp(prefix="NB4"))
m.write_rv(workdir=workdir)
```

It can be seen that .rvX files were generated in the indicated path. This makes the Ravenpy platform more flexible for various use-cases and can be exported/imported as needed.

The above code only created the .rvX files. The model did not actually run yet. To do so, we must ask it to, as follows. Don't worry about the warnings: Raven is informing us that it has generated some internal parameters to build the model configuration based on some of the parameters we provided.

```
# If we want to import our own raven configuration files and forcing data, we can do so␣
↪by importing them
# using the ravenpy.run  method. This will run the model exactly as the users will have␣
↪designed it.
from ravenpy import OutputReader
```

(continues on next page)

(continued from previous page)

```
from ravenpy.ravenpy import run

# This is used to specify the raven configuration files prefixes. In this case, we will␣
↪retake the previously created files
run_name = run_name

# This is the path where the files were uploaded by the user. Model outputs will also be␣
↪placed there in a
# subfolder called "outputs"
configdir = workdir

# Run the model and get the path to the outputs folder that can be used in the output␣
↪reader.
outputs_path = run(modelname=run_name, configdir=configdir)

# Get the outputs using the Output Reader object.
outputs = OutputReader(run_name=run_name, path=outputs_path)
```

```
# If we already have a model configuration that we built in-memory (such as the "m"␣
↪GR4JCN model we built above),
# then we can use the Emulator object to simply emulate the model we were working on and␣
↪get outputs directly
from ravenpy import Emulator

# Prepare the emulator by writing files on disk
e = Emulator(config=m)

# Run the model and get the outputs.
outputs = e.run()
```

However, the above demonstration shows how to use one of the eight emulators to run a Raven simulation. Ravenpy also contains other powerful tools to run other user-defined raven models by reading existing configuration files and running the model in Ravenpy. This means that users that already have Raven models of their systems can upload the configuration and hydrometeorological data netcdf files to their private account on the PAVICS-Hydro server and run their model there. Here is an example of how this could be done.

At this stage, no matter the method we used, we have the outputs of the model simulations in the "outputs" object. Let's explore it:

```
# Show the files available in the outputs. Each of these can be accessed to get␣
↪information about the simulation.
outputs.files
```

The outputs are as follow:

- hydrograph: The actual simulated hydrograph (q_sim), in netcdf format. It also contains the observed discharge (q_obs) if observed streamflow was provided as a forcing file.
- storage: The state variables of the simulation duration, in netcdf format
- solution: The state variables at the end of the simulation, which are saved as a ".rvc" file that can be used to hot-start a model (for forecasting, for example)
- messages: A list of messages returned by Raven when executing the run.

You can explore the outputs using othe following syntax. This loads the data into memory to be used directly in another cell for processing or analysis.

```
# The model outputs are actually already loaded as Python objects in memory, thus we can␣
↪access the data directly.
print("----------------HYDROGRAPH----------------")
display(outputs.hydrograph)
print("")
print("-----------------STORAGE------------------")
display(outputs.storage)
print("")
print("-----------------SOLUTION-----------------")
display(outputs.solution)
print("")
```

We can see in the "hydrograph" section that the model has generated a simulation using the forcing data we provided, but it only used the period between the start_date and end_date we asked it for. We can see that the dates of the ERA5 data we requested in the previous notebook cover the period 1980-01-01 to 1991-01-01. In our simulation, we only ask to run over the period from 1985-01-01 to 1990-01-01. Raven takes care of subsetting the data for the required period. We can look at the simulated streamflow from Raven to confirm this:

```
# Import the graphing utility built to handle Raven model outputs
from ravenpy.utilities.nb_graphs import hydrographs

hydrograph_objects = outputs.hydrograph
hydrographs(hydrograph_objects)
```

As you can see, the simulated flow covers only the period we asked for. The results probably don't look good, but that's OK! We will soon calibrate our model to get reasonable parameters.

We could also simply do basic plots using:

```
outputs.hydrograph.q_sim.plot()
```

Finally, we can inspect and work with other state variables in the model outputs. For example, say we want to investigate the snow water equivalent timeseries. We can first get the list of available state variables:

```
print(list(outputs.storage.keys()))
```

And then plot the variable of interest:

```
# Plot the "Snow" variable
outputs.storage["Snow"].plot()
```

As you can see, PAVICS-Hydro makes it easy to build a hydrological model, run it with forcing data, and then interact with the results! In the next notebooks, we will see how to adjust configuration files (the .rvX files) to setup and run a model, and also how to calibrate its parameters.

Supplementary information on Hydrological response unit definition

Raven requires a description of the watershed streamflow is simulated in. Different models require different parameters, but minimally, area, elevation, latitude and longitude are required. These data need to be provided for a few reasons:

- Area is required since the size of the watershed will directly influence the simulated streamflow. Units are in square kilometers (km^2).
- Elevation (average elevation of the watershed) is required, although in many models the value is not actually used and therefore can be set to an arbitrary number. We strongly recommend using the real elevation as that will ensure that the value is present if you decide to switch to another model that requires elevation. Elevation is expressed in meters above mean sea level.
- Latitude and longitude refer to the catchment centroid, and are used, among others, for evapotranspiration computations. They are expressed in decimal degrees (°), with longitudes within [-180, 180].

These values should be either precomputed externally, or they can be computed using the PAVICS-Hydro geophysical extraction toolbox that we used in the second tutorial notebook.

Supplementary information on model parameters

Each model requires a set of tuning parameters to represent and compensate for unknown quantities in certain hydrological processes. Some models have more parameters than others, for example:

- GR4JCN = 6 parameters
- HMETS = 21 parameters
- MOHYSE = 10 parameters
- HBVEC = 21 parameters

These parameters are found through calibration by tuning their values until the simulated streamflow matches the observations as much as possible. PAVICS-Hydro provides an integrated calibration toolbox that will be explored in the the 6th step of this tutorial. For now, we simply provided a set of parameters but it is not yet fully calibrated. This explains the poor quality of the simulated hydrograph.

Explore!

With this information in mind, you can now explore running different models and parameters and on different periods, and display the simulated hydrographs. You can change the start and end dates, the area, latitude, and even add other options that you might find in the documentation or in later tutorials.

If you want to run other models than GR4JCN, you can use these preset models:

HMETS:

```
m = HMETS(
    params=(
        9.5019,
        0.2774,
        6.3942,
        0.6884,
        1.2875,
        5.4134,
```

(continues on next page)

(continued from previous page)

```
        2.3641,
        0.0973,
        0.0464,
        0.1998,
        0.0222,
        -1.0919,
        2.6851,
        0.3740,
        1.0000,
        0.4739,
        0.0114,
        0.0243,
        0.0069,
        310.7211,
        916.1947,
    ),
    **default_emulator_config,
)
```

Mohyse:

```
m = Mohyse(
    params=(1.0, 0.0468, 4.2952, 2.658, 0.4038, 0.0621, 0.0273, 0.0453, 0.9039, 5.6167),
    **default_emulator_config,
)
```

HBVEC:

```
m = HBVEC(
    params=(
        0.059845,
        4.07223,
        2.00157,
        0.034737,
        0.09985,
        0.506,
        3.4385,
        38.32455,
        0.46066,
        0.06304,
        2.2778,
        4.8737,
        0.5718813,
        0.04505643,
        0.877607,
        18.94145,
        2.036937,
        0.4452843,
```

(continues on next page)

(continued from previous page)

```
        0.6771759,
        1.141608,
        1.024278,
    ),
    **default_emulator_config,
)
```

CanadianShield:

```
# The CanadianShield model needs atleast 2 HRUs. We have to modify the default config␣
↪before executing it.
default_emulator_config["HRUs"] = [hru, hru]

m = CanadianShield(
    params=(
        4.72304300e-01,
        8.16392200e-01,
        9.86197600e-02,
        3.92699900e-03,
        4.69073600e-02,
        4.95528400e-01,
        6.803492000e00,
        4.33050200e-03,
        1.01425900e-05,
        1.823470000e00,
        5.12215400e-01,
        9.017555000e00,
        3.077103000e01,
        5.094095000e01,
        1.69422700e-01,
        8.23412200e-02,
        2.34595300e-01,
        7.30904000e-02,
        1.284052000e00,
        3.653415000e00,
        2.306515000e01,
        2.402183000e00,
        2.522095000e00,
        5.80344900e-01,
        1.614157000e00,
        6.031781000e00,
        3.11129800e-01,
        6.71695100e-02,
        5.83759500e-05,
        9.824723000e00,
        9.00747600e-01,
        8.04057300e-01,
        1.179003000e00,
        7.98001300e-01,
    ),
```

(continues on next page)

(continued from previous page)

```
    **default_emulator_config,
)
```

HYPR:

```
m = HYPR(
    params=(
        -1.856410e-01,
        2.92301100e00,
        3.1194200e-02,
        4.3982810e-01,
        4.6509760e-01,
        1.1770040e-01,
        1.31236800e01,
        4.0417950e-01,
        1.21225800e00,
        5.91273900e01,
        1.6612030e-01,
        4.10501500e00,
        8.2296110e-01,
        4.15635200e01,
        5.85111700e00,
        6.9090140e-01,
        9.2459950e-01,
        1.64358800e00,
        1.59920500e00,
        2.51938100e00,
        1.14820100e00,
    ),
    **default_emulator_config,
)
```

SACSMA:

```
m = SACSMA(
    params=(
        0.0100000,
        0.0500000,
        0.3000000,
        0.0500000,
        0.0500000,
        0.1300000,
        0.0250000,
        0.0600000,
        0.0600000,
        1.0000000,
        40.000000,
        0.0000000,
```

(continues on next page)

(continued from previous page)

```
        0.0000000,
        0.1000000,
        0.0000000,
        0.0100000,
        1.5000000,
        0.4827523,
        4.0998200,
        1.0000000,
        1.0000000,
    ),
    **default_emulator_config,
)
```

Blended:

```
m = Blended(
    params=(
        2.930702e-02,
        2.211166e00,
        2.166229e00,
        0.0002254976,
        2.173976e01,
        1.565091e00,
        6.211146e00,
        9.313578e-01,
        3.486263e-02,
        0.251835,
        0.0002279250,
        1.214339e00,
        4.736668e-02,
        0.2070342,
        7.806324e-02,
        -1.336429e00,
        2.189741e-01,
        3.845617e00,
        2.950022e-01,
        4.827523e-01,
        4.099820e00,
        1.283144e01,
        5.937894e-01,
        1.651588e00,
        1.705806,
        3.719308e-01,
        7.121015e-02,
        1.906440e-02,
        4.080660e-01,
        9.415693e-01,
        -1.856108e00,
        2.356995e00,
        1.0e00,
```

(continues on next page)

(continued from previous page)

```
        1.0e00,
        7.510967e-03,
        5.321608e-01,
        2.891977e-02,
        9.605330e-01,
        6.128669e-01,
        9.558293e-01,
        1.008196e-01,
        9.275730e-02,
        7.469583e-01,
    ),
    **default_emulator_config,
)
```

7.1.6 05 - Advanced RavenPy configuration

In this notebook, we will explore alternative ways to setup a Raven model and how to parameterize and customize a raven-based hydrological model

Running Raven using pre-existing configuration files

To run Raven, we need configuration (`.rvX`) files defining hydrological processes, watersheds and meteorological data. If you already have those configuration files ready, or want to see how to import an existing Raven model into PAVICS-Hydro, this tutorial is for you. It shows how to run Raven from a Python programming environment using RavenPy.

Let's start by importing some utilities that will make our life easier to get data on the servers. If you already have raven model setups, you could simply upload the files here and create your own "config" list:

```
# Utility that simplifies getting data hosted on the remote PAVICS-Hydro data server.
from ravenpy.utilities.testdata import get_file
```

A note on datasets

For this part of the tutorial, we will use pre-existing datasets that are hosted on the PAVICS-Hydro servers to setup the Raven model. This means that the .rv files are all built and the forcing file already exists. We could apply all of the same logic to a RavenPy model we would have built at the previous step, but this way lets us show that we can also work on an imported model. Let's import the configuration files:

```
# Get the .rv files. It could also be the .rv files returned from the previous notebook,␣
↪but here we are using a new basin that contains observed streamflow
# to make the calibration possible in the next notebook. Note that these configuration␣
↪files also include links to the
# required hydrometeorological database (NetCDF file).
config = [
    get_file(f"raven-gr4j-cemaneige/raven-gr4j-salmon.{ext}")
    for ext in ["rvt", "rvc", "rvi", "rvh", "rvp"]
]
config
```

So "config" is just a set of paths to the various .rvX files (.rvt, .rvc, .rvi. .rvh and .rvp). Therefore, if you have your own .rv files that describe your model, you can upload them and replace "config" with your own files!

Building a hydrological model on-the-fly using existing configuration files.

Here we create a Raven model instance, configuring it using the pre-defined configuration files and running it by providing the full path to the NetCDF driving datasets. The configuration we provide is for a GR4J-CN model emulator that Raven will run for us. We provide the configuration files for GR4J-CN as well as the forcing data (precipitation, temperature, observed streamflow, etc.) that will be used to run the model.

```
from ravenpy import OutputReader
from ravenpy.ravenpy import run

run_name = "raven-gr4j-salmon"  # As can be seen in the config above, this is the name␣
↪of the .rvX files.
configdir = config[
    0
].parent  # We can get the path to the folder containing the .rvX files this way

# Run the model and get the path to outputs
outputs_path = run(modelname=run_name, configdir=configdir, overwrite=True)

# Note. The modelname parameter can be confusing. You need to give the FILES extension␣
↪name (run_name in our case),
# not the name of the model.

outputs_path
```

```
# Read the output files at the output_path

outputs = OutputReader(run_name=None, path=outputs_path)  # Get the outputs
# Note. We set up the run_name to None, because we didn't rename the output files. If you␣
↪gave a different name to your file
# compared to the one above, you should change the run_name value to this new name. It's␣
↪important though that you keep the end
# of the filename the same

# Show the list of files that were retrived by the OutputReader
outputs.files
```

The model should have run! But you also might have seen some warnings that Raven is giving us, depending on the input files used:

- Some might be saying that we are providing rain and snow independently, but in the configuration files, we are asking the model to recompute the separation using an algorithm based on total precipitation and air temperature. This is OK, and we can live with this (alternatively, we could reconfigure the model to remove this but that will be for another notebook!).
- Others could be saying that we supply PET data, but the model is configured to compute PET from the available temperature and latitude/longitude data. This is also acceptable to us for now, so these warnings can be disregarded.
- And others might simply explain that our configuration provided some parameters but others were computed internally based on our parameter set rather than being explicitly set in our configuration, which is OK.

Evaluating the model response

That's it! The code above has launched the GR4J-CN model using weather data and the configuration we provided. There are many other options we could provide, but for now we left everything to the default options to keep things simple. We will explore those in a future tutorial as well.

Now, let's look at the modeled hydrographs. Note that there is a "q_obs" hydrograph, representing the observations we provided ourselves. This is to facilitate the comparison between observations and simulations, and it is not required per se to run the model. The "q_sim" variable is the simulated streamflow and is the one we are interested in.

Note that RavenPy assumes that model outputs are always saved in netCDF format, and relies on xarray to access data.

To see results, we must first tell the model to read them from the files Raven has written in the output folder:

We can visualize the simulated streamflow using xarray's built-in plotting tool, as follows:

```
outputs.hydrograph.q_sim.plot()
```

We also now have access to diagnostics! This is because along with the simulated discharge, the model has access to observed discharge to compute error metrics such as RMSE and NSE. Let's see where the file has been generated:

```
print("-----------------DIAGNOSTICS-----------------")
print(outputs.diagnostics)
print("")

print("-----------------NASH_SUTCLIFFE-----------------")
print(outputs.diagnostics["DIAG_NASH_SUTCLIFFE"])
print("")

print("-----------------RMSE-----------------")
print(outputs.diagnostics["DIAG_RMSE"])
```

We can see that the Nash-Sutcliffe value is quite poor. This is due to the short simulation period in the configuration (see the hydrograph above!) and the lack of a spin-up period, combined to a poor parameter set choice. We will improve upon all of these shortcomings in the next notebooks!

Advanced RavenPy configuration options

Raven can perform many operations and has multiple configuration options. Here we provide a list of configuration options to explore which you can eventually use to taylor the codes to your own specifications. These can only be run on RavenPy-built hydrological models, and will not operate on Raven models imported by users since those configuration files are not modifiable for the time being.

We will give an overview of the various configuration keywords after this code block, but users should read the Raven documentation for more options for each of these processes.

Let's first define some variables we will need for all of our tests:

```
# Get required packages
import datetime as dt

import matplotlib.pyplot as plt

from ravenpy import Emulator
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
```

(continues on next page)

(continued from previous page)

```
# Observed weather data for the Salmon river. We extracted this using Tutorial Notebook␣
↪03 and the
# salmon_river.geojson file as the contour.
ts = get_file("notebook_inputs/ERA5_weather_data_Salmon.nc")

# Set alternate variable names in the timeseries data file
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# Provide the type of data made available to Raven
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]

# Prepare the catchment properties
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Add some information regarding station data
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
}

# Start and end dates of the simulation
start_date = dt.datetime(1985, 1, 1)
end_date = dt.datetime(1990, 1, 1)

# Set parameters
parameters = [0.529, -3.396, 407.29, 1.072, 16.9, 0.947]
```

We can now perform a “basic” run, with no modifications.

```
# Run the model (See Notebook 04 for more details on implementation)
m = GR4JCN(
    params=parameters,
    Gauge=[
        rc.Gauge.from_nc(
            ts,
            data_type=data_type,  # Note that this is the list of all the variables
            alt_names=alt_names,  # Note that all variables here are mapped to their␣
↪names in the netcdf file.
```

(continues on next page)

(continued from previous page)

```
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="NB05_test1",
    # GlobalParameter={"AVG_ANNUAL_RUNOFF": 208.480},
)

# Run the model and get the outputs.
outputs1 = Emulator(m).run()

# Plot the generated hydrograph
outputs1.hydrograph.q_sim.plot.line(x="time", label="Base case")
plt.legend(loc="upper left")
plt.show()
```

We can now run another model by adding some other properties. To start, we can add some Global Parameters to the model to make Raven adjust the simulations based on the information we provide. Some options of Global Parameters are indicated here, but more can be found in the official Raven documentation.

Examples of GlobalParameter options (Note that some are only available for certain models and others can be mutually exclusive. Please refer to the documentation for this type of adjustment):

Temperature interval of transformation between rain and snow. Set the midpoint of the range and the width of the range, in degrees C:

“RAINSNOW_TEMP”: midpoint_temp // Ex: “RAINSNOW_TEMP”: -1.0

“RAINSNOW_DELTA”: delta_temp // Ex: “RAINSNOW_DELTA”: 3.0

Maximum liquid water content of snow, as a percentage of SWE (0-1). Usually ~0.05.

“SNOW_SWI”: saturation // Ex: “SNOW_SWI”: 0.1

Average annual snow for the entire watershed in mm of SWE. Used in CemaNeige.

“AVG_ANNUAL_SNOW”: average_snow_per_year // Ex: “AVG_ANNUAL_SNOW”: 400.0

There are many others, but this should clarify the implementation. Let’s try some of them out!

```
# Run the model (See Notebook 04 for more details on implementation)
m = GR4JCN(
    params=parameters,
    Gauge=[
        rc.Gauge.from_nc(
            ts,
            data_type=data_type,  # Note that this is the list of all the variables
            alt_names=alt_names,  # Note that all variables here are mapped to their␣
↪names in the netcdf file.
```

(continues on next page)

(continued from previous page)

```
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="NB05_test2",
    GlobalParameter={"AVG_ANNUAL_SNOW": 350.0},
)

# Run the model and get the outputs.
outputs2 = Emulator(m).run()

# Plot the generated hydrograph
outputs1.hydrograph.q_sim.plot.line(x="time", label="Base case")
outputs2.hydrograph.q_sim.plot.line(x="time", label="With AVG_ANNUAL_SNOW")

plt.legend(loc="upper left")
plt.show()
```

We can also adjust the time series data to play with the scaling of units.

By default, RavenPy and Raven will detect units from the forcing data netcdf files. However, in some instances, units might be lacking, or their format might require some tinkering. One such case is for precipitation data that is cumulative in the netcdf file. In these cases, Raven can decumulate the precipitation, but the scaling might lead to undesirable results. For this reason, it is highly recommended to pass the scaling and offsetting variables directly. To do so, add some context in the data_kwds:

```
# Add some information regarding station data
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
    # HOW TO PROCESS THE PRECIPITATION DATA: For the Precip variable, we tell Raven we␣
↪want to Deaccumulate
    # values, shift them in time by 6 hours (for UTC time zone management), and then␣
↪apply a linear transform
    # to the values to get new scaled values. The linear transform can take two inputs:
    #     "scale"   is the "a" variable in the linear relationship y = ax + b. Usually␣
↪used to multiply precipitation.
    #     "offset"  is the "b" variable in the linear relationship y = ax + b. Usually␣
↪used to convert temperatures(K to °C)
    "PRECIP": {
        "Deaccumulate": True,
        "TimeShift": -0.25,
        "LinearTransform": {
            "scale": 1000.0  # # Converting meters to mm (multiply by 1000).
        },
    },
    "TEMP_AVE": {
        "TimeShift": -0.25,
```

(continues on next page)

(continued from previous page)

```
    },
}
```

In our example, our precipitation is not actually accumulated and the timestep is daily, so we don't need the "Deaccumulate" or the "TimeShift" parameters. So let's generate a new data_kwds that is applicable in our case. More complex cases that require "Deaccumulate" and "TimeShift" will be presented in later notebooks that use accumulated precipitation in forecasting applications, in Notebook 12.

```
# Add some information regarding station data
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
    # Let's simulate a very rough estimation of the impacts of climate change where␣
↪precipitation is expected
    # to increase by 10% and temperatures to increase by 3°C. This will be applied to␣
↪all data on the entire
    # period and is thus not realistic. We will explore more realistic methods in␣
↪Notebook 08.
    "PRECIP": {"LinearTransform": {"scale": 1.1}},
    "TEMP_AVE": {"LinearTransform": {"offset": 3.0}},
}
```

Now we can use this new setup to generate another series of streamflow

```
# Run the model (See Notebook 04 for more details on implementation)
m = GR4JCN(
    params=parameters,
    Gauge=[
        rc.Gauge.from_nc(
            ts,
            data_type=data_type,  # Note that this is the list of all the variables
            alt_names=alt_names,  # Note that all variables here are mapped to their␣
↪names in the netcdf file.
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="NB05_test3",
    GlobalParameter={"AVG_ANNUAL_SNOW": 350.0},
)

# Run the model and get the outputs.
outputs3 = Emulator(m).run()

# Plot the generated hydrograph
outputs1.hydrograph.q_sim.plot.line(x="time", label="Base case")
outputs2.hydrograph.q_sim.plot.line(x="time", label="With AVG_ANNUAL_SNOW")
```

(continues on next page)

(continued from previous page)

```
outputs3.hydrograph.q_sim.plot.line(x="time", label="With AVG_ANNUAL_SNOW and Scaling")

plt.legend(loc="upper left")
plt.show()
```

We can see that the scaling increased the flows almost everywhere except in the first year which is the warm-up period.

Other options that can be implemented are indicated here, although more exist and are documented in the official Raven manual.

RainSnowFraction:

Algorithm to use to separate the total precipitation into rainfall and snowfall.

Ex: RainSnowFraction='RAINSNOW_DINGMAN'

Evaporation

Evaporation: Formula to use to compute the evapotranspiration from the land HRUs.

Ex: Evaporation="PET_OUDIN"

Suppress model outputs / files

Boolean that indicates if you wish for Raven to provide information after the model evaluation by writing to file. For a single run this can be left to **False**, but for calibration and other intensive tasks, it is faster to leave it to **True**.

Ex: SuppressOutputs=True

Finally, let's see how to implement these commands:

```
# Run the model (See Notebook 04 for more details on implementation)
m = GR4JCN(
    params=parameters,
    Gauge=[
        rc.Gauge.from_nc(
            ts,
            data_type=data_type,  # Note that this is the list of all the variables
            alt_names=alt_names,  # Note that all variables here are mapped to their␣
↪names in the netcdf file.
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="NB05_test3",
    GlobalParameter={"AVG_ANNUAL_SNOW": 350.0},
    RainSnowFraction="RAINSNOW_DINGMAN",
    Evaporation="PET_HARGREAVES_1985",
    SuppressOutput=False,  # We can't read the hydrographs if they are not written to␣
↪disk, so set to False here.
```

(continues on next page)

(continued from previous page)

```
)

# Run the model and get the outputs.
outputs4 = Emulator(m).run()

# Plot the generated hydrograph
outputs1.hydrograph.q_sim.plot.line(x="time", label="Base case")
outputs2.hydrograph.q_sim.plot.line(x="time", label="With AVG_ANNUAL_SNOW")
outputs3.hydrograph.q_sim.plot.line(x="time", label="With AVG_ANNUAL_SNOW and Scaling")
outputs4.hydrograph.q_sim.plot.line(
    x="time", label="With AVG_ANNUAL_SNOW, Scaling and Options"
)

plt.legend(loc="upper left")
plt.show()
```

A note on the above results

We can see that the results change significantly according to the options we have passed, namely the evaporation algorithm modified the hydrograph quite significantly. However, this is caused by the fact that the parameter set we have used has not been calibrated using this PET method, and thereore the model cannot be expected to perform as well. This means that when using these model options, it is important to recalibrate the model parameters such that they represent the actual model being used!

Finally, we can also ask Raven to supply custom outputs using this line in the model configuration:

CustomOutput=rc.CustomOutput() and by providing a list of desired pre-processed variables. Here we ask for the yearly average of precipitation over the entire watershed:

CustomOutput=rc.CustomOutput("YEARLY", "AVERAGE", "PRECIP", "ENTIRE_WATERSHED")

Please see the documentation for more details on using custom outputs.

7.1.7 06 - Calibration of a Raven hydrological model

Calibration of a Raven model

In this notebook, we show how to calibrate a Raven model using the GR4J-CN predefined structure. Users can refer to the documentation for the parameterization of other hydrological model structures.

Let's start by importing the packages that will do the work.

```
import datetime as dt

import spotpy

from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities.calibration import SpotSetup
```

Preparing the model to be calibrated on a given watershed

Our test watershed from the last notebook is selected for this test. It can be replaced with any desired watershed.

```
from ravenpy.utilities.testdata import get_file

# We get the netCDF for testing on a server. You can replace the getfile method by a␣
↪string containing the path to your own netCDF
nc_file = get_file(
    "raven-gr4j-cemaneige/Salmon-River-Near-Prince-George_meteo_daily.nc"
)

# Display the dataset that we will be using
print(nc_file)
```

The process is very similar to setting up a hydrological model. We first need to create the model with its configuration. We must provide the same information as before, except for the model parameters since those need to be calibrated.

```
# Here, we need to give the name of your different dataset in order to match with Raven␣
↪models.
alt_names = {
    "RAINFALL": "rain",
    "SNOWFALL": "snow",
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PET": "pet",
    "HYDROGRAPH": "qobs",
}

# The HRU of your watershed
hru = dict(area=4250.6, elevation=843.0, latitude=54.4848, longitude=-123.3659)

# You can decide the evaluation metrics that will be used to calibrate the parameters of␣
↪your model. You need atleast
# 1 evaluation metric, but you can do any combinaison of evaluations from this list :
#
# NASH_SUTCLIFFE,
# LOG_NASH,
# RMSE,
# PCT_BIAS,
# ABSERR,
# ABSMAX,
# PDIFF,
# TMVOL,
# RCOEFF,
# NSC,
# KLING_GUPTA
eval_metrics = ("NASH_SUTCLIFFE",)


# We need to create the desired model with its parameters the same way as in the␣
↪Notebook 04_Emulating_hydrological_models.
model_config = GR4JCN(
    ObservationData=[rc.ObservationData.from_nc(nc_file, alt_names="qobs")],
```

(continues on next page)

(continued from previous page)

```
    Gauge=[
        rc.Gauge.from_nc(
            nc_file,
            alt_names=alt_names,
            data_kwds={"ALL": {"elevation": hru["elevation"]}},
        )
    ],
    HRUs=[hru],
    StartDate=dt.datetime(1990, 1, 1),
    EndDate=dt.datetime(1999, 12, 31),
    RunName="test",
    EvaluationMetrics=eval_metrics,  # We add this code to tell Raven which objective␣
↪function we want to pass.
    SuppressOutput=True,
)
```

Spotpy Calibration

Once you've created your model, you need to create a SpotSetup, which will be used to calibrate your model.

```
# In order to calibrate your model, you need to give the lower and higher bounds of the␣
↪model. In this case, we are passing
# the boundaries for a GR4JCN, but it's important to change them, if you are using␣
↪another model. Note that the list of these
# boundaries for each model is at the end of this notebook.
low_params = (0.01, -15.0, 10.0, 0.0, 1.0, 0.0)
high_params = (2.5, 10.0, 700.0, 7.0, 30.0, 1.0)


spot_setup = SpotSetup(
    config=model_config,
    low=low_params,
    high=high_params,
)
```

Now that the model is setup and configured and that `SpotSetup` object exists, we need to create a sampler from `spotpy` module which will optimize the hydrological model paramaters. You can see that we are using the DDS algorithm to optimize the parameters:

[Tolson, B.A. and Shoemaker, C.A., 2007. Dynamically dimensioned search algorithm for computationally efficient watershed model calibration. Water Resources Research, 43(1)].

If you want to use another algorithm, please refer to the Spotpy documentation here : https://spotpy.readthedocs.io/

Finally, we run the sampler by the amount of desired repetitions.

```
# Number of total model evaluations in the calibration. This value should be over 500␣
↪for real optimisation,
# and upwards of 10000 evaluations for models with many parameters. This will take a␣
↪LONG period of time so
# be sure of all the configuration above before executing with a high number of model␣
↪evaluations.
model_evaluations = 10
```

(continues on next page)

(continued from previous page)

```
# Set up the spotpy sampler with the method, the setup configuration, a run name and␣
↪other options. Please refer to
# the spotpy documentation for more options. We recommend sticking to this format for␣
↪efficiency of most applications.
sampler = spotpy.algorithms.dds(
    spot_setup, dbname="RAVEN_model_run", dbformat="ram", save_sim=False
)

# Launch the actual optimization. Multiple trials can be launched, where the entire␣
↪process is repeated and
# the best overall value from all trials is returned.
sampler.sample(model_evaluations, trials=1)
```

Analysing the calibration results

The best parameters as well as the objective functions can be analyzed.

```
# Get all the values of each iteration
results = sampler.getdata()

print("The best Nash-Sutcliffe value is : ")

# Get the raw resutlts directly in an array
bestindex, bestobjfun = spotpy.analyser.get_maxlikeindex(
    results
)  # Want to get the MAX NSE (change for min for RMSE)
best_model_run = list(
    results[bestindex][0]
)  # Get the parameter set returning the best NSE
optimized_parameters = best_model_run[
    1:-1
]  # Remove the NSE value (position 0) and the ID at the last position to get the actual␣
↪parameter set.

print("\nThe best parameters are : ")
# Display the parameter set ready to use in a future run:
print(optimized_parameters)
```

Next steps

In the next notebooks, we will apply the model to specific use-cases, including making and using hotstart files for forecasting, performing hindcasting and forecasting, applying data assimilation and evaluating the impacts of climate change on the hydrology of a watershed. In the meantime, you can explore calibration with any of the emulated models below with the provided low and high bounds. You can also provide your own for specific cases.

List of Model-Boundaries

GR4J-CN :

HMETS :

Mohyse :

HBV-EC :

CanadianShield :

HYPR :

SACSMA :

Blended :

7.1.8 07 - Making and using hostart files

Create a hotstart file to resume a simulation from given hydrological conditions

Hydrological models have state variables that describe the snow pack, soil moisture, underground reservoirs, etc. Typically, those cannot be measured empirically, so one way to estimate those values is to run the model for a period before the period we are actually interested in, and save the state variables at the end of this *warm-up* simulation.

This notebook shows how to save those state variables and use them to configure another Raven simulation. These *states* are configured by the `:HRUStateVariableTable` and `:BasinStateVariables` commands, but `ravenpy` has a convenience function `set_solution` to update those directly from the `solution.rvc` simulation output.

In the following, we run the model on two years then save the final states. Next, we use those final states to configure the initial state of a second simulation over the next two years. If everything is done correctly, these two series should be identical to a simulation over the full four years.

Model configuration

At this point the following blocks of code should be quite familiar! If not, please go back to notebook "04 - Emulating hydrological models" to understand what is happening.

```
# Import packages
import datetime as dt
import warnings

from matplotlib import pyplot as plt

from ravenpy import Emulator, RavenWarning
from ravenpy.config import commands as rc

# Import the GR4JCN model
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities.testdata import get_file
```

```
# Start and end date for full simulation
# Make sure the end date is before the end of the hydrometeorological data NetCDF file.
start_date = dt.datetime(1986, 1, 1)
end_date = dt.datetime(1988, 1, 1)
```

(continues on next page)

(continued from previous page)

```
# Define HRU
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Get dataset:
ERA5_full = get_file("notebook_inputs/ERA5_weather_data.nc")

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
}

# Model configuration
config = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            ERA5_full,
            data_type=data_type,
            alt_names=alt_names,
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="full",
)
```

```
# Silence the Raven warnings
warnings.simplefilter("ignore", category=RavenWarning)

# Run the model and get the outputs.
out1 = Emulator(config=config).run()
```

(continues on next page)

(continued from previous page)

```
# Plot the model output
out1.hydrograph.q_sim.plot(label="Part 1")
plt.legend()
```

Now let's run the model for the next two years, setting the initial conditions to the final states of the first simulation.

The path to the `solution.rvc` file can be found in `out1.files["solution"]`. The content itself can be displayed with `out1.solution`

```
# The path to the solution (final model states)
hotstart = out1.files["solution"]

# Configure and run the model, this time with the next two years first 3 years (1988-
↪1990).
conf2 = config.set_solution(hotstart)
conf2.start_date = dt.datetime(1988, 1, 1)
conf2.end_date = dt.datetime(1990, 1, 1)
conf2.run_name = "part_2"

out2 = Emulator(config=conf2).run()

# Plot the model output
out2.hydrograph.q_sim.plot(label="Part 2")
plt.legend()
```

Compare with simulation over entire period

Now in theory, those two simulations should be identical to one simulation over the whole period of four years, let's confirm this.

```
full = config.copy()
full.end_date = dt.datetime(1990, 1, 1)
full.run_name = "full"

out = Emulator(config=full).run(overwrite=True)

out.hydrograph.q_sim.plot(label="Full", color="gray", lw=4)
out1.hydrograph.q_sim.plot(
    label="Part 1",
    color="blue",
    lw=0.5,
)
out2.hydrograph.q_sim.plot(label="Part 2", color="orange", lw=0.5)
plt.legend()
```

And now if we look at the difference between both hydrographs, we can see that there are differences in the second part at machine precision levels, due to rounding in the hotstart file (note that the y-axis is 1e-6, which is essentially 0!). But the rest is perfect!

Therefore, we can provide forecasting abilities by saving simulation final states and using those to initialize model states for the forecasting runs. This will be used in other notebooks such as notebook #12 on hindcasting.

```
import numpy as np

delta1 = np.abs(out1.hydrograph.q_sim - out.hydrograph.q_sim)
delta2 = np.abs(out2.hydrograph.q_sim - out.hydrograph.q_sim)

delta1.plot(label="Part 1")
delta2.plot(label="Part 2")
plt.title("Difference between two parts and full simulation")
```

7.1.9 08 - Getting and bias-correcting CMIP6 climate data

Applying bias correction on climate model data to perform climate change impact studies on hydrology

This notebook will guide you on how to conduct bias correction of climate model outputs that will be fed as inputs to the hydrological model Raven to perform climate change impact studies on hydrology.

Geographic data

In this tutorial, we will be using the shapefile or GeoJSON file for watershed contours as generated in previous notebooks. The file can be uploaded to your workspace here and used directly in the cells below. In this notebook, we present a quick demonstration of the bias-correction approach on a small and predetermined dataset, but you can use your own basin according to your needs.

```
import warnings

from numba.core.errors import NumbaDeprecationWarning

warnings.simplefilter("ignore", category=NumbaDeprecationWarning)
```

```
import datetime as dt
import tempfile
from pathlib import Path

import gcsfs
import intake
import numpy as np
import xarray as xr
import xclim
import xclim.sdba as sdba
from clisops.core import average, subset

from ravenpy.utilities.testdata import get_file

tmp = Path(tempfile.mkdtemp())
```

Application to a real catchment and test-case.

In this notebook, we will perform bias-correction on a real catchment using real data! You can change the input file for the contours, the catchment properties and other such parameters. The previous notebooks show how to extract basin area, latitude, and longitude, so use those to generate the required information if it is not readily available for your catchment.

Let's first start by providing some basic information:

- basin_contour: The shapefile or geojson of the watershed boundaries (if it is a shapefile, it has to be a zip-file containing the .shp, .shx and .prj files)
- reference_start_day: The start day of the reference period
- reference_end_day: The end day of the reference period
- future_start_day: The start day of the future period
- future_end_day: The end day of the future period
- climate_model: The name of the climate model. Must be selected from the available list for this notebook. However, if you want to use other data, the bias-correction step will still be applicable if you follow the same logic and formats as shown here.

```
# We get the basin contour for testing on a server. You can replace the getfile method␣
↪by a string containing the path
# to your own geojson

# Get basin contour
basin_contour = get_file("notebook_inputs/input.geojson")

reference_start_day = dt.datetime(1980, 12, 31)
reference_end_day = dt.datetime(1991, 1, 1)
# Notice we are using one day before and one day after the desired period of 1981-01-01␣
↪to 1990-12-31.
# This is to account for any UTC shifts that might require getting data in a previous or␣
↪later time.

future_start_day = dt.datetime(2080, 12, 31)
future_end_day = dt.datetime(2091, 1, 1)
# Notice we are using one day before and one day after the desired period of 1981-01-01␣
↪to 1990-12-31.
# This is to account for any UTC shifts that might require getting data in a previous or␣
↪later time.

"""
Choose a climate model from the list below, which have the daily data required for Raven.
↪ Depending on the period required, it is possible that some
models will cause errors that need to be adressed specifically using date conversions.␣
↪In those cases, please select another model or adjust the datetime
data to your needs.

ACCESS-CM2
ACCESS-ESM1-5
AWI-CM-1-1-MR
BCC-CSM2-MR
CESM2-WACCM
```

(continues on next page)

(continued from previous page)

```
CMCC-CM2-SR5
CMCC-ESM2
CanESM5
EC-Earth3
EC-Earth3-CC
EC-Earth3-Veg
EC-Earth3-Veg-LR
FGOALS-g3
GFDL-CM4
GFDL-ESM4
INM-CM4-8
INM-CM5-0
IPSL-CM6A-LR
KACE-1-0-G
KIOST-ESM
MIROC6
MPI-ESM1-2-HR
MPI-ESM1-2-LR
MRI-ESM2-0
NESM3
NorESM2-LM
NorESM2-MM
"""

climate_model = "MIROC6"
```

Get CMIP6 data from the cloud

Accessing and downloading climate data can be a painful and time-consuming endeavour. PAVICS-Hydro provides a method to gather data quickly and efficiently, with as little user-input as possible. We use the PanGEO catalog for cloud climate data, and with a few simple keywords, we can automatically extract the required data from the climate model simulations. Furthermore, we can also automatically subset it to our precise location as defined by the watershed boundaries, and also extract only the time period of interest.

Let's start by opening the catalog of available data:

```
# Prepare the filesystem that allows reading data. Data is read on the Google Cloud␣
↪Services, which host a copy of the CMIP6 (and other) data.
fsCMIP = gcsfs.GCSFileSystem(token="anon", access="read_only")

# Get the catalog info from the pangeo dataset, which basically is a list of links to␣
↪the various products.
col = intake.open_esm_datastore(
    "https://storage.googleapis.com/cmip6/pangeo-cmip6.json"
)

# Print the contents of the catalog, so we can see the classification system
display(col)
```

We can see that there are a lot of climate models (source_id), experiments, members, and other classifications. Let's see the list of available models, for example (source_id):

```
# Get the list of models. Replace "source_id" with any of the catalog categories (table_
↪id, activity_id, variable_id, etc.)
list(col.df.source_id.unique())
```

For this notebook, we will work with MIROC6, but you can use any other model from the list established previously.

Now, we can be more selective about what we want to get from the CMIP6 project data:

- source_id: The climate model, in this case 'MIROC6'
- experiment_id: The forcing scenario. Here we will use 'historical' (for the historical period) and for future data we could use any of the SSP simulations, such as 'ssp585' or 'ssp245'.
- table_id: The timestep of the model simulation. Here we will use 'day' for daily data, but some models have monthly and 3-hourly data, for example.
- variable_id: The codename for the variable of interest. Here we will want 'tasmin', 'tasmax', and 'pr' for minimum temperature, maximum temperature and total precipitation, respectively.
- member_id: The code identifying the model member. Some models are run multiple times with varying initial conditions to represent natural variability. Here we will only focus on the first member 'r1i1p1f1'.

You can find more information about available data on the CMIP6 project webpage and [data nodes] (https://esgf-node.llnl.gov/projects/cmip6/).

Let's now see what the PanGEO catalog returns when we ask to filter according to all of these criteria:

```
# Build a query dictionary for all of our requests, for tasmin.
query = dict(
    experiment_id="historical",
    table_id="day",
    variable_id="tasmin",
    member_id="r1i1p1f1",
    source_id=climate_model,
)
col_subset = col.search(
    require_all_on=["source_id"], **query
)  # Command that will return the filtered list

# Show the filtered list:
display(col_subset.df)
```

We can see that the list contains only one item: The daily tasmin variable, for the historical period of member r1i1p1f1 from the MIROC6 model, as requested! We can also see the path where that file resides on the "zstore", which is where it is stored on the Google Cloud service. We can now get the data:

```
# Get the object locator object
mapper = fsCMIP.get_mapper(col_subset.df.zstore[0])
```

The final step is to open the dataset with xarray by using the 'open_zarr()' function. The following block performs multiple operations to get the data that we want:

- It opens the data using xarray
- It extracts only the times that we need for the reference/historical period
- It then subsets it spatially by getting only the points within the catchment boundaries. If your catchments is too small and this fails, try with a larger basin or apply a buffer around your boundaries.
- Since we are running a lumped model, it take the spatial average.

- It will then remove unnecessary coordinates that could cause problems later ('height', in this case)
- It will then rechunk the data into a format that makes it much faster to read and process

Finally, we will display the output of this entire process.

```
# Get the CMIP6 data from Google Cloud and read it in memory using xarray. This is done␣
↪via "lazy loading" and is not actually reading the data in memory
# yet, but is keeping track of what it will need to get, eventually.
ds = xr.open_zarr(mapper, consolidated=True)

# Convert to numpy.datetime64 object to be compatbile
if type(ds.time[0].values) is not type(np.datetime64("1980-01-01")):
    ds = ds.convert_calendar("standard")

# Extract only the dates that we really want. Again, this is done via lazy loading, and␣
↪is not actually using memory at this point.
ds = ds.sel(time=slice(reference_start_day, reference_end_day))

# Use the clisops subsetting tools to extract the data for the watershed boundaries and␣
↪take the spatial average
ds = average.average_shape(ds, basin_contour)

# Correct the coordinates that are unnecessary for our variable
ds = ds.reset_coords("height", drop=True)

# Rechunk the data so it is much faster to read (single chunk rather than 1 chunk per␣
↪day)
historical_tasmin = ds["tasmin"].chunk(-1)

# Show the end result!
display(historical_tasmin)
```

We can see that we have a single chunk of 10 years of tasmin data, as expected! However, you might also have noticed that there is no metadata, such as units and variable properties left in the data array. We can fix that by wrapping the code in a block that forces xarray to keep the metadata.

Also, since we will need to use this block of code for each variable, it might become tedious. Therefore, to simplify the code, we can combine everything into a function.

```
def extract_and_average(mapper, start, end, geometry):
    with xr.set_options(keep_attrs=True):
        ds = xr.open_zarr(mapper, consolidated=True)

        # Convert to numpy.datetime64 object to be compatbile
        if type(ds.time[0].values) is not type(np.datetime64("1980-01-01")):
            ds = ds.convert_calendar("standard")

        # Compute average over region
        out = average.average_shape(ds.sel(time=slice(start, end)), geometry)

        # Convert geometry variables into attributes
        attrs = {
            key: out[key].values.item()
            for key in out.coords
```

(continues on next page)

(continued from previous page)

```
            if key not in ["time", "time_bnds", "lon", "lat"]
        }
        out = out.isel(geom=0).reset_coords(attrs.keys(), drop=True)
        out.attrs.update(attrs)

        return out.chunk(-1)
```

Much better! we have all the information we need. Let's repeat the process for the 3 variables and for the reference and future periods using ssp585. You probably don't have to change anything in this following block of code, but you can taylor it to your needs knowing how everything is built now.

```
# We will add a wrapper to ensure that the following operations will preserve the␣
↪original data attributes, such as units and variable names.
with xr.set_options(keep_attrs=True):
    # Load the files from the PanGEO catalogs, for reference and future variables of␣
↪temperature and precipitation.
    out = {}
    for exp in ["historical", "ssp585"]:
        if exp == "historical":
            period_start = reference_start_day
            period_end = reference_end_day
        else:
            period_start = future_start_day
            period_end = future_end_day

        out[exp] = {}
        for variable in ["tasmin", "tasmax", "pr"]:
            print(exp, variable)
            query = dict(
                experiment_id=exp,
                table_id="day",
                variable_id=variable,
                member_id="r1i1p1f1",
                source_id=climate_model,
            )
            col_subset = col.search(require_all_on=["source_id"], **query)
            mapper = fsCMIP.get_mapper(col_subset.df.zstore[0])
            ds = xr.open_zarr(mapper, consolidated=True)

            out[exp][variable] = extract_and_average(
                mapper, period_start, period_end, basin_contour
            )[variable]

# We can now extract the variables that we will need later:
historical_tasmax = out["historical"]["tasmax"]
historical_tasmin = out["historical"]["tasmin"]
historical_pr = out["historical"]["pr"]
future_tasmax = out["ssp585"]["tasmax"]
future_tasmin = out["ssp585"]["tasmin"]
future_pr = out["ssp585"]["pr"]
```

Reference data to prepare bias correction

We have extracted the historical period and future period data from the GCM. Now we need the reference data to use as the baseline for bias-correction. Here we will use ERA5 and we will gather it again, since we can't be sure that the dates we selected in the 3rd notebook are still valid.

```
# Regenerate the ERA5 data to be sure. In an operational context, you could combine␣
↪everything onto one notebook and ensure that the
# dates and locations are constant!

# Get the ERA5 data from the Wasabi/Amazon S3 server.
catalog_name = "https://raw.githubusercontent.com/hydrocloudservices/catalogs/main/
↪catalogs/atmosphere.yaml"
cat = intake.open_catalog(catalog_name)
ds = cat.era5_reanalysis_single_levels.to_dask()

"""
Get the ERA5 data. We will rechunk it to a single chunck to make it compatible with␣
↪other codes on the platform, especially bias-correction.
We are also taking the daily min and max temperatures as well as the daily total␣
↪precipitation.
"""
# We will add a wrapper to ensure that the following operations will preserve the␣
↪original data attributes, such as units and variable names.
with xr.set_options(keep_attrs=True):
    ERA5_reference = subset.subset_shape(
        ds.sel(time=slice(reference_start_day, reference_end_day)), basin_contour
    ).mean({"latitude", "longitude"})
    ERA5_tmin = ERA5_reference["t2m"].resample(time="1D").min().chunk(-1, -1, -1)
    ERA5_tmax = ERA5_reference["t2m"].resample(time="1D").max().chunk(-1, -1, -1)
    ERA5_pr = ERA5_reference["tp"].resample(time="1D").sum().chunk(-1, -1, -1)
```

```
ERA5_pr.plot()
```

```
# Here we need to make sure that our units are all in the correct format. You can play␣
↪around with the tools we've seen thus far to explore the units
# and make sure everything is consistent.

# Let's start with precipitation:
ERA5_pr = xclim.core.units.convert_units_to(ERA5_pr, "mm", context="hydro")
# The CMIP data is a rate rather than an absolute value, so let's get the absolute values:
historical_pr = xclim.core.units.rate2amount(historical_pr)
future_pr = xclim.core.units.rate2amount(future_pr)

# Now we can actually convert units in absolute terms.
historical_pr = xclim.core.units.convert_units_to(historical_pr, "mm", context="hydro")
future_pr = xclim.core.units.convert_units_to(future_pr, "mm", context="hydro")

# Now let's do temperature:
ERA5_tmin = xclim.core.units.convert_units_to(ERA5_tmin, "degC")
ERA5_tmax = xclim.core.units.convert_units_to(ERA5_tmax, "degC")
historical_tasmin = xclim.core.units.convert_units_to(historical_tasmin, "degC")
historical_tasmax = xclim.core.units.convert_units_to(historical_tasmax, "degC")
```

(continues on next page)

(continued from previous page)

```
future_tasmin = xclim.core.units.convert_units_to(future_tasmin, "degC")
future_tasmax = xclim.core.units.convert_units_to(future_tasmax, "degC")
```

The model is now going to be trained to find correction factors between the reference dataset (observations) and historical dataset (climate model outputs for the same time period). The correction factors obtained are then applied to both reference and future climate outputs to correct them. This step is called the bias correction. In this test-case, we apply a method named `detrended quantile mapping`.

Here we use the `xclim` utilities to bias-correct CMIP6 GCM data using ERA5 reanalysis data as the reference. See `xclim` documentation for more options! (https://xclim.readthedocs.io/en/stable/notebooks/sdba.html)

> **Warning** This following block of code will take a while to run, and some warning messages will appear during the process (related to longitude wrapping and other information on calendar types). Unless an error message appears, the code should run just fine!

```
# Use xclim utilities (sbda) to give information on the type of window used for the bias␣
↪correction.
group_month_window = sdba.utils.Grouper("time.dayofyear", window=15)

# This is an adjusting function. It builds the tool that will perform the corrections.
Adjustment = sdba.DetrendedQuantileMapping.train(
    ref=ERA5_pr, hist=historical_pr, nquantiles=50, kind="+", group=group_month_window
)

# Apply the correction factors on the reference period
corrected_ref_precip = Adjustment.adjust(historical_pr, interp="linear")

# Apply the correction factors on the future period
corrected_fut_precip = Adjustment.adjust(future_pr, interp="linear")

# Ensure that the precipitation is non-negative, which can happen with some climate␣
↪models
corrected_ref_precip = corrected_ref_precip.where(corrected_ref_precip > 0, 0)
corrected_fut_precip = corrected_fut_precip.where(corrected_fut_precip > 0, 0)

# Train the model to find the correction factors for the maximum temperature (tasmax)␣
↪data
Adjustment = sdba.DetrendedQuantileMapping.train(
    ref=ERA5_tmax,
    hist=historical_tasmax,
    nquantiles=50,
    kind="+",
    group=group_month_window,
)

# Apply the correction factors on the reference period
corrected_ref_tasmax = Adjustment.adjust(historical_tasmax, interp="linear")

# Apply the correction factors on the future period
corrected_fut_tasmax = Adjustment.adjust(future_tasmax, interp="linear")

# Train the model to find the correction factors for the minimum temperature (tasmin)␣
↪data
```

(continues on next page)

(continued from previous page)

```
Adjustment = sdba.DetrendedQuantileMapping.train(
    ref=ERA5_tmin,
    hist=historical_tasmin,
    nquantiles=50,
    kind="+",
    group=group_month_window,
)

# Apply the correction factors on the reference period
corrected_ref_tasmin = Adjustment.adjust(historical_tasmin, interp="linear")

# Apply the correction factors on the future period
corrected_fut_tasmin = Adjustment.adjust(future_tasmin, interp="linear")
```

The corrected reference and future data are then converted to netCDF files. This will take a while to run (perhaps a minute or two), since it will need to write the datasets to disk after having processed everything via lazy loading.

```
# Convert the reference corrected data into netCDF file. We will then apply a special␣
↪code to remove a dimension in the dataset to make it applicable to the RAVEN models.
ref_dataset = xr.merge(
    [
        corrected_ref_precip.to_dataset(name="pr"),
        corrected_ref_tasmax.to_dataset(name="tasmax"),
        corrected_ref_tasmin.to_dataset(name="tasmin"),
    ]
)

# Write to temporary folder
fn_ref = tmp / "reference_dataset.nc"
ref_dataset.to_netcdf(fn_ref)

# Convert the future corrected data into netCDF file
fut_dataset = xr.merge(
    [
        corrected_fut_precip.to_dataset(name="pr"),
        corrected_fut_tasmax.to_dataset(name="tasmax"),
        corrected_fut_tasmin.to_dataset(name="tasmin"),
    ]
)

fn_fut = tmp / "future_dataset.nc"
fut_dataset.to_netcdf(fn_fut)
```

```
# Show the corrected future precipitation.
corrected_fut_precip.plot()
```

```
# Compare it to the future precipitation without bias-correction.
future_pr.plot()
```

7.1.10 09 - Hydrological impacts of climate change

Performing bias correction on climate model data to perform climate change impact studies on hydrology

This notebook will allow evaluating the impacts of climate change on the hydrology of a catchment. We will use the data we previously generated in notebook "08 - Getting and bias-correcting CMIP6 data", where we produced both reference and future forcing datasets.

You can apply this notebook to other models, climate datasets, and generally pick and choose parts of various notebooks to build your own complete workflow.

```
"""
Import the required packages
"""
import datetime as dt
import warnings

from matplotlib import pyplot as plt

from ravenpy import Emulator
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities.testdata import get_file

warnings.filterwarnings("ignore")
```

Simulate the flows on the reference period

In this step, we will take the reference period climate data and run the GR4J-CN hydrological model with it. We will then plot a graph to see the streamflow representative of the reference period.

```
# Define the hydrological response unit. We can use the information from the tutorial␣
↪notebook #02! Here we are using
# arbitrary data for a test catchment.
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Define the start and end dates of the reference period:
start_date = dt.datetime(1981, 1, 1)
end_date = dt.datetime(1990, 12, 31)

# We get the netCDF for testing on a server. You can replace the getfile method by a␣
↪string containing the path
# to your own netCDF

reference_ds = get_file("notebook_inputs/reference_dataset.nc")
```

(continues on next page)

(continued from previous page)

```
# Alternate names for the data in the climate data NetCDF files
alt_names = {
    "TEMP_MIN": "tasmin",
    "TEMP_MAX": "tasmax",
    "PRECIP": "pr",
}

# Types of data required by the Raven GR4JCN instance
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
}
# Start a model instance, in this case a GR4JCN model emulator.
m = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            reference_ds,  # path to the reference period dataset.
            data_type=data_type,
            alt_names=alt_names,
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="test",
)

# Prepare the emulator by writing files on disk
e = Emulator(config=m)

# Run the model and get the outputs.
outputs_reference = e.run()

outputs_reference.hydrograph.q_sim.plot(label="Reference", color="blue", lw=0.5)
plt.legend()
```

Now do the same but for the future period!

We will copy the block of code from above, changing only the file path (from reference dataset to future dataset) as well as the start and end dates.

```
# Define the start and end dates of the reference period:
start_date = dt.datetime(2081, 1, 1)
end_date = dt.datetime(2090, 12, 31)

# Get the future period dataset (path)
future_ds = get_file("notebook_inputs/future_dataset.nc")

# Start a new model instance, again in this case a GR4JCN model emulator.
m = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            # name of the future period dataset.
            future_ds,
            data_type=data_type,
            alt_names=alt_names,
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="test",
)

# Prepare the emulator by writing files on disk
e = Emulator(config=m)

# Run the model and get the outputs.
outputs_future = e.run()

outputs_future.hydrograph.q_sim.plot(label="Future", color="orange", lw=0.5)
plt.legend()
```

You have just generated streamflows for the reference and future periods!

We can analyze these hydrographs with many tools in PAVICS-Hydro, or export them to use elsewhere, or use them as inputs to another process!

```
# Get the path to the hydrograph file:
outputs_future.files["hydrograph"]
```

```
# Work with the hydrograph data directly:
outputs_future.hydrograph
```

7.1.11 10 - Data Assimilation

Using ravenpy to perform data assimilation of streamflow to prepare the model states for a forecast.

Here we apply the Ensemble Kalman Filter (EnKF) data assimilation method to the initial states of a Raven hydrological model, which will allow improving the estimation of the initial states to reduce the initial model bias. This also helps improve the forecast skill for shorter-term forecasts (up to a few days lead-time), and in some instances, can also improve longer-term forecasts.

We will first start by importing important packages, gathering important datasets and configuration settings as we have seen previously.

```
import warnings

from numba.core.errors import NumbaDeprecationWarning

warnings.simplefilter("ignore", category=NumbaDeprecationWarning)
```

```
# Import packages
import datetime as dt
import tempfile
from pathlib import Path

import matplotlib.pyplot as plt
import xarray as xr

from ravenpy import Emulator, EnsembleReader
from ravenpy.config import commands as rc
from ravenpy.config import options as o
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities.testdata import get_file

# Import hydrometeorological data
salmon_meteo = get_file(
    "raven-gr4j-cemaneige/Salmon-River-Near-Prince-George_meteo_daily.nc"
)

# Define HRU
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Alternative names for variables in meteo forcing file
alt_names = {
    "RAINFALL": "rain",
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "SNOWFALL": "snow",
}
```

(continues on next page)

(continued from previous page)

```
# The types of meteorological data available in the file
data_type = ["RAINFALL", "TEMP_MIN", "TEMP_MAX", "SNOWFALL"]

# Additional information about the weather station gauge required by Raven
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
}

# Force a test path.
tmp_path = Path(tempfile.mkdtemp())

# Generate the meteorological gauge data required by raven
gauge = [
    rc.Gauge.from_nc(
        salmon_meteo,
        data_type=data_type,
        alt_names=alt_names,
        data_kwds=data_kwds,
    ),
]
```

We will now start the assimilation with a spinup period

Data assimilation is best performed on a series of initial states that are already somewhat reasonable. Starting a model from empty states and applying assimilation will work but will take more time to converge, and might in some instances create numerical instability. In this example, we perform a 1-year simulation to generate reasonable model states, and at the last time step, Raven will apply the Ensemble Kalman Filter (EnKF) to assimilate the states for the next step (forecasting or closed-loop assimilation).

```
# Spin up the model. This period will be used to do an initial spinup, at the end of␣
↪which the model states
# will be assimilated to better represent the observed streamflow and thus setting up␣
↪parameters for the next
# steps. We first need to specify the spinup dates:
start_date = dt.datetime(1996, 9, 1)
end_date = dt.datetime(1997, 8, 31)

# Prepare the configuration for the spinup. Since we have added information about␣
↪Ensemble Kalman Filter data
# assimilation, a ".rve" file will also be written to disk and Raven will use this to␣
↪perform the assimilation.
conf_spinup = GR4JCN(
    # Model parameters
    params=[0.14, -0.005, 576, 7.0, 1.1, 0.92],
    # Meteorological gauge data from the Salmon river
    Gauge=gauge,
```

(continues on next page)

(continued from previous page)

```
    # Streamflow observations. Very important for data assimilation, or else there is no␣
↪target to attain.
    ObservationData=[rc.ObservationData.from_nc(salmon_meteo, alt_names="qobs")],
    # Sepcify the HRUs composing the watershed. Here we are using a lumped model, so␣
↪there is a single HRU.
    HRUs=[hru],
    # Start and end dates of the simulation. EnKF will be applied at the last date␣
↪(EndDate)
    StartDate=start_date,
    EndDate=end_date,
    # Specify which mode of EnKF we want to use. We want the spinup for now, but later␣
↪we will use other
    # options. We are also using 25 members in the ensemble, but this can be changed␣
↪according to your needs.
    EnsembleMode=rc.EnsembleMode(n=25),
    EnKFMode=o.EnKFMode.SPINUP,
    # Run name of the spinup period. This is important because it will be required in␣
↪the next step.
    RunName="spinup",
    # Let's specify some metrics to assess the model performance.
    EvaluationMetrics=("NASH_SUTCLIFFE",),
    # The folder where the ensemble runs will be generated. By default, the runs are␣
↪called ens_1... ens_N.
    OutputDirectoryFormat="./ens_*",
    # We need to tell Raven which inputs to perturb. the perturbation is applied␣
↪following a distribution
    # that should realistically represent the uncertainty of the observations of these␣
↪variables. Here we
    # use precipitation, but we could also add temperature for example.
    ForcingPerturbation=[
        rc.ForcingPerturbation(
            forcing="PRECIP",
            dist="DIST_NORMAL",
            p1=1.0,
            p2=0.5,
            adj="MULTIPLICATIVE",
        ),
        rc.ForcingPerturbation(
            forcing="TEMP_MAX",
            dist="DIST_NORMAL",
            p1=0.0,
            p2=2.0,
            adj="ADDITIVE",
        ),
        rc.ForcingPerturbation(
            forcing="TEMP_MIN",
            dist="DIST_NORMAL",
            p1=0.0,
            p2=2.0,
            adj="ADDITIVE",
        ),
    ],
```

(continues on next page)

(continued from previous page)

```
    # Define the HRU Groups the assimilation will be applied on. Here we apply to all␣
↪HRUs (single HRU)
    DefineHRUGroups=["All"],
    HRUGroup=[{"name": "All", "groups": ["1"]}],
    # Define which variables we want to assimilate.
    # Here we only adjust the water content of the 2 first layers of soil (SOIL[0] and␣
↪SOIL[1])
    AssimilatedState=[
        rc.AssimilatedState(state="SOIL[0]", group="All"),
        rc.AssimilatedState(state="SOIL[1]", group="All"),
    ],
    # Define which subbasin id the streamflow is associated with
    AssimilateStreamflow=[rc.AssimilateStreamflow(sb_id=1)],
    # Define the error model for the observed streamflow. We will have a STD equal to 7%␣
↪of the streamflow
    # value for each day, following a normal distribution.
    ObservationErrorModel=[
        rc.ObservationErrorModel(
            state="STREAMFLOW",
            dist="DIST_NORMAL",
            p1=1,
            p2=0.07,
            adj="MULTIPLICATIVE",
        )
    ],
    # Set to true for more details (verbosity)
    DebugMode=False,
    NoisyMode=False,
)

# Now that the configuration is completed, we can actually launch Raven to do the␣
↪assimilation
spinup = Emulator(config=conf_spinup, workdir=tmp_path, overwrite=True).run(
    overwrite=True
)
```

We have now run the model and obtained an ensemble of simulations that each have perturbed meteorological data and new initial states. We can read-in the generated hydrographs and see what our spinup period flows look like.

```
# Get the paths to all the ens_1...ens_N folders, one per member
paths_spinup = list(tmp_path.glob("ens_*"))

# Read those into memory in an EnsembleReader object
ens_spinup = EnsembleReader(run_name=conf_spinup.run_name, paths=paths_spinup)

# We can now plot the results
ens_spinup.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False, lw=0.5)
ens_spinup.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="Forecasts", lw=0.5)
ens_spinup.hydrograph.q_obs[1, :, 0].plot.line(
    x="time", color="black", label="Observation"
)
plt.legend(loc="upper left")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Streamlfow (m$^3$/s)")
plt.title("Spinup period")
plt.show()
```

Start converging model states by closed-loop assimilation

We have completed the spinup period, which has left us with a set of 25 initial states that can be used to sample initial state uncertainty. However, we need to do a few more assimilation passes before the model starts to converge to appropriate values. From the assimilated states of the spinup period, let's now do a single 3-day simulation and see what happens:

```
# Set the start date equal to the assimilated date of the prior run, as we want to start␣
↪from the assimilated
# states. The end date is set 3 days later, after which assimilation will be␣
↪automatically performed.
start_date = end_date
end_date = end_date + dt.timedelta(days=3)

# Closed-Loop assimilation. From the previous configuration, we can make a copy and only␣
↪change the required
# parameters, such as the run name, start and end dates, and the type of EnKF (switch␣
↪from spinup to closed-loop).
conf_loop = conf_spinup.duplicate(
    EnKFMode=o.EnKFMode.CLOSED_LOOP,
    # This will be the name of the output files in the closed-loop run.
    RunName="loop",
    # This is the name of the run we will start from, i.e. the assimilated spinup states␣
↪from earlier!
    SolutionRunName="spinup",
    # We need to tell the model not to set the default initial conditions (it will use␣
↪the assimilated states)
    UniformInitialConditions=None,
    # Set the new dates
    StartDate=start_date,
    EndDate=end_date,
)

# Now that the configuration is ready, launch the assimilation run. Raven will run 25␣
↪times: Once for each member
# With the same perturbed meteorological and hydrometric data and parameters as defined␣
↪previously, but for this
# new 3-day period.
loop = Emulator(config=conf_loop, workdir=tmp_path, overwrite=True).run(overwrite=True)

# Get the paths to all the ens_1...ens_N folders, one per member
paths_loop = list(tmp_path.glob("ens_*"))

# Repeat the same process as the spinup to look at model results:
ens_loop = EnsembleReader(run_name=conf_loop.run_name, paths=paths_loop)

# We can now plot the results
```

(continues on next page)

(continued from previous page)

```
ens_loop.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False, lw=0.5)
ens_loop.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="Forecasts", lw=0.5)
ens_loop.hydrograph.q_obs[1, :, 0].plot.line(
    x="time", color="black", label="Observation"
)
plt.legend(loc="lower left")
plt.ylabel("Streamlfow (m$^3$/s)")
plt.title("First closed-loop period")
plt.show()
```

We can see that the assimilation has not converged very well. This is expected, as there have only been 2 assimilation steps performed as of yet: One after the spinup period, and this one that happens 3 days later. We will iterate the assimilation loop to help the model converge after multiple assimilation steps. Here we will loop over 30 steps of 3 days each.

```
# Let's store the hydrograph from the previous 3-day run in a variable that we will␣
↪append to at each time step.
total_hydrograph = ens_loop.hydrograph

# Here is where the assimilation loop is performed. We will apply the assimilation 30␣
↪successive times, advancing
# in time by 3 days each iteration.
for i in range(0, 30):
    # Set the new start_date and end_dates
    start_date = end_date
    end_date = end_date + dt.timedelta(days=3)

    # Again, copy the configuration object and change some elements
    conf_loop = conf_loop.duplicate(
        # Here we will set RunName and SolutionRunName to the same values such that the␣
↪model will read the "loop"
        # run, perform the assimilation, and save the results to "loop" again, making␣
↪them available for the
        # next run, effectively overwriting the results at each step. We could preserve␣
↪each run's result by changing
        # these run names dynamically, but in our case it is not important nor required␣
↪to do so.
        RunName="loop",
        SolutionRunName="loop",
        # Again, set the initial conditions to None to preserve the assimilated ones.
        UniformInitialConditions=None,
        # Set the start and end date of the simulation period, with the assimilation␣
↪being performed on the final date.
        StartDate=start_date,
        EndDate=end_date,
    )

    # Perform the actual simulation and assimilation for this 3-day step.
    new_loop = Emulator(config=conf_loop, workdir=tmp_path, overwrite=True).run(
        overwrite=True
    )
```

(continues on next page)

(continued from previous page)

```
    # Extract the results for this 3-day hydrograph and store it into our "total_
↪hydrograph" which keeps track
    # of the flows for each of the 3-day periods.
    ens_loop = EnsembleReader(run_name=conf_loop.run_name, paths=paths_loop)
    total_hydrograph = xr.concat([total_hydrograph, ens_loop.hydrograph], dim="time")


# Once the loop is complete, plot the results:
total_hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False, lw=0.5)
total_hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="Forecasts", lw=0.5)
total_hydrograph.q_obs[1, :, 0].plot.line(x="time", color="black", label="Observation")
plt.legend(loc="upper left")
plt.ylabel("Streamlfow (m$^3$/s)")
plt.title("All closed-loop periods")
plt.show()
```

Before going any further, let's compare the assimilated results to those obtained using a simple non-assimilated run (open-loop):

```
# Reset the start and end-dates to cover the entire period (spinup + 30 3-day steps)
start_date = dt.datetime(1996, 9, 1)
end_date = dt.datetime(1997, 8, 31) + dt.timedelta(days=30 * 3)

# Setup a standard GR4JCN model
conf_openloop = GR4JCN(
    params=[0.14, -0.005, 576, 7.0, 1.1, 0.92],
    Gauge=gauge,
    ObservationData=[rc.ObservationData.from_nc(salmon_meteo, alt_names="qobs")],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="OPEN_LOOP",
    EvaluationMetrics=("NASH_SUTCLIFFE",),
)

openloop = Emulator(config=conf_openloop, workdir=tmp_path, overwrite=True).run(
    overwrite=True
)

openloop.hydrograph.q_sim.plot.line("r", x="time", label="Open-loop simulation")
total_hydrograph.q_sim[:, :, 0].mean(dim="member").plot.line(
    "b", x="time", label="Closed-loop assimilation"
)
openloop.hydrograph.q_obs.plot.line(x="time", color="black", label="Observations")

plt.xlim([dt.date(1997, 9, 1), dt.date(1997, 12, 1)])
plt.ylim([0, 50])
plt.legend(loc="upper left")
plt.ylabel("Streamlfow (m$^3$/s)")
plt.title("Closed-loop vs. Open-loop simulations")
plt.show()
```

We can see that the data assimilation as vastly improved most of the hydrograph. Making the assimilation more frequent,

changing other state variables, or adjusting the error model hyperparameters could also lead to better simulations.

Once we are satisfied with the initial states, our model would now be ready for forecasting, using the ensemble initial states as initial conditions for generating the forecasts. This can be done using the EnKF forecating method:

```
# Set up the forecast configuration, basing it on the previous (final) assimilation step.
conf_forecast = conf_loop.duplicate(
    EnKFMode=o.EnKFMode.FORECAST,
    RunName="forecast",
    SolutionRunName="loop",
    UniformInitialConditions=None,
    # Set the start date equal to the end date of the last assimilation run.
    StartDate=end_date,
    # Here we will do a 30-day forecast using the observed meteorological data as␣
↪forecast data. However it is
    # possible to replace the Gauge forcing data with that of a forecast, as we have␣
↪done before.
    EndDate=end_date + dt.timedelta(days=30),
)

forecast = Emulator(config=conf_forecast, workdir=tmp_path, overwrite=True).run(
    overwrite=True
)

# We will plot the resulting forecast. Note that since we have 25 members, we also have␣
↪25 forecasts, i.e. one
# per possible initial state. We could take the mean hydrograph to get the best␣
↪estimator of the forecasted flow.
ens = EnsembleReader(run_name=conf_forecast.run_name, paths=paths_loop)
ens.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", label=None, lw=0.5)
ens.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="Forecast", lw=0.5)
ens.hydrograph.q_obs[1, :, 0].plot.line("black", x="time", label="Observation")
plt.legend(loc="upper left")
plt.ylabel("Streamlfow (m$^3$/s)")
plt.title("Forecast after assimilation")
plt.xlim([dt.date(1997, 11, 29), dt.date(1997, 12, 29)])
plt.show()
```

7.1.12 11 - Climatological ESP forecasting

Extended Streamflow Prediction (ESP) forecasts from climatological time series

This notebook shows how to perform a climatological Extended Streamflow Prediction (ESP) forecast, using historical weather as a proxy for future weather.

The general idea is to initialize the state of the hydrological model to represent current conditions, but instead of using weather forecasts to predict future flows, we run the model with observed, historical weather series from past years. So for example if we have 30 years of weather observations, we get 30 different forecasts. The accuracy of this forecast ensemble can then be evaluated by different probabilistic metrics.

```
import datetime as dt

from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities import forecasting
from ravenpy.utilities.testdata import get_file
```

Run the model simulations

Here we set model parameters somewhat arbitrarily, but you can set the parameters to the calibrated parameters as seen in the "06_Raven_calibration" notebook we previously encountered.

We also need to choose the forecast issue date. Each forecast will start with the same day and month. For example, jun-06-1980 will compare the climatology using all jun-06's from the dataset. Finally, we can provide the forecast duration (in number of days) as well as the historical meteorological years we want to use to generate the ESP forecast. This allows selecting years that we want to include in the forecast. For example, perhaps we only want to generate a forecast using wet or dry years.

```
# Get the selected watershed's time series. You can use your own time-series for your␣
↪catchment by replacing
# this line with the name / path of your input file.
ts = get_file("raven-gr4j-cemaneige/Salmon-River-Near-Prince-George_meteo_daily.nc")

# This is the forecast start date, on which the forecasts will be launched.
start_date = dt.datetime(1980, 6, 1)

# Provide the length of the forecast, in days:
forecast_duration = 100

# Define HRU to build the hydrological model
hru = {}
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "RAINFALL": "rain",
    "SNOWFALL": "snow",
}

# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "RAINFALL", "SNOWFALL"]
data_kwds = {
    "ALL": {
        "elevation": hru[
```

(continues on next page)

(continued from previous page)

```
            "elevation"
        ],  # No need for lat/lon as they are included in the netcdf file already
    }
}
# Model configuration
model_config = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            ts, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    Duration=forecast_duration,
    RunName="full",
)
```

Issuing the ESP forecast

Here we launch the code that will perform the ESP forecast. Depending on the number of years in the historical dataset and the forecast duration, it might take a while to return a forecast result.

```
# Simulate the climatological ESP:
ESP_sims = forecasting.climatology_esp(
    config=model_config,
    years=[
        1982,
        1998,
        2003,
        2004,
    ],  # List of years to use in the forecast. Optional. Will use all years by default.
)

# Show the results in an xarray dataset, ready to use:
ESP_sims.hydrograph
```

We can now inspect and graph the resulting climatological ESP:

```
ESP_sims.hydrograph.q_sim[:, :, 0].plot.line(x="time")
plt.title("GR4JCN climatological ESP for 1980-06-01")
plt.xticks(rotation=90)
plt.grid("on")
plt.xlabel("Time [days]")
plt.ylabel("Streamflow $[m^3s^{-1}]$")
plt.show()
```

Compute the forecast scores

There are different metric to evaluate the performance of forecasts. As an example, here we are computing the CRPS metric, using the xskillscore library included in PAVICS-Hydro.

```
import numpy as np
import xarray as xr
import xskillscore as xs

# Align time axes to get the observed streamflow time series for the same time frame as␣
↪the ESP forecast ensemble
q_obs, q_sims = xr.align(xr.open_dataset(ts).qobs, ESP_sims.hydrograph, join="inner")

# Adjust the streamflow to convert missing data from -1.2345 format to NaN. Set all␣
↪negative values to NaN.
q_obs = q_obs.where(q_obs > 0, np.nan)

# Compute the Continuous Ranked Probability Score using xskillscore
xs.crps_ensemble(q_obs, q_sims, dim="time").q_sim.values[0]
```

Performing a climatology ESP hindcast

In this section, we make the hindcasts for each initialization date that we desire. Here we will extract ESP forecasts for a given calendar date for the years in "hindcast_years" as hindcast dates. Each ESP hindcast uses all available data in the `ts` dataset, so in this case we will have 56/57 members for each hindcast initialization depending on the date that we start on, UNLESS we specify a list of years manually. The "hindcasts" dataset generated contains all of the flow data from the ESP hindcasts for the initialization dates. The `q_obs` dataset contains all q_obs in the timeseries: Climpred will sort it all out during its processing. Note that the format of these datasets is tailor-made to be used in climpred, and thus has specific dimension names.

This is a slimmed down example of how we would run an ESP forecast over multiple years to assess the skill of such a forecast.

```
hindcasts = forecasting.hindcast_climatology_esp(
    config=model_config,  # Note that the forecast duration is already set-up in the␣
↪model_config above.
    warm_up_duration=365,  # number of days for the warm-up
    years=[1985, 1986, 1987, 1988, 1989, 1990],
    hindcast_years=[2001, 2002, 2003, 2004, 2005, 2006, 2007],
)
```

Evaluate the forecast using different metrics

Once we have the correctly formatted datasets, Make the hindcast object for climpred

These three functions respectively compute the rank histogram, the CRPS and the reliability for the set of initialized dates (i.e. forecast issue dates, here 1 day per year at the same calendar day).

```
# Once we have the correctly formatted datasets, Make the hindcast object for climpred

# We first need to get the observed streamflow:
q_obs = xr.open_dataset(ts)
```

(continues on next page)

(continued from previous page)

```
# However, our simulated streamflow is named "q_sim" and climpred requires the␣
↪observation to be named the same thing
# so let's rename it. While we're at it, we need to make sure that the identifier is the␣
↪same. In our observation
# dataset, it is called "nstations" but in our simulated streamflow it's called "nbasins".
↪ Here we standardize.
q_obs = q_obs.rename({"qobs": "q_sim", "nstations": "nbasins"})

# Make the hindcasting object we can use to compute statistics and metrics
hindcast_object = forecasting.to_climpred_hindcast_ensemble(hindcasts, q_obs)


# This function is used to convert to binary to see if yes/no forecast is larger than␣
↪observations
def pos(x):
    return x > 0  # Check for binary outcome


# Rank histogram verification metric
rank_histo_verif = hindcast_object.verify(
    metric="rank_histogram",
    comparison="m2o",
    dim=["member", "init"],
    alignment="same_inits",
)
# CRPS verification metric
crps_verif = hindcast_object.verify(
    metric="crps",
    comparison="m2o",
    dim=["member", "init"],
    alignment="same_inits",
)
```

```
# We can explore and plot the CRPS as a function of lead-time, for example. Results are␣
↪stored as a dataset and
# can thus be integrated into any simulation or processes.
plt.plot(crps_verif.q_sim)
plt.xlabel("Lead time [days]")
plt.ylabel("CRPS $[m^3s^{-1}]$")
plt.grid("on")
plt.show()
```

7.1.13 Hindcasting with CaSPAr-Archived ECCC forecasts

This notebook shows how to perform a streamflow hindcast, using CaSPar archived weather forecasts. It generates the hindcasts and plots them.

CaSPAr (Canadian Surface Prediction Archive) is an archive of historical ECCC forecasts developed by Juliane Mai at the University of Waterloo, Canada. More details on CaSPAr can be found here https://caspar-data.ca/.

Mai, J., Kornelsen, K.C., Tolson, B.A., Fortin, V., Gasset, N., Bouhemhem, D., Schäfer, D., Leahy, M., Anctil, F. and Coulibaly, P., 2020. The Canadian Surface Prediction Archive (CaSPAr): A Platform to Enhance Environmental Modeling in Canada and Globally. Bulletin of the American Meteorological Society, 101(3), pp.E341-E356.

```
# This entire section is cookie-cutter template to import required packages and prepare␣
↪the temporary writing space.
import datetime as dt
import tempfile
from pathlib import Path

import xarray as xr
from clisops.core import average, subset

from ravenpy import Emulator, RavenWarning
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.extractors.forecasts import get_CASPAR_dataset
from ravenpy.utilities import forecasting
from ravenpy.utilities.testdata import get_file

tmp = Path(tempfile.mkdtemp())
```

Run the model simulations

Here we set model parameters somewhat arbitrarily, but you can set the parameters to the calibrated parameters as seen in the "06_Raven_calibration" notebook we previously encountered. We can then specify the start date for the hindcast ESP simulations and run the simulations.This means we need to choose the forecast (hindcast) date. Available data include May 2017 onwards.

```
# Date of the hindcast
hdate = dt.datetime(2018, 6, 1)

# Get the Forecast data from GEPS via CASPAR
ts_hindcast, _ = get_CASPAR_dataset("GEPS", hdate)

# Get basin contour
basin_contour = get_file("notebook_inputs/salmon_river.geojson")

# Subset the data for the region of interest and take the mean to get a single vector
with xr.set_options(keep_attrs=True):
    ts_subset = subset.subset_shape(ts_hindcast, basin_contour).mean(
        dim=("rlat", "rlon")
    )
ts_subset = ts_subset.resample(time="6H").nearest(
    tolerance="1H"
)  # To make the timesteps identical accross the entire duration
```

```
# See how many members we have available
len(ts_subset.members)
```

Now that we have the correct weather forecasts, we can setup the hydrological model for a warm-up run:

```
# Prepare a RAVEN model run using historical data, GR4JCN in this case.
# This is a dummy run to get initial states. In a real forecast situation,
# this run would end on the day before the forecast, but process is the same.

# Here we need a file of observation data to run a simulation to generate initial␣
↪conditions for our forecast.
# ts = str(
#     get_file("raven-gr4j-cemaneige/Salmon-River-Near-Prince-George_meteo_daily.nc")
# )

# TODO: We will use ERA5 data for Salmon River because it covers the correct period.
ts = get_file("notebook_inputs/ERA5_weather_data_Salmon.nc")

# This is the model start date, on which the simulation will be launched for a certain␣
↪duration
# to set up the initial states. We will then save the final states as a launching point␣
↪for the
# forecasts.

start_date = dt.datetime(2000, 1, 1)
end_date = dt.datetime(2018, 6, 2)

# Define HRU to build the hydrological model
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
}
# Model configuration
```

(continues on next page)

(continued from previous page)

```
model_config_warmup = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            ts, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    EndDate=end_date,
    RunName="NB12_warmup_run",
)

# Run the model and get the outputs.
out1 = Emulator(config=model_config_warmup).run()


# Extract the path to the final states file that will be used as the next initial states
hotstart = out1.files["solution"]
```

We now have the initial states ready for the next step, which is to launch the forecasts in hindcasting mode:

```
# Explore the forecast data to see which variables we have:
display(ts_subset)
```

```
# Configure and run a new model by setting the initial states (equal to the previous run
↪'s final states) and prepare
# the configuration for the forecasts (including forecast start date, which should be␣
↪equal to the final simulation
# date + 1, as well as the forecast duration.)

# We need to write the hindcast data as a file for Raven to be able to access it.
fname = tmp / "hindcast.nc"
ts_subset.to_netcdf(fname)

# We need to adjust the data_type and alt_names according to the data in the forecast:
# Set alternative names for netCDF variables
alt_names = {
    "TEMP_AVE": "tas",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_AVE", "PRECIP"]


# We will need to reuse this for GR4J. Update according to your needs. For example, here␣
↪we will also pass
# the catchment latitude and longitude as our CaSPAr data has been averaged at the␣
↪catchment scale.
# We also need to tell the model to deaccumulate the precipitation and shift it in time␣
↪by 6 hours for our
```

(continues on next page)

(continued from previous page)

```
# catchment (UTC timezones):
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
    "PRECIP": {
        "Deaccumulate": True,
        "TimeShift": -0.25,
        "LinearTransform": {
            "scale": 1000.0
        },  # Since we are deaccumulating, we need to manually specify scale.
    },  # Converting meters to mm (multiply by 1000).
    "TEMP_AVE": {
        "TimeShift": -0.25,
    },
}


# Model configuration for forecasting, including correct start date and forecast duration
model_config_fcst = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            fname, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=end_date + dt.timedelta(days=1),
    Duration=7,
    RunName="NB12_forecast_run",
)

# Update the initial states
model_config_fcst = model_config_fcst.set_solution(hotstart)

# Generate the hindcast by providing all necessary information to generate virtual␣
↪stations representing
# the forecast members
hindcast = forecasting.hindcast_from_meteo_forecast(
    model_config_fcst,
    forecast=fname,
    # We also need to provide the necessary information to create gauges inside the␣
↪forecasting model:
    data_kwds=data_kwds,
    data_type=data_type,
    alt_names=alt_names,
)
```

Explore the hindcast data:

```
hindcast.hydrograph
```

And, for visual representation of the forecasts:

```
import matplotlib.pyplot as plt

# Simulate an observed streamflow timeseries: Here we take a member from the ensemble,␣
↪but you should use your own
# observed timeseries:
qq = hindcast.hydrograph.q_sim[0, :, 0]

hindcast.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False)
hindcast.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="forecasts")
qq.plot.line("r", x="time", label="observations")
plt.legend(loc="upper left")
plt.show()
```

7.2 Advanced workflows

7.2.1 Comparing hindcasts to a climatological ensemble streamflow prediction (ESP)

This notebook shows how to use climatological weather to perform a Climatology-based Extended Streamflow Prediction (ESP) forecast. Then using the same initial states, uses the CaSPar archived weather forecasts to generate streamflow hindcasts over the same period. It is thus possible to compare both approaches.

CaSPAr (Canadian Surface Prediction Archive) is an archive of historical ECCC forecasts developed by Juliane Mai at the University of Waterloo, Canada. More details on CaSPAr can be found here https://caspar-data.ca/.

Mai, J., Kornelsen, K.C., Tolson, B.A., Fortin, V., Gasset, N., Bouhemhem, D., Schäfer, D., Leahy, M., Anctil, F. and Coulibaly, P., 2020. The Canadian Surface Prediction Archive (CaSPAr): A Platform to Enhance Environmental Modeling in Canada and Globally. Bulletin of the American Meteorological Society, 101(3), pp.E341-E356.

```
%matplotlib inline
# This entire section is cookie-cutter template to allow calling the servers and␣
↪instantiating the connection
# to the WPS server. Do not modify this block.
import datetime as dt

import matplotlib.pyplot as plt
import xarray as xr
from clisops.core import average, subset

from ravenpy import Emulator
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.extractors.forecasts import get_CASPAR_dataset
from ravenpy.utilities import forecasting
from ravenpy.utilities.testdata import get_file, open_dataset
```

Setting up the warm-up file

Here we tell the model that we want to forecast over the Salmon River catchment and provide its properties (area, lat/long, elevation). We will run it using the GR4JCN hydrological model and have provided some parameters. Other information on the forecast conditions is provided. Thr first step is to generate a hotstart file to prepare the model to generate forecasts.

```
# Define the warmup period dates
start_date_wu = dt.datetime(2010, 1, 1)
end_date_wu = dt.datetime(2018, 6, 30)

# Define the catchment contour. Here we use the Salmon River file we previously␣
↪generated using the Delineator
# in Tutorial Notebook 01.
basin_contour = get_file("notebook_inputs/salmon_river.geojson")

# Define some of the catchment properties. Could also be replaced by a call to the␣
↪properties WPS as in
# the Tutorial Notebook 02.
hru = {}
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Observed weather data for the Salmon river. We extracted this using Tutorial Notebook␣
↪03 and the
# salmon_river.geojson file as the contour.
ts = get_file("notebook_inputs/ERA5_weather_data_Salmon.nc")

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
}

# Model configuration
model_config_warmup = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
```

(continues on next page)

(continued from previous page)

```
    Gauge=[
        rc.Gauge.from_nc(
            ts, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=start_date_wu,
    EndDate=end_date_wu,
    RunName="ESP_vs_NWP_warmup",
)

# Run the model and get the outputs.
out1 = Emulator(config=model_config_warmup).run()

# Extract the path to the final states file that will be used as the next initial states
hotstart = out1.files["solution"]
```

```
dss = open_dataset(ts)
dss
```

Hindcasting using Climatological Ensemble Streamflow Prediction (ESP)

Now that we have the hotstart file ready to go, we can configure our model for forecasting in climatology ESP mode:

```
# Date of the hindcast
hdate = dt.datetime(2018, 7, 1)

# Duration of the hindcast, in days
duration = 7

# Build a new model config:
# Model configuration
model_config_ESP = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            ts, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=hdate,
    Duration=duration,
    RunName="ESP_vs_NWP_ESPfcst",
)

# Set the initial states of this new config to the correct values, i.e. the end of the␣
↪previous forecast.
model_config_ESP = model_config_ESP.set_solution(hotstart)

# Simulate the climatological ESP:
ESP_sims = forecasting.climatology_esp(config=model_config_ESP)
```

(continues on next page)

(continued from previous page)

```
# Show the results in an xarray dataset, ready to use:
ESP_sims.hydrograph
```

We have now run the hindcast using Climatological ESP and retrieved the results. Let's take a look at the resulting forecast.

```
# Invent an observation so we can compute metrics later, and display as Qobs here. TODO:␣
↪Add real streamflow data.
qq = ESP_sims.hydrograph.q_sim[0, :, 0]

# This is to be replaced with a call to the forecast graphing WPS as soon as merged.
# model.q_sim.plot.line("b", x="time")
ESP_sims.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False)
ESP_sims.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="ESP forecasts")
qq.plot.line("r", x="time", label="observations")
plt.legend(loc="upper left")
plt.show()
```

Hindcasting using archived weather forecasts from a weather forecast model

In this next part, we will use the CaSPAr dataset (archived weather forecasts from Environment and Climate Change Canada) to forecast flows on the same period using the same hotstart file.

```
# Get the Forecast data from GEPS via CASPAR.
# Take an extra day to ensure time-shift doesn't remove a part of our day
ts_hindcast, _ = get_CASPAR_dataset("GEPS", hdate - dt.timedelta(days=1))

# Subset the data for the region of interest and take the mean to get a single vector
with xr.set_options(keep_attrs=True):
    ts_subset = subset.subset_shape(ts_hindcast, basin_contour).mean(
        dim=("rlat", "rlon")
    )

ts_subset = ts_subset.resample(time="6H").nearest(
    tolerance="1H"
)  # To make the timesteps identical accross the entire duration

# We need to write the hindcast data as a file for Raven to be able to access it.
fname = "/tmp/hindcast.nc"
ts_subset.to_netcdf(fname)

# We need to adjust the data_type and alt_names according to the data in the forecast:
# Set alternative names for netCDF variables
alt_names = {
    "TEMP_AVE": "tas",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_AVE", "PRECIP"]
```

(continues on next page)

(continued from previous page)

```
# We will need to reuse this for GR4J. Update according to your needs. For example, here␣
↪we will also pass
# the catchment latitude and longitude as our CaSPAr data has been averaged at the␣
↪catchment scale.
# We also need to tell the model to deaccumulate the precipitation and shift it in time␣
↪by 6 hours for our
# catchment (UTC timezones):
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
    "PRECIP": {
        "Deaccumulate": True,
        "TimeShift": -0.25,
        "LinearTransform": {
            "scale": 1000.0
        },  # Since we are deaccumulating, we need to manually specify scale.
    },  # Converting meters to mm (multiply by 1000).
    "TEMP_AVE": {
        "TimeShift": -0.25,
    },
}

# Model configuration for forecasting, including correct start date and forecast duration
model_config_fcst = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            fname, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=hdate,
    Duration=duration,
    RunName="NB12_forecast_run",
)

# Update the initial states
model_config_fcst = model_config_fcst.set_solution(hotstart)

# Generate the hindcast by providing all necessary information to generate virtual␣
↪stations representing
# the forecast members
hindcast_sims = forecasting.hindcast_from_meteo_forecast(
    model_config_fcst,
    forecast=fname,
    overwrite=True,
    # We also need to provide the necessary information to create gauges inside the␣
↪forecasting model:
```

(continues on next page)

(continued from previous page)

```
    data_kwds=data_kwds,
    data_type=data_type,
    alt_names=alt_names,
)

# Display the hydrographs
display(hindcast_sims.hydrograph)
```

```
hindcast_sims.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False)
hindcast_sims.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="hindcasts")
qq.plot.line("r", x="time", label="observations")
plt.legend(loc="upper left")
plt.show()
```

The model has run in forecast mode and we can now easily compare results:

```
hindcast_sims.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False)
ESP_sims.hydrograph.q_sim[:, :, 0].plot.line("g", x="time", add_legend=False)
ESP_sims.hydrograph.q_sim[1, :, 0].plot.line("g", x="time", label="ESP forecasts")
hindcast_sims.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="hindcasts")
qq.plot.line("r", x="time", label="observations")
plt.legend(loc="upper left")
plt.show()
```

7.2.2 Real-time flow forecasts with ECCC weather forecasts

This notebook shows how to perform a streamflow forecast, using ECCC weather forecasts. Generates the forecasts and plots them.

```
%matplotlib inline

# Import the required packages

import datetime as dt

import fiona
import matplotlib.pyplot as plt
import xarray as xr
from clisops.core import average, subset

from ravenpy import Emulator
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.extractors.forecasts import get_recent_ECCC_forecast
from ravenpy.utilities import forecasting
from ravenpy.utilities.testdata import get_file, open_dataset
```

```
# Define the catchment contour. Here we use the Salmon River file we previously␣
↪generated using the Delineator
# in Tutorial Notebook 01.
```

(continues on next page)

(continued from previous page)

```
basin_contour = get_file("notebook_inputs/salmon_river.geojson")

# Get the most recent ECCC forecast data from the Geomet extraction tool:
forecast_data = get_recent_ECCC_forecast(
    fiona.open(basin_contour), climate_model="GEPS"
)
display(forecast_data)

# We need to write the forecast data as a file for Raven to be able to access it.
fname = "/tmp/forecast.nc"
forecast_data.to_netcdf(fname)
```

```
# Define the warmup period dates. Our weather file ends before the forecast date so our␣
↪states will not be as
# good as those of a model run operationally.
start_date_wu = dt.datetime(2010, 1, 1)
end_date_wu = dt.datetime(2020, 3, 30)

# Define some catchment properties. Could also be replaced by a call to the properties␣
↪WPS as in
# the Tutorial Notebook 02.
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Observed weather data for the Salmon river. We extracted this using Tutorial Notebook␣
↪03 and the
# salmon_river.geojson file as the contour. Used for the model warm-up.
ts = get_file("notebook_inputs/ERA5_weather_data_Salmon.nc")

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
}

# Model configuration
```

(continues on next page)

(continued from previous page)

```
model_config_warmup = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            ts, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=start_date_wu,
    EndDate=end_date_wu,
    RunName="ESP_vs_NWP_warmup",
)

# Run the model and get the outputs.
out1 = Emulator(config=model_config_warmup).run()

# Extract the path to the final states file that will be used as the next initial states
hotstart = out1.files["solution"]
```

```
# Length of the desired forecast, in days
duration = 7

# We need to adjust the data_type and alt_names according to the data in the forecast:
# Set alternative names for netCDF variables
alt_names = {
    "TEMP_AVE": "tas",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_AVE", "PRECIP"]

# We will need to reuse this for GR4J. Update according to your needs. For example, here␣
↪we will also pass
# the catchment latitude and longitude as our CaSPAr data has been averaged at the␣
↪catchment scale.
# We also need to tell the model to deaccumulate the precipitation and shift it in time␣
↪by 6 hours for our
# catchment (UTC timezones):
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
    "PRECIP": {
        "Deaccumulate": True,
        "TimeShift": -0.25,
        "LinearTransform": {
            "scale": 1.0
        },  # Since we are deaccumulating, we need to manually specify scale.
    },  # We are already in mm, so leave it like so (scale = 1.0).
```

(continues on next page)

(continued from previous page)

```
    "TEMP_AVE": {
        "TimeShift": -0.25,
    },
}

# ECCC forecast time format is a bit complex to work with, so we will use cftime to make␣
↪it more manageable.
fcst_tmp = open_dataset(fname, use_cftime=True)

# Get the first timestep that will be used for the model simulation
start_date = fcst_tmp.time.data[0] + dt.timedelta(days=1)

# Model configuration for forecasting, including correct start date and forecast␣
↪duration and initial state
model_config_fcst = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            fname, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=start_date,
    Duration=duration,
    RunName="Realtime_forecast_NB",
).set_solution(hotstart, timestamp=False)
```

```
# Generate the forecast by providing all necessary information to generate virtual␣
↪stations representing
# the forecast members. Note that we are using the hindcasting tools, becasue there is␣
↪effectively no difference
# between operational hindcasting and operational forecasting except for the forecast␣
↪issue time and data
# availability, which we solved by using the most recent ECCC forecasts with a warmed-up␣
↪model and hotstart file.

forecast_sims = forecasting.hindcast_from_meteo_forecast(
    model_config_fcst,
    forecast=fname,
    ens_dim="members",
    # We also need to provide the necessary information to create gauges inside the␣
↪forecasting model:
    data_kwds=data_kwds,
    data_type=data_type,
    alt_names=alt_names,
)

display(forecast_sims.hydrograph)
```

And, for visual representation of the forecasts:

```
# Simulate an observed streamflow timeseries: Here we take a member from the ensemble, but you should use your own
# observed timeseries:
qq = forecast_sims.hydrograph.q_sim[0, :, 0]

# This is to be replaced with a call to the forecast graphing WPS as soon as merged.
# model.q_sim.plot.line("b", x="time")
forecast_sims.hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False)
forecast_sims.hydrograph.q_sim[1, :, 0].plot.line("b", x="time", label="forecasts")
qq.plot.line("r", x="time", label="observations")
plt.legend(loc="upper left")
plt.show()
```

7.2.3 Probabilistic flood risk assessment

In this notebook, we combine the forecasting abilities and the time series analysis capabilities in a single seamless process to estimate the flood risk of a probabilistic forecast. As an example, we first perform a frequency analysis on an observed time series, then estimate the streamflow associated to a 2-year return period. We then perform a climatological ESP forecast (to ensure repeatability, but a realtime forecast would work too!) and estimate the probability of flooding (exceeding the threshold) given the ensemble of members in the probabilistic forecast.

```
import warnings

from numba.core.errors import NumbaDeprecationWarning

warnings.simplefilter("ignore", category=NumbaDeprecationWarning)
```

```
%matplotlib inline

import datetime as dt

import xclim
from matplotlib import pyplot as plt

from ravenpy.utilities.testdata import get_file, open_dataset
```

Perform the time series analysis on observed data for the catchment using the frequency analysis WPS capabilities.

```
# Get the data that we will be using for the demonstration.
file = "raven-gr4j-cemaneige/Salmon-River-Near-Prince-George_meteo_daily.nc"
ts = open_dataset(file).qobs

# Perform the frequency analysis for various return periods. We compute 2, 5, 10, 25, 50 and 100 year return
# periods, but later on we will only compare the forecasts to the 2 year return period.
out = xclim.generic.return_level(
    ts, mode="max", t=(2, 5, 10, 25, 50, 100), dist="gumbel_r"
)
out
```

```
# Plot the results of the flows as a function of return period.
fig, ax = plt.subplots(1)
lines = out.plot(ax=ax)

# Get 2-year return period from the frequency analysis
threshold = out.sel(return_period=2).values
print(f"Threshold: {threshold:.1f}")

pt = ax.plot([2], [threshold], "ro")

ax.annotate(
    "Flow threshold, set at 2-year return period",
    (2, threshold),
    xytext=(25, 10),
    textcoords="offset points",
    arrowprops=dict(arrowstyle="->", connectionstyle="arc3"),
)
```

Probabilistic forecast

In this example, we will perform an ensemble hydrological forecast and will then compute the probability of flooding given a flooding threshold. Start by building the model configuration as in the Tutorial Notebook 11:

```
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities.forecasting import climatology_esp, compute_forecast_flood_risk

# Choose the forecast date. Each forecast will start with the same day and month.
# For example, jan-05-2001 will compare the climatology using all jan-05ths from the␣
↪dataset)
fdate = dt.datetime(2003, 4, 13)

# The dataset to use to get the forecast timeseries:
duration = 30  # Length in days of the climatological ESP forecast

# Define HRU to build the hydrological model
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "RAINFALL": "rain",
    "SNOWFALL": "snow",
}
```

(continues on next page)

(continued from previous page)

```
# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "RAINFALL", "SNOWFALL"]
data_kwds = {
    "ALL": {
        "elevation": hru[
            "elevation"
        ],  # No need for lat/lon as they are included in the netcdf file already
    }
}
# Model configuration
model_config = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    Gauge=[
        rc.Gauge.from_nc(
            get_file(file),
            data_type=data_type,
            alt_names=alt_names,
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=fdate,
    Duration=duration,
    RunName="Probabilistic_flood_risk_NB",
)
```

Now that the configuration is ready, launch the ESP forecasting tool to generate an ensemble hydrological forecast:

```
# Launch the ESP forecasting method
ESP_sims = climatology_esp(
    config=model_config,
)

# Show the results in an xarray dataset, ready to use:
ESP_sims.hydrograph
```

```
# Plot the forecasts and the 2-year threshold previously estimated
fig, ax = plt.subplots(1)
ESP_sims.hydrograph.q_sim[:, :, 0].plot.line(
    ax=ax, hue="member", add_legend=False, color="gray", lw=0.5
)
t = ax.axhline(threshold, color="red")
```

```
# Now compute the flood risk given the probabilistic forecast and the threshold␣
↪associated to the 2-year return
# period.

threshold = out.sel(return_period=2).values

# Run the flood forecast risk tool to extract the probability of exceedance in netcdf␣
↪format and xarray Dataset format
flood_risk_data = compute_forecast_flood_risk(
```

(continues on next page)

(continued from previous page)

```
    forecast=ESP_sims.hydrograph.q_sim,
    flood_level=threshold,
)

# Extract the data and plot
fig, ax = plt.subplots(1)
l = flood_risk_data.exceedance_probability.plot()
ax.set_ylabel("Flood risk")
```

Results analysis

We can see from the above figure that there is no risk of exceeding the 2-year return period for the selected dates of the forecast.

7.2.4 Distributed hydrological modelling

Using Ravenpy to build a distributed hydrological model

In this notebook, we will demonstrate how to build a distributed hydrological model using Raven as well as "Routing product" (Generated by BasinMaker), a database of subbasins and how they link to one another in a river network. Currently, Routing product is only available for North American catchments. However, if in time it becomes available on a larger scale, it would be trivial to change the setup apply it to other supported regions.

```
# Import the list of possible model templates for distributed hydrological modelling
from ravenpy.config.emulators import (
    GR4JCN,
    HBVEC,
    HMETS,
    HYPR,
    SACSMA,
    Blended,
    CanadianShield,
    Mohyse,
)
```

```
import datetime as dt
import tempfile
from pathlib import Path

import matplotlib.pyplot as plt
import xarray as xr

from ravenpy import Emulator
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.extractors.routing_product import (
    BasinMakerExtractor,
    GridWeightExtractor,
    open_shapefile,
```

(continues on next page)

(continued from previous page)

```
    upstream_from_coords,
)
from ravenpy.utilities.testdata import get_file, open_dataset

tmp_path = Path(tempfile.mkdtemp())
```

In the next step, we will get the Routing product file for our catchment. These can be downloaded here: http://hydrology.uwaterloo.ca/basinmaker/download_regional.html

```
# Get path to pre-downloaded BasinMaker Routing product database for our catchment
shp_path = get_file("basinmaker/drainage_region_0175_v2-1/finalcat_info_v2-1.zip")

# Note that for this to work, the coordinates must be in the small
# BasinMaker example (drainage_region_0175)
df = open_shapefile(shp_path)

# Gauge station for observations at Matapedia
# SubId: 175000128
# -67.12542 48.10417
sub = upstream_from_coords(-67.12542, 48.10417, df)

# Extract the subbasins and HRUs (one HRU per sub-basin)
bm = BasinMakerExtractor(
    df=sub,
    hru_aspect_convention="ArcGIS",
)

# Get the .rvh file that we will provide to the config and that links HRUs/subbasins to␣
↪the river network
rvh = bm.extract(hru_from_sb=True)
```

Now that we have the HRUs and river network all setup, let's get the hydrometeorological data. We first get the database of streamflows and then do the same for weather. You can provide your own for your own catchments, here we are using our datasets to keep things tidy.

```
# Streamflow observations file
qobs_fn = get_file("matapedia/Qobs_Matapedia_01BD009.nc")

# Make an obervation gauge from the observed streamflow
qobs = rc.ObservationData.from_nc(qobs_fn, alt_names=("discharge",))
```

Now prepare the meteorological data using the Gauge format. Note that this dataset of stations is a combination of stations that we iterate on, making a Gauge object for each station in our dataset:

```
# Meteo observations file
meteo_grid_fn = get_file("matapedia/Matapedia_meteo_data_stations.nc")

# Alternate names for variables in the files
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}
```

(continues on next page)

(continued from previous page)

```
# Make virtual Gauges
meteo_forcing_stations = [
    rc.Gauge.from_nc(
        meteo_grid_fn,
        data_type=alt_names.keys(),
        station_idx=i + 1,
        alt_names=alt_names,
    )
    for i in range(6)  # Since we have 6 stations
]
```

Now that we have the data, we can run the distributed model as usual. Note that we must provide the AVG_ANNUAL_RUNOFF parameter to initialize the catchment's hydrological states for distributed models:

```
# Prepare the model configuration
model_config = GR4JCN(
    params=[0.529, -3.396, 407.29, 1.072, 16.9, 0.947],
    StartDate=dt.datetime(1998, 1, 1),
    EndDate=dt.datetime(2020, 12, 31),
    ObservationData=[qobs],
    Gauge=meteo_forcing_stations,
    GlobalParameter={"AVG_ANNUAL_RUNOFF": 40.65},
    **rvh,
)

# Run the model with the configuration we just built
distributed_outputs = Emulator(model_config).run(overwrite=True)
```

Explore the results, just like for any other model. However, this time we have a few gauges because the Routing Product integrates some gauges already. We want data for the first gauge:

```
# Show the hydrographs object
display(distributed_outputs.hydrograph)
```

```
# Plot the resulting streamflow
distributed_outputs.hydrograph.q_sim.isel(nbasins=0).plot.line(
    x="time", label="Distributed model", color="blue", lw=1.5
)

# Plot the observed streamflow
qobs_data = open_dataset(qobs_fn)
qobs_data.discharge.plot.line(x="time", label="Observations", color="red", lw=1.5)

plt.legend()
```

7.2.5 Managing Jupyter Environments

This Notebook shows how to customize your Jupyter environment to install packages, reset the environment to defaults, and exporting the environment for reproducibility. We also provide some information on general guidelines on using the PAVICS-Hydro JupyterLab instance.

Installing packages

It is possible to install packages to the environment if they are not currently installed. To do so, we should prioritize "mamba" which can be seen as a faster/more efficient conda, and use pip if mamba fails. We can install packages by issuing the command in a notebook cell. Here we will try importing the "seaborn" package, which is not installed by default on PAVICS.

```
# Attempt to install seaborn. This will fail when run for the first time!

# UNCOMMENT THE FOLLOWING LINE TO TEST THE EXISTENCE OF THE SEABORN PACKAGE. It is␣
↪currently commented to ensure the automatic notebook checks do not fail for an obvious␣
↪reason.
# import seaborn
```

This has failed because the package is not currently installed. Let's install it using mamba. The same command can be used with pip, simply replace "mamba" with "pip".

```
# Install using mamba, and provide the "--yes" option to pre-confirm installation
!mamba install seaborn --yes

# This will take a few seconds to download, install and confirm installation.
```

We can now import the newly installed package:

```
# This will now work.
import seaborn
```

Resetting the environment

If a package is installed that causes conflicts or causes code to break, it is possible to reset the environment by closing the server and respawning a new one, that will have the default packages installed. To do so, simply go to: –> File –> Hub Control Panel –> Stop my server.

Doing so will kill the server, but it will nonetheless keep all of your files. Respawning the server will open a fresh default environment. You can test this now! When you try and re-run the notebook, the first cell will fail again because 'seaborn' will have not been installed yet on this server instance.

Exporting your environment

To export your environment to replicate it elsewhere (such as a local installation, or to make a backup in case of future updates), you need to export two elements:

- The data
- The installed packages

The data can be exported using the explorer on the left. You can select the files you want to download directly, or you can select "Download current folder as an archive". This will allow you to keep a copy of your data on your personal computer. However, note that data stored on this server is not removed or purged. Users are encouraged to use storage on an as-needed basis and to remove data that is not required to free-up resources for other users. PAVICS developers will contact users that use unreasonable amounts of storage space in order to find an alternative solution. The same reasoning also applies to computing power. Users can run multiple kernels/notebooks in parallel, but users are encouraged to use resources on an as-needed basis, with power users potentially being contacted to find alternative solutions.

The environment can be exported using the following commands:

Export it to text in this Notebook:

```
conda env export
```

Other methods to export environments

You can also export the environment to files, using these commands:

Export it to file with explicit packages and channels:

```
conda list --explicit>ENV.txt
```

Export it cross-platform:

```
conda env export --from-history>ENV.yml
```

7.2.6 Regionalization of model parameters

Here we call the Regionalization WPS service to provide estimated streamflow (best estimate and ensemble) at an ungauged site using three pre-calibrated hydrological models and a large hydrometeorological database with catchment attributes (Extended CANOPEX). Multiple regionalization strategies are allowed.

```
import datetime as dt

from matplotlib import pyplot as plt

from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.utilities.regionalization import (
    read_gauged_params,
    read_gauged_properties,
    regionalize,
)
from ravenpy.utilities.testdata import get_file
```

We can first start by setting up our model. This model will be setup on our ungauged basin, for which we want to generate streamflow. We still need to provide meteorological forcings and other descriptors (HRUs), however we do not provide a parameter set. This will be done by regionalization later.

```
# Get the forcing dataset for the ungauged watershed
ts = get_file("notebook_inputs/ERA5_weather_data_Salmon.nc")

# Get HRUs of ungauged watershed
hru = dict(
    area=4250.6,
    elevation=843.0,
    latitude=54.4848,
    longitude=-123.3659,
    hru_type="land",
)

# Set alternative names for netCDF variables
alt_names = {
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PRECIP": "pr",
}

# Data types to extract from netCDF
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]
data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "Latitude": hru["latitude"],
        "Longitude": hru["longitude"],
    },
}

# Model configuration for the ungauged watershed. Notice we are not providing parameters,
↪ because,
# by definition, we do not have the optimal parameters for an ungauged basin.
# Also note that, for now, only the GR4JCN, HMETS and MOHYSE models are supported, as␣
↪they are the only ones
# for which we have a pre-computed database of parameters to use to estimate␣
↪relationships between descriptors
# and model parameters.
model_config = GR4JCN(
    Gauge=[
        rc.Gauge.from_nc(
            ts, data_type=data_type, alt_names=alt_names, data_kwds=data_kwds
        )
    ],
    HRUs=[hru],
    StartDate=dt.datetime(1990, 1, 1),
    EndDate=dt.datetime(2010, 12, 31),
    RunName="regionalization",
)
```

We can now start working on the regionalization method and the required information.

```
# We need to provide the name of the model structure we are using. Can be "GR4JCN",
↪"HMETS" or "MOHYSE"
model_structure = "GR4JCN"

# Read the table of model parameters and calibrated NSE values for all the basins in the␣
↪donors dataset
nash, params = read_gauged_params(model_structure)

# Which variables do we want to use to estimate the parameter relationships?
# Possible values and their description are provided here:
"""
latitude  (catchment centroid latitude, degrees)
longitude (catchment centroid longitude, degrees)
area      (drainage area, km^2)
gravelius (Gravelius index)
perimeter (catchment perimeter, m)
elevation (mean catchment elevation, m)
slope     (mean catchment slope, %)
aspect    (catchment orientation vs. North, degrees)
forest    (Land-use percentage as forest (%))
grass     (Land-use percentage as grass (%))
wetland   (Land-use percentage as wetlands (%))
urban     (Land-use percentage as urban areas (%))
shrubs    (Land-use percentage as shrubs (%))
crops     (Land-use percentage as crops (%))
snowIce   (Land-use percentage as permanent snow/ice (%))
"""
variables = ["latitude", "longitude", "area", "forest"]

# Read the desired properties from the donors table
props = read_gauged_properties(variables)

# Provide the values for the desired variables for the ungauged basin (used to estimate␣
↪relationships)
ungauged_props = {
    "latitude": 40.4848,
    "longitude": -103.3659,
    "area": 4250.6,
    "forest": 0.4,
}

# Choice of the regionalization method. You can choose between the following methods␣
↪(with their description):
"""
SP   (Spatial Proximity: Uses the latitude and longitude only by default, returns the␣
↪nearest donors)
PS   (Physical Similarity: Finds the most similar donor catchments according to your␣
↪desired variables)
MLR  (Multiple Linear Regression: Build a linear regression between the desired␣
↪variables and the model
     parameters from the donor database. Then estimate parameters from the linear␣
↪regression using
     the ungauged basin's properties.)
```

(continues on next page)

(continued from previous page)

```
SP-IDW (Spatial Proximity but average the results of multiple donors using the inverse␣
↪distance weighting
        based on distance)
PS-IDW (Physical Similarity but average the results of multiple donors using the inverse␣
↪distance weighting
        of degree of similarity)
SP-IDW-RA (SP-IDW while adding regression-based parameters to the donor parameter dataset
          [Arsenault and Brissette, 2014])
PS-IDW-RA (PS-IDW while adding regression-based parameters to the donor parameter dataset
          [Arsenault and Brissette, 2014])
---
Arsenault, R., and Brissette, F. P. (2014), Continuous streamflow prediction in ungauged␣
↪basins:
The effects of equifinality and parameter set selection on uncertainty in␣
↪regionalization approaches,
Water Resour. Res., 50, 6135-6153, doi:10.1002/2013WR014898.
"""
regionalization_method = "SP-IDW-RA"

# Here we provide a threshold to exclude donor catchments. Basically, any donors whose␣
↪calibration NSE is lower
# than this threshold is considered unreliable and is removed from the database prior to␣
↪processing. 0.6-0.7 are
# generally well-accepted values in the literature. The higher the threshold, the fewer␣
↪donors remain so an
# equilibrium must be found.
minimum_donor_NSE = 0.7

# Finally, we can choose how many donors we want to use. The value is only used for SP-␣
↪and PS-based methods.
# The hydrographs generated by running the model using the parameters of multiple donors␣
↪are averaged (either
# using a simple mean, or using IDW if we used the IDW tag) which results in generally␣
↪better hydrographs than
# any of the single hydrographs.
number_donors = 5

# Launch the regionalization method and get
#  - hydrograph: the mean hydrograph, and
#  - ensemble_hydrograph: the hydrographs of each of the individual donors before␣
↪averaging
hydrograph, ensemble_hydrograph = regionalize(
    config=model_config,
    method=regionalization_method,
    nash=nash,
    params=params,
    props=props,
    target_props=ungauged_props,
    min_NSE=minimum_donor_NSE,
    size=number_donors,
)
```

The `hydrograph` and `ensemble` outputs are netCDF files storing the time series. These files are opened by default

using `xarray`, which provides convenient and powerful time series analysis and plotting tools.

```
display(hydrograph)
```

```
display(ensemble_hydrograph)
```

```
qq = ensemble_hydrograph.q_sim[0, :, 0]

ensemble_hydrograph.q_sim[:, :, 0].plot.line("b", x="time", add_legend=False)
ensemble_hydrograph.q_sim[1, :, 0].plot.line(
    "b", x="time", label="Regionalized hydrographs"
)
qq.plot.line("r", x="time", label="observations")
plt.legend(loc="upper right")
plt.show()
```

```
print("Max: ", hydrograph.max())
print("Mean: ", hydrograph.mean())
print("Monthly means: ", hydrograph.groupby("time.month").mean(dim="time"))
```

7.2.7 Forcing HMETS with the extended CANOPEX dataset

Here we use ravenpy to launch the HMETS hydrological model and analyze the output. We also prepare and gather data directly from the CANOPEX dataset made available freely for all users.

```
import warnings

from numba.core.errors import NumbaDeprecationWarning

warnings.simplefilter("ignore", category=NumbaDeprecationWarning)
```

```
# Cookie-cutter template necessary to provide the tools, packages and paths for the␣
↪project. All notebooks
# need this template (or a slightly adjusted one depending on the required packages)
import datetime as dt
import tempfile
from pathlib import Path

import pandas as pd
import spotpy
import xarray as xr

from ravenpy.config import commands as rc
from ravenpy.config.emulators import HMETS
from ravenpy.utilities.calibration import SpotSetup
from ravenpy.utilities.testdata import get_file

# Make a temporary folder
tmp = Path(tempfile.mkdtemp())
```

```
# DATA MAIN SOURCE - DAP link to CANOPEX dataset.
CANOPEX_DAP = "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/dodsC/birdhouse/ets/
↪Watersheds_5797_cfcompliant.nc"
ds = xr.open_dataset(CANOPEX_DAP)

# Explore the dataset:
display(ds)
```

```
# We could explore the dataset and find a watershed of interest, but for now, let's pick␣
↪one at random
# from the dataset:
watershedID = 5600

# And show what it includes:
ts = ds.isel({"watershed": watershedID})
```

```
# Let's write the file to disk to make it more efficient to retrieve:
fname = tmp / "CANOPEX_extracted.nc"
ts.to_netcdf(fname)
ds.close()
ts.close()
```

```
# With this info, we can gather some properties from the CANOPEX database. This same␣
↪database is used for
# regionalization, so let's query it there where more information is available:
tmp = pd.read_csv(get_file("regionalisation_data/gauged_catchment_properties.csv"))

basin_area = float(tmp["area"][watershedID])
basin_latitude = float(tmp["latitude"][watershedID])
basin_longitude = float(tmp["longitude"][watershedID])
basin_elevation = float(tmp["elevation"][watershedID])
basin_name = ds.watershed.data

print("Basin name: ", basin_name)
print("Latitude: ", basin_latitude, " °N")
print("Area: ", basin_area, " km^2")
```

Now, we might have the model and data, but we don't have model parameters! We need to calibrate. This next snippets show how to configure the model and the calibration.

```
# We will also calibrate on only a subset of the years for now to keep the computations␣
↪faster in this notebook.
start_calib = dt.datetime(1998, 1, 1)
end_calib = dt.datetime(1999, 12, 31)

# General parameters depending on the data source. We can find them by exploring the␣
↪CANOPEX dataset in the
# cells above.
data_type = ["TEMP_MAX", "TEMP_MIN", "PRECIP"]

alt_names = {
    "TEMP_MIN": "tasmin",
```

(continues on next page)

(continued from previous page)

```
    "TEMP_MAX": "tasmax",
    "PRECIP": "pr",
}

hru = {}
hru = dict(
    area=basin_area,
    elevation=basin_elevation,
    latitude=basin_latitude,
    longitude=basin_longitude,
    hru_type="land",
)

data_kwds = {
    "ALL": {
        "elevation": hru["elevation"],
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
}
# Set the evaluation metrics to be calculated by Raven
eval_metrics = ("NASH_SUTCLIFFE",)

model_config = HMETS(
    ObservationData=[
        rc.ObservationData.from_nc(fname, alt_names="discharge", station_idx=1)
    ],
    Gauge=[
        rc.Gauge.from_nc(
            fname,
            station_idx=1,
            data_type=data_type,  # Note that this is the list of all the variables
            alt_names=alt_names,  # Note that all variables here are mapped to their␣
↪names in the netcdf file.
            data_kwds=data_kwds,
        )
    ],
    HRUs=[hru],
    StartDate=start_calib,
    EndDate=end_calib,
    RunName="CANOPEX_test",
    EvaluationMetrics=eval_metrics,
    RainSnowFraction="RAINSNOW_DINGMAN",
    SuppressOutput=True,
)
```

```
# The model parameters bounds can either be set independently or we can use the defaults.
low_params = (
    0.3,
    0.01,
    0.5,
    0.15,
```

(continues on next page)

(continued from previous page)

```
    0.0,
    0.0,
    -2.0,
    0.01,
    0.0,
    0.01,
    0.005,
    -5.0,
    0.0,
    0.0,
    0.0,
    0.0,
    0.00001,
    0.0,
    0.00001,
    0.0,
    0.0,
)
high_params = (
    20.0,
    5.0,
    13.0,
    1.5,
    20.0,
    20.0,
    3.0,
    0.2,
    0.1,
    0.3,
    0.1,
    2.0,
    5.0,
    1.0,
    3.0,
    1.0,
    0.02,
    0.1,
    0.01,
    0.5,
    2.0,
)

# Setup the spotpy optimizer
spot_setup = SpotSetup(
    config=model_config,
    low=low_params,
    high=high_params,
)
```

Finally, we can run the optimizer:

```
# We'll definitely want to adjust the random seed and number of model evaluations:
```

(continues on next page)

(continued from previous page)

```
model_evaluations = (
    50  # This is to keep computing time fast for the demo, increase as necessary
)

# Setup the spotpy sampler with the method, the setup configuration, a run name and␣
↪other options. Please refer to
# the spotpy documentation for more options. We recommend sticking to this format for␣
↪efficiency of most applications.
sampler = spotpy.algorithms.dds(
    spot_setup,
    dbname="CANOPEX_test",
    dbformat="ram",
    save_sim=False,
)

# Launch the actual optimization. Multiple trials can be launched, where the entire␣
↪process is repeated and
# the best overall value from all trials is returned.
sampler.sample(model_evaluations, trials=1)
```

```
# Get the model diagnostics
diag = spot_setup.diagnostics

# Print the NSE and the parameter set in 2 different ways:
print("Nash-Sutcliffe value is: " + str(diag["DIAG_NASH_SUTCLIFFE"]))

# Get all the values of each iteration
results = sampler.getdata()

# Get the raw resutlts directly in an array
params = spotpy.analyser.get_best_parameterset(results)[0]
params
```

At this stage, we have calibrated the model on the observations for the desired dates. Now, let's run the model on a longer time period and look at the hydrograph

```
from ravenpy import Emulator

conf = model_config.set_params(params)
conf.suppress_output = False
out = Emulator(conf).run()
```

The `hydrograph` and `storage` outputs are netCDF files storing the time series. These files are opened by default using `xarray`, which provides convenient and powerful time series analysis and plotting tools.

```
q = out.hydrograph.q_sim
```

```
# You can also get statistics from the data directly.
print("Max: ", q.max().values)
print("Mean: ", q.mean().values)
print(
    "Monthly means: ",
```

(continues on next page)

(continued from previous page)

```
    q.groupby("time.month").mean(dim="time").values,
)
```

```
# Plot the simulated hydrograph
from pandas.plotting import register_matplotlib_converters

register_matplotlib_converters()
q.plot()
```

7.2.8 Performing a sensitivity analysis

In this notebook, we perform a sensitivity analysis on GR4JCN to determine the importance of each parameter using the Sobol' sensitivity analysis method. The example shown herein is done using very few parameter samples, and as such, results will be poor and should not be interpreted as-is. However, it is possible to use this code locally using RavenPy to run a much larger sampling on a local computer.

Prepare data for GR4JCN

We will use GR4JCN for this analysis. Since the sensitivity analysis acts on a model response to different inputs, we must find a metric that can be used to measure the impacts of parameters on the model response. In this case, we will use the Nash-Sutcliffe and Absolute Error metrics as responses. It could be any scalar value: mean flow, peak flow, lowest flow, flow volume, etc. But for this exercice we suppose that we want to know the impact of a parameter set on an objective function value. We therefore use a dataset that contains observed streamflow to compute the evaluation metrics.

Let's now import the required packages, get the correct data and setup the model HRU for physiographic information.

```
# Import required packages:
import datetime as dt
import tempfile
from pathlib import Path

import numpy as np
from SALib.analyze import sobol as sobol_analyzer
from SALib.sample import sobol as sobol_sampler
from tqdm.notebook import tqdm

from ravenpy import OutputReader
from ravenpy.config import commands as rc
from ravenpy.config.emulators import GR4JCN
from ravenpy.ravenpy import run
from ravenpy.utilities.testdata import get_file

# We get the netCDF from a server. You can replace the `get_file` function by a string␣
↪containing the path to your own netCDF.
nc_file = get_file(
    "raven-gr4j-cemaneige/Salmon-River-Near-Prince-George_meteo_daily.nc"
)

# Here, we need to give the name of your different dataset in order to match with Raven␣
```

(continues on next page)

(continued from previous page)

```
↪models.
alt_names = {
    "RAINFALL": "rain",
    "SNOWFALL": "snow",
    "TEMP_MIN": "tmin",
    "TEMP_MAX": "tmax",
    "PET": "pet",
    "HYDROGRAPH": "qobs",
}

# The HRU of your watershed
hru = dict(area=4250.6, elevation=843.0, latitude=54.4848, longitude=-123.3659)

# The evaluation metrics. Multiple options are possible, as can be found in Tutorial␣
↪Notebook 06. Here we use Nash-Sutcliffe and Absolute Error.
eval_metrics = ("NASH_SUTCLIFFE", "ABSERR")

# Data keywords for meteorological data stations
data_kwds = {
    "ALL": {
        "elevation": hru[
            "elevation"
        ],  # extract the values directly from the "hru" we previously built
        "latitude": hru["latitude"],
        "longitude": hru["longitude"],
    }
}
```

Sensitivity analysis step 1: Define the Sobol problem to analyze

Sobol sensitivity analysis requires three distinct steps:

1. Sample parameter sets from the possible parameter space;
2. Run the model and gather the model response for each of these parameter spaces;
3. Analyze the change in model responses as a function of changes in parameter sets.

Therefore, the first step is to sample the parameter space.

```
# Define the model inputs:
problem = {
    "num_vars": 6,  # Number of variables
    "names": [
        "x1",
        "x2",
        "x3",
        "x4",
        "CN1",
        "CN2",
    ],  # Names of these variables, to make it easier to follow. Can be any string␣
↪defined by the user
    "bounds": [
```

(continues on next page)

(continued from previous page)

```
        [
            0.01,
            2.5,
        ],  # We must provide lower and upper bounds for each parameter to sample. Must␣
↪be adjusted for each model.
        [-15.0, 10.0],
        [10.0, 700.0],
        [0.0, 7.0],
        [1.0, 30.0],
        [0.0, 1.0],
    ],
}

# Generate samples. The number of parameter sets to generate will be N * (2D + 2), where␣
↪N is defined below and D is the number of
# model inputs (6 for GR4JCN).
N = 4
param_values = sobol_sampler.sample(problem, N)

# Display the size of the param_values matrix. We will run the model with each set of␣
↪parameters, i.e. one per row.
print("The number of parameter sets to evaluate is " + str(param_values.shape[0]))
```

Sensitivity analysis step 2: Run the model for each parameter set

In this stage, we have our sampled parameter sets according to the Sobol / Saltelli sampling methods. We now need to run the GR4JCN model for each of these parameter sets and compute the objective function (model response) that we want. Here we ask the model to pre-compute two objective functions (NSE and MAE), so we will be able to perform the sensitivity analysis on both metrics while only running the model once for each parameter set.

We use a simple loop to run the model here, but advanced users could parallelize this as it is an "embarassingly parallelizable" problem.

```
# Set the working directory at one place so all files are overwritten at the same place␣
↪(Avoids creating hundreds or thousands
# of folders with each run's data)
workdir = Path(tempfile.mkdtemp())

# Pre-define the results matrix based on the number of parameters we will test (and thus␣
↪how many runs we will need to do). We will test SA with
# two objective functions (NSE and AbsErr). Let's pre-define both vectors now.
Y_NSE = np.zeros([param_values.shape[0]])
Y_ABS = np.zeros([param_values.shape[0]])

# Define a run name for files
run_name = "SA_Sobol"

config = dict(
    ObservationData=[rc.ObservationData.from_nc(nc_file, alt_names="qobs")],
    Gauge=[rc.Gauge.from_nc(nc_file, alt_names=alt_names, data_kwds=data_kwds)],
    HRUs=[hru],
```

(continues on next page)

(continued from previous page)

```
    StartDate=dt.datetime(1990, 1, 1),
    EndDate=dt.datetime(1999, 12, 31),
    RunName=run_name,
    EvaluationMetrics=eval_metrics,  # We add this code to tell Raven which objective␣
↪function we want to pass.
    SuppressOutput=True,  # This suppresses the writing of files to disk, returning only␣
↪basic information such as the evaluation metrics values.
)

# Now we have a loop that runs the model iteratively, once per parameter set:
for i, X in enumerate(tqdm(param_values)):
    # We need to create the desired model with its parameters the same way as in the␣
↪Notebook 04_Emulating_hydrological_models.
    m = GR4JCN(
        params=X.tolist(),  # Here is where we pass the paramter sets to the model, from␣
↪the loop enumerator X.
        **config,
    )

    # Write the files to disk, and overwrite existing files in the folder (we already␣
↪got the values we needed from previous runs)
    m.write_rv(workdir=workdir, overwrite=True)

    # Run the model and get the path to the outputs folder that can be used in the␣
↪output reader.
    outputs_path = run(modelname=run_name, configdir=workdir)

    # Get the outputs using the Output Reader object.
    outputs = OutputReader(run_name=run_name, path=outputs_path)

    # Gather the results for both of the desired objective functions. We will see how␣
↪the choice of objective function impacts sensitivity.
    Y_NSE[i] = outputs.diagnostics["DIAG_NASH_SUTCLIFFE"][0]
    Y_ABS[i] = outputs.diagnostics["DIAG_ABSERR"][0]
```

Sensitivity analysis step 3: Analyze results and obtain parameter sensitivity indices

At this point, we have a model response for each of the parameter sets. We can analyze the results using the code below. We will display only the total and 1st order sensitivities.

```
# Perform analysis for the NSE objective function first
Si = sobol_analyzer.analyze(problem, Y_NSE, print_to_console=True)

# Print the first-order sensitivity indices
print(Si["S1"])
```

```
# Now perform the sensitivity analysis for the Absolute Error objective function
Si = sobol_analyzer.analyze(problem, Y_ABS, print_to_console=True)

# Print the first-order sensitivity indices
print(Si["S1"])
```

Result analysis

We can see that parameters x2 and x3 are more sensitive than the other with total (ST) and 1st order (S1) sensitivities higher than the other parameters. This is true for both objective functions, but could also be different for other metrics, so it is important to keep this in mind when using a sensitivity analysis to determine parameter importance! A common example is a parameter related to snowmelt. This parameter will have no impact if there is no snow during the period used in the model, but would become critical if there were to be snow in following years.

Note that the tables above present the sobol sensitivity in the left column and the confidence interval (95%) in the right column. Values are strange because we are using way too few parameter sets to adequately sample the parameter space here, but increasing the value of "N" to 1024 or 2048 would allow for a much better estimation for a 6-parameter model.

7.2.9 Analyzing time series

We will use the 'xclim' package and it's powerful time-series analysis tools to analyze the streamflow observations of the Salmon River basin. We will compute a few indicators, but you can refer to the xclim documentation to see how you can best make use of it for your specific needs.

```
%matplotlib inline

import xarray as xr
import xclim
from pandas.plotting import register_matplotlib_converters

from ravenpy.utilities.testdata import get_file, open_dataset

register_matplotlib_converters()

# Get the file we will use to analyze flows
file = "hydro_simulations/raven-gr4j-cemaneige-sim_hmets-0_Hydrographs.nc"
ds = open_dataset(file)
```

Base flow index

The base flow index is the minimum 7-day average flow divided by the mean flow.

```
help(xclim.land.base_flow_index)
```

The base flow index needs as input arguments a DataArray storing the stream flow time series, and the frequency at which the index is computed (`YS`: yearly, `QS-DEC`: seasonally).

```
out = xclim.land.base_flow_index(ds.q_sim)
out.plot()
```

To compute generic statistics of a time series, use the `stats` process.

```
help(xclim.generic.stats)
```

```
# Here we compute the annual summer (JJA) minimum
out = xclim.generic.stats(ds.q_sim, op="min", season="JJA")
out.plot()
```

Frequency analysis

The process `freq_analysis` is similar to the previous stat in that it fits a series of annual maxima or minima to a statistical distribution, and returns the values corresponding to different return periods.

```
help(xclim.generic.return_level)
```

For example, computing the Q(2,7), the minimum 7-days streamflow with a two-year reoccurrence, can be done using the following.

```
out = xclim.generic.return_level(ds.q_sim, mode="min", t=2, dist="gumbel_r", window=7)
out
```

An array of return periods can be passed.

```
out = xclim.generic.return_level(
    ds.q_sim, mode="max", t=(2, 5, 10, 25, 50, 100), dist="gumbel_r"
)
out.plot()
```

Getting the parameters of the distribution and comparing the fit

It's sometimes more useful to store the fitted parameters of the distribution rather than storing only the quantiles. In the example below, we're first computing the annual maxima of the simulated time series, then fitting them to a gumbel distribution using the `fit` process.

```
import json

with xclim.set_options(
    check_missing="pct", missing_options={"pct": {"tolerance": 0.05}}
):
    ts = xclim.generic.stats(ds.q_sim, op="max")

ts
```

```
with xclim.set_options(check_missing="skip"):
    pa = xclim.generic.fit(ts.isel(nbasins=0), dist="gumbel_r")
pa
```

CHAPTER

EIGHT

CONTRIBUTING

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

You can contribute in many ways:

8.1 Types of Contributions

8.1.1 Report Bugs

Report bugs at https://github.com/CSHS-CWRA/ravenpy/issues.

If you are reporting a bug, please include:

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

8.1.2 Fix Bugs

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

8.1.3 Implement Features

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

8.1.4 Write Documentation

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

8.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/CSHS-CWRA/ravenpy/issues.

If you are proposing a feature:

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

8.2 Get Started!

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

1. Fork the *ravenpy* repo on GitHub.
2. Clone your fork locally:

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

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

```
$ mkvirtualenv ravenpy
$ cd ravenpy/
$ pip install -e ".[dev]"
```

4. To ensure a consistent style, please install the pre-commit hooks to your repo:

```
$ pre-commit install
```

   Special style and formatting checks will be run when you commit your changes. You can always run the hooks on their own with:

```
$ pre-commit run -a
```

5. Create a branch for local development:

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

   Now you can make your changes locally.

6. When you're done making changes, check that your changes pass flake8, black, and the tests, including testing other Python versions with tox:

```
$ flake8 ravenpy tests
$ black --check ravenpy tests
$ pytest tests
$ tox
```

   To get flake8, black, and tox, just pip install them into your virtualenv.

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

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

7. If you are editing the docs, compile and open them with:

```
$ make docs
# or to simply generate the html
$ cd docs/
$ make html
```

8. Submit a pull request through the GitHub website.

8.3 Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.8, 3.9, 3.10, and 3.11. Check https://github.com/CSHS-CWRA/RavenPy/actions/workflows/main.yml and make sure that the tests pass for all supported Python versions.

8.4 Tips

To run a subset of tests:

```
$ pytest tests.test_ravenpy
```

8.5 Versioning/Tagging

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

8.6 Packaging

When a new version has been minted (features have been successfully integrated test coverage and stability is adequate), maintainers should update the pip-installable package (wheel and source release) on PyPI as well as the binary on conda-forge.

8.6.1 The Automated Approach

The simplest way to package *ravenpy* is to "publish" a version on GitHuh. GitHub CI Actions are presently configured to build the library and publish the packages on PyPI automatically.

Tagged versions will trigger a GitHub Workflow (*tag-testpypi.yml*) that will attempt to build and publish the release on TestPyPI.

Note: Should this step fail, changes may be needed in the package; Be sure to remove this tag on GitHub and locally, address any existing problems, and recreate the tag.

To upload a new version to PyPI, simply create a new "Published" release version on GitHub to trigger the upload workflow (*publish-pypi.yml*). When publishing on GitHub, the maintainer can either set the release notes manually (based on the *HISTORY.rst*), or set GitHub to generate release notes automatically. The choice of method is up to the maintainer.

Warning: A published version on TestPyPI/PyPI can never be overwritten. Be sure to verify that the package published at https://test.pypi.org/project/ravenpy/ matches expectations before publishing a release version on GitHub.

8.6.2 The Manual Approach

The manual approach to library packaging for general support (pip wheels) requires that the flit library is installed.

From the command line on your Linux distribution, simply run the following from the clone's main dev branch:

```
# To build the packages (sources and wheel)
$ flit build

# To upload to PyPI
$ flit publish
```

The new version based off of the version checked out will now be available via *pip* (*$ pip install ravenpy*).

8.6.3 Releasing on conda-forge

Initial Release

In order to prepare an initial release on conda-forge, we *strongly* suggest consulting the following links:

- https://conda-forge.org/docs/maintainer/adding_pkgs.html
- https://github.com/conda-forge/staged-recipes

Before updating the main conda-forge recipe, we echo the conda-forge documentation and *strongly* suggest performing the following checks:

- Ensure that dependencies and dependency versions correspond with those of the PyPI published version, with open or pinned versions for the *host* requirements.
- If possible, configure tests within the conda-forge build CI, e.g.:

```
test:
  source_files:
    - tests
  requires:
    - pip
    - pytest
    - pytest-xdist
  imports:
    - ravenpy
  commands:
    - pip check
    - pytest
```

Subsequent releases

If the conda-forge feedstock recipe is built from PyPI, then when a new release is published on PyPI, *regro-cf-autotick-bot* will open Pull Requests automatically on the conda-forge feedstock. It is up to the conda-forge feedstock maintainers to verify that the package is building properly before merging the Pull Request to the main branch.

CHAPTER

NINE

CREDITS

9.1 Development Lead

- David Huard <huard.david@ouranos.ca> @huard

9.2 Co-Developers

- Christian Jauvin <jauvin.christian@ouranos.ca> @cjauvin
- Trevor James Smith <smith.trevorj@ouranos.ca> @Zeitsperre
- Julie Mai <juliane.mai@uwaterloo.ca> @julemai
- Ming Han <ming.han@uwaterloo.ca> @dustming

CHAPTER

TEN

HISTORY

10.1 0.15.0 (unreleased)

- Fixed bug in *Config.duplicate* dating from the switch to Pydantic V2 in 0.13 (PR #367)

10.2 0.14.1 (2024-05-07)

- Upgraded *owslib* to *>=0.29.1*. (PR #358)
- All operations that open NetCDF files or DAP links accept an *engine* argument. The default for all of these is *h5netcdf*. (PR #358)
- Added *pydap* as an alternate backend for opening DAP links. (PR #358)
- Fixed buggy CustomOutput command. (PR #360)
- Make sure config and output paths are absolute. (PR #360)

10.2.1 Internal changes

- Added some development dependencies that were missing to the *environment.yml*. (PR #358)
- *test_climpred_hindcast_verif* is now skipped for Python3.10 builds. It seems to only fail on the particular version of Python. When examining the dependencies, other than the Python version (and ABI version), there are no differences in the environments between Python3.10 and Python3.11. Possibly an issue with *climpred*. (PR #358)
- Temporarily disabled tests for macOS on GitHub due to architecture changes. (PR #358)
- Pinned *pyogrio* below v0.8.0 until *geopandas* supports it. (PR #363)
- Updated linting dependencies to the latest versions. (PR #363)

10.3 0.14.0 (2024-03-13)

- Add support for new processes and methods added in Raven v3.8. (PR #335)
- Add Interpolation command options. (PR #338)
- Let VegetationClass records contain symbolic expressions. (PR #338)
- Add support for custom RV subclasses. (PR #338)
- Use HRU_ID (if available) instead of SubId in BasinMaker reservoirs extraction logic. (PR #338)
- Added support for Python 3.12 and dropped support for Python3.8. (PR #341, PR #343)
- Added support for *raven-hydro* v0.3.0 and *RavenHydroFramework* to v3.8. (PR #341, PR #351)
- *ravenpy* now requires *xclim* >= v0.48.2, *xarray* >= v2023.11.0, and *pandas* >= 2.2.0. (PR #341)
- Now automatically filters HRUs based on the `hru_type`. (issue #340, PR #334)

10.3.1 Internal changes

- Updated GitHub publishing workflows to use Trusted Publisher for TestPyPI/PyPI releases. (PR #341)
- Added Dependabot to keep dependencies up-to-date. (PR #342)
- Now using step-security/harden-runner action to harden GitHub Actions runners. (PR #341)
- Adjusted GitHub Workflows to test against Python 3.9, 3.10, 3.11, and 3.12. (PR #341, PR #343)
- Updated the build-system requirements when testing with *tox* to use newer *setuptools* and *wheel* versions when building *gdal*. (PR #341)

10.4 0.13.0 (2024-01-10)

- Fixed problem with scalar elevation in netCDF files parsed with *nc_specs*. (issue #279, PR #323)
- Added notebook on sensitivity analysis. (PR #320)
- Updated Notebooks 03 and 04. (PR #319)
- Upgrade to *pydantic* v2.0. (PR #326)
- Pin *cf-xarray* for Python3.8. (PR #325)
- Fix *Coveralls* Workflows. (PR #328)
- Fix notebook execution. (PR #329)
- Refactor and simplify testing data fetching. (PR #332)

10.4.1 Breaking changes

- Update to *pydantic* v2.0. (PR #326)
- Added *h5netcdf* as a core dependency to provide a stabler backend for *xarray.open_dataset*. (PR #332)
- Switched from *autodoc_pydantic* to *autodoc-pydantic* for *pydantic* v2.0+ support in documentation. (PR #326)

10.4.2 Internal changes

- Removed some redundant *pytest* fixtures for running *emulators* tests.
- *"session"*-scoped *pytest* fixtures used for hindcasting/forecasting are now always yielded and copied to new objects within tests.

10.5 0.12.3 (2023-10-02)

- *RavenPy* now uses *platformdirs* to write *raven_testing* to the user's cache directory. Dynamic paths are now used to cache data dependent on the user's operating system. Developers can now safely delete the *.raven_testing_data* folder in their home directory without affecting the functionality of *RavenPy*.
- Updated *raven-hydro* to v0.2.4 to address CMake build issues.

10.5.1 Breaking changes

- In tests, set *xclim*'s missing value option to `skip`. As of *xclim* v0.45, missing value checks are applied to the `fit` indicator, meaning that parameters will be set to *None* if missing values are found in the fitted time series. Wrap calls to `fit` with `xclim.set_options(check_missing="skip")` to reproduce the previous behavior of xclim.
- The *_determine_upstream_ids* function under *ravenpy.utilities.geoserver* has been removed as it was a duplicate of *ravenpy.utilities.geo.determine_upstream_ids*. The latter function is now used in its place.

10.5.2 Internal changes

- Added a GitHub Actions workflow to remove obsolete GitHub Workflow cache files.
- *RavenPy* now accepts a *RAVENPY_THREDDS_URL* for setting the URL globally to the THREDDS-hosted climate data service. Defaults to *https://pavics.ouranos.ca/twitcher/ows/proxy/thredds*.
- ***RavenPy* processes and tests that depend on remote GeoServer calls now allow for optional server URL and file location targets. The server URL can be set globally with the following environment variable:**
  - ***RAVENPY_GEOSERVER_URL*: URL to the GeoServer-hosted vector/raster data. Defaults to *https://pavics.ouranos.ca/geoserver*. This environment variable was previously called *GEO_URL* but was renamed to narrow its scope to *RavenPy*.**
    - *GEO_URL* is still supported for backward compatibility but may eventually be removed in a future release.
- *RavenPy* has temporarily pinned *xarray* below v2023.9.0 due to incompatibilities with *xclim* v0.45.0`.

10.6 0.12.2 (2023-07-04)

This release is primarily a bugfix to address issues arising from dependencies.

10.6.1 Breaking changes

- *raven-hydro* version has been bumped from v0.2.1 to v0.2.3. This version provides better support for builds on Windows and MacOS.
- Due to major breaking changes, *pydantic* has been pinned below v2.0 until changes can be made to adapt to their new API.
- *numpy* has been pinned below v1.25.0 to ensure compatibility with *numba*.

10.6.2 Internal changes

- `test_geoserver::test_select_hybas_ar_domain_point` is now temporarily skipped when testing on MacOS due to a mysterious domain identification error.

10.7 0.12.1 (2023-06-01)

This release is largely a bugfix to better stabilize performance and enhance the documentation.

- Avoid repeatedly calling *xr.open_dataset* in *OutputReader*'s *hydrograph* and *storage* properties. This seems to cause kernel failures in Jupyter notebooks.

10.7.1 Internal changes

- Hyperlinks to documented functions now points to entries in the *User API* section.
- Docstrings are now more conformant to numpy-docstring conventions and formatting errors raised from badly-formatted pydantic-style docstrings have been addressed.
- In order to prevent timeout and excessive memory usage, Jupyter notebooks have been adjusted to no longer run on ReadTheDocs. All notebooks have been updated to the latest RavenPy and remain tested against RavenPy externally.
- Documentation built on ReadTheDocs is now set to *fail_on_warning*.

10.8 0.12.0 (2023-05-25)

This release includes major breaking changes. It completely overhauls how models are defined, and how to run simulations, and any code relying on the previous release will most likely break. Please check the documentation to see how to use the new improved interface.

10.8.1 Breaking changes

- The entire model configuration and simulation interface (see PR #269).
- The Raven model executable is now updated to v3.7.
- Added support for Ensemble Kalman Filter using RavenC.
- Now employing the *spotpy* package for model calibration instead of *ostrich*.
- BasinMaker importer assumes *SubBasin=HRU* in order to work with files downloaded from the BasinMaker web site.
- Ravenpy now employs a new method for installing the Raven model using the raven-hydro python package (based on *scikit-build-core*) (see PR #278).
- Replaced *setup.py*, *requirements.txt*, and *Manifest.in* for PEP 517 compliance (*pyproject.toml*) using the flit backend (see PR #278).
- Dealt with an import-based error that occurred due to the sequence in which modules are loaded at import (attempting to call ravenpy before it is installed).
- Updated pre-commit hooks to include formatters and checkers for TOML files.
- The build recipes no longer build on each other, so when installing the *dev* or *docs* recipe, you must also install the gis recipe.
- Updated the GeoServer API calls to work with the GeoPandas v0.13.0.

10.9 0.11.0 (2023-02-16)

- Update RavenC executable to v3.6.
- Update xclim library to v0.40.0.
- Update fiona library to v1.9.
- Address some failures that can be caused when attempting to run CLI commands without the proper GIS dependencies installed.
- Addressed warnings raised in conda-forge compilation due to badly-configured MANIFEST.in.
- Update installation documentation to reflect most recent changes.

10.10 0.10.0 (2022-12-21)

- Update Raven executable to 3.5. Due to a bug in RavenC, simulations storing reservoir information to netCDF will fail. We expect this to be resolved in the next release. Note that we only test RavenPy with one Raven version. There is no guarantee it will work with other versions.
- Relax geo test to avoid failures occurring due to GDAL 3.6.
- Pin numpy below 1.24 (see https://github.com/numba/numba/issues/8615)

10.11 0.9.0 (2022-11-16)

10.11.1 Breaking changes

- HRUState's signature has changed. Instead of passing variables as keyword arguments (e.g. *soil0=10.*), it now expects a *state* dictionary keyed by variables' Raven name (e.g. *{"SOIL[0]": 10}). This change makes `rvc* files easier to read, and avoids Raven warnings regarding 'initial conditions for state variables not in model'.
- *nc_index* renamed to *meteo_idx* to enable the specification of distinct indices for observed streamflow using *hydro_idx*. *nc_index* remains supported for backward compatibility.
- The distributed python testing library, *pytest-xdist* is now a testing and development requirement.
- *xarray* has been pinned below "2022.11.0" due to incompatibility with *climpred==*"2.2.0"*.

10.11.2 New features

- Add support for hydrometric gauge data distinct from meteorological input data. Configuration parameter *hydro_idx* identifies the gauge station index, while *meteo_idx* (previously *nc_index*) stands for the meteo station index.
- Add support for multiple gauge observations. If a list of *hydro_idx* is provided, it must be accompanied with a list of corresponding subbasin identifiers (*gauged_sb_ids*) of the same length.
- Automatically infer scale and offset *:LinearTransform* parameters from netCDF file metadata, so that input data units are automatically converted to Raven-compliant units whenever possible.
- Add support for the command *:RedirectToFile*. Tested for grid weights only.
- Add support for the command *:WriteForcingFunctions*.
- Add support for the command *:CustomOutput*.
- Multiple other new RavenCommand objects added, but not integrated in the configuration, including *:SoilParameterList*, *:VegetationParameterList* and *:LandUseParameterList*.
- Multichoice options (e.g. calendars) moved from RV classes to *config.options*, but aliases created for backward compatibility.
- Patch directory traversal vulnerability (CVE-2007-4559).
- A local copy of the raven-testdata with environment variable (*RAVENPY_TESTDATA_PATH*) set to that location is now no longer needed in order to run the testing suite. Test data is fetched automatically and now stored at *~/.raven_testing_data*.
- **RavenPy now leverages *pytest-xdist* to distribute tests among Python workers and significantly speed up the testing suite, depending on number of available CPUs. File access within the testing suite has also been completely rewritten for thread safety.**
  - On pytest launch with "*–numprocesses* > 0", testing data will be fetched automatically from *Ouranosinc/raven-testdata* by one worker, blocking others until this step is complete. Spawned pytest workers will then copy the testing data to their respective temporary directories before beginning testing.
- **To aid with development and debugging purposes, two new environment variables and pytest fixtures are now available:**
  - In order to skip the data collection step: *export SKIP_TEST_DATA=true*
  - In order to target a specific branch of *Ouranosinc/raven-testdata* for data retrieval: *export MAIN_TESTDATA_BRANCH="my_branch"*

  - In order to fetch testing data using the user-set raven-testdata branch, pytest fixtures for *get_file* and *get_local_testdata* are now available for convenience

10.12 0.8.1 (2022-10-26)

- Undo change related to *suppress_output*, as it breaks multiple tests in raven. New *Raven._execute* method runs models but does not parse results.

10.13 0.8.0

10.13.1 Breaking changes

- Parallel parameters must be provided explicitly using the *parallel* argument when calling emulators.
- Multiple *nc_index* values generate multiple *gauges*, instead of being parallelized.
- Python3.7 is no longer supported.
- Documentation now uses sphinx-apidoc at build-time to generate API pages.
- Add `generate-hrus-from-routing-product` script.
- Do not write RV zip file and merge outputs when *suppress_output* is True. Zipping rv files during multiple calibration runs leads to a non-linear performance slow-down.
- Fixed issues with coverage reporting via tox and GitHub Actions
- Add partial support for *:RedirectToFile* command, tested with GridWeights only.

10.14 0.7.8

- Added functionalities in Data Assimilation utils and simplified tests.
- Removed pin on setuptools.
- Fixed issues related to symlinks, working directory, and output filenames.
- Fixed issues related to GDAL version handling in conda-forge.
- Updated jupyter notebooks.

10.15 0.7.7

- Updated internal shapely calls to remove deprecated `.to_wkt()` methods.

10.16 0.7.6

- Automate release pipeline to PyPI using GitHub CI actions.
- Added coverage monitoring GitHub CI action.
- Various documentation adjustments.
- Various metadata adjustments.
- Pinned owslib to 0.24.1 and above.
- Circumvented a bug in GitHub CI that was causing tests to fail at collection stage.

10.17 0.7.5

- Update test so that it works with xclim 0.29.

10.18 0.7.4

- Pinned climpred below v2.1.6.

10.19 0.7.3

- Pinned xclim below v0.29.

10.20 0.7.2

- Update cruft.
- Subclass `derived_parameters` in Ostrich emulators to avoid having to pass `params`.

10.21 0.7.0

- Add support for V2.1 of the Routing Product in `ravenpy.extractors.routing_product`.
- Add `collect-subbasins-upstream-of-gauge` CLI script.
- Modify WFS request functions to use spatial filtering (`Intersects`) supplied by OWSLib.

10.22 0.6.0

- Add support for EvaluationPeriod commands. Note that as a result of this, the model's `diagnostics` property contains one list per key, instead of a single scalar. Also note that for calibration, Ostrich will use the first period and the first evaluation metric.
- Add `SACSMA`, `CANADIANSHIELD` and `HYPR` model emulators.

10.23 0.5.2

- Simplify RVC configuration logic.
- Add `ravenpy.utilities.testdata.file_md5_checksum` (previously in `xarray.tutorial`).

10.24 0.5.1

- Some adjustments and bugfixes needed for RavenWPS.
- Refactoring of some internal logic in `ravenpy.config.rvs.RVT`.
- Improvements to typing with the help of mypy.

10.25 0.5.0

- Refactoring of the RV config subsystem:
  - The config is fully encapsulated into its own class: `ravenpy.config.rvs.Config`.
  - The emulator RV templates are inline in their emulator classes.
- The emulators have their own submodule: `ravenpy.models.emulators`.
- The "importers" have been renamed to "extractors" and they have their own submodule: `ravenpy.extractors`.

10.26 0.4.2

- Update to RavenC revision 318 to fix OPeNDAP access for StationForcing commands.
- Fix grid_weights set to None by default.
- Pass nc_index to ObservationData command.
- Expose more cleanly RavenC errors and warnings.

10.27 0.4.1

- Add notebook about hindcast verification skill.
- Add notebook about routing capability.
- Modify geoserver functions to have them return GeoJSON instead of GML.
- Collect upstream watershed aggregation logic.
- Fix RVC bug.

10.28 0.4.0

This is an interim version making one step toward semi-distributed modeling support. Model configuration is still in flux and will be significantly modified with 0.5. The major change in this version is that model configuration supports passing multiple HRU objects, instead of simply passing area, latitude, longitude and elevation for a single HRU.

- GR4JCN emulator now supports routing mode.
- Add BLENDED model emulator.
- DAP links for forcing files are now supported.
- Added support for `tox`-based localized installation and testing with python-pip.
- Now supporting Python 3.7, 3.8, and 3.9.
- Build testing for `pip` and `conda`-based builds with GitHub CI.

10.29 0.3.1

- Update external dependencies (Raven, OSTRICH) to facilitate Conda packaging.

10.30 0.3.0

- Migration and refactoring of GIS and IO utilities (`utils.py`, `utilities/gis.py`) from RavenWPS to RavenPy.
- RavenPy can now be installed from PyPI without GIS dependencies (limited functionality).
- Hydro routing product is now supported from `geoserver.py` (a notebook has been added to demonstrate the new functions).
- New script `ravenpy aggregate-forcings-to-hrus` to aggregate NetCDF files and compute updated grid weights.
- Add the basis for a new routing emulator option (WIP).
- Add climpred verification capabilities.

10.31 0.2.3

- Regionalisation data is now part of the package.
- Fix tests that were not using testdata properly.
- Add tests for external dataset access.
- `utilities.testdata.get_local_testdata` now raises an exception when it finds no dataset corresponding to the user pattern.

10.32 0.2.2

- Set wcs.getCoverage timeout to 120 seconds.
- Fix `Raven.parse_results` logic when no flow observations are present and no diagnostic file is created.
- Fix ECCC test where input was cached and shadowed forecast input data.

10.33 0.2.1

- Fix xarray caching bug in regionalization.

10.34 0.2.0

- Refactoring of `ravenpy.utilities.testdata` functions.
- Bump xclim to 0.23.

10.35 0.1.7

- Fix xarray caching bug affecting climatological ESP forecasts (#33).
- Fix deprecation issue with Fiona.

10.36 0.1.6 (2021-01-15)

- Correct installer bugs.

10.37 0.1.5 (2021-01-14)

- Release with docs.

10.38 0.1.0 (2020-12-20)

- First release on PyPI.

CHAPTER

ELEVEN

UTILITY SCRIPTS

11.1 ravenpy generate-grid-weights

Generate grid weights in various formats.

INPUT_FILE: File containing model discretization. Can be either:

(A) NetCDF file containing at least 1D or 2D latitudes and 1D or 2D longitudes where this grid needs to be representative of model outputs that are then required to be routed. The names of the dimensions and the variables holding the lat/lon information should be specified with options –dim-names (-d) and –var-names (-v).

(B) Shapefile (either a .shp or a .zip) that contains shapes of subbasins and one attribute in this shapefile that is defining its index in the NetCDF model output file (numbering needs to be [0 … N-1]). The name of this attribute should be specified via option –netcdf-input-field (-f).

ROUTING_FILE: Shapefile (either a .shp or a .zip) that contains all information of the routing toolbox for the catchment of interest (and maybe some more catchments). This file should contain an attribute with a unique identifier for the HRUs: the default is "HRU_ID", but it can be set with –routing-id-field (-c).

The script will output the results as RVT file with a single :GridWeights command block containing the weights (that the user is then free to embed or reference, in her own config context).

```
ravenpy generate-grid-weights [OPTIONS] INPUT_FILE ROUTING_FILE
```

Options

-d, --dim-names <dim_names>

Ordered dimension names of longitude (x) and latitude (y) in the NetCDF INPUT_FILE.

Default
lon_dim, lat_dim

-v, --var-names <var_names>

Variable name of 1D or 2D longitude and latitude variables in the NetCDF INPUT_FILE (in this order).

Default
longitude, latitude

-c, --routing-id-field <routing_id_field>

Name of column in routing information shapefile (ROUTING_FILE) containing a unique key for each dataset.

Default
SubId

-f, --netcdf-input-field <netcdf_input_field>

Attribute name in INPUT_FILE shapefile that defines the index of the shape in NetCDF model output file (numbering needs to be [0 … N-1]).

Default
NetCDF_col

-g, --gauge-id <gauge_ids>

Streamflow gauge IDs of interest (corresponds to 'Gauge_ID' in the ROUTING_FILE shapefile).

-s, --sub-id <sub_ids>

IDs of subbasins of interest (containing usually a gauge station, corresponds to 'SubId' in the ROUTING_FILE shapefile).

-e, --area-error-threshold <area_error_threshold>

Threshold (as fraction) of allowed mismatch in areas between subbasins from routing information (ROUTING_FILE) and overlay with grid-cells or subbasins (INPUT_FILE). If error is smaller than this threshold the weights will be adjusted such that they sum up to exactly 1. Raven will exit gracefully in case weights do not sum up to at least 0.95.

Default
0.05

-o, --output <output>

Text field that will contain the results as a single :GridWeights Raven command containing the weights.

Arguments

INPUT_FILE

Required argument

ROUTING_FILE

Required argument

11.2 ravenpy aggregate-forcings-to-hrus

Aggregates NetCDF files containing 3-dimensional forcing variables like precipitation and temperature over (x,y,time) into 2-dimensional forcings for each of the n HRUs of a specific basin over (n,time). The 3-dimensional NetCDF files are usually used in `:GriddedForcing` commands in Raven while the 2-dimensional ones can be used in `:StationForcing` commands. The NetCDF files generated with this function will only contain the forcings required to simulate an individual basin and hence file sizes are smaller and Raven runtimes can decrease drastically under certain conditions.

INPUT_NC_FILE: NetCDF file containing 3-dimensional variables that will be aggregated. Either all variables will be aggregated or only a subset specified using –var-to-aggregate (e.g., [precip,temp]). The name of the spatial dimensions of the NetCDF are assumed to be (lon_dim, lat_dim). Otherwise they will need to be specified using –dim-names. The order of the three dimensions for each variable does not matter; the function will arrange them as required.

INPUT_WEIGHT_FILE: A text file containing the grid weights derived using the script "generate-grid-weights" for the basin forcings are required and the specified NetCDF file. The content of this file must be formatted as a valid `:GridWeights` Raven command.

The script outputs two files:

(1) Aggregated NetCDF file that can be used in a `:StationForcing` command in a Raven config.

(2) A text file (with the same format as INPUT_WEIGHT_FILE) with the updated grid weights, that a `:StationForcing` command will require.

```
ravenpy aggregate-forcings-to-hrus [OPTIONS] INPUT_NC_FILE INPUT_WEIGHT_FILE
```

Options

-d, --dim-names <dim_names>

Ordered dimension names of longitude (x) and latitude (y) in the NetCDF INPUT_NC_FILE.

Default
`lon_dim, lat_dim`

-v, --var-to-aggregate <variables_to_aggregate>

Required Variables to aggregate in INPUT_NC_FILE (at least one).

--output-nc-file <output_nc_file>

--output-weight-file <output_weight_file>

Arguments

INPUT_NC_FILE

Required argument

INPUT_WEIGHT_FILE

Required argument

11.3 ravenpy collect-subbasins-upstream-of-gauge

Find the subbasins upstream of a gauge from a Routing Product shapefile, and save them in a new shapefile.

INPUT_FILE: Routing Product shapefile (e.g. “drainage_region_0003_v2-1/finalcat_info_v2-1.shp”).

GAUGE_ID: ID of the target gauge, to be found in the “Obs_NM” column (e.g. “02LE024”).

```
ravenpy collect-subbasins-upstream-of-gauge [OPTIONS] INPUT_FILE GAUGE_ID
```

Options

-o, --output <output>

Name of the output shapefile.

Arguments

INPUT_FILE

Required argument

GAUGE_ID

Required argument

11.4 ravenpy generate-hrus-from-routing-product

Create a new HRU shapefile by splitting every subbasin row of a Routing Product V2.1 shapefile into at least a land HRU and possibly an additional lake HRU.

INPUT_FILE: Routing Product V2.1 shapefile (e.g. "drainage_region_0003_v2-1/finalcat_info_v2-1.shp").

```
ravenpy generate-hrus-from-routing-product [OPTIONS] INPUT_FILE
```

Options

-o, --output <output>

Output shapefiles (will create a folder if no extension)

Arguments

INPUT_FILE

Required argument

CHAPTER

TWELVE

USER API

12.1 Execution

Main module.

class ravenpy.ravenpy.**Emulator**(*config:* Config, *workdir: str | PathLike | None = None*, *modelname: str | None = None*, *overwrite: bool = False*)

property config: *Config*

Read-only model configuration.

property modelname: **str**

File name stem of configuration files.

property output: *OutputReader*

Return simulation output object.

property output_path: **Path | None**

Path to model outputs.

resume(*timestamp: bool = True*) → *Config*

Return new model configuration using state variables from the end of the run.

timestamp: bool
If False, ignore time stamp information in the solution. If True, the solution will set StartDate to the solution's timestamp.

run(*overwrite: bool = False*) → *OutputReader*

Run the model. This will write RV files if not already done.

Parameters
overwrite (*bool*) – If True, overwrite existing files.

property workdir: **Path**

Path to RV files and output subdirectory.

class ravenpy.ravenpy.**EnsembleReader**(**, run_name: str | None = None*, *paths: List[str | PathLike] | None = None*, *runs: List[*OutputReader*] | None = None*, *dim: str = 'member'*)

property files

property hydrograph

property storage

class ravenpy.ravenpy.**OutputReader**(*run_name: str | None = None*, *path: str | Path | None = None*)

property diagnostics: dict | None

Return model diagnostics.

property files: dict

Return paths to output files.

property hydrograph: Dataset

Return the hydrograph.

property messages: str | None

property path: Path

Path to output directory.

property solution: dict | None

Return solution file content.

property storage: Dataset

Return the storage variables.

exception ravenpy.ravenpy.**RavenError**

This is an error that is meant to be raised whenever a message of type "ERROR" is found in the Raven_errors.txt file resulting from a Raven (i.e. the C program) run.

exception ravenpy.ravenpy.**RavenWarning**

This is a warning corresponding to a message of type "WARNING" in the Raven_errors.txt file resulting from a Raven (i.e. the C program) run.

ravenpy.ravenpy.**run**(*modelname: str*, *configdir: str | Path*, *outputdir: str | Path | None = None*, *overwrite: bool = True*, *verbose: bool = False*) → Path

Run Raven given the path to an existing model configuration.

Parameters

- **modelname** (*str*) – Configuration files stem, i.e. the file name without extension.
- **configdir** (*Path or str*) – Path to configuration files directory.
- **outputdir** (*Path or str, optional*) – Path to model simulation output. If None, will write to configdir/output.
- **overwrite** (*bool*) – If True, overwrite existing files.
- **verbose** (*bool*) – If True, always display Raven warnings. If False, warnings will only be printed if an error occurs.

Returns

Path to model outputs.

Return type

Path

12.2 Configuration

class ravenpy.config.commands.**AssimilateStreamflow**(*, *sb_id: str*)

Subbasin ID to assimilate streamflow for.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'sb_id': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

sb_id: str

class ravenpy.config.commands.**AssimilatedState**(*, *state: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | Literal['STREAMFLOW']*, *group: str*)

group: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'group': FieldInfo(annotation=str, required=True), 'state': FieldInfo(annotation=Union[Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'], Literal['STREAMFLOW']], required=True)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

state: `Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | Literal['STREAMFLOW']`

class ravenpy.config.commands.**BasinIndex**(*, *sb_id: int = 1*, *name: str = 'watershed'*, *ChannelStorage: float = 0.0*, *RivuletStorage: float = 0.0*, *Qout: Sequence[float] = (1.0, 0.0, 0.0)*, *Qlat: Sequence[float] | None = None*, *Qin: Sequence[float] | None = None*)

Initial conditions for a flow segment.

channel_storage: `float`

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'channel_storage': FieldInfo(annotation=float, required=False, default=0.0, alias='ChannelStorage', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default='watershed'), 'qin': FieldInfo(annotation=Union[Sequence[float], NoneType], required=False, default=None, alias='Qin', alias_priority=2), 'qlat': FieldInfo(annotation=Union[Sequence[float], NoneType], required=False, default=None, alias='Qlat', alias_priority=2), 'qout': FieldInfo(annotation=Sequence[float], required=False, default=(1.0, 0.0, 0.0), alias='Qout', alias_priority=2), 'rivulet_storage': FieldInfo(annotation=float, required=False, default=0.0, alias='RivuletStorage', alias_priority=2), 'sb_id': FieldInfo(annotation=int, required=False, default=1)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

classmethod parse(*s*)

qin: Sequence[float] | None

qlat: Sequence[float] | None

qout: Sequence[float]

rivulet_storage: float

sb_id: int

class ravenpy.config.commands.**BasinStateVariables**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.BasinIndex], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod parse(*sol*)

root: Sequence[*BasinIndex*]

class ravenpy.config.commands.**ChannelProfile**(*, *name: str = 'chn_XXX'*, *bed_slope: float = 0*, *survey_points: Tuple[Tuple[float, float], ...] = ()*, *roughness_zones: Tuple[Tuple[float, float], ...] = ()*)

ChannelProfile command (RVP).

bed_slope: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'bed_slope': FieldInfo(annotation=float, required=False, default=0), 'name': FieldInfo(annotation=str, required=False, default='chn_XXX'), 'roughness_zones': FieldInfo(annotation=Tuple[Tuple[float, float], ...], required=False, default=()), 'survey_points': FieldInfo(annotation=Tuple[Tuple[float, float], ...], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

roughness_zones: Tuple[Tuple[float, float], ...]

survey_points: Tuple[Tuple[float, float], ...]

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**CustomOutput**(*, *time_per: Literal['DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS']*, *stat: Literal['AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES']*, *variable: str*, *space_agg: Literal['BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED']*, *filename: str = ''*)

Create custom output file to track a single variable, parameter or forcing function over time at a number of basins, HRUs, or across the watershed.

Parameters

- **time_per** (*{'DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS'}*) – Time period.
- **stat** (*{'AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES', 'HISTOGRAM [min] [max] [# bins]'*) – Statistic reported for each time interval.
- **variable** (*str*) – Variable or parameter name. Consult the Raven documentation for the list of allowed names.
- **space_agg** (*{'BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED'}*) – Spatial evaluation domain.
- **filename** (*str*) – Output file name. Defaults to something approximately like *<run name>_<variable>_<time_per>_<stat>_<space_agg>.nc*

filename: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'filename': FieldInfo(annotation=str, required=False, default=''), 'space_agg': FieldInfo(annotation=Literal['BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED'], required=True), 'stat': FieldInfo(annotation=Literal['AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES'], required=True), 'time_per': FieldInfo(annotation=Literal['DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS'], required=True), 'variable': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

space_agg: Literal['BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED']

stat: Literal['AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES']

time_per: Literal['DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS']

variable: str

class ravenpy.config.commands.**Data**(*, *data_type: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] = ''*, *units: str = ''*, *ReadFromNetCDF:* ReadFromNetCDF)

data_type: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR']

classmethod from_nc(*fn*, *data_type*, *station_idx=1*, *alt_names=()*, ***kwds*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data_type': FieldInfo(annotation=Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], required=False, default=''), 'read_from_netcdf': FieldInfo(annotation=ReadFromNetCDF, required=True, alias='ReadFromNetCDF', alias_priority=2), 'units': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

read_from_netcdf: *ReadFromNetCDF*

units: str

class ravenpy.config.commands.**EnsembleMode**(*, *mode: Literal['ENSEMBLE_ENKF'] = 'ENSEMBLE_ENKF'*, *n: int*)

mode: Literal['ENSEMBLE_ENKF']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'mode': FieldInfo(annotation=Literal['ENSEMBLE_ENKF'], required=False, default='ENSEMBLE_ENKF'), 'n': FieldInfo(annotation=int, required=True, description='Number of members')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

n: int

class ravenpy.config.commands.**EvaluationPeriod**(*, *name: str*, *start: date*, *end: date*)

:EvaluationPeriod [period_name] [start yyyy-mm-dd] [end yyyy-mm-dd]

end: date

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'end': FieldInfo(annotation=date, required=True), 'name': FieldInfo(annotation=str, required=True), 'start': FieldInfo(annotation=date, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

start: date

class ravenpy.config.commands.**ForcingPerturbation**(*, *forcing: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR']*, *dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']*, *p1: float*, *p2: float*, *adj: Literal['ADDITIVE', 'MULTIPLICATIVE']*, *hru_grp: str = ''*)

adj: Literal['ADDITIVE', 'MULTIPLICATIVE']

dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']

forcing: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR']

hru_grp: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'adj': FieldInfo(annotation=Literal['ADDITIVE', 'MULTIPLICATIVE'], required=True), 'dist': FieldInfo(annotation=Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA'], required=True), 'forcing': FieldInfo(annotation=Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], required=True), 'hru_grp': FieldInfo(annotation=str, required=False, default=''), 'p1': FieldInfo(annotation=float, required=True), 'p2': FieldInfo(annotation=float, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

p1: float

p2: float

class ravenpy.config.commands.**Gauge**(*, *name: str = 'default'*, *Latitude: float*, *Longitude: float*, *Elevation: float | None = None*, *RainCorrection: Variable | Expression | float | None = None*, *SnowCorrection: Variable | Expression | float | None = None*, *MonthlyAveEvaporation: Sequence | None = None*, *MonthlyAveTemperature: Sequence | None = None*, *MonthlyMinTemperature: Sequence | None = None*, *MonthlyMaxTemperature: Sequence | None = None*, *Data: Sequence[Data] | None = None*)

classmethod confirm_monthly(*v*)

data: Sequence[Data] | None

property ds: Dataset

Return xarray Dataset with forcing variables keyed by Raven forcing names.

elevation: float | None

classmethod from_nc(*fn: str | Path | Sequence[Path]*, *data_type: Sequence[str] | None = None*, *station_idx: int = 1*, *alt_names: Dict[str, str] | None = None*, *mon_ave: bool = False*, *data_kwds: Dict[str, Any] | None = None*, *engine: str = 'h5netcdf'*, ***kwds*) → *Gauge*

Return Gauge instance with configuration options inferred from the netCDF itself.

Parameters

- **fn** (*str or Path or Sequence[Path]*) – NetCDF file path or paths.
- **data_type** (*Sequence[str], optional*) – Raven data types to extract from netCDF files, e.g. 'PRECIP', 'AVE_TEMP'. The algorithm tries to find all forcings in each file until one is found, then it stops searching for it in the following files.
- **station_idx** (*int*) – Index along station dimension. Starts at 1. Should be the same for all netCDF files.

- **alt_names** (*dict*) – Alternative variable names keyed by data type. Use this if variables do not correspond to CF standard defaults.
- **mon_ave** (*bool*) – If True, compute the monthly average.
- **data_kwds** (*dict[options.Forcings, dict[str, str]]*) – Additional *:Data* parameters keyed by forcing type and station id. Overrides inferred parameters. Use keyword "ALL" to pass parameters to all variables.
- **engine** (*{"h5netcdf", "netcdf4", "pydap"}*) – The engine used to open the dataset. Default is 'h5netcdf'.
- ****kwds** – Additional arguments for Gauge.

Returns
Gauge instance.

Return type
Gauge

latitude: float

longitude: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.Data], NoneType], required=False, default=None, alias='Data', alias_priority=2), 'elevation': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='Elevation', alias_priority=2), 'latitude': FieldInfo(annotation=float, required=True, alias='Latitude', alias_priority=2), 'longitude': FieldInfo(annotation=float, required=True, alias='Longitude', alias_priority=2), 'monthly_ave_evaporation': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyAveEvaporation', alias_priority=2), 'monthly_ave_temperature': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyAveTemperature', alias_priority=2), 'monthly_max_temperature': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyMaxTemperature', alias_priority=2), 'monthly_min_temperature': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyMinTemperature', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default='default'), 'rain_correction': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=None, alias='RainCorrection', alias_priority=2, description='Rain correction'), 'snow_correction': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=None, alias='SnowCorrection', alias_priority=2, description='Snow correction')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

monthly_ave_evaporation: Sequence | None

monthly_ave_temperature: Sequence | None

monthly_max_temperature: Sequence | None

monthly_min_temperature: Sequence | None

name: str

rain_correction: Variable | Expression | float | None

snow_correction: Variable | Expression | float | None

class ravenpy.config.commands.**GridWeights**(*, *NumberHRUs: int = 1*, *NumberGridCells: int = 1*, *data: Sequence[GWRecord] = (GWRecord(root=(1, 0, 1.0)),)*)

GridWeights command.

Notes

command can be embedded in both a *GriddedForcing* or a *StationForcing*.

The default is to have a single cell that covers an entire single HRU, with a weight of 1.

class GWRecord(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Tuple[int, int, float], required=False, default=(1, 0, 1.0))}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Tuple[int, int, float]

data: Sequence[*GWRecord*]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Sequence[ravenpy.config.commands.GridWeights.GWRecord], required=False, default=(GWRecord(root=(1, 0, 1.0)),)), 'number_grid_cells': FieldInfo(annotation=int, required=False, default=1, alias='NumberGridCells', alias_priority=2), 'number_hrus': FieldInfo(annotation=int, required=False, default=1, alias='NumberHRUs', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

number_grid_cells: int

number_hrus: int

classmethod parse(*s*)

class ravenpy.config.commands.**GriddedForcing**(*, *FileNameNC: Url | Path*, *VarNameNC: str*, *DimNamesNC: Sequence[str]*, *station_idx: int | None = None*, *TimeShift: float | None = None*, *LinearTransform: LinearTransform | None = None*, *Deaccumulate: bool | None = None*, *LatitudeVarNameNC: str | None = None*, *LongitudeVarNameNC: str | None = None*, *ElevationVarNameNC: str | None = None*, *name: str = ''*, *ForcingType: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] | None = None*, *GridWeights: GridWeights | RedirectToFile = GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),))*)

GriddedForcing command (RVT).

classmethod check_dims(*v*)

deaccumulate: bool | None

dim_names_nc: Sequence[str]

elevation_var_name_nc: str | None

file_name_nc: HttpUrl | Path

forcing_type: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] | None

grid_weights: *GridWeights* | *RedirectToFile*

latitude_var_name_nc: str | None

linear_transform: *LinearTransform* | None

longitude_var_name_nc: str | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'deaccumulate': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None, alias='Deaccumulate', alias_priority=2), 'dim_names_nc': FieldInfo(annotation=Sequence[str], required=True, alias='DimNamesNC', alias_priority=2), 'elevation_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='ElevationVarNameNC', alias_priority=2), 'file_name_nc': FieldInfo(annotation=Union[Annotated[pydantic_core._pydantic_core.Url, UrlConstraints(max_length=2083, allowed_schemes=['http', 'https'], host_required=None, default_host=None, default_port=None, default_path=None)], Path], required=True, alias='FileNameNC', alias_priority=2, description='NetCDF file name.'), 'forcing_type': FieldInfo(annotation=Union[Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], NoneType], required=False, default=None, alias='ForcingType', alias_priority=2), 'grid_weights': FieldInfo(annotation=Union[GridWeights, RedirectToFile], required=False, default=GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),)), alias='GridWeights', alias_priority=2), 'latitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LatitudeVarNameNC', alias_priority=2), 'linear_transform': FieldInfo(annotation=Union[LinearTransform, NoneType], required=False, default=None, alias='LinearTransform', alias_priority=2), 'longitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LongitudeVarNameNC', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default=''), 'station_idx': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'time_shift': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='TimeShift', alias_priority=2, description='Time stamp shift in days.'), 'var_name_nc': FieldInfo(annotation=str, required=True, alias='VarNameNC', alias_priority=2, description='NetCDF variable name.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

station_idx: int | None

time_shift: float | None

var_name_nc: str

class ravenpy.config.commands.**HRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = '[NONE]'*, *veg_class: str = '[NONE]'*, *soil_profile: str = '[NONE]'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Record to populate :HRUs command internal table (RVH).

aquifer_profile: str

area: Variable | Expression | float | None

aspect: float

elevation: float

hru_id: int

hru_type: str | None

land_use_class: str

latitude: float

longitude: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='[NONE]')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

slope: float

soil_profile: str

subbasin_id: int

terrain_class: str

veg_class: str

class ravenpy.config.commands.**HRUGroup**(*, *name: str*, *groups: _Rec*)

groups: _Rec

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'groups': FieldInfo(annotation=HRUGroup._Rec, required=True), 'name': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

class ravenpy.config.commands.**HRUState**(*, *hru_id: int = 1*, *data: Dict[str, Variable | Expression | float | None] = None*)

data: Dict[str, Variable | Expression | float | None]

hru_id: int

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Dict[str, Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=False, default_factory=dict), 'hru_id': FieldInfo(annotation=int, required=False, default=1)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod parse(*sol*, *names=None*)

class ravenpy.config.commands.**HRUStateVariableTable**(*root: RootModelRootType = PydanticUndefined*)

Table of HRU state variables.

If the HRUState include different attributes, the states will be modified to include all attributes.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.HRUState], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

classmethod parse(*sol: str*)

root: Sequence[*HRUState*]

set_attributes()

class ravenpy.config.commands.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

HRUs command (RVH).

classmethod ignore_unrecognized_hrus(*values*)

Ignore HRUs with unrecognized hru_type.

HRUs are ignored only if all allowed HRU classes define *hru_type*, and if the values passed include it.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.HRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*HRU*]

ravenpy.config.commands.**LU**

alias of *LandUseClass*

class ravenpy.config.commands.**LandUseClass**(*, *name: str = ''*, *impermeable_frac: Variable | Expression | float | None = 0.0*, *forest_coverage: Variable | Expression | float | None = 0.0*)

forest_coverage: Variable | Expression | float | None

impermeable_frac: Variable | Expression | float | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'forest_coverage': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'impermeable_frac': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'name': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

class ravenpy.config.commands.**LandUseClasses**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.LandUseClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*LandUseClass*]

class ravenpy.config.commands.**LandUseParameterList**(*, *Parameters: Sequence[Literal['FOREST_COVERAGE', 'IMPERMEABLE_FRAC', 'ROUGHNESS', 'FOREST_SPARSENESS', 'DEP_MAX', 'MAX_DEP_AREA_FRAC', 'DD_MELT_TEMP', 'MELT_FACTOR', 'DD_REFREEZE_TEMP', 'MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_AGGRADATION', 'SNOW_PATCH_LIMIT', 'HBV_MELT_FOR_CORR', 'HBV_MELT_ASP_CORR', 'GLAC_STORAGE_COEFF', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'HBV_GLACIER_AG', 'CC_DECAY_COEFF', 'SCS_CN', 'SCS_IA_FRACTION', 'PARTITION_COEFF', 'MAX_SAT_AREA_FRAC', 'B_EXP', 'ABST_PERCENT', 'DEP_MAX_FLOW', 'DEP_N', 'DEP_SEEP_K', 'DEP_K', 'DEP_THRESHOLD', 'PDMROF_B', 'PONDED_EXP', 'OW_PET_CORR', 'LAKE_PET_CORR', 'LAKE_REL_COEFF', 'FOREST_PET_CORR', 'GAMMA_SCALE', 'GAMMA_SHAPE', 'GAMMA_SCALE2', 'GAMMA_SHAPE2', 'HMETS_RUNOFF_COEFF', 'AET_COEFF', 'GR4J_X4', 'UBC_ICEPT_FACTOR', 'STREAM_FRACTION', 'BF_LOSS_FRACTION']] | None = None*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['FOREST_COVERAGE', 'IMPERMEABLE_FRAC', 'ROUGHNESS', 'FOREST_SPARSENESS', 'DEP_MAX', 'MAX_DEP_AREA_FRAC', 'DD_MELT_TEMP', 'MELT_FACTOR', 'DD_REFREEZE_TEMP', 'MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_AGGRADATION', 'SNOW_PATCH_LIMIT', 'HBV_MELT_FOR_CORR', 'HBV_MELT_ASP_CORR', 'GLAC_STORAGE_COEFF', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'HBV_GLACIER_AG', 'CC_DECAY_COEFF', 'SCS_CN', 'SCS_IA_FRACTION', 'PARTITION_COEFF', 'MAX_SAT_AREA_FRAC', 'B_EXP', 'ABST_PERCENT', 'DEP_MAX_FLOW', 'DEP_N', 'DEP_SEEP_K', 'DEP_K', 'DEP_THRESHOLD', 'PDMROF_B', 'PONDED_EXP', 'OW_PET_CORR', 'LAKE_PET_CORR', 'LAKE_REL_COEFF', 'FOREST_PET_CORR', 'GAMMA_SCALE', 'GAMMA_SHAPE', 'GAMMA_SCALE2', 'GAMMA_SHAPE2', 'HMETS_RUNOFF_COEFF', 'AET_COEFF', 'GR4J_X4', 'UBC_ICEPT_FACTOR', 'STREAM_FRACTION', 'BF_LOSS_FRACTION']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['FOREST_COVERAGE', 'IMPERMEABLE_FRAC', 'ROUGHNESS', 'FOREST_SPARSENESS', 'DEP_MAX', 'MAX_DEP_AREA_FRAC', 'DD_MELT_TEMP', 'MELT_FACTOR', 'DD_REFREEZE_TEMP', 'MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_AGGRADATION', 'SNOW_PATCH_LIMIT', 'HBV_MELT_FOR_CORR', 'HBV_MELT_ASP_CORR', 'GLAC_STORAGE_COEFF', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'HBV_GLACIER_AG', 'CC_DECAY_COEFF', 'SCS_CN', 'SCS_IA_FRACTION', 'PARTITION_COEFF', 'MAX_SAT_AREA_FRAC', 'B_EXP', 'ABST_PERCENT', 'DEP_MAX_FLOW', 'DEP_N', 'DEP_SEEP_K', 'DEP_K', 'DEP_THRESHOLD', 'PDMROF_B', 'PONDED_EXP', 'OW_PET_CORR', 'LAKE_PET_CORR', 'LAKE_REL_COEFF', 'FOREST_PET_CORR', 'GAMMA_SCALE', 'GAMMA_SHAPE', 'GAMMA_SCALE2', 'GAMMA_SHAPE2', 'HMETS_RUNOFF_COEFF', 'AET_COEFF', 'GR4J_X4', 'UBC_ICEPT_FACTOR', 'STREAM_FRACTION', 'BF_LOSS_FRACTION']] | None

pl: Sequence[ParameterList]

units: Sequence[str] | None

class ravenpy.config.commands.**LinearTransform**(*, *scale: float = 1*, *offset: float = 0*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'offset': FieldInfo(annotation=float, required=False, default=0), 'scale': FieldInfo(annotation=float, required=False, default=1)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

offset: float

scale: float

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**ObservationData**(*, *data_type: Literal['HYDROGRAPH'] = 'HYDROGRAPH'*, *units: str = ''*, *ReadFromNetCDF:* ReadFromNetCDF, *uid: str = '1'*)

data_type: Literal['HYDROGRAPH']

classmethod from_nc(*fn*, *station_idx: int = 1*, *alt_names=()*, *engine='h5netcdf'*, ***kwds*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data_type': FieldInfo(annotation=Literal['HYDROGRAPH'], required=False, default='HYDROGRAPH'), 'read_from_netcdf': FieldInfo(annotation=ReadFromNetCDF, required=True, alias='ReadFromNetCDF', alias_priority=2), 'uid': FieldInfo(annotation=str, required=False, default='1'), 'units': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

read_from_netcdf: *ReadFromNetCDF*

uid: str

units: str

class ravenpy.config.commands.**ObservationErrorModel**(*, *state: Literal['STREAMFLOW']*, *dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']*, *p1: float*, *p2: float*, *adj: Literal['ADDITIVE', 'MULTIPLICATIVE']*)

adj: Literal['ADDITIVE', 'MULTIPLICATIVE']

dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'adj': FieldInfo(annotation=Literal['ADDITIVE', 'MULTIPLICATIVE'], required=True), 'dist': FieldInfo(annotation=Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA'], required=True), 'p1': FieldInfo(annotation=float, required=True), 'p2': FieldInfo(annotation=float, required=True), 'state': FieldInfo(annotation=Literal['STREAMFLOW'], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

p1: float

p2: float

state: Literal['STREAMFLOW']

class ravenpy.config.commands.**Process**(*, *algo: str = 'RAVEN_DEFAULT'*, *source: str | None = None*, *to: Sequence[str] = ()*)

Process type embedded in HydrologicProcesses command.

See processes.py for list of processes.

algo: str

classmethod is_list(*v*)

classmethod is_source_state_variable(*v: str*)

classmethod is_to_state_variable(*v: Sequence*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=str, required=False, default='RAVEN_DEFAULT'), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

source: str | None

to: Sequence[str]

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**RainSnowTransition**(*, *temp: Variable | Expression | float | None*, *delta: Variable | Expression | float | None*)

Specify the range of temperatures over which there will be a rain/snow mix when partitioning total precipitation into rain and snow components.

delta: Variable | Expression | float | None

Range [C].

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'delta': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True), 'temp': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

temp: Variable | Expression | float | None

Midpoint of the temperature range [C].

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**ReadFromNetCDF**(*, *FileNameNC: Url | Path*, *VarNameNC: str*, *DimNamesNC: Sequence[str]*, *StationIdx: int = 1*, *TimeShift: float | None = None*, *LinearTransform: LinearTransform | None = None*, *Deaccumulate: bool | None = None*, *LatitudeVarNameNC: str | None = None*, *LongitudeVarNameNC: str | None = None*, *ElevationVarNameNC: str | None = None*)

property da: DataArray

Return DataArray from configuration.

deaccumulate: bool | None

dim_names_nc: Sequence[str]

elevation_var_name_nc: str | None

file_name_nc: Url | Path

classmethod from_nc(*fn*, *data_type*, *station_idx=1*, *alt_names=()*, *engine='h5netcdf'*, ***kwds*)

Instantiate class from netCDF dataset.

latitude_var_name_nc: `str | None`

linear_transform: *LinearTransform* `| None`

longitude_var_name_nc: `str | None`

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'deaccumulate': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None, alias='Deaccumulate', alias_priority=2), 'dim_names_nc': FieldInfo(annotation=Sequence[str], required=True, alias='DimNamesNC', alias_priority=2), 'elevation_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='ElevationVarNameNC', alias_priority=2), 'file_name_nc': FieldInfo(annotation=Union[Annotated[pydantic_core._pydantic_core.Url, UrlConstraints(max_length=2083, allowed_schemes=['http', 'https'], host_required=None, default_host=None, default_port=None, default_path=None)], Path], required=True, alias='FileNameNC', alias_priority=2, description='NetCDF file name.'), 'latitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LatitudeVarNameNC', alias_priority=2), 'linear_transform': FieldInfo(annotation=Union[LinearTransform, NoneType], required=False, default=None, alias='LinearTransform', alias_priority=2), 'longitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LongitudeVarNameNC', alias_priority=2), 'station_idx': FieldInfo(annotation=int, required=False, default=1, alias='StationIdx', alias_priority=2, description='NetCDF index along station dimension. Starts at 1.'), 'time_shift': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='TimeShift', alias_priority=2, description='Time stamp shift in days.'), 'var_name_nc': FieldInfo(annotation=str, required=True, alias='VarNameNC', alias_priority=2, description='NetCDF variable name.')}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod reorder_time(*v*)

TODO: Return dimensions as x, y, t. Currently only puts time at the end.

station_idx: `int`

time_shift: `float | None`

var_name_nc: `str`

class ravenpy.config.commands.**RedirectToFile**(*root: RootModelRootType = PydanticUndefined*)

RedirectToFile command (RVT).

Notes

For the moment, this command can only be used in the context of a *GriddedForcingCommand* or a *StationForcingCommand*, as a *grid_weights* field replacement when inlining is not desired.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Path, required=True, metadata=[PathType(path_type='file')])}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Path

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**Reservoir**(*, *name: str = 'Lake_XXX'*, *SubBasinID: int = 0*, *HRUID: int = 0*, *Type: str = 'RESROUTE_STANDARD'*, *WeirCoefficient: float = 0*, *CrestWidth: float = 0*, *MaxDepth: float = 0*, *LakeArea: float = 0*)

Reservoir command (RVH).

crest_width: float

hru_id: int

lake_area: float

max_depth: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'crest_width': FieldInfo(annotation=float, required=False, default=0, alias='CrestWidth', alias_priority=2), 'hru_id': FieldInfo(annotation=int, required=False, default=0, alias='HRUID', alias_priority=2), 'lake_area': FieldInfo(annotation=float, required=False, default=0, alias='LakeArea', alias_priority=2, description='Lake area in m2'), 'max_depth': FieldInfo(annotation=float, required=False, default=0, alias='MaxDepth', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default='Lake_XXX'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=0, alias='SubBasinID', alias_priority=2), 'type': FieldInfo(annotation=str, required=False, default='RESROUTE_STANDARD', alias='Type', alias_priority=2), 'weir_coefficient': FieldInfo(annotation=float, required=False, default=0, alias='WeirCoefficient', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

subbasin_id: int

type: str

weir_coefficient: float

ravenpy.config.commands.**SB**

alias of *SubBasin*

class ravenpy.config.commands.**SBGroupPropertyMultiplier**(*, *group_name: str*, *parameter_name: str*, *mult: float*)

group_name: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'group_name': FieldInfo(annotation=str, required=True), 'mult': FieldInfo(annotation=float, required=True), 'parameter_name': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

mult: float

parameter_name: str

ravenpy.config.commands.**SC**

alias of *SoilClass*

ravenpy.config.commands.**SP**

alias of *SoilProfile*

class ravenpy.config.commands.**SeasonalRelativeHeight**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands._MonthlyRecord], required=False, default=(<ravenpy.config.commands._MonthlyRecord object>,))}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[_MonthlyRecord]

class ravenpy.config.commands.**SeasonalRelativeLAI**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands._MonthlyRecord], required=False, default=(<ravenpy.config.commands._MonthlyRecord object>,))}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[_MonthlyRecord]

class ravenpy.config.commands.**SoilClasses**(*root: RootModelRootType = PydanticUndefined*)

class **SoilClass**(**, name: str, mineral: Tuple[float, float, float] | None = None, organic: float | None = None*)

mineral: Tuple[float, float, float] | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'mineral': FieldInfo(annotation=Union[Tuple[float, float, float], NoneType], required=False, default=None), 'name': FieldInfo(annotation=str, required=True), 'organic': FieldInfo(annotation=Union[float, NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

organic: float | None

classmethod validate_mineral(*v*)

Assert sum of mineral fraction is 1.

classmethod validate_mineral_pct(*v: Tuple*)

classmethod validate_organic_pct(*v: float*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.SoilClasses.SoilClass], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*SoilClass*]

class ravenpy.config.commands.**SoilModel**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=int, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: int

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**SoilParameterList**(*, *Parameters: Sequence[Literal['SAND_CON', 'CLAY_CON', 'SILT_CON', 'ORG_CON', 'POROSITY', 'STONE_FRAC', 'SAT_WILT', 'FIELD_CAPACITY', 'BULK_DENSITY', 'HYDRAUL_COND', 'CLAPP_B', 'CLAPP N,CLAPP M', 'SAT_RES', 'AIR_ENTRY_PRESSURE', 'WILTING_PRESSURE', 'HEAT_CAPACITY', 'THERMAL_COND', 'WETTING_FRONT_PSI', 'EVAP_RES_FC', 'SHUTTLEWORTH_B', 'ALBEDO_WET', 'ALBEDO_DRY', 'VIZ_ZMIN', 'VIC_ZMAX', 'VIC ALPHA', 'VIC_EVAP_GAMMA', 'MAX_PERC_RATE', 'PERC_N', 'PERC_COEFF', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'UNAVAIL_FRAC', 'HBV_BETA', 'MAX_BASEFLOW_RATE', 'BASEFLOW_N', 'BASEFLOW_COEFF', 'BASEFLOW_COEFF2', 'BASEFLOW_THRESH', 'BF_LOSS_FRACTION', 'STORAGE_THRESHOLD', 'MAX_CAP_RISE_RATE', 'MAX_INTERFLOW_RATE', 'INTERFLOW_COEF', 'UBC_EVAL_SOIL_DEF', 'UBC_INFIL_SOIL_DEF', 'GR4J_X2', 'GR4J_X3', 'B_EXP', 'PET_CORRECTION']] | None = None*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['SAND_CON', 'CLAY_CON', 'SILT_CON', 'ORG_CON', 'POROSITY', 'STONE_FRAC', 'SAT_WILT', 'FIELD_CAPACITY', 'BULK_DENSITY', 'HYDRAUL_COND', 'CLAPP_B', 'CLAPP N,CLAPP M', 'SAT_RES', 'AIR_ENTRY_PRESSURE', 'WILTING_PRESSURE', 'HEAT_CAPACITY', 'THERMAL_COND', 'WETTING_FRONT_PSI', 'EVAP_RES_FC', 'SHUTTLEWORTH_B', 'ALBEDO_WET', 'ALBEDO_DRY', 'VIZ_ZMIN', 'VIC_ZMAX', 'VIC ALPHA', 'VIC_EVAP_GAMMA', 'MAX_PERC_RATE', 'PERC_N', 'PERC_COEFF', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'UNAVAIL_FRAC', 'HBV_BETA', 'MAX_BASEFLOW_RATE', 'BASEFLOW_N', 'BASEFLOW_COEFF', 'BASEFLOW_COEFF2', 'BASEFLOW_THRESH', 'BF_LOSS_FRACTION', 'STORAGE_THRESHOLD', 'MAX_CAP_RISE_RATE', 'MAX_INTERFLOW_RATE', 'INTERFLOW_COEF', 'UBC_EVAL_SOIL_DEF', 'UBC_INFIL_SOIL_DEF', 'GR4J_X2', 'GR4J_X3', 'B_EXP', 'PET_CORRECTION']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['SAND_CON', 'CLAY_CON', 'SILT_CON', 'ORG_CON', 'POROSITY', 'STONE_FRAC', 'SAT_WILT', 'FIELD_CAPACITY', 'BULK_DENSITY', 'HYDRAUL_COND', 'CLAPP_B', 'CLAPP N,CLAPP M', 'SAT_RES', 'AIR_ENTRY_PRESSURE', 'WILTING_PRESSURE', 'HEAT_CAPACITY', 'THERMAL_COND', 'WETTING_FRONT_PSI', 'EVAP_RES_FC', 'SHUTTLEWORTH_B', 'ALBEDO_WET', 'ALBEDO_DRY', 'VIZ_ZMIN', 'VIC_ZMAX', 'VIC ALPHA', 'VIC_EVAP_GAMMA', 'MAX_PERC_RATE', 'PERC_N', 'PERC_COEFF', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'UNAVAIL_FRAC', 'HBV_BETA', 'MAX_BASEFLOW_RATE', 'BASEFLOW_N', 'BASEFLOW_COEFF', 'BASEFLOW_COEFF2', 'BASEFLOW_THRESH', 'BF_LOSS_FRACTION', 'STORAGE_THRESHOLD', 'MAX_CAP_RISE_RATE', 'MAX_INTERFLOW_RATE', 'INTERFLOW_COEF', 'UBC_EVAL_SOIL_DEF', 'UBC_INFIL_SOIL_DEF', 'GR4J_X2', 'GR4J_X3', 'B_EXP', 'PET_CORRECTION']] | None

pl: Sequence[ParameterList]

units: Sequence[str] | None

class ravenpy.config.commands.**SoilProfile**(*, *name: str = ''*, *soil_classes: Sequence[str] = ()*, *thicknesses: Sequence[Variable | Expression | float | None] = ()*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'name': FieldInfo(annotation=str, required=False, default=''), 'soil_classes': FieldInfo(annotation=Sequence[str], required=False, default=()), 'thicknesses': FieldInfo(annotation=Sequence[Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

soil_classes: Sequence[str]

thicknesses: Sequence[Variable | Expression | float | None]

class ravenpy.config.commands.**SoilProfiles**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.SoilProfile], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[*SoilProfile*]

class ravenpy.config.commands.**StationForcing**(*, *FileNameNC: Url | Path*, *VarNameNC: str*, *DimNamesNC: Sequence[str]*, *station_idx: int | None = None*, *TimeShift: float | None = None*, *LinearTransform: LinearTransform | None = None*, *Deaccumulate: bool | None = None*, *LatitudeVarNameNC: str | None = None*, *LongitudeVarNameNC: str | None = None*, *ElevationVarNameNC: str | None = None*, *name: str = ''*, *ForcingType: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] | None = None*, *GridWeights: GridWeights | RedirectToFile = GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),))*)

StationForcing command (RVT).

classmethod check_dims(*v*)

deaccumulate: bool | None

dim_names_nc: Sequence[str]

elevation_var_name_nc: str | None

file_name_nc: HttpUrl | Path

forcing_type: options.Forcings | None

classmethod from_nc(*fn*, *data_type*, *alt_names=()*, ***kwds*)

Instantiate class from netCDF dataset.

grid_weights: *GridWeights* | *RedirectToFile*

latitude_var_name_nc: str | None

linear_transform: *LinearTransform* | None

longitude_var_name_nc: str | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'deaccumulate': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None, alias='Deaccumulate', alias_priority=2), 'dim_names_nc': FieldInfo(annotation=Sequence[str], required=True, alias='DimNamesNC', alias_priority=2), 'elevation_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='ElevationVarNameNC', alias_priority=2), 'file_name_nc': FieldInfo(annotation=Union[Annotated[pydantic_core._pydantic_core.Url, UrlConstraints(max_length=2083, allowed_schemes=['http', 'https'], host_required=None, default_host=None, default_port=None, default_path=None)], Path], required=True, alias='FileNameNC', alias_priority=2, description='NetCDF file name.'), 'forcing_type': FieldInfo(annotation=Union[Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], NoneType], required=False, default=None, alias='ForcingType', alias_priority=2), 'grid_weights': FieldInfo(annotation=Union[GridWeights, RedirectToFile], required=False, default=GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),)), alias='GridWeights', alias_priority=2), 'latitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LatitudeVarNameNC', alias_priority=2), 'linear_transform': FieldInfo(annotation=Union[LinearTransform, NoneType], required=False, default=None, alias='LinearTransform', alias_priority=2), 'longitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LongitudeVarNameNC', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default=''), 'station_idx': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'time_shift': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='TimeShift', alias_priority=2, description='Time stamp shift in days.'), 'var_name_nc': FieldInfo(annotation=str, required=True, alias='VarNameNC', alias_priority=2, description='NetCDF variable name.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

station_idx: int | None

time_shift: float | None

var_name_nc: str

class ravenpy.config.commands.**SubBasin**(*, *subbasin_id: int = 1*, *name: str = 'sub_001'*, *downstream_id: int = -1*, *profile: str = 'NONE'*, *reach_length: float = 0*, *gauged: bool = True*, *gauge_id: str | None = ''*)

Record to populate RVH :SubBasins command internal table.

downstream_id: int

gauge_id: str | None

gauged: bool

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'downstream_id': FieldInfo(annotation=int, required=False, default=-1), 'gauge_id': FieldInfo(annotation=Union[str, NoneType], required=False, default=''), 'gauged': FieldInfo(annotation=bool, required=False, default=True), 'name': FieldInfo(annotation=str, required=False, default='sub_001'), 'profile': FieldInfo(annotation=str, required=False, default='NONE'), 'reach_length': FieldInfo(annotation=float, required=False, default=0), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)])}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

profile: str

reach_length: float

subbasin_id: int

class ravenpy.config.commands.**SubBasinGroup**(*, *name: str = ''*, *sb_ids: Sequence[int] = ()*)

SubBasinGroup command (RVH).

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'name': FieldInfo(annotation=str, required=False, default=''), 'sb_ids': FieldInfo(annotation=Sequence[int], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

sb_ids: Sequence[int]

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**SubBasinProperties**(*, *Parameters: Sequence[Literal['TIME_TO_PEAK', 'TIME_CONC', 'TIME_LAG', 'NUM_RESERVOIRS', 'RES_CONSTANT', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'Q_REFERENCE', 'MANNINGS_N', 'SLOPE', 'DIFFUSIVITY', 'CELERITY', 'RAIN_CORR', 'SNOW_CORR']] | None = None*, *records: Sequence[SubBasinProperty] | None = None*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['TIME_TO_PEAK', 'TIME_CONC', 'TIME_LAG', 'NUM_RESERVOIRS', 'RES_CONSTANT', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'Q_REFERENCE', 'MANNINGS_N', 'SLOPE', 'DIFFUSIVITY', 'CELERITY', 'RAIN_CORR', 'SNOW_CORR']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'records': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.SubBasinProperty], NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['TIME_TO_PEAK', 'TIME_CONC', 'TIME_LAG', 'NUM_RESERVOIRS', 'RES_CONSTANT', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'Q_REFERENCE', 'MANNINGS_N', 'SLOPE', 'DIFFUSIVITY', 'CELERITY', 'RAIN_CORR', 'SNOW_CORR']] | None

records: Sequence[*SubBasinProperty*] | None

class ravenpy.config.commands.**SubBasinProperty**(*, *sb_id: str*, *values: Sequence[Variable | Expression | float | None]*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'sb_id': FieldInfo(annotation=str, required=True), 'values': FieldInfo(annotation=Sequence[Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

sb_id: str

values: Sequence[Variable | Expression | float | None]

class ravenpy.config.commands.**SubBasins**(*root: RootModelRootType = PydanticUndefined*)

SubBasins command (RVH).

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.SubBasin], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*SubBasin*]

ravenpy.config.commands.**TC**

alias of *TerrainClass*

class ravenpy.config.commands.**TerrainClass**(**, name: str, hillslope_length: Variable | Expression | float | None, drainage_density: Variable | Expression | float | None, topmodel_lambda: Variable | Expression | float | None = None*)

drainage_density: Variable | Expression | float | None

hillslope_length: Variable | Expression | float | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'drainage_density': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True), 'hillslope_length': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True), 'name': FieldInfo(annotation=str, required=True), 'topmodel_lambda': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=None)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: `str`

topmodel_lambda: `Variable | Expression | float | None`

class ravenpy.config.commands.**TerrainClasses**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.TerrainClass], required=True)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: `Sequence[TerrainClass]`

ravenpy.config.commands.**VC**

alias of *VegetationClass*

class ravenpy.config.commands.**VegetationClass**(*, *name: str = ''*, *max_ht: Variable | Expression | float | None = 0.0*, *max_lai: Variable | Expression | float | None = 0.0*, *max_leaf_cond: Variable | Expression | float | None = 0.0*)

max_ht: `Variable | Expression | float | None`

max_lai: `Variable | Expression | float | None`

max_leaf_cond: `Variable | Expression | float | None`

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'max_ht': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'max_lai': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'max_leaf_cond': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'name': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

class ravenpy.config.commands.**VegetationClasses**(*root: RootModelRootType = PydanticUndefined*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.VegetationClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*VegetationClass*]

class ravenpy.config.commands.**VegetationParameterList**(*, *Parameters: Sequence[Literal['MAX_HEIGHT', 'MAX_LEAF_COND', 'MAX_LAI', 'SVF_EXTINCTION', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT', 'RAIN_ICEPT_FACT', 'SNOW_ICEPT_FACT', 'SAI_HT_RATIO', 'TRUNK_FRACTION', 'STEMFLOW_FRAC', 'ALBEDO', 'ALBEDO_WET', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'ROOT_EXTINCT', 'MAX_ROOT_LENGTH', 'MIN_RESISTIVITY', 'XYLEM_FRAC', 'ROOTRADIUS', 'PSI_CRITICAL', 'DRIP_PROPORTION', 'MAX_INTERCEPT_RATE', 'CHU_MATURITY', 'VEG_DIAM', 'VEG_MBETA', 'VEG_DENSPET_VEG_CORR', 'TFRAIN', 'TFSNOW', 'RELATIVE_HT', 'RELATIVE_LAI', 'CAP_LAI_RATIO', 'SNOCAP_LAI_RATIO']] | None = None*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['MAX_HEIGHT', 'MAX_LEAF_COND', 'MAX_LAI', 'SVF_EXTINCTION', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT', 'RAIN_ICEPT_FACT', 'SNOW_ICEPT_FACT', 'SAI_HT_RATIO', 'TRUNK_FRACTION', 'STEMFLOW_FRAC', 'ALBEDO', 'ALBEDO_WET', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'ROOT_EXTINCT', 'MAX_ROOT_LENGTH', 'MIN_RESISTIVITY', 'XYLEM_FRAC', 'ROOTRADIUS', 'PSI_CRITICAL', 'DRIP_PROPORTION', 'MAX_INTERCEPT_RATE', 'CHU_MATURITY', 'VEG_DIAM', 'VEG_MBETA', 'VEG_DENSPET_VEG_CORR', 'TFRAIN', 'TFSNOW', 'RELATIVE_HT', 'RELATIVE_LAI', 'CAP_LAI_RATIO', 'SNOCAP_LAI_RATIO']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

```
parameters: Sequence[Literal['MAX_HEIGHT', 'MAX_LEAF_COND', 'MAX_LAI',
'SVF_EXTINCTION', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT', 'RAIN_ICEPT_FACT',
'SNOW_ICEPT_FACT', 'SAI_HT_RATIO', 'TRUNK_FRACTION', 'STEMFLOW_FRAC', 'ALBEDO',
'ALBEDO_WET', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'ROOT_EXTINCT',
'MAX_ROOT_LENGTH', 'MIN_RESISTIVITY', 'XYLEM_FRAC', 'ROOTRADIUS', 'PSI_CRITICAL',
'DRIP_PROPORTION', 'MAX_INTERCEPT_RATE', 'CHU_MATURITY', 'VEG_DIAM', 'VEG_MBETA',
'VEG_DENSPET_VEG_CORR', 'TFRAIN', 'TFSNOW', 'RELATIVE_HT', 'RELATIVE_LAI',
'CAP_LAI_RATIO', 'SNOCAP_LAI_RATIO']] | None

pl: Sequence[ParameterList]

units: Sequence[str] | None
```

class ravenpy.config.rvs.**Config**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: Any = None*, *SoilClasses: SoilClasses | None = None*, *SoilProfiles: SoilProfiles | None = None*, *VegetationClasses: VegetationClasses | None = None*, *LandUseClasses: LandUseClasses | None = None*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList | None = None*, *LandUseParameterList: LandUseParameterList | None = None*, *VegetationParameterList: VegetationParameterList | None = None*, *ChannelProfile: Sequence[ChannelProfile] | None = None*, *GlobalParameter: Dict[str, Variable | Expression | float | None] | None = {}*, *RainSnowTransition: RainSnowTransition | None = None*, *SeasonalRelativeLAI: SeasonalRelativeLAI | None = None*, *SeasonalRelativeHeight: SeasonalRelativeHeight | None = None*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*, *SubBasins: SubBasins | None = None*, *SubBasinGroup: Sequence[SubBasinGroup] | None = None*, *SubBasinProperties: SubBasinProperties | None = None*, *SBGroupPropertyMultiplier: Sequence[SBGroupPropertyMultiplier] | None = None*, *HRUs: HRUs | None = None*, *HRUGroup: Sequence[HRUGroup] | None = None*, *Reservoirs: Sequence[Reservoir] | None = None*, *HRUStateVariableTable: HRUStateVariableTable | None = None*, *BasinStateVariables: BasinStateVariables | None = None*, *UniformInitialConditions: Dict[str, Variable | Expression | float | None] | None = None*, *SilentMode: bool | None = None*, *NoisyMode: bool | None = None*, *RunName: str | None = None*, *Calendar: Calendar | None = None*, *StartDate: date | datetime | datetime | None = None*, *AssimilationStartTime: date | datetime | datetime | None = None*, *EndDate: date | datetime | datetime | None = None*, *Duration: float | None = None*, *TimeStep: float | str | None = None*, *Routing: Routing | None = None*, *CatchmentRoute: CatchmentRoute | None = None*, *Evaporation: Evaporation | None = None*, *OW_Evaporation: Evaporation | None = None*, *SWRadiationMethod: SWRadiationMethod | None = None*, *SWCloudCorrect: SWCloudCorrect | None = None*, *SWCanopyCorrect: SWCanopyCorrect | None = None*, *LWRadiationMethod: LWRadiationMethod | None = None*, *WindspeedMethod: WindspeedMethod | None = None*, *RainSnowFraction: RainSnowFraction | None = None*, *PotentialMeltMethod: PotentialMeltMethod | None = None*, *OroTempCorrect: OroTempCorrect | None = None*, *OroPrecipCorrect: OroPrecipCorrect | None = None*, *OroPETCorrect: OroPETCorrect | None = None*, *CloudCoverMethod: CloudCoverMethod | None = None*, *PrecipIceptFract: PrecipIceptFract | None = None*, *SubdailyMethod: SubdailyMethod | None = None*, *MonthlyInterpolationMethod: MonthlyInterpolationMethod | None = None*, *SoilModel: SoilModel | None = None*, *TemperatureCorrection: bool | None = None*, *LakeStorage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH',*

assimilate_streamflow: Sequence[rc.AssimilateStreamflow] | None

assimilated_state: Sequence[rc.AssimilatedState] | None

assimilation_start_time: date | None

basin_state_variables: rc.BasinStateVariables | None

calendar: o.Calendar | None

catchment_route: o.CatchmentRoute | None

channel_profile: Sequence[rc.ChannelProfile] | None

cloud_cover_method: o.CloudCoverMethod | None

custom_output: Sequence[rc.CustomOutput] | None

debug_mode: bool | None

define_hru_groups: Sequence[str] | None

deltares_fews_mode: bool | None

direct_evaporation: bool | None

dont_write_watershed_storage: bool | None

duplicate(***kwds*)

Duplicate this model, changing the values given in the keywords.

duration: float | None

end_date: date | None

enkf_mode: o.EnKFMode | None

ensemble_mode: rc.EnsembleMode | None

evaluation_metrics: Sequence[o.EvaluationMetrics] | None

evaluation_period: Sequence[rc.EvaluationPeriod] | None

evaporation: o.Evaporation | None

extra_rvt_filename: str | None

forcing_perturbation: Sequence[rc.ForcingPerturbation] | None

forecast_rvt_filename: str | None

gauge: Sequence[rc.Gauge] | None

global_parameter: Dict[str, Sym] | None

gridded_forcing: Sequence[rc.GriddedForcing] | None

header(*rv*)

Return the header to print at the top of each RV file.

hru_group: Sequence[rc.HRUGroup] | None

hru_state_variable_table: rc.HRUStateVariableTable | None

hrus: rc.HRUs | None

hydrologic_processes: Sequence[rc.Process | rp.Conditional | rp.ProcessGroup] | None

property is_symbolic

Return True if configuration contains symbolic expressions.

lake_storage: o.StateVariables | None

land_use_classes: rc.LandUseClasses | None

land_use_parameter_list: rc.LandUseParameterList | None

lw_radiation_method: o.LWRadiationMethod | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow],
NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow',
alias_priority=2, validate_default=False), 'assimilated_state':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState],
NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState',
alias_priority=2, validate_default=False), 'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'basin_state_variables':
FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False,
default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Union[Calendar,
NoneType], required=False, default_factory=<lambda>, alias='Calendar',
alias_priority=2, validate_default=False), 'catchment_route':
FieldInfo(annotation=Union[CatchmentRoute, NoneType], required=False,
default_factory=<lambda>, alias='CatchmentRoute', alias_priority=2,
validate_default=False), 'channel_profile':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile],
NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile',
alias_priority=2, validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'enkf_mode':
FieldInfo(annotation=Union[EnKFMode, NoneType], required=False,
default_factory=<lambda>, alias='EnKFMode', alias_priority=2,
validate_default=False), 'ensemble_mode':  FieldInfo(annotation=Union[EnsembleMode,
NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode',
alias_priority=2, validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Union[Evaporation, NoneType], required=False,
default_factory=<lambda>, alias='Evaporation', alias_priority=2,
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

monthly_interpolation_method: o.MonthlyInterpolationMethod | None

netcdf_attribute: Dict[str, str] | None

noisy_mode: bool | None

observation_data: Sequence[rc.ObservationData] | None

observation_error_model: Sequence[rc.ObservationErrorModel] | None

oro_pet_correct: o.OroPETCorrect | None

oro_precip_correct: o.OroPrecipCorrect | None

oro_temp_correct: o.OroTempCorrect | None

output_directory_format: str | Path | None

ow_evaporation: o.Evaporation | None

params: Any

pavics_mode: bool | None

potential_melt_method: o.PotentialMeltMethod | None

precip_icept_frac: o.PrecipIceptFract | None

rain_snow_fraction: o.RainSnowFraction | None

rain_snow_transition: rc.RainSnowTransition | None

reservoirs: Sequence[rc.Reservoir] | None

routing: o.Routing | None

run_name: str | None

property rvc

property rve

property rvh

property rvi

property rvp

property rvt

sb_group_property_multiplier: Sequence[rc.SBGroupPropertyMultiplier] | None

seasonal_relative_height: rc.SeasonalRelativeHeight | None

seasonal_relative_lai: rc.SeasonalRelativeLAI | None

set_params(*params: Dict | Sequence*) → *Config*

Return a new instance of Config with params frozen to their numerical values.

set_solution(*fn: Path*, *timestamp: bool = True*) → *Config*

Return a new instance of Config with hru, basin states and start date set from an existing solution.

Parameters

- **fn** (*Path*) – Path to solution file.
- **timestamp** (*bool*) – If False, ignore time stamp information in the solution. If True, the solution will set StartDate to the solution's timestamp.

Returns
Config with internal state set from the solution file.

Return type
Config

silent_mode: bool | None

soil_classes: rc.SoilClasses | None

soil_model: rc.SoilModel | None

soil_parameter_list: rc.SoilParameterList | None

soil_profiles: rc.SoilProfiles | None

solution_run_name: str | None

start_date: date | None

station_forcing: Sequence[rc.StationForcing] | None

sub_basin_group: Sequence[rc.SubBasinGroup] | None

sub_basin_properties: rc.SubBasinProperties | None

sub_basins: rc.SubBasins | None

subdaily_method: o.SubdailyMethod | None

suppress_output: bool | None

sw_canopy_correct: o.SWCanopyCorrect | None

sw_cloud_correct: o.SWCloudCorrect | None

sw_radiation_method: o.SWRadiationMethod | None

temperature_correction: bool | None

terrain_classes: rc.TerrainClasses | None

time_step: float | str | None

truncate_hindcasts: bool | None

uniform_initial_conditions: Dict[str, Sym] | None

vegetation_classes: rc.VegetationClasses | None

vegetation_parameter_list: `rc.VegetationParameterList | None`

window_size: `int | None`

windspeed_method: `o.WindspeedMethod | None`

write_forcing_functions: `bool | None`

write_local_flows: `bool | None`

write_netcdf_format: `bool | None`

write_rv(*workdir: str | Path*, *modelname=None*, *overwrite=False*, *header=True*)

Write configuration files to disk.

Parameters

- **workdir** (*str, Path*) – A directory where rv files will be written to disk.
- **modelname** (*str*) – File name stem for rv files. If not given, defaults to *RunName* if set, otherwise *raven*.
- **overwrite** (*bool*) – If True, overwrite existing configuration files.
- **header** (*bool*) – If True, write a header at the top of each RV file.

write_subbasin_file: `bool | None`

zip(*workdir: str | Path*, *modelname=None*, *overwrite=False*)

Write configuration to zip file.

Parameters

- **workdir** (*Path, str*) – Path to zip archive storing RV files.
- **modelname** (*str*) – File name stem for rv files. If not given, defaults to *RunName* if set, otherwise *raven*.
- **overwrite** (*bool*) – If True, overwrite existing configuration zip file.

class ravenpy.config.rvs.**RVC**(*, *HRUStateVariableTable:* HRUStateVariableTable | *None* = *None*, *BasinStateVariables:* BasinStateVariables | *None* = *None*, *UniformInitialConditions: Dict[str, Variable | Expression | float | None] | None* = *None*)

basin_state_variables: *BasinStateVariables* | None

hru_state_variable_table: *HRUStateVariableTable* | None

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: *ClassVar[dict[str, FieldInfo]]* = {'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'hru_state_variable_table': FieldInfo(annotation=Union[HRUStateVariableTable, NoneType], required=False, default_factory=<lambda>, alias='HRUStateVariableTable', alias_priority=2, validate_default=False), 'uniform_initial_conditions': FieldInfo(annotation=Union[Dict[str, Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], NoneType], required=False, default_factory=<lambda>, alias='UniformInitialConditions', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

uniform_initial_conditions: *Dict[str, Variable | Expression | float | None] | None*

class ravenpy.config.rvs.**RVE**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*)

assimilate_streamflow: *Sequence[AssimilateStreamflow] | None*

assimilated_state: *Sequence[AssimilatedState] | None*

enkf_mode: *EnKFMode | None*

extra_rvt_filename: *str | None*

forcing_perturbation: *Sequence[ForcingPerturbation] | None*

forecast_rvt_filename: *str | None*

model_computed_fields: *ClassVar[dict[str, ComputedFieldInfo]]* = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: *ClassVar[ConfigDict]* = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'extra_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='ExtraRVTFilename', alias_priority=2, validate_default=False), 'forcing_perturbation': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ForcingPerturbation], NoneType], required=False, default_factory=<lambda>, alias='ForcingPerturbation', alias_priority=2, validate_default=False), 'forecast_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='ForecastRVTFilename', alias_priority=2, validate_default=False), 'observation_error_model': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ObservationErrorModel], NoneType], required=False, default_factory=<lambda>, alias='ObservationErrorModel', alias_priority=2, validate_default=False), 'output_directory_format': FieldInfo(annotation=Union[str, Path, NoneType], required=False, default_factory=<lambda>, alias='OutputDirectoryFormat', alias_priority=2, validate_default=False), 'solution_run_name': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='SolutionRunName', alias_priority=2, validate_default=False), 'truncate_hindcasts': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='TruncateHindcasts', alias_priority=2, validate_default=False), 'window_size': FieldInfo(annotation=Union[int, NoneType], required=False, default_factory=<lambda>, alias='WindowSize', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

observation_error_model: Sequence[*ObservationErrorModel*] | None

output_directory_format: str | Path | None

solution_run_name: str | None

truncate_hindcasts: bool | None

window_size: int | None

class ravenpy.config.rvs.**RVH**(*, *SubBasins:* SubBasins | *None* = *None*, *SubBasinGroup:* *Sequence[*SubBasinGroup*]* | *None* = *None*, *SubBasinProperties:* SubBasinProperties | *None* = *None*, *SBGroupPropertyMultiplier:* *Sequence[*SBGroupPropertyMultiplier*]* | *None* = *None*, *HRUs:* HRUs | *None* = *None*, *HRUGroup: Sequence[*HRUGroup*]* | *None* = *None*, *Reservoirs:* *Sequence[*Reservoir*]* | *None* = *None*)

hru_group: Sequence[*HRUGroup*] | None

hrus: *HRUs* | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'hru_group': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.HRUGroup], NoneType], required=False, default_factory=<lambda>, alias='HRUGroup', alias_priority=2, validate_default=False), 'hrus': FieldInfo(annotation=Union[HRUs, NoneType], required=False, default_factory=<lambda>, alias='HRUs', alias_priority=2, validate_default=False), 'reservoirs': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.Reservoir], NoneType], required=False, default_factory=<lambda>, alias='Reservoirs', alias_priority=2, validate_default=False), 'sb_group_property_multiplier': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.SBGroupPropertyMultiplier], NoneType], required=False, default_factory=<lambda>, alias='SBGroupPropertyMultiplier', alias_priority=2, validate_default=False), 'sub_basin_group': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.SubBasinGroup], NoneType], required=False, default_factory=<lambda>, alias='SubBasinGroup', alias_priority=2, validate_default=False), 'sub_basin_properties': FieldInfo(annotation=Union[SubBasinProperties, NoneType], required=False, default_factory=<lambda>, alias='SubBasinProperties', alias_priority=2, validate_default=False), 'sub_basins': FieldInfo(annotation=Union[SubBasins, NoneType], required=False, default_factory=<lambda>, alias='SubBasins', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

reservoirs: Sequence[*Reservoir*] | None

sb_group_property_multiplier: Sequence[*SBGroupPropertyMultiplier*] | None

sub_basin_group: Sequence[*SubBasinGroup*] | None

sub_basin_properties: *SubBasinProperties* | None

sub_basins: *SubBasins* | None

class ravenpy.config.rvs.**RVI**(*, *SilentMode: bool | None = None*, *NoisyMode: bool | None = None*, *RunName: str | None = None*, *Calendar: Calendar | None = None*, *StartDate: date | datetime | datetime | None = None*, *AssimilationStartTime: date | datetime | datetime | None = None*, *EndDate: date | datetime | datetime | None = None*, *Duration: float | None = None*, *TimeStep: float | str | None = None*, *Routing: Routing | None = None*, *CatchmentRoute: CatchmentRoute | None = None*, *Evaporation: Evaporation | None = None*, *OW_Evaporation: Evaporation | None = None*, *SWRadiationMethod: SWRadiationMethod | None = None*, *SWCloudCorrect: SWCloudCorrect | None = None*, *SWCanopyCorrect: SWCanopyCorrect | None = None*, *LWRadiationMethod: LWRadiationMethod | None = None*, *WindspeedMethod: WindspeedMethod | None = None*, *RainSnowFraction: RainSnowFraction | None = None*, *PotentialMeltMethod: PotentialMeltMethod | None = None*, *OroTempCorrect: OroTempCorrect | None = None*, *OroPrecipCorrect: OroPrecipCorrect | None = None*, *OroPETCorrect: OroPETCorrect | None = None*, *CloudCoverMethod: CloudCoverMethod | None = None*, *PrecipIceptFract: PrecipIceptFract | None = None*, *SubdailyMethod: SubdailyMethod | None = None*, *MonthlyInterpolationMethod: MonthlyInterpolationMethod | None = None*, *SoilModel: SoilModel | None = None*, *TemperatureCorrection: bool | None = None*, *LakeStorage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | None = None*, *DefineHRUGroups: Sequence[str] | None = None*, *HydrologicProcesses: Sequence[Process | Conditional | ProcessGroup] | None = None*, *EvaluationMetrics: Sequence[EvaluationMetrics] | None = None*, *EvaluationPeriod: Sequence[EvaluationPeriod] | None = None*, *EnsembleMode: EnsembleMode | None = None*, *WriteNetcdfFormat: bool | None = None*, *NetCDFAttribute: Dict[str, str] | None = None*, *CustomOutput: Sequence[CustomOutput] | None = None*, *DirectEvaporation: bool | None = None*, *DeltaresFEWSMode: bool | None = None*, *DebugMode: bool | None = None*, *DontWriteWatershedStorage: bool | None = None*, *PavicsMode: bool | None = None*, *SuppressOutput: bool | None = None*, *WriteForcingFunctions: bool | None = None*, *WriteSubbasinFile: bool | None = None*, *WriteLocalFlows: bool | None = None*)

assimilation_start_time: date | datetime | datetime | None

calendar: Calendar | None

catchment_route: CatchmentRoute | None

cloud_cover_method: CloudCoverMethod | None

custom_output: Sequence[*CustomOutput*] | None

classmethod dates2cf(*v*, *info*)

Convert dates to cftime dates.

debug_mode: bool | None

define_hru_groups: Sequence[str] | None

deltares_fews_mode: bool | None

direct_evaporation: bool | None

dont_write_watershed_storage: bool | None

duration: float | None

end_date: date | datetime | datetime | None

ensemble_mode: *EnsembleMode* | None

evaluation_metrics: Sequence[EvaluationMetrics] | None

evaluation_period: Sequence[*EvaluationPeriod*] | None

evaporation: Evaporation | None

hydrologic_processes: Sequence[*Process* | Conditional | ProcessGroup] | None

classmethod init_soil_model(*v*)

lake_storage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | None

lw_radiation_method: LWRadiationMethod | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Union[Calendar,
NoneType], required=False, default_factory=<lambda>, alias='Calendar',
alias_priority=2, validate_default=False), 'catchment_route':
FieldInfo(annotation=Union[CatchmentRoute, NoneType], required=False,
default_factory=<lambda>, alias='CatchmentRoute', alias_priority=2,
validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'ensemble_mode':
FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False,
default_factory=<lambda>, alias='EnsembleMode', alias_priority=2,
validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Union[Evaporation, NoneType], required=False,
default_factory=<lambda>, alias='Evaporation', alias_priority=2,
validate_default=False), 'hydrologic_processes':
FieldInfo(annotation=Union[Sequence[Union[ravenpy.config.commands.Process,
ravenpy.config.processes.Conditional, ravenpy.config.processes.ProcessGroup]],
NoneType], required=False, default_factory=<lambda>, alias='HydrologicProcesses',
alias_priority=2, validate_default=False), 'lake_storage':
FieldInfo(annotation=Union[Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP',
'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY',
'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW',
'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR',
'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP',
'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE',
'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT',
'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'], NoneType],
required=False, default_factory=<lambda>, alias='LakeStorage', alias_priority=2,
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

monthly_interpolation_method: MonthlyInterpolationMethod | None

netcdf_attribute: Dict[str, str] | None

noisy_mode: bool | None

oro_pet_correct: OroPETCorrect | None

oro_precip_correct: OroPrecipCorrect | None

oro_temp_correct: OroTempCorrect | None

ow_evaporation: Evaporation | None

pavics_mode: bool | None

potential_melt_method: PotentialMeltMethod | None

precip_icept_frac: PrecipIceptFract | None

rain_snow_fraction: RainSnowFraction | None

routing: Routing | None

run_name: str | None

silent_mode: bool | None

soil_model: *SoilModel* | None

start_date: date | datetime | datetime | None

subdaily_method: SubdailyMethod | None

suppress_output: bool | None

sw_canopy_correct: SWCanopyCorrect | None

sw_cloud_correct: SWCloudCorrect | None

sw_radiation_method: SWRadiationMethod | None

temperature_correction: bool | None

time_step: float | str | None

windspeed_method: WindspeedMethod | None

write_forcing_functions: bool | None

write_local_flows: bool | None

write_netcdf_format: bool | None

write_subbasin_file: bool | None

class ravenpy.config.rvs.RVP(*, *params: Any = None*, *SoilClasses:* SoilClasses | *None = None*, *SoilProfiles:* SoilProfiles | *None = None*, *VegetationClasses:* VegetationClasses | *None = None*, *LandUseClasses:* LandUseClasses | *None = None*, *TerrainClasses:* TerrainClasses | *None = None*, *SoilParameterList:* SoilParameterList | *None = None*, *LandUseParameterList:* LandUseParameterList | *None = None*, *VegetationParameterList:* VegetationParameterList | *None = None*, *ChannelProfile: Sequence[*ChannelProfile*]* | *None = None*, *GlobalParameter: Dict[str, Variable* | *Expression* | *float* | *None]* | *None = {}*, *RainSnowTransition:* RainSnowTransition | *None = None*, *SeasonalRelativeLAI:* SeasonalRelativeLAI | *None = None*, *SeasonalRelativeHeight:* SeasonalRelativeHeight | *None = None*)

channel_profile: Sequence[*ChannelProfile*] | None

global_parameter: Dict[str, Variable | Expression | float | None] | None

land_use_classes: *LandUseClasses* | None

land_use_parameter_list: *LandUseParameterList* | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'global_parameter': FieldInfo(annotation=Union[Dict[str, Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], NoneType], required=False, default={}, alias='GlobalParameter', alias_priority=2), 'land_use_classes': FieldInfo(annotation=Union[LandUseClasses, NoneType], required=False, default_factory=<lambda>, alias='LandUseClasses', alias_priority=2, validate_default=False), 'land_use_parameter_list': FieldInfo(annotation=Union[LandUseParameterList, NoneType], required=False, default_factory=<lambda>, alias='LandUseParameterList', alias_priority=2, validate_default=False), 'params': FieldInfo(annotation=Any, required=False, default=None), 'rain_snow_transition': FieldInfo(annotation=Union[RainSnowTransition, NoneType], required=False, default_factory=<lambda>, alias='RainSnowTransition', alias_priority=2, validate_default=False), 'seasonal_relative_height': FieldInfo(annotation=Union[SeasonalRelativeHeight, NoneType], required=False, default_factory=<lambda>, alias='SeasonalRelativeHeight', alias_priority=2, validate_default=False), 'seasonal_relative_lai': FieldInfo(annotation=Union[SeasonalRelativeLAI, NoneType], required=False, default_factory=<lambda>, alias='SeasonalRelativeLAI', alias_priority=2, validate_default=False), 'soil_classes': FieldInfo(annotation=Union[SoilClasses, NoneType], required=False, default_factory=<lambda>, alias='SoilClasses', alias_priority=2, validate_default=False), 'soil_parameter_list': FieldInfo(annotation=Union[SoilParameterList, NoneType], required=False, default_factory=<lambda>, alias='SoilParameterList', alias_priority=2, validate_default=False), 'soil_profiles': FieldInfo(annotation=Union[SoilProfiles, NoneType], required=False, default_factory=<lambda>, alias='SoilProfiles', alias_priority=2, validate_default=False), 'terrain_classes': FieldInfo(annotation=Union[TerrainClasses, NoneType], required=False, default_factory=<lambda>, alias='TerrainClasses', alias_priority=2, validate_default=False), 'vegetation_classes': FieldInfo(annotation=Union[VegetationClasses, NoneType], required=False, default_factory=<lambda>, alias='VegetationClasses', alias_priority=2, validate_default=False), 'vegetation_parameter_list': FieldInfo(annotation=Union[VegetationParameterList, NoneType], required=False, default_factory=<lambda>, alias='VegetationParameterList', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

params: Any

rain_snow_transition: *RainSnowTransition* | None

seasonal_relative_height: *SeasonalRelativeHeight* | None

seasonal_relative_lai: *SeasonalRelativeLAI* | None

soil_classes: *SoilClasses* | None

soil_parameter_list: *SoilParameterList* | None

soil_profiles: *SoilProfiles* | None

terrain_classes: *TerrainClasses* | None

vegetation_classes: *VegetationClasses* | None

vegetation_parameter_list: *VegetationParameterList* | None

class ravenpy.config.rvs.**RVT**(*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*)

gauge: Sequence[*Gauge*] | None

gridded_forcing: Sequence[*GriddedForcing*] | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'gauge': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.Gauge], NoneType], required=False, default_factory=<lambda>, alias='Gauge', alias_priority=2, validate_default=False), 'gridded_forcing': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.GriddedForcing], NoneType], required=False, default_factory=<lambda>, alias='GriddedForcing', alias_priority=2, validate_default=False), 'observation_data': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ObservationData], NoneType], required=False, default_factory=<lambda>, alias='ObservationData', alias_priority=2, validate_default=False), 'station_forcing': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.StationForcing], NoneType], required=False, default_factory=<lambda>, alias='StationForcing', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

observation_data: Sequence[*ObservationData*] | None

station_forcing: Sequence[*StationForcing*] | None

ravenpy.config.rvs.**is_symbolic**(*params: dict*) → bool

Return True if parameters include a symbolic variable.

12.3 Emulators

ravenpy.config.emulators.**get_model**(*name*)

Return the corresponding Raven emulator configuration class.

Parameters
name (*str*) – Model class name or model identifier.

Return type
Raven model configuration class

12.4 Extractors

class ravenpy.extractors.routing_product.**BasinMakerExtractor**(*df*, *hru_aspect_convention='GRASS'*, *routing_product_version='2.1'*)

This is a class to encapsulate the logic of converting the Routing Product into the required data structures to generate the RVH file format.

Parameters

- **df** (*GeoDataFrame*) – Sub-basin information.
- **hru_aspect_convention** (*{"GRASS", "ArcGIS"}*) – How sub-basin aspect is defined.
- **routing_product_version** (*{"2.1", "1.0"}*) – Version of the BasinMaker data.

HRU_ASPECT_CONVENTION = 'GRASS'

MANNING_DEFAULT = 0.035

MAX_RIVER_SLOPE = 1e-05

ROUTING_PRODUCT_VERSION = '2.1'

USE_LAKE_AS_GAUGE = False

USE_LAND_AS_GAUGE = False

USE_MANNING_COEFF = False

WEIR_COEFFICIENT = 0.6

extract(*hru_from_sb: bool = False*) → dict

Extract data from the Routing Product shapefile and return dictionaries that can be parsed into Raven Commands.

Parameters
hru_from_sb (*bool*) – If True, draw HRU information from subbasin information. This is likely to yield crude results.

Returns

“sub_basins”
Sequence of dictionaries with *SubBasin* attributes.

”sub_basin_group”
Sequence of dictionaries with *SubBasinGroup* attributes.

"reservoirs"
Sequence of dictionaries with *Reservoir* attributes.

"channel_profile"
Sequence of dictionaries with *ChannelProfile* attributes.

"hrus"
Sequence of dictionaries with *HRU* attributes.

Return type
dict

class ravenpy.extractors.routing_product.**GridWeightExtractor**(*input_file_path*, *routing_file_path*, *dim_names=('lon_dim', 'lat_dim')*, *var_names=('longitude', 'latitude')*, *routing_id_field='SubId'*, *netcdf_input_field='NetCDF_col'*, *gauge_ids=None*, *sub_ids=None*, *area_error_threshold=0.05*)

Class to extract grid weights.

Notes

The original version of this algorithm can be found at: https://github.com/julemai/GridWeightsGenerator

AREA_ERROR_THRESHOLD = 0.05

CRS_CAEA = 3573

CRS_LLDEG = 4326

DIM_NAMES = ('lon_dim', 'lat_dim')

NETCDF_INPUT_FIELD = 'NetCDF_col'

ROUTING_ID_FIELD = 'SubId'

VAR_NAMES = ('longitude', 'latitude')

extract() → dict
Return dictionary to create a GridWeights command.

ravenpy.extractors.routing_product.**open_shapefile**(*path: str | PathLike*)
Return GeoDataFrame from shapefile path.

ravenpy.extractors.routing_product.**upstream_from_coords**(*lon: float*, *lat: float*, *df: DataFrame | geopandas.GeoDataFrame*) → DataFrame | geopandas.GeoDataFrame

Return the sub-basins located upstream from outlet.

Parameters

- **lon** (*float*) – Longitude of outlet.
- **lat** (*float*) – Latitude of outlet.
- **df** (*pandas.DataFrame or geopandas.GeoDataFrame*) – Routing product.

Returns
Sub-basins located upstream from outlet.

Return type
pandas.DataFrame or geopandas.GeoDataFrame

ravenpy.extractors.routing_product.**upstream_from_id**(*fid: int*, *df: DataFrame | geopandas.GeoDataFrame*) → DataFrame | geopandas.GeoDataFrame

Return upstream sub-basins by evaluating the downstream networks.

Parameters

- **fid** (*int*) – feature ID of the downstream feature of interest.
- **df** (*pandas.DataFrame or geopandas.GeoDataFrame*) – A GeoDataframe comprising the watershed attributes.

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pandas.DataFrame or geopandas.GeoDataFrame

ravenpy.extractors.forecasts.**get_CASPAR_dataset**(*climate_model: str*, *date: datetime*, *thredds: str = 'https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/'*, *directory: str = 'dodsC/birdhouse/disk2/caspar/daily/'*) → Tuple[Dataset, List[DatetimeIndex | Series | Timestamp | Any]]

Return CASPAR dataset.

Parameters

- **climate_model** (*str*) – Type of climate model, for now only "GEPS" is supported.
- **date** (*dt.datetime*) – The date of the forecast.
- **thredds** (*str*) – The thredds server url. Default: "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/"
- **directory** (*str*) – The directory on the thredds server where the data is stored. Default: "dodsC/birdhouse/disk2/caspar/daily/"

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.forecasts.**get_ECCC_dataset**(*climate_model: str*, *thredds: str = 'https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/'*, *directory: str = 'dodsC/datasets/forecasts/eccc_geps/'*) → Tuple[Dataset, List[DatetimeIndex | Series | Timestamp | Any]]

Return latest GEPS forecast dataset.

Parameters

- **climate_model** (*str*) – Type of climate model, for now only "GEPS" is supported.
- **thredds** (*str*) – The thredds server url. Default: "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/"
- **directory** (*str*) – The directory on the thredds server where the data is stored. Default: "dodsC/datasets/forecasts/eccc_geps/"

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.forecasts.**get_hindcast_day**(*region_coll: fiona.Collection*, *date*, *climate_model='GEPS'*)

Generate a forecast dataset that can be used to run raven.

Data comes from the CASPAR archive and must be aggregated such that each file contains forecast data for a single day, but for all forecast timesteps and all members.

The code takes the region shapefile, the forecast date required, and the climate_model to use, here GEPS by default, but eventually could be GEPS, GDPS, REPS or RDPS.

ravenpy.extractors.forecasts.**get_recent_ECCC_forecast**(*region_coll: fiona.Collection*, *climate_model: str = 'GEPS'*) → Dataset

Generate a forecast dataset that can be used to run raven.

Data comes from the ECCC datamart and collected daily. It is aggregated such that each file contains forecast data for a single day, but for all forecast timesteps and all members.

The code takes the region shapefile and the climate_model to use, here GEPS by default, but eventually could be GEPS, GDPS, REPS or RDPS.

Parameters

- **region_coll** (*fiona.Collection*) – The region vectors.
- **climate_model** (*str*) – Type of climate model, for now only "GEPS" is supported.

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.forecasts.**get_subsetted_forecast**(*region_coll: fiona.Collection*, *ds: Dataset*, *times: datetime | DataArray*, *is_caspar: bool*) → Dataset

Get Subsetted Forecast.

This function takes a dataset, a region and the time sampling array and returns the subsetted values for the given region and times.

Parameters

- **region_coll** (*fiona.Collection*) – The region vectors.
- **ds** (*xr.Dataset*) – The dataset containing the raw, worldwide forecast data
- **times** (*dt.datetime or xr.DataArray*) – The array of times required to do the forecast.
- **is_caspar** (*bool*) – True if the data comes from Caspar, false otherwise. Used to define lat/lon on rotated grid.

Returns
The forecast dataset.

Return type
xr.Dataset

12.5 Utilities

12.5.1 Geospatial

Tools for reading and writing geospatial data formats.

ravenpy.utilities.io.**address_append**(*address: str | Path*) → str

Format a URL/URI to be more easily read with libraries such as "rasterstats".

Parameters
address (*Union[str, Path]*) – URL/URI to a potential zip or tar file

Returns
URL/URI prefixed for archive type

Return type
str

ravenpy.utilities.io.**archive_sniffer**(*archives: str | Path | List[str | Path]*, *working_dir: str | Path | None = None*, *extensions: Sequence[str] | None = None*) → List[str | Path]

Return a list of locally unarchived files that match the desired extensions.

Parameters

- **archives** (*str or Path or list of str or Path*) – Archive location or list of archive locations.
- **working_dir** (*str or Path, optional*) – String or Path to a working location.
- **extensions** (*Sequence of str, optional*) – List of accepted extensions.

Returns
List of files with matching accepted extensions.

Return type
list of str or Path

ravenpy.utilities.io.**crs_sniffer**(**args: str | Path | Sequence[str | Path]*) → List[int | str] | str | int

Return the list of CRS found in files.

Parameters
args (*Union[str, Path, Sequence[Union[str, Path]]]*) – Path(s) to the file(s) to examine.

Returns
Returns either a list of CRSes or a single CRS definition, depending on the number of instances found.

Return type
Union[List[str], str]

ravenpy.utilities.io.**generic_extract_archive**(*resources: str | Path | List[bytes | str | Path]*, *output_dir: str | Path | None = None*) → List[str]

Extract archives (tar/zip) to a working directory.

Parameters

- **resources** (*str or Path or list of bytes or str or Path*) – List of archive files (if netCDF files are in list, they are passed and returned as well in the return).

- **output_dir** (*str or Path, optional*) – String or Path to a working location (default: temporary folder).

Returns
List of original or of extracted files.

Return type
list

ravenpy.utilities.io.**get_bbox**(*vector: str | Path*, *all_features: bool = True*) → Tuple[float, float, float, float]

Return bounding box of all features or the first feature in file.

Parameters

- **vector** (*str or Path*) – A path to file storing vector features.
- **all_features** (*bool*) – Return the bounding box for all features. Default: True.

Returns
Geographic coordinates of the bounding box (lon0, lat0, lon1, lat1).

Return type
float, float, float, float

ravenpy.utilities.io.**is_within_directory**(*directory: str | PathLike*, *target: str | PathLike*) → bool

ravenpy.utilities.io.**raster_datatype_sniffer**(*file: str | Path*) → str

Return the type of the raster stored in the file.

Parameters
file (*Union[str, Path]*) – Path to file.

Returns
rasterio datatype of array values

Return type
str

ravenpy.utilities.io.**safe_extract**(*tar: TarFile*, *path: str = '.'*, *members=None*, ***, *numeric_owner=False*) → None

Tools for performing geospatial translations and transformations.

ravenpy.utilities.geo.**determine_upstream_ids**(*fid: str*, *df: DataFrame | geopandas.GeoDataFrame*, *basin_field: str | None = None*, *downstream_field: str | None = None*, *basin_family: str | None = None*) → DataFrame | geopandas.GeoDataFrame

Return a list of upstream features by evaluating the downstream networks.

Parameters

- **fid** (*str*) – feature ID of the downstream feature of interest.
- **df** (*pd.DataFrame*) – A Dataframe comprising the watershed attributes.
- **basin_field** (*str*) – The field used to determine the id of the basin according to hydro project.
- **downstream_field** (*str*) – The field identifying the downstream sub-basin for the hydro project.
- **basin_family** (*str, optional*) – Regional watershed code (For HydroBASINS dataset).

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pd.DataFrame

ravenpy.utilities.geo.**find_geometry_from_coord**(*lon: float*, *lat: float*, *df: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Return the geometry containing the given coordinates.

lon
[float] Longitude.

lat
[float] Latitude.

df
[GeoDataFrame] Data.

Returns
Record whose geometry contains the point.

Return type
GeoDataFrame

ravenpy.utilities.geo.**generic_raster_clip**(*raster: str | Path*, *output: str | Path*, *geometry: shapely.geometry.Polygon | shapely.geometry.MultiPolygon | List[shapely.geometry.Polygon | shapely.geometry.MultiPolygon]*, *touches: bool = False*, *fill_with_nodata: bool = True*, *padded: bool = True*, *raster_compression: str = 'lzw'*) → None

Crop a raster file to a given geometry.

Parameters

- **raster** (*Union[str, Path]*) – Path to input raster.
- **output** (*Union[str, Path]*) – Path to output raster.
- **geometry** (*Union[Polygon, MultiPolygon, List[Union[Polygon, MultiPolygon]]]*) – Geometry defining the region to crop.
- **touches** (*bool*) – Whether to include cells that intersect the geometry or not. Default: True.
- **fill_with_nodata** (*bool*) – Whether to keep pixel values for regions outside of shape or set as nodata or not. Default: True.
- **padded** (*bool*) – Whether to add a half-pixel buffer to shape before masking or not. Default: True.
- **raster_compression** (*str*) – Level of data compression. Default: 'lzw'.

Return type
None

ravenpy.utilities.geo.**generic_raster_warp**(*raster: str | Path*, *output: str | Path*, *target_crs: str | dict | pyproj.CRS*, *raster_compression: str = 'lzw'*) → None

Reproject a raster file.

Parameters

- **raster** (*Union[str, Path]*) – Path to input raster.

- **output** (*Union[str, Path]*) – Path to output raster.
- **target_crs** (*str or dict*) – Target projection identifier.
- **raster_compression** (*str*) – Level of data compression. Default: 'lzw'.

Return type
None

ravenpy.utilities.geo.**generic_vector_reproject**(*vector: str | Path*, *projected: str | Path*, *source_crs: str | pyproj.CRS = 4326*, *target_crs: str | pyproj.CRS | None = None*) → None

Reproject all features and layers within a vector file and return a GeoJSON

Parameters

- **vector** (*Union[str, Path]*) – Path to a file containing a valid vector layer.
- **projected** (*Union[str, Path]*) – Path to a file to be written.
- **source_crs** (*Union[str, pyproj.crs.CRS]*) – CRS for the source geometry. Default: 4326.
- **target_crs** (*Union[str, pyproj.crs.CRS]*) – CRS for the target geometry.

Return type
None

ravenpy.utilities.geo.**geom_transform**(*geom: shapely.geometry.GeometryCollection | shapely.geometry.shape*, *source_crs: str | int | pyproj.CRS = 4326*, *target_crs: str | int | pyproj.CRS | None = None*) → shapely.geometry.GeometryCollection

Change the projection of a geometry.

Assuming a geometry's coordinates are in a *source_crs*, compute the new coordinates under the *target_crs*.

Parameters

- **geom** (*Union[GeometryCollection, shape]*) – Source geometry.
- **source_crs** (*Union[str, int, CRS]*) – Projection identifier (proj4) for the source geometry, e.g. '+proj=longlat +datum=WGS84 +no_defs'.
- **target_crs** (*Union[str, int, CRS]*) – Projection identifier (proj4) for the target geometry.

Returns
Reprojected geometry.

Return type
GeometryCollection

GeoServer interaction operations.

Working assumptions for this module: * Point coordinates are passed as shapely.geometry.Point instances. * BBox coordinates are passed as (lon1, lat1, lon2, lat2). * Shapes (polygons) are passed as shapely.geometry.shape parsable objects. * All functions that require a CRS have a CRS argument with a default set to WGS84. * GEOSERVER_URL points to the GeoServer instance hosting all files. * For legacy reasons, we also accept the *GEO_URL* environment variable.

TODO: Refactor to remove functions that are just 2-lines of code. For example, many function's logic essentially consists in creating the layer name. We could have a function that returns the layer name, and then other functions expect the layer name.

ravenpy.utilities.geoserver.**filter_hydro_routing_attributes_wfs**(*attribute: str | None = None*, *value: str | float | int | None = None*, *level: int = 12*, *lakes: str = '1km'*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → str

Return a URL that formats and returns a remote GetFeatures request from hydro routing dataset.

For geographic rasters, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **attribute** (*list*) – Attributes/fields to be queried.
- **value** (*str or int or float*) – The requested value for the attribute.
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
URL to the GeoJSON-encoded WFS response.

Return type
str

ravenpy.utilities.geoserver.**filter_hydrobasins_attributes_wfs**(*attribute: str*, *value: str | float | int*, *domain: str*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → str

Return a URL that formats and returns a remote GetFeatures request from the USGS HydroBASINS dataset.

For geographic raster grids, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **attribute** (*str*) – Attribute/field to be queried.
- **value** (*str or float or int*) – Value for attribute queried.
- **domain** (*{"na", "ar"}*) – The domain of the HydroBASINS data.
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
URL to the GeoJSON-encoded WFS response.

Return type
str

ravenpy.utilities.geoserver.**get_hydro_routing_attributes_wfs**(*attribute: Sequence[str]*, *level: int = 12*, *lakes: str = '1km'*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → str

Return a URL that formats and returns a remote GetFeatures request from hydro routing dataset.

For geographic rasters, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **attribute** (*list*) – Attributes/fields to be queried.
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
URL to the GeoJSON-encoded WFS response.

Return type
str

ravenpy.utilities.geoserver.**get_hydro_routing_location_wfs**(*coordinates: Tuple[str | float | int, str | float | int]*, *lakes: str*, *level: int = 12*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → dict

Return features from the hydro routing data set using bounding box coordinates.

For geographic rasters, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **coordinates** (*Tuple[str or float or int, str or float or int]*) – Geographic coordinates of the bounding box (left, down, right, up).
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
A GeoJSON-derived dictionary of vector features (FeatureCollection).

Return type
dict

ravenpy.utilities.geoserver.**get_hydrobasins_location_wfs**(*coordinates: Tuple[str | float | int, str | float | int]*, *domain: str | None = None*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → Dict[str, str | int | float]

Return features from the USGS HydroBASINS data set using bounding box coordinates.

For geographic raster grids, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **coordinates** (*Tuple[str or float or int, str or float or int]*) – Geographic coordinates of the bounding box (left, down, right, up).
- **domain** (*{"na", "ar"}*) – The domain of the HydroBASINS data.
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
A GeoJSON-encoded vector feature.

Return type
dict

ravenpy.utilities.geoserver.**get_raster_wcs**(*coordinates: Iterable | Sequence[float | str]*, *geographic: bool = True*, *layer: str | None = None*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → bytes

Return a subset of a raster image from the local GeoServer via WCS 2.0.1 protocol.

For geographic raster grids, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **coordinates** (*Sequence of int or float or str*) – Geographic coordinates of the bounding box (left, down, right, up)
- **geographic** (*bool*) – If True, uses "Long" and "Lat" in WCS call. Otherwise, uses "E" and "N".
- **layer** (*str*) – Layer name of raster exposed on GeoServer instance, e.g. 'public:CEC_NALCMS_LandUse_2010'
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
A GeoTIFF array.

Return type
bytes

ravenpy.utilities.geoserver.**hydro_routing_upstream**(*fid: str | float | int*, *level: int = 12*, *lakes: str = '1km'*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → Series

Return a list of hydro routing features located upstream.

Parameters

- **fid** (*str or float or int*) – Basin feature ID code of the downstream feature.
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pd.Series

ravenpy.utilities.geoserver.**hydrobasins_aggregate**(*gdf: DataFrame*) → DataFrame

Aggregate multiple HydroBASINS watersheds into a single geometry.

Parameters
gdf (*pd.DataFrame*) – Watershed attributes indexed by HYBAS_ID

Return type
pd.DataFrame

ravenpy.utilities.geoserver.**hydrobasins_upstream**(*feature: dict*, *domain: str*) → DataFrame

Return a list of HydroBASINS features located upstream.

Parameters

- **feature** (*dict*) – Basin feature attributes, including the fields ["HYBAS_ID", "NEXT_DOWN", "MAIN_BAS"].
- **domain** (*{"na", "ar"}*) – Domain of the feature, North America or Arctic.

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pd.Series

ravenpy.utilities.geoserver.**select_hybas_domain**(*bbox: Tuple[int | float, int | float, int | float, int | float] | None = None*, *point: Tuple[int | float, int | float] | None = None*) → str

Provided a given coordinate or boundary box, return the domain name of the geographic region the coordinate is located within.

Parameters

- **bbox** (*Optional[Tuple[Union[float, int], Union[float, int], Union[float, int], Union[float, int]]]*) – Geographic coordinates of the bounding box (left, down, right, up).
- **point** (*Optional[Tuple[Union[float, int], Union[float, int]]]*) – Geographic coordinates of an intersecting point (lon, lat).

Returns
The domain that the coordinate falls within. Possible results: "na", "ar".

Return type
str

12.5.2 Graphics

Library to perform graphs for the streamflow time series analysis.

The following graphs can be plotted:

- hydrograph
- mean_annual_hydrograph
- spaghetti_annual_hydrograph

ravenpy.utilities.graphs.**forecast**(*file: str | Path*, *fcst_var: str = 'q_sim'*) → Figure

Create a graphic of the hydrograph for each forecast member.

Parameters

- **file** (*str or Path*) – Raven output file containing simulated streamflows.
- **fcst_var** (*str*) – Name of the streamflow variable.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.graphs.**hindcast**(*file: str | Path*, *fcst_var: str*, *qobs: str | Path*, *qobs_var: str*) → Figure

Create a graphic of the hydrograph for each hindcast member.

Parameters

- **file** (*str or Path*) – Raven output file containing simulated streamflows.
- **fcst_var** (*str*) – Name of the streamflow variable.
- **qobs** (*str or Path*) – Streamflow observation file, with times matching the hindcast.
- **qobs_var** (*str*) – Name of the streamflow observation variable.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.graphs.**hydrograph**(*file_list: Sequence[str | Path]*)

Create a graphic of the hydrograph for each model simulation.

Parameters
file_list (*Sequence of str or Path*) – Raven output files containing simulated streamflows.

ravenpy.utilities.graphs.**mean_annual_hydrograph**(*file_list: Sequence[str | Path]*)

Create a graphic of the mean hydrological cycle for each model simulation.

Parameters
file_list (*Sequence of str or Path*) – Raven output files containing simulated streamflows.

ravenpy.utilities.graphs.**spaghetti_annual_hydrograph**(*file: str | Path*)

Create a spaghetti plot of the mean hydrological cycle for one model simulations.

The mean simulation is also displayed.

Parameters
file (*str or Path*) – Raven output files containing simulated streamflows of one model.

ravenpy.utilities.graphs.**ts_fit_graph**(*ts: DataArray*, *params: DataArray*) → Figure

Create graphic showing a histogram of the data and the distribution fitted to it.

The graphic contains one panel per watershed.

Parameters

- **ts** (*xr.DataArray*) – Stream flow time series with dimensions (time, nbasins).
- **params** (*xr.DataArray*) – Fitted distribution parameters returned by *xclim.land.fit* indicator.

Returns
Figure showing a histogram and the parameterized pdf.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.graphs.**ts_graphs**(*file*, *trend=True*, *alpha=0.05*)

Create a figure with the statistics so one can see a trend in the data.

Graphs for time series statistics.

Parameters
file (*str or Path*) – xarray-compatible file containing streamflow statistics for one run.

12.5.3 Regionalization

Tools for hydrological regionalization.

ravenpy.utilities.regionalization.**IDW**(*qsims: DataArray*, *dist: Series*) → DataArray

Inverse distance weighting.

Parameters

- **qsims** (*xr.DataArray*) – Ensemble of hydrogram stacked along the *members* dimension.
- **dist** (*pd.Series*) – Distance from catchment which generated each hydrogram to target catchment.

Returns
Inverse distance weighted average of ensemble.

Return type
xr.DataArray

ravenpy.utilities.regionalization.**distance**(*gauged: DataFrame*, *ungauged: Series*) → Series

Return geographic distance [km] between ungauged and database of gauged catchments.

Parameters

- **gauged** (*pd.DataFrame*) – Table containing columns for longitude and latitude of catchment's centroid.
- **ungauged** (*pd.Series*) – Coordinates of the ungauged catchment.

Return type
pd.Series

ravenpy.utilities.regionalization.**multiple_linear_regression**(*source: DataFrame*, *params: DataFrame*, *target: DataFrame*) → Tuple[List[Any], List[int]]

Multiple Linear Regression for model parameters over catchment properties.

Uses known catchment properties and model parameters to estimate model parameter over an ungauged catchment using its properties.

Parameters

- **source** (*pd.DataFrame*) – Properties of gauged catchments.
- **params** (*pd.DataFrame*) – Model parameters of gauged catchments.
- **target** (*pd.DataFrame*) – Properties of the ungauged catchment.

Returns
A named tuple of the estimated model parameters and the R2 of the linear regression.

Return type
list of Any, list of int

ravenpy.utilities.regionalization.**read_gauged_params**(*model*)

Return table of NASH-Sutcliffe Efficiency values and model parameters for North American catchments.

Returns

- *pd.DataFrame* – Nash-Sutcliffe Efficiency keyed by catchment ID.
- *pd.DataFrame* – Model parameters keyed by catchment ID.

ravenpy.utilities.regionalization.**read_gauged_properties**(*properties*) → DataFrame

Return table of gauged catchments properties over North America.

Returns
Catchment properties keyed by catchment ID.

Return type
pd.DataFrame

ravenpy.utilities.regionalization.**regionalization_params**(*method: str*, *gauged_params: DataFrame*, *gauged_properties: DataFrame*, *ungauged_properties: DataFrame*, *filtered_params: DataFrame*, *filtered_prop: DataFrame*) → List[float]

Return the model parameters to use for the regionalization.

Parameters

- **method** (*{'MLR', 'SP', 'PS', 'SP_IDW', 'PS_IDW', 'SP_IDW_RA', 'PS_IDW_RA'}*) – Name of the regionalization method to use.
- **gauged_params** (*pd.DataFrame*) – A DataFrame of parameters for donor catchments (size = number of donors)
- **gauged_properties** (*pd.DataFrame*) – A DataFrame of properties of the donor catchments (size = number of donors)
- **ungauged_properties** (*pd.DataFrame*) – A DataFrame of properties of the ungauged catchment (size = 1)
- **filtered_params** (*pd.DataFrame*) – A DataFrame of parameters of all filtered catchments (size = all catchments with NSE > min_NSE)
- **filtered_prop** (*pd.DataFrame*) – A DataFrame of properties of all filtered catchments (size = all catchments with NSE > min_NSE)

Returns
List of model parameters to be used for the regionalization.

Return type
list

ravenpy.utilities.regionalization.**regionalize**(*config: Config*, *method: str*, *nash: Series*, *params: DataFrame | None = None*, *props: DataFrame | None = None*, *target_props: Series | dict | None = None*, *size: int = 5*, *min_NSE: float = 0.6*, *workdir: str | Path | None = None*, *overwrite: bool = False*, ***kwds*) → Tuple[DataArray, Dataset]

Perform regionalization for catchment whose outlet is defined by coordinates.

Parameters

- **config** (ravenpy.config.rvs.Config) – Symbolic emulator configuration. Only GR4JCN, HMETS and Mohyse are supported.
- **method** (*{'MLR', 'SP', 'PS', 'SP_IDW', 'PS_IDW', 'SP_IDW_RA', 'PS_IDW_RA'}*) – Name of the regionalization method to use.
- **nash** (*pd.Series*) – NSE values for the parameters of gauged catchments.
- **params** (*pd.DataFrame*) – Model parameters of gauged catchments. Needed for all but MRL method.
- **props** (*pd.DataFrame*) – Properties of gauged catchments to be analyzed for the regionalization. Needed for MLR and RA methods.
- **target_props** (*pd.Series or dict*) – Properties of ungauged catchment. Needed for MLR and RA methods.
- **size** (*int*) – Number of catchments to use in the regionalization.
- **min_NSE** (*float*) – Minimum calibration NSE value required to be considered as a donor.
- **workdir** (*Union[str, Path]*) – Work directory. If None, a temporary directory will be created.
- **overwrite** (*bool*) – If True, existing files will be overwritten.
- ****kwds** – Model configuration parameters, including the forcing files (ts).

Returns

- *(qsim, ensemble)*
- **qsim** (*DataArray (time,)*) – Multi-donor averaged predicted streamflow.
- **ensemble** (*Dataset*) –

  q_sim
  [DataArray (realization, time)] Ensemble of members based on number of donors.

  parameter
  [DataArray (realization, param)] Parameters used to run the model.

ravenpy.utilities.regionalization.**similarity**(*gauged: DataFrame*, *ungauged: DataFrame*, *kind: str = 'ptp'*) → Series

Return similarity measure between gauged and ungauged catchments.

Parameters

- **gauged** (*pd.DataFrame*) – Gauged catchment properties.
- **ungauged** (*pd.DataFrame*) – Ungauged catchment properties
- **kind** (*{'ptp', 'std', 'iqr'}*) – Normalization method: peak to peak (maximum - minimum), standard deviation, inter-quartile range.

Return type
pd.Series

CHAPTER

THIRTEEN

RAVENPY

13.1 ravenpy package

A Python package to help run Raven, the hydrologic modelling framework.

13.1.1 Subpackages

ravenpy.cli package

ravenpy.cli.**main**()

Submodules

ravenpy.cli.aggregate_forcings_to_hrus module

ravenpy.cli.collect_subbasins_upstream_of_gauge module

ravenpy.cli.generate_grid_weights module

ravenpy.cli.generate_hrus_from_routing_product module

ravenpy.config package

Subpackages

ravenpy.config.emulators package

ravenpy.config.emulators.**get_model**(*name*)

Return the corresponding Raven emulator configuration class.

Parameters

name (*str*) – Model class name or model identifier.

Return type

Raven model configuration class

Submodules

ravenpy.config.emulators.blended module

class ravenpy.config.emulators.blended.**Blended**(*, *EnKFMode: ~ravenpy.config.options.EnKFMode | None = None, WindowSize: int | None = None, SolutionRunName: str | None = None, ExtraRVTFilename: str | None = None, OutputDirectoryFormat: str | ~pathlib.Path | None = None, ForecastRVTFilename: str | None = None, TruncateHindcasts: bool | None = None, ForcingPerturbation: ~typing.Sequence[~ravenpy.config.commands.ForcingPerturbation] | None = None, AssimilatedState: ~typing.Sequence[~ravenpy.config.commands.AssimilatedState] | None = None, AssimilateStreamflow: ~typing.Sequence[~ravenpy.config.commands.AssimilateStreamflow] | None = None, ObservationErrorModel: ~typing.Sequence[~ravenpy.config.commands.ObservationErrorModel] | None = None, params: ~types.Params = Params(X01=Variable('X01'), X02=Variable('X02'), X03=Variable('X03'), X04=Variable('X04'), X05=Variable('X05'), X06=Variable('X06'), X07=Variable('X07'), X08=Variable('X08'), X09=Variable('X09'), X10=Variable('X10'), X11=Variable('X11'), X12=Variable('X12'), X13=Variable('X13'), X14=Variable('X14'), X15=Variable('X15'), X16=Variable('X16'), X17=Variable('X17'), X18=Variable('X18'), X19=Variable('X19'), X20=Variable('X20'), X21=Variable('X21'), X22=Variable('X22'), X23=Variable('X23'), X24=Variable('X24'), X25=Variable('X25'), X26=Variable('X26'), X27=Variable('X27'), X28=Variable('X28'), X29=Variable('X29'), X30=Variable('X30'), X31=Variable('X31'), X32=Variable('X32'), X33=Variable('X33'), X34=Variable('X34'), X35=Variable('X35'), R01=Variable('R01'), R02=Variable('R02'), R03=Variable('R03'), R04=Variable('R04'), R05=Variable('R05'), R06=Variable('R06'), R07=Variable('R07'), R08=Variable('R08')), SoilClasses: ~ravenpy.config.commands.SoilClasses = [{'name': 'TOPSOIL'}, {'name': 'PHREATIC'}, {'name': 'DEEP_GW'}], SoilProfiles: ~ravenpy.config.commands.SoilProfiles = [{'name': 'LAKE'}, {'name': 'ROCK'}, {'name': 'DEFAULT_P', 'soil_classes': ('TOPSOIL', 'PHREATIC', 'DEEP_GW'), 'thicknesses': (Variable('X29'), Variable('X30'), 1000000.0)}], VegetationClasses: ~ravenpy.config.commands.VegetationClasses = [{'max_ht': 4, 'max_lai': 5, 'max_leaf_cond': 5, 'name': 'FOREST'}], LandUseClasses: ~ravenpy.config.commands.LandUseClasses = [LandUseClass(name='FOREST', impermeable_frac=0.0, forest_coverage=0.02345)], TerrainClasses: ~ravenpy.config.commands.TerrainClasses = [TerrainClass(name='DEFAULT_T', hillslope_length=1.0, drainage_density=1.0, topmodel_lambda=Variable('X07'))], SoilParameterList: ~ravenpy.config.commands.SoilParameterList = {'parameters': ('POROSITY', 'PERC_COEFF',*

Bases: *Config*

Blended hydrological model for blending outputs from different submodules.

References

Mai, J., Craig, J. R., and Tolson, B. A.: Simultaneously determining global sensitivities of model parameters and model structure, Hydrol. Earth Syst. Sci., 24, 5835–5858, https://doi.org/10.5194/hess-24-5835-2020, 2020.

Chlumsky, R., Mai, J., Craig, J. R., & Tolson, B. A. (2021). Simultaneous calibration of hydrologic model structure and parameters using a blended model. Water Resources Research, 57, e2020WR029229. https://doi.org/10.1029/2020WR029229

calendar: **Calendar**

catchment_route: **CatchmentRoute**

evaporation: **Evaporation**

global_parameter: **Dict**

hru_state_variable_table: *HRUStateVariableTable*

hrus: **HRUs**

hydrologic_processes: **Sequence[*Process* | ProcessGroup]**

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

model_computed_fields: **ClassVar[dict[str, ComputedFieldInfo]] = {}**

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: **ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}**

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow],
NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow',
alias_priority=2, validate_default=False), 'assimilated_state':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState],
NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState',
alias_priority=2, validate_default=False), 'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'basin_state_variables':
FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False,
default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Calendar, required=False,
default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2),
'catchment_route':  FieldInfo(annotation=CatchmentRoute, required=False,
default='ROUTE_DUMP', alias='CatchmentRouting', alias_priority=2),
'channel_profile':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile],
NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile',
alias_priority=2, validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'enkf_mode':
FieldInfo(annotation=Union[EnKFMode, NoneType], required=False,
default_factory=<lambda>, alias='EnKFMode', alias_priority=2,
validate_default=False), 'ensemble_mode':  FieldInfo(annotation=Union[EnsembleMode,
NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode',
alias_priority=2, validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Evaporation, required=False, default=<Evaporation.OUDIN:
'PET_OUDIN'>, alias='Evaporation', alias_priority=2), 'extra_rvt_filename':
FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>,
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: `Dict[str, str]`

params: `Params`

potential_melt_method: `PotentialMeltMethod`

rain_snow_fraction: `RainSnowFraction`

routing: `Routing`

seasonal_relative_height: *SeasonalRelativeHeight*

seasonal_relative_lai: *SeasonalRelativeLAI*

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basins: *SubBasins*

terrain_classes: *TerrainClasses*

time_step: `float | str`

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: `bool`

class ravenpy.config.emulators.blended.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.emulators.blended.LandHRU], required=True)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[LandHRU]

class ravenpy.config.emulators.blended.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'FOREST'*, *veg_class: str = 'FOREST'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = 'DEFAULT_T'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Bases: *HRU*

aquifer_profile: str

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='FOREST'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='DEFAULT_T'), 'veg_class': FieldInfo(annotation=str, required=False, default='FOREST')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

ravenpy.config.emulators.canadianshield module

class ravenpy.config.emulators.canadianshield.**BedRockHRU**(*, *hru_id: int = 2*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'FOREST'*, *veg_class: str = 'FOREST'*, *soil_profile: str = 'SOILP_BEDROCK'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Bases: *HRU*

aquifer_profile: str

hru_id: int

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=2), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='FOREST'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='SOILP_BEDROCK'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='FOREST')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.canadianshield.**CanadianShield**(*, *EnKFMode: ~ravenpy.config.options.EnKFMode | None = None, WindowSize: int | None = None, SolutionRunName: str | None = None, ExtraRVTFilename: str | None = None, OutputDirectoryFormat: str | ~pathlib.Path | None = None, ForecastRVTFilename: str | None = None, TruncateHindcasts: bool | None = None, ForcingPerturbation: ~typing.Sequence[~ravenpy.config.commands.ForcingPertu | None = None, AssimilatedState: ~typing.Sequence[~ravenpy.config.commands.AssimilatedSt | None = None, AssimilateStreamflow: ~typing.Sequence[~ravenpy.config.commands.AssimilateStr | None = None, ObservationErrorModel: ~typing.Sequence[~ravenpy.config.commands.ObservationE | None = None, params: ~types.Params = Params(X01=Variable('X01'), X02=Variable('X02'), X03=Variable('X03'), X04=Variable('X04'), X05=Variable('X05'), X06=Variable('X06'), X07=Variable('X07'), X08=Variable('X08'), X09=Variable('X09'), X10=Variable('X10'), X11=Variable('X11'), X12=Variable('X12'), X13=Variable('X13'), X14=Variable('X14'), X15=Variable('X15'), X16=Variable('X16'), X17=Variable('X17'), X18=Variable('X18'), X19=Variable('X19'), X20=Variable('X20'), X21=Variable('X21'), X22=Variable('X22'), X23=Variable('X23'), X24=Variable('X24'), X25=Variable('X25'), X26=Variable('X26'), X27=Variable('X27'), X28=Variable('X28'), X29=Variable('X29'), X30=Variable('X30'), X31=Variable('X31'), X32=Variable('X32'), X33=Variable('X33'), X34=Variable('X34')), SoilClasses: ~ravenpy.config.commands.SoilClasses = [{'name': 'TOPSOIL'}, {'name':*

Bases: *Config*

The Canadian Shield model is a useful configuration of Raven for Canadian shield basins characterized by shallow soils atop rock, with ample exposed rock and lakes. It is a custom model for this type of region, but there is no reference to it in the literature.

calendar: Calendar

catchment_route: CatchmentRoute

classmethod equal_area(*hrus*)

evaporation: Evaporation

global_parameter: Dict

hru_state_variable_table: *HRUStateVariableTable*

hrus: HRUs

hydrologic_processes: Sequence[*Process* | Conditional]

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'assimilation_start_time': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2, validate_default=False), 'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'calendar': FieldInfo(annotation=Calendar, required=False, default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2), 'catchment_route': FieldInfo(annotation=CatchmentRoute, required=False, default='ROUTE_TRI_CONVOLUTION', alias='CatchmentRouting', alias_priority=2), 'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'cloud_cover_method': FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False, default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2, validate_default=False), 'custom_output': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput], NoneType], required=False, default_factory=<lambda>, alias='CustomOutput', alias_priority=2, validate_default=False), 'debug_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DebugMode', alias_priority=2, validate_default=False), 'define_hru_groups': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2, validate_default=False), 'deltares_fews_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode', alias_priority=2, validate_default=False), 'direct_evaporation': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2, description='Rainfall is automatically reduced through evapotranspiration up to the limit of the calculated PET.', validate_default=False), 'dont_write_watershed_storage': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage', alias_priority=2, description='Do not write watershed storage variables to disk.', validate_default=False), 'duration': FieldInfo(annotation=Union[float, NoneType], required=False, default_factory=<lambda>, alias='Duration', alias_priority=2, validate_default=False), 'end_date': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'ensemble_mode': FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode', alias_priority=2, validate_default=False), 'evaluation_metrics': FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics], NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics', alias_priority=2, validate_default=False), 'evaluation_period': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod], NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod', alias_priority=2, validate_default=False), 'evaporation': FieldInfo(annotation=Evaporation, required=False, default=<Evaporation.HARGREAVES_1985: 'PET_HARGREAVES_1985'>, alias='Evaporation', alias_priority=2), 'extra_rvt_filename': FieldInfo(annotation=Union[str, NoneType],

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

monthly_interpolation_method: MonthlyInterpolationMethod

netcdf_attribute: Dict[str, str]

ow_evaporation: Evaporation

params: Params

potential_melt_method: PotentialMeltMethod

precip_icept_frac: PrecipIceptFract

rain_snow_fraction: RainSnowFraction

routing: Routing

seasonal_relative_height: *SeasonalRelativeHeight*

seasonal_relative_lai: *SeasonalRelativeLAI*

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basins: *SubBasins*

sw_canopy_correct: SWCanopyCorrect

time_step: float | str

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: bool

class ravenpy.config.emulators.canadianshield.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for CanadianShield.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Tuple[ravenpy.config.emulators.canadianshield.OrganicHRU, ravenpy.config.emulators.canadianshield.BedRockHRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Tuple[OrganicHRU, BedRockHRU]

class ravenpy.config.emulators.canadianshield.**OrganicHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'FOREST'*, *veg_class: str = 'FOREST'*, *soil_profile: str = 'SOILP_ORG'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Bases: *HRU*

aquifer_profile: str

hru_id: int

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='FOREST'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='SOILP_ORG'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='FOREST')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

ravenpy.config.emulators.gr4jcn module

class ravenpy.config.emulators.gr4jcn.**GR4JCN**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: P = P(GR4J_X1=Variable('GR4J_X1'), GR4J_X2=Variable('GR4J_X2'), GR4J_X3=Variable('GR4J_X3'), GR4J_X4=Variable('GR4J_X4'), CEMANEIGE_X1=Variable('CEMANEIGE_X1'), CEMANEIGE_X2=Variable('CEMANEIGE_X2'))*, *SoilClasses: SoilClasses = [{'name': 'SOIL_PROD'}, {'name': 'SOIL_ROUT'}, {'name': 'SOIL_TEMP'}, {'name': 'SOIL_GW'}, {'name': 'AQUIFER'}]*, *SoilProfiles: SoilProfiles = [{'name': 'DEFAULT_P', 'soil_classes': ['SOIL_PROD', 'SOIL_ROUT', 'SOIL_TEMP', 'SOIL_GW'], 'thicknesses': [Variable('GR4J_X1'), 0.3, 1, 1]}, {'name': 'LAKE'}]*, *VegetationClasses: VegetationClasses = [{'name': 'VEG_ALL'}, {'name': 'VEG_WATER'}]*, *LandUseClasses: LandUseClasses = [{'name': 'LU_ALL'}, {'name': 'LU_WATER'}]*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList = {'parameters': ('POROSITY', 'GR4J_X3', 'GR4J_X2'), 'pl': [ParameterList(name='[DEFAULT]', values=(1.0, Variable('GR4J_X3'), Variable('GR4J_X2')))]}*, *LandUseParameterList: LandUseParameterList = {'parameters': ('GR4J_X4', 'MELT_FACTOR'), 'pl': [ParameterList(name='[DEFAULT]', values=(Variable('GR4J_X4'), 7.73))]}*, *VegetationParameterList: VegetationParameterList | None = None*, *ChannelProfile: Sequence[ChannelProfile] | None = None*, *GlobalParameter: Dict[str, Variable | Expression | float | None] = {'ADIABATIC_LAPSE': 0.0065, 'AIRSNOW_COEFF': Sum((1, Product((-1, Variable('CEMANEIGE_X2'))))), 'AVG_ANNUAL_SNOW': Variable('CEMANEIGE_X1'), 'PRECIP_LAPSE': 0.0004}*, *RainSnowTransition: RainSnowTransition = {'delta': 1, 'temp': 0}*, *SeasonalRelativeLAI: SeasonalRelativeLAI | None = None*, *SeasonalRelativeHeight: SeasonalRelativeHeight | None = None*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*, *SubBasins: SubBasins = [SubBasin(subbasin_id=1, name='sub_001', downstream_id=-1, profile='NONE', reach_length=0, gauged=True, gauge_id='')]*, *SubBasinGroup: Sequence[SubBasinGroup] | None = None*, *SubBasinProperties: SubBasinProperties | None = None*, *SBGroupPropertyMultiplier:*

Bases: *Config*

GR4J + Cemaneige global hydrological model

References

Perrin, C., C. Michel and V. Andréassian (2003). Improvement of a parsimonious model for streamflow simulation. Journal of Hydrology, 279(1-4), 275-289. doi: 10.1016/S0022-1694(03)00225-7.

Valéry, Audrey, Vazken Andréassian, and Charles Perrin. 2014. "'As Simple as Possible but Not Simpler': What Is Useful in a Temperature-Based Snow-Accounting Routine? Part 2 - Sensitivity Analysis of the Cemaneige Snow Accounting Routine on 380 Catchments." Journal of Hydrology, no. 517(0): 1176–87, doi: 10.1016/j.jhydrol.2014.04.058.

calendar: Calendar

catchment_route: CatchmentRoute

evaporation: Evaporation

global_parameter: Dict[str, Variable | Expression | float | None]

hrus: HRUs

hydrologic_processes: Sequence[*Process* | Conditional]

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow],
NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow',
alias_priority=2, validate_default=False), 'assimilated_state':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState],
NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState',
alias_priority=2, validate_default=False), 'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'basin_state_variables':
FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False,
default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Calendar, required=False,
default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2),
'catchment_route':  FieldInfo(annotation=CatchmentRoute, required=False,
default='ROUTE_DUMP', alias='CatchmentRoute', alias_priority=2), 'channel_profile':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile],
NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile',
alias_priority=2, validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'enkf_mode':
FieldInfo(annotation=Union[EnKFMode, NoneType], required=False,
default_factory=<lambda>, alias='EnKFMode', alias_priority=2,
validate_default=False), 'ensemble_mode':  FieldInfo(annotation=Union[EnsembleMode,
NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode',
alias_priority=2, validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Evaporation, required=False, default='PET_OUDIN',
alias='Evaporation', alias_priority=2), 'extra_rvt_filename':
FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>,
alias='ExtraRVTFilename', alias_priority=2, validate_default=False),
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: Dict[str, str]

oro_precip_correct: OroPrecipCorrect

oro_temp_correct: OroTempCorrect

params: P

potential_melt: PotentialMeltMethod

rain_snow_fraction: RainSnowFraction

rain_snow_transition: *RainSnowTransition*

routing: Routing

soil_classes: *SoilClasses*

soil_model: int | *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basins: *SubBasins*

time_step: float | str

uniform_initial_conditions: Dict[str, Variable | Expression | float | None] | None

vegetation_classes: *VegetationClasses*

write_netcdf_format: bool

class ravenpy.config.emulators.gr4jcn.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[Union[ravenpy.config.emulators.gr4jcn.LandHRU, ravenpy.config.emulators.gr4jcn.LakeHRU]], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[LandHRU | LakeHRU]

class ravenpy.config.emulators.gr4jcn.**LakeHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'LU_WATER'*, *veg_class: str = 'VEG_WATER'*, *soil_profile: str = 'LAKE'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: Literal['lake'] = 'lake'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['lake']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['lake'], required=False, default='lake'), 'land_use_class': FieldInfo(annotation=str, required=False, default='LU_WATER'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='LAKE'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='VEG_WATER')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.gr4jcn.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'LU_ALL'*, *veg_class: str = 'VEG_ALL'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: Literal['land'] = 'land'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['land']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['land'], required=False, default='land'), 'land_use_class': FieldInfo(annotation=str, required=False, default='LU_ALL'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='VEG_ALL')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.gr4jcn.**P**(*GR4J_X1: Variable | Expression | float | None = Variable('GR4J_X1')*, *GR4J_X2: Variable | Expression | float | None = Variable('GR4J_X2')*, *GR4J_X3: Variable | Expression | float | None = Variable('GR4J_X3')*, *GR4J_X4: Variable | Expression | float | None = Variable('GR4J_X4')*, *CEMANEIGE_X1: Variable | Expression | float | None = Variable('CEMANEIGE_X1')*, *CEMANEIGE_X2: Variable | Expression | float | None = Variable('CEMANEIGE_X2')*)

Bases: Params

CEMANEIGE_X1: Variable | Expression | float | None = Variable('CEMANEIGE_X1')

CEMANEIGE_X2: Variable | Expression | float | None = Variable('CEMANEIGE_X2')

GR4J_X1: Variable | Expression | float | None = Variable('GR4J_X1')

GR4J_X2: Variable | Expression | float | None = Variable('GR4J_X2')

GR4J_X3: Variable | Expression | float | None = Variable('GR4J_X3')

GR4J_X4: Variable | Expression | float | None = Variable('GR4J_X4')

ravenpy.config.emulators.hbvec module

class ravenpy.config.emulators.hbvec.**HBVEC**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: P = P(X01=Variable('X01'), X02=Variable('X02'), X03=Variable('X03'), X04=Variable('X04'), X05=Variable('X05'), X06=Variable('X06'), X07=Variable('X07'), X08=Variable('X08'), X09=Variable('X09'), X10=Variable('X10'), X11=Variable('X11'), X12=Variable('X12'), X13=Variable('X13'), X14=Variable('X14'), X15=Variable('X15'), X16=Variable('X16'), X17=Variable('X17'), X18=Variable('X18'), X19=Variable('X19'), X20=Variable('X20'), X21=Variable('X21'))*, *SoilClasses: SoilClasses = [{'mineral': (1, 0, 0), 'name': 'TOPSOIL', 'organic': 0}, {'mineral': (1, 0, 0), 'name': 'SLOW_RES', 'organic': 0}, {'mineral': (1, 0, 0), 'name': 'FAST_RES', 'organic': 0}]*, *SoilProfiles: SoilProfiles = [{'name': 'DEFAULT_P', 'soil_classes': ['TOPSOIL', 'FAST_RES', 'SLOW_RES'], 'thicknesses': (Variable('X17'), 100, 100)}]*, *VegetationClasses: VegetationClasses = [{'max_ht': 25, 'max_lai': 6, 'max_leaf_cond': 5.3, 'name': 'VEG_ALL'}]*, *LandUseClasses: LandUseClasses = [{'forest_coverage': 1, 'impermeable_frac': 0, 'name': 'LU_ALL'}]*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList = {'parameters': ['POROSITY', 'FIELD_CAPACITY', 'SAT_WILT', 'HBV_BETA', 'MAX_CAP_RISE_RATE', 'MAX_PERC_RATE', 'BASEFLOW_COEFF', 'BASEFLOW_N'], 'pl': [ParameterList(name='[DEFAULT]', values=(Variable('X05'), Variable('X06'), Variable('X14'), Variable('X07'), Variable('X16'), 0.0, 0.0, 0.0)), ParameterList(name='FAST_RES', values=('_DEFAULT', '_DEFAULT', 0.0, '_DEFAULT', '_DEFAULT', Variable('X08'), Variable('X09'), Sum((Variable('X15'), 1)))), ParameterList(name='SLOW_RES', values=('_DEFAULT', '_DEFAULT', 0.0, '_DEFAULT', '_DEFAULT', '_DEFAULT', Variable('X10'), 1.0))]}*, *LandUseParameterList: LandUseParameterList = {'parameters': ['MELT_FACTOR', 'MIN_MELT_FACTOR', 'HBV_MELT_FOR_CORR', 'REFREEZE_FACTOR', 'HBV_MELT_ASP_CORR', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'GLAC_STORAGE_COEFF', 'HBV_GLACIER_AG'], 'pl': [ParameterList(name='[DEFAULT]', values=(Variable('X02'), 2.2, Variable('X18'), Variable('X03'), 0.48, 1.64, 0.05, Variable('X19'), 0.05))]}*, *VegetationParameterList: VegetationParameterList = {'parameters': ['MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'TFRAIN', 'TFSNOW'], 'pl': [ParameterList(name='VEG_ALL', values=(10000.0, 10000.0, 0.88, 0.88))]}*, *ChannelProfile:*

Bases: *Config*

Hydrologiska Byråns Vattenbalansavdelning – Environment Canada (HBV-EC)

References

Lindström, G., et al., 1997. Development and test of the distributed HBV-96 hydrological model. Journal of Hydrology, 201 (1–4), 272–288. doi:10.1016/S0022-1694(97)00041-3

Hamilton, A.S., Hutchinson, D.G., and Moore, R.D., 2000. Estimating winter streamflow using conceptual streamflow model. Journal of Cold Regions Engineering, 14 (4), 158–175. doi:10.1061/(ASCE)0887-381X(2000)14:4(158)

Canadian Hydraulics Centre, 2010. Green kenue reference manual. Ottawa, Ontario: National Research Council.

calendar: Calendar

catchment_route: CatchmentRoute

cloud_cover_method: CloudCoverMethod

evaporation: Evaporation

global_parameter: Dict

hru_state_variable_table: *HRUStateVariableTable*

hrus: HRUs

hydrologic_processes: Sequence[*Process* | Conditional]

lake_storage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE']

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

lw_radiation_method: LWRadiationMethod

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'assimilation_start_time': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2, validate_default=False), 'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'calendar': FieldInfo(annotation=Calendar, required=False, default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2), 'catchment_route': FieldInfo(annotation=CatchmentRoute, required=False, default='TRIANGULAR_UH', alias='CatchmentRoute', alias_priority=2), 'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'cloud_cover_method': FieldInfo(annotation=CloudCoverMethod, required=False, default='CLOUDCOV_NONE', alias='CloudCoverMethod', alias_priority=2), 'custom_output': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput], NoneType], required=False, default_factory=<lambda>, alias='CustomOutput', alias_priority=2, validate_default=False), 'debug_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DebugMode', alias_priority=2, validate_default=False), 'define_hru_groups': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2, validate_default=False), 'deltares_fews_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode', alias_priority=2, validate_default=False), 'direct_evaporation': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2, description='Rainfall is automatically reduced through evapotranspiration up to the limit of the calculated PET.', validate_default=False), 'dont_write_watershed_storage': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage', alias_priority=2, description='Do not write watershed storage variables to disk.', validate_default=False), 'duration': FieldInfo(annotation=Union[float, NoneType], required=False, default_factory=<lambda>, alias='Duration', alias_priority=2, validate_default=False), 'end_date': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'ensemble_mode': FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode', alias_priority=2, validate_default=False), 'evaluation_metrics': FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics], NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics', alias_priority=2, validate_default=False), 'evaluation_period': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod], NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod', alias_priority=2, validate_default=False), 'evaporation': FieldInfo(annotation=Evaporation, required=False, default='PET_FROMMONTHLY', alias='Evaporation', alias_priority=2), 'extra_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='ExtraRVTFilename', alias_priority=2, validate_default=False),

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

monthly_interpolation_method: MonthlyInterpolationMethod

netcdf_attribute: Dict[str, str]

oro_pet_correct: OroPETCorrect

oro_precip_correct: OroPrecipCorrect

oro_temp_correct: OroTempCorrect

ow_evaporation: Evaporation

params: P

potential_melt_method: PotentialMeltMethod

precip_icept_frac: PrecipIceptFract

rain_snow_fraction: RainSnowFraction

rain_snow_transition: *RainSnowTransition*

routing: Routing

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basin_properties: *SubBasinProperties*

sub_basins: *SubBasins*

sw_canopy_correct: SWCanopyCorrect

sw_cloud_correct: SWCloudCorrect

sw_radiation_method: SWRadiationMethod

time_step: float | str

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: bool

class ravenpy.config.emulators.hbvec.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.emulators.hbvec.LandHRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[LandHRU]

class ravenpy.config.emulators.hbvec.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'LU_ALL'*, *veg_class: str = 'VEG_ALL'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: Literal['land'] = 'land'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['land']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['land'], required=False, default='land'), 'land_use_class': FieldInfo(annotation=str, required=False, default='LU_ALL'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='VEG_ALL')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.hbvec.P(*X01: Variable | Expression | float | None = Variable('X01')*, *X02: Variable | Expression | float | None = Variable('X02')*, *X03: Variable | Expression | float | None = Variable('X03')*, *X04: Variable | Expression | float | None = Variable('X04')*, *X05: Variable | Expression | float | None = Variable('X05')*, *X06: Variable | Expression | float | None = Variable('X06')*, *X07: Variable | Expression | float | None = Variable('X07')*, *X08: Variable | Expression | float | None = Variable('X08')*, *X09: Variable | Expression | float | None = Variable('X09')*, *X10: Variable | Expression | float | None = Variable('X10')*, *X11: Variable | Expression | float | None = Variable('X11')*, *X12: Variable | Expression | float | None = Variable('X12')*, *X13: Variable | Expression | float | None = Variable('X13')*, *X14: Variable | Expression | float | None = Variable('X14')*, *X15: Variable | Expression | float | None = Variable('X15')*, *X16: Variable | Expression | float | None = Variable('X16')*, *X17: Variable | Expression | float | None = Variable('X17')*, *X18: Variable | Expression | float | None = Variable('X18')*, *X19: Variable | Expression | float | None = Variable('X19')*, *X20: Variable | Expression | float | None = Variable('X20')*, *X21: Variable | Expression | float | None = Variable('X21')*)

Bases: Params

X01: Variable | Expression | float | None = Variable('X01')

X02: Variable | Expression | float | None = Variable('X02')

X03: Variable | Expression | float | None = Variable('X03')

X04: Variable | Expression | float | None = Variable('X04')

X05: Variable | Expression | float | None = Variable('X05')

X06: Variable | Expression | float | None = Variable('X06')

X07: Variable | Expression | float | None = Variable('X07')

X08: Variable | Expression | float | None = Variable('X08')

X09: Variable | Expression | float | None = Variable('X09')

X10: Variable | Expression | float | None = Variable('X10')

X11: Variable | Expression | float | None = Variable('X11')

X12: Variable | Expression | float | None = Variable('X12')

X13: Variable | Expression | float | None = Variable('X13')

X14: Variable | Expression | float | None = Variable('X14')

X15: Variable | Expression | float | None = Variable('X15')

X16: Variable | Expression | float | None = Variable('X16')

X17: Variable | Expression | float | None = Variable('X17')

X18: Variable | Expression | float | None = Variable('X18')

X19: Variable | Expression | float | None = Variable('X19')

X20: Variable | Expression | float | None = Variable('X20')

X21: Variable | Expression | float | None = Variable('X21')

ravenpy.config.emulators.hmets module

class ravenpy.config.emulators.hmets.**ForestHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'FOREST'*, *veg_class: str = 'FOREST'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: Literal['land'] = 'land'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['land']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['land'], required=False, default='land'), 'land_use_class': FieldInfo(annotation=str, required=False, default='FOREST'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='FOREST')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.hmets.**HMETS**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: P = P(GAMMA_SHAPE=Variable('GAMMA_SHAPE'), GAMMA_SCALE=Variable('GAMMA_SCALE'), GAMMA_SHAPE2=Variable('GAMMA_SHAPE2'), GAMMA_SCALE2=Variable('GAMMA_SCALE2'), MIN_MELT_FACTOR=Variable('MIN_MELT_FACTOR'), MAX_MELT_FACTOR=Variable('MAX_MELT_FACTOR'), DD_MELT_TEMP=Variable('DD_MELT_TEMP'), DD_AGGRADATION=Variable('DD_AGGRADATION'), SNOW_SWI_MIN=Variable('SNOW_SWI_MIN'), SNOW_SWI_MAX=Variable('SNOW_SWI_MAX'), SWI_REDUCT_COEFF=Variable('SWI_REDUCT_COEFF'), DD_REFREEZE_TEMP=Variable('DD_REFREEZE_TEMP'), REFREEZE_FACTOR=Variable('REFREEZE_FACTOR'), REFREEZE_EXP=Variable('REFREEZE_EXP'), PET_CORRECTION=Variable('PET_CORRECTION'), HMETS_RUNOFF_COEFF=Variable('HMETS_RUNOFF_COEFF'), PERC_COEFF=Variable('PERC_COEFF'), BASEFLOW_COEFF_1=Variable('BASEFLOW_COEFF_1'), BASEFLOW_COEFF_2=Variable('BASEFLOW_COEFF_2'), TOPSOIL=Variable('TOPSOIL'), PHREATIC=Variable('PHREATIC'))*, *SoilClasses: SoilClasses = [{'name': 'TOPSOIL'}, {'name': 'PHREATIC'}]*, *soil_profiles: SoilProfiles = [{'name': 'LAKE'}, {'name': 'ROCK'}, {'name': 'DEFAULT_P', 'soil_classes': ('TOPSOIL', 'PHREATIC'), 'thicknesses': (Quotient(Variable('TOPSOIL'), 1000.0), Quotient(Variable('PHREATIC'), 1000.0))}]*, *VegetationClasses: VegetationClasses = [{'max_ht': 4, 'max_lai': 5, 'max_leaf_cond': 5, 'name': 'FOREST'}]*, *LandUseClasses: LandUseClasses = [{'forest_coverage': 1, 'name': 'FOREST'}]*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList = {'parameters': ('POROSITY', 'PERC_COEFF', 'PET_CORRECTION', 'BASEFLOW_COEFF'), 'pl': [ParameterList(name='TOPSOIL', values=(1.0, Variable('PERC_COEFF'), Variable('PET_CORRECTION'), Variable('BASEFLOW_COEFF_1'))), ParameterList(name='PHREATIC', values=(1.0, 0.0, 0.0, Variable('BASEFLOW_COEFF_2')))]}*, *LandUseParameterList: LandUseParameterList = {'parameters': ['MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'DD_MELT_TEMP', 'DD_AGGRADATION', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_REFREEZE_TEMP', 'HMETS_RUNOFF_COEFF', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'GAMMA_SHAPE2', 'GAMMA_SCALE2'], 'pl':*

Bases: *Config*

Hydrology Model - École de technologie supérieure (HMETS)

References

Martel, J.-L., Demeester, K., Brissette, F., Arsenault, R., Poulin, A. 2017. HMETS: A simple and efficient hydrology model for teaching hydrological modelling, flow forecasting and climate change impacts. Int. J. Eng. Educ., 33, 1307–1316.

calendar: Calendar

catchment_route: CatchmentRoute

evaporation: Evaporation

global_parameter: Dict[str, Variable | Expression | float | None]

hrus: HRUs

hydrologic_processes: Sequence[*Process*]

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow],
NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow',
alias_priority=2, validate_default=False), 'assimilated_state':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState],
NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState',
alias_priority=2, validate_default=False), 'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'basin_state_variables':
FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False,
default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Calendar, required=False,
default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2),
'catchment_route':  FieldInfo(annotation=CatchmentRoute, required=False,
default='ROUTE_DUMP', alias='CatchmentRoute', alias_priority=2), 'channel_profile':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile],
NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile',
alias_priority=2, validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'enkf_mode':
FieldInfo(annotation=Union[EnKFMode, NoneType], required=False,
default_factory=<lambda>, alias='EnKFMode', alias_priority=2,
validate_default=False), 'ensemble_mode':  FieldInfo(annotation=Union[EnsembleMode,
NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode',
alias_priority=2, validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Evaporation, required=False, default='PET_OUDIN',
alias='Evaporation', alias_priority=2), 'extra_rvt_filename':
FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>,
alias='ExtraRVTFilename', alias_priority=2, validate_default=False),
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: `Dict[str, str]`

params: `P`

potential_melt_method: `PotentialMeltMethod`

rain_snow_fraction: `RainSnowFraction`

routing: `Routing`

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basins: *SubBasins*

time_step: `float | str`

uniform_initial_conditions: `Dict[str, Variable | Expression | float | None]`

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: `bool`

class ravenpy.config.emulators.hmets.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for HMETS.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.emulators.hmets.ForestHRU], required=True)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[ForestHRU]

class ravenpy.config.emulators.hmets.**P**(*GAMMA_SHAPE: Variable | Expression | float | None = Variable('GAMMA_SHAPE')*, *GAMMA_SCALE: Variable | Expression | float | None = Variable('GAMMA_SCALE')*, *GAMMA_SHAPE2: Variable | Expression | float | None = Variable('GAMMA_SHAPE2')*, *GAMMA_SCALE2: Variable | Expression | float | None = Variable('GAMMA_SCALE2')*, *MIN_MELT_FACTOR: Variable | Expression | float | None = Variable('MIN_MELT_FACTOR')*, *MAX_MELT_FACTOR: Variable | Expression | float | None = Variable('MAX_MELT_FACTOR')*, *DD_MELT_TEMP: Variable | Expression | float | None = Variable('DD_MELT_TEMP')*, *DD_AGGRADATION: Variable | Expression | float | None = Variable('DD_AGGRADATION')*, *SNOW_SWI_MIN: Variable | Expression | float | None = Variable('SNOW_SWI_MIN')*, *SNOW_SWI_MAX: Variable | Expression | float | None = Variable('SNOW_SWI_MAX')*, *SWI_REDUCT_COEFF: Variable | Expression | float | None = Variable('SWI_REDUCT_COEFF')*, *DD_REFREEZE_TEMP: Variable | Expression | float | None = Variable('DD_REFREEZE_TEMP')*, *REFREEZE_FACTOR: Variable | Expression | float | None = Variable('REFREEZE_FACTOR')*, *REFREEZE_EXP: Variable | Expression | float | None = Variable('REFREEZE_EXP')*, *PET_CORRECTION: Variable | Expression | float | None = Variable('PET_CORRECTION')*, *HMETS_RUNOFF_COEFF: Variable | Expression | float | None = Variable('HMETS_RUNOFF_COEFF')*, *PERC_COEFF: Variable | Expression | float | None = Variable('PERC_COEFF')*, *BASEFLOW_COEFF_1: Variable | Expression | float | None = Variable('BASEFLOW_COEFF_1')*, *BASEFLOW_COEFF_2: Variable | Expression | float | None = Variable('BASEFLOW_COEFF_2')*, *TOPSOIL: Variable | Expression | float | None = Variable('TOPSOIL')*, *PHREATIC: Variable | Expression | float | None = Variable('PHREATIC')*)

Bases: Params

BASEFLOW_COEFF_1: Variable | Expression | float | None = Variable('BASEFLOW_COEFF_1')

BASEFLOW_COEFF_2: Variable | Expression | float | None = Variable('BASEFLOW_COEFF_2')

DD_AGGRADATION: Variable | Expression | float | None = Variable('DD_AGGRADATION')

DD_MELT_TEMP: Variable | Expression | float | None = Variable('DD_MELT_TEMP')

DD_REFREEZE_TEMP: Variable | Expression | float | None = Variable('DD_REFREEZE_TEMP')

GAMMA_SCALE: Variable | Expression | float | None = Variable('GAMMA_SCALE')

GAMMA_SCALE2: Variable | Expression | float | None = Variable('GAMMA_SCALE2')

GAMMA_SHAPE: Variable | Expression | float | None = Variable('GAMMA_SHAPE')

GAMMA_SHAPE2: Variable | Expression | float | None = Variable('GAMMA_SHAPE2')

HMETS_RUNOFF_COEFF: Variable | Expression | float | None = Variable('HMETS_RUNOFF_COEFF')

MAX_MELT_FACTOR: Variable | Expression | float | None = Variable('MAX_MELT_FACTOR')

MIN_MELT_FACTOR: Variable | Expression | float | None = Variable('MIN_MELT_FACTOR')

PERC_COEFF: Variable | Expression | float | None = Variable('PERC_COEFF')

PET_CORRECTION: Variable | Expression | float | None = Variable('PET_CORRECTION')

PHREATIC: Variable | Expression | float | None = Variable('PHREATIC')

REFREEZE_EXP: Variable | Expression | float | None = Variable('REFREEZE_EXP')

REFREEZE_FACTOR: Variable | Expression | float | None = Variable('REFREEZE_FACTOR')

SNOW_SWI_MAX: Variable | Expression | float | None = Variable('SNOW_SWI_MAX')

SNOW_SWI_MIN: Variable | Expression | float | None = Variable('SNOW_SWI_MIN')

SWI_REDUCT_COEFF: Variable | Expression | float | None = Variable('SWI_REDUCT_COEFF')

TOPSOIL: Variable | Expression | float | None = Variable('TOPSOIL')

ravenpy.config.emulators.hypr module

class ravenpy.config.emulators.hypr.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.emulators.hypr.LandHRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[LandHRU]

class ravenpy.config.emulators.hypr.**HYPR**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: Params = Params(X01=Variable('X01'), X02=Variable('X02'), X03=Variable('X03'), X04=Variable('X04'), X05=Variable('X05'), X06=Variable('X06'), X07=Variable('X07'), X08=Variable('X08'), X09=Variable('X09'), X10=Variable('X10'), X11=Variable('X11'), X12=Variable('X12'), X13=Variable('X13'), X14=Variable('X14'), X15=Variable('X15'), X16=Variable('X16'), X17=Variable('X17'), X18=Variable('X18'), X19=Variable('X19'), X20=Variable('X20'), X21=Variable('X21'))*, *SoilClasses: SoilClasses = [{'name': 'TOPSOIL'}, {'name': 'SLOW_RES'}, {'name': 'FAST_RES'}]*, *SoilProfiles: SoilProfiles = [{'name': 'DEFAULT_P', 'soil_classes': ('TOPSOIL', 'FAST_RES', 'SLOW_RES'), 'thicknesses': (Variable('X11'), 1e+99, 1e+99)}]*, *VegetationClasses: VegetationClasses = [{'max_ht': 0, 'max_lai': 0, 'max_leaf_cond': 1e+99, 'name': 'FOREST'}]*, *LandUseClasses: LandUseClasses = [LandUseClass(name='OPEN_1', impermeable_frac=0.0, forest_coverage=0.0)]*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList = {'parameters': ('POROSITY', 'FIELD_CAPACITY', 'SAT_WILT', 'HBV_BETA', 'MAX_CAP_RISE_RATE', 'MAX_PERC_RATE', 'BASEFLOW_COEFF', 'BASEFLOW_N', 'BASEFLOW_COEFF2', 'STORAGE_THRESHOLD'), 'pl': [ParameterList(name='[DEFAULT]', values=(1.0, Variable('X04'), 0.0, Variable('X16'), 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)), ParameterList(name='FAST_RES', values=('_DEFAULT', '_DEFAULT', 0.0, '_DEFAULT', '_DEFAULT', 0.0, Variable('X06'), 1.0, Variable('X05'), Variable('X10'))), ParameterList(name='SLOW_RES', values=('_DEFAULT', '_DEFAULT', 0.0, '_DEFAULT', '_DEFAULT', '_DEFAULT', 0.01, 1.0, 0.05, 0.0))]}*, *LandUseParameterList: LandUseParameterList = {'parameters': ('MELT_FACTOR', 'MIN_MELT_FACTOR', 'HBV_MELT_FOR_CORR', 'REFREEZE_FACTOR', 'HBV_MELT_ASP_CORR', 'DD_MELT_TEMP', 'FOREST_COVERAGE', 'PONDED_EXP', 'PDMROF_B', 'DEP_MAX', 'MAX_DEP_AREA_FRAC'), 'pl': [ParameterList(name='[DEFAULT]', values=(Variable('X17'), Variable('X18'), Variable('X13'), Variable('X12'), 0.0, Variable('X01'), 0.0, Variable('X09'), Variable('X07'), Variable('X14'), Variable('X08')))]}*, *VegetationParameterList: VegetationParameterList = {'parameters': ('SAI_HT_RATIO', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'TFRAIN', 'TFSNOW'), 'pl': [ParameterList(name='[DEFAULT]', values=(0.0, 10000.0, 10000.0, 1.0, 1.0))]}*, *ChannelProfile: Sequence[ChannelProfile] | None = None*, *global_parameter: Dict = {'AdiabaticLapseRate': Variable('X19'), 'PrecipLapse':*

Bases: *Config*

The HYdrological model for Prairie Region (HYPR) is based on the conceptual Hydrologiska Byrans Vattenbalansavdelning (HBV)-light model. HBV is modified to work in the prairies by incorporating a conceptual lateral-flow component to represent the pothole storage complexities. HYPR can be used for prairie streamflow simulation.

References

Mohamed I. Ahmed, Amin Elshorbagy, and Alain Pietroniro (2020). Toward Simple Modeling Practices in the Complex Canadian Prairie Watersheds. Journal of Hydrologic Engineering, 25(6), 04020024. doi: 10.1061/(ASCE)HE.1943-5584.0001922.

calendar: Calendar

catchment_route: CatchmentRoute

cloud_cover_method: CloudCoverMethod

evaporation: Evaporation

global_parameter: Dict

hru_state_variable_table: *HRUStateVariableTable*

hrus: HRUs

hydrologic_processes: Sequence[*Process* | Conditional]

lake_storage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE']

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

lw_radiation_method: LWRadiationMethod

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'assimilation_start_time': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2, validate_default=False), 'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'calendar': FieldInfo(annotation=Calendar, required=False, default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2), 'catchment_route': FieldInfo(annotation=CatchmentRoute, required=False, default='ROUTE_TRI_CONVOLUTION', alias='CatchmentRouting', alias_priority=2), 'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'cloud_cover_method': FieldInfo(annotation=CloudCoverMethod, required=False, default=<CloudCoverMethod.NONE: 'CLOUDCOV_NONE'>, alias='CloudCoverMethod', alias_priority=2), 'custom_output': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput], NoneType], required=False, default_factory=<lambda>, alias='CustomOutput', alias_priority=2, validate_default=False), 'debug_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DebugMode', alias_priority=2, validate_default=False), 'define_hru_groups': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2, validate_default=False), 'deltares_fews_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode', alias_priority=2, validate_default=False), 'direct_evaporation': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2, description='Rainfall is automatically reduced through evapotranspiration up to the limit of the calculated PET.', validate_default=False), 'dont_write_watershed_storage': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage', alias_priority=2, description='Do not write watershed storage variables to disk.', validate_default=False), 'duration': FieldInfo(annotation=Union[float, NoneType], required=False, default_factory=<lambda>, alias='Duration', alias_priority=2, validate_default=False), 'end_date': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'ensemble_mode': FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode', alias_priority=2, validate_default=False), 'evaluation_metrics': FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics], NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics', alias_priority=2, validate_default=False), 'evaluation_period': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod], NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod', alias_priority=2, validate_default=False), 'evaporation': FieldInfo(annotation=Evaporation, required=False, default=<Evaporation.FROMMONTHLY: 'PET_FROMMONTHLY'>, alias='Evaporation', alias_priority=2), 'extra_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>,

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: Dict[str, str]

ow_evaporation: Evaporation

params: Params

potential_melt_method: PotentialMeltMethod

precip_icept_frac: PrecipIceptFract

rain_snow_fraction: RainSnowFraction

routing: Routing

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basin_properties: *SubBasinProperties*

sub_basins: *SubBasins*

sw_canopy_correct: SWCanopyCorrect

sw_cloud_correct: SWCloudCorrect

sw_radiation_method: SWRadiationMethod

time_step: float | str

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: bool

class ravenpy.config.emulators.hypr.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'OPEN_1'*, *veg_class: str = 'FOREST'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Bases: *HRU*

aquifer_profile: str

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='OPEN_1'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='FOREST')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

ravenpy.config.emulators.mohyse module

class ravenpy.config.emulators.mohyse.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.emulators.mohyse.LandHRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[LandHRU]

class ravenpy.config.emulators.mohyse.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'LU_ALL'*, *veg_class: str = 'VEG_ALL'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: Literal['land'] = 'land'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['land']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['land'], required=False, default='land'), 'land_use_class': FieldInfo(annotation=str, required=False, default='LU_ALL'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='VEG_ALL')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.mohyse.**Mohyse**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: P = P(X01=Variable('X01'), X02=Variable('X02'), X03=Variable('X03'), X04=Variable('X04'), X05=Variable('X05'), X06=Variable('X06'), X07=Variable('X07'), X08=Variable('X08'), X09=Variable('X09'), X10=Variable('X10'))*, *SoilClasses: SoilClasses = [{'name': 'TOPSOIL'}, {'name': 'GWSOIL'}]*, *SoilProfiles: SoilProfiles = [{'name': 'LAKE'}, {'name': 'ROCK'}, {'name': 'DEFAULT_P', 'soil_classes': ['TOPSOIL', 'GWSOIL'], 'thicknesses': [Variable('X05'), 10.0]}]*, *VegetationClasses: VegetationClasses = [{'name': 'VEG_ALL'}]*, *LandUseClasses: LandUseClasses = [{'forest_coverage': 1, 'impermeable_frac': 0, 'name': 'LU_ALL'}]*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList = {'parameters': ['POROSITY', 'PET_CORRECTION', 'HBV_BETA', 'BASEFLOW_COEFF', 'PERC_COEFF'], 'pl': [ParameterList(name='TOPSOIL', values=(1.0, 1.0, 1.0, Variable('X07'), Variable('X06'))), ParameterList(name='GWSOIL', values=(1.0, 1.0, 1.0, Variable('X08'), 0.0))]}*, *LandUseParameterList: LandUseParameterList = {'parameters': ['MELT_FACTOR', 'AET_COEFF', 'FOREST_SPARSENESS', 'DD_MELT_TEMP'], 'pl': [ParameterList(name='[DEFAULT]', values=(Variable('X03'), Variable('X02'), 0.0, Variable('X04')))]}*, *VegetationParameterList: VegetationParameterList = {'parameters': ['SAI_HT_RATIO', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT'], 'pl': [ParameterList(name='[DEFAULT]', values=(0.0, 0.0, 0.0))]}*, *ChannelProfile: Sequence[ChannelProfile] | None = None*, *GlobalParameter: Dict = {'MOHYSE_PET_COEFF': Variable('X01'), 'RAINSNOW_TEMP': -2, 'TOC_MULTIPLIER': 1}*, *RainSnowTransition: RainSnowTransition | None = None*, *SeasonalRelativeLAI: SeasonalRelativeLAI | None = None*, *SeasonalRelativeHeight: SeasonalRelativeHeight | None = None*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*, *SubBasins: SubBasins = [SubBasin(subbasin_id=1, name='sub_001', downstream_id=-1, profile='NONE', reach_length=0, gauged=True, gauge_id='')]*, *SubBasinGroup: Sequence[SubBasinGroup] | None = None*, *SubBasinProperties: SubBasinProperties = {'parameters':*

Bases: *Config*

Modèle Hydrologique Simplifié à l'Extrême (MOHYSE)

References

Fortin, V.; Turcotte, R. Le modèle hydrologique MOHYSE. In Note de Cours Pour SCA7420, Université du Québec à Montréal: Montréal, QC, Canada, 2007; p. 14.

Troin, M., Arsenault, R. and Brissette, F., 2015. Performance and uncertainty evaluation of snow models on snowmelt flow simulations over a Nordic catchment (Mistassibi, Canada). Hydrology, 2(4), pp.289-317.

calendar: Calendar

catchment_route: CatchmentRoute

direct_evaporation: bool

evaporation: Evaporation

global_parameter: Dict

hrus: HRUs

hydrologic_processes: Sequence[*Process*]

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'assimilation_start_time': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2, validate_default=False), 'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'calendar': FieldInfo(annotation=Calendar, required=False, default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2), 'catchment_route': FieldInfo(annotation=CatchmentRoute, required=False, default='ROUTE_GAMMA_CONVOLUTION', alias='CatchmentRoute', alias_priority=2), 'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'cloud_cover_method': FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False, default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2, validate_default=False), 'custom_output': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput], NoneType], required=False, default_factory=<lambda>, alias='CustomOutput', alias_priority=2, validate_default=False), 'debug_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DebugMode', alias_priority=2, validate_default=False), 'define_hru_groups': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2, validate_default=False), 'deltares_fews_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode', alias_priority=2, validate_default=False), 'direct_evaporation': FieldInfo(annotation=bool, required=False, default=True, alias='DirectEvaporation', alias_priority=2), 'dont_write_watershed_storage': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage', alias_priority=2, description='Do not write watershed storage variables to disk.', validate_default=False), 'duration': FieldInfo(annotation=Union[float, NoneType], required=False, default_factory=<lambda>, alias='Duration', alias_priority=2, validate_default=False), 'end_date': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'ensemble_mode': FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode', alias_priority=2, validate_default=False), 'evaluation_metrics': FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics], NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics', alias_priority=2, validate_default=False), 'evaluation_period': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod], NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod', alias_priority=2, validate_default=False), 'evaporation': FieldInfo(annotation=Evaporation, required=False, default='PET_MOHYSE', alias='Evaporation', alias_priority=2), 'extra_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='ExtraRVTFilename', alias_priority=2, validate_default=False), 'forcing_perturbation':

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: `Dict[str, str]`

params: `P`

potential_melt_method: `PotentialMeltMethod`

rain_snow_fraction: `RainSnowFraction`

routing: `Routing`

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basin_properties: *SubBasinProperties*

sub_basins: *SubBasins*

time_step: `float | str`

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: `bool`

class ravenpy.config.emulators.mohyse.**P**(*X01: Variable | Expression | float | None = Variable('X01')*, *X02: Variable | Expression | float | None = Variable('X02')*, *X03: Variable | Expression | float | None = Variable('X03')*, *X04: Variable | Expression | float | None = Variable('X04')*, *X05: Variable | Expression | float | None = Variable('X05')*, *X06: Variable | Expression | float | None = Variable('X06')*, *X07: Variable | Expression | float | None = Variable('X07')*, *X08: Variable | Expression | float | None = Variable('X08')*, *X09: Variable | Expression | float | None = Variable('X09')*, *X10: Variable | Expression | float | None = Variable('X10')*)

Bases: Params

X01: `Variable | Expression | float | None = Variable('X01')`

X02: `Variable | Expression | float | None = Variable('X02')`

X03: `Variable | Expression | float | None = Variable('X03')`

X04: `Variable | Expression | float | None = Variable('X04')`

X05: `Variable | Expression | float | None = Variable('X05')`

X06: `Variable | Expression | float | None = Variable('X06')`

```
X07:  Variable | Expression | float | None = Variable('X07')

X08:  Variable | Expression | float | None = Variable('X08')

X09:  Variable | Expression | float | None = Variable('X09')

X10:  Variable | Expression | float | None = Variable('X10')
```

ravenpy.config.emulators.routing module

class ravenpy.config.emulators.routing.**BasicRoute**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: Any = None*, *SoilClasses:* SoilClasses | *None = None*, *SoilProfiles:* SoilProfiles | *None = None*, *VegetationClasses:* VegetationClasses | *None = None*, *LandUseClasses:* LandUseClasses | *None = None*, *TerrainClasses:* TerrainClasses | *None = None*, *SoilParameterList:* SoilParameterList | *None = None*, *LandUseParameterList:* LandUseParameterList | *None = None*, *VegetationParameterList:* VegetationParameterList | *None = None*, *ChannelProfile: Sequence[ChannelProfile] | None = None*, *GlobalParameter: Dict[str, Variable | Expression | float | None] | None = {}*, *RainSnowTransition:* RainSnowTransition | *None = None*, *SeasonalRelativeLAI:* SeasonalRelativeLAI | *None = None*, *SeasonalRelativeHeight:* SeasonalRelativeHeight | *None = None*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*, *SubBasins:* SubBasins = *[SubBasin(subbasin_id=1, name='sub_001', downstream_id=-1, profile='NONE', reach_length=0, gauged=True, gauge_id='')]*, *SubBasinGroup: Sequence[SubBasinGroup] | None = None*, *SubBasinProperties:* SubBasinProperties | *None = None*, *SBGroupPropertyMultiplier: Sequence[SBGroupPropertyMultiplier] | None = None*, *HRUs: HRUs = [LandHRU(hru_id=1, area=0, elevation=0, latitude=0, longitude=0, subbasin_id=1, land_use_class='Landuse_Land_HRU', veg_class='Veg_Land_HRU', soil_profile='Soil_Land_HRU', aquifer_profile='[NONE]', terrain_class='[NONE]', slope=0.0, aspect=0.0, hru_type='land')]*, *HRUGroup: Sequence[HRUGroup] | None = None*, *Reservoirs: Sequence[Reservoir] | None = None*, *HRUStateVariableTable:* HRUStateVariableTable | *None = None*, *BasinStateVariables:* BasinStateVariables | *None = None*, *UniformInitialConditions: Dict[str, Variable | Expression | float | None] | None = None*,

Bases: *Config*

Raven configuration performing routing only.

calendar: Calendar

catchment_route: CatchmentRoute

hrus: HRUs

hydrologic_processes: Sequence[*Process*]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow],
NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow',
alias_priority=2, validate_default=False), 'assimilated_state':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState],
NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState',
alias_priority=2, validate_default=False), 'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'basin_state_variables':
FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False,
default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Calendar, required=False,
default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2),
'catchment_route':  FieldInfo(annotation=CatchmentRoute, required=False,
default='ROUTE_DUMP', alias='CatchmentRoute', alias_priority=2,
description='Catchment routing method, used to convey water from the catchment
tributaries and rivulets to the subbasin outlets.'), 'channel_profile':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile],
NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile',
alias_priority=2, validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'enkf_mode':
FieldInfo(annotation=Union[EnKFMode, NoneType], required=False,
default_factory=<lambda>, alias='EnKFMode', alias_priority=2,
validate_default=False), 'ensemble_mode':  FieldInfo(annotation=Union[EnsembleMode,
NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode',
alias_priority=2, validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Union[Evaporation, NoneType], required=False,
default_factory=<lambda>, alias='Evaporation', alias_priority=2,
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: `Dict[str, str]`

potential_melt_method: `PotentialMeltMethod`

precip_icept_frac: `PrecipIceptFract`

routing: `Routing`

soil_model: *SoilModel*

sub_basins: *SubBasins*

time_step: `float | str`

write_netcdf_format: `bool`

class ravenpy.config.emulators.routing.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[Union[ravenpy.config.emulators.routing.LandHRU, ravenpy.config.emulators.routing.LakeHRU]], required=True)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: `Sequence[LandHRU | LakeHRU]`

class ravenpy.config.emulators.routing.**LakeHRU**(**, hru_id: int = 1, area: Variable | Expression | float | None = 0, elevation: float = 0, latitude: float = 0, longitude: float = 0, subbasin_id: int = 1, land_use_class: str = 'Landuse_Lake_HRU', veg_class: str = 'Veg_Lake_HRU', soil_profile: str = 'Soil_Lake_HRU', aquifer_profile: str = '[NONE]', terrain_class: str = '[NONE]', slope: float = 0.0, aspect: float = 0.0, hru_type: Literal['lake'] = 'lake'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['lake']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['lake'], required=False, default='lake'), 'land_use_class': FieldInfo(annotation=str, required=False, default='Landuse_Lake_HRU'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='Soil_Lake_HRU'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='Veg_Lake_HRU')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.routing.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'Landuse_Land_HRU'*, *veg_class: str = 'Veg_Land_HRU'*, *soil_profile: str = 'Soil_Land_HRU'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: Literal['land'] = 'land'*)

Bases: *HRU*

aquifer_profile: str

hru_type: Literal['land']

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Literal['land'], required=False, default='land'), 'land_use_class': FieldInfo(annotation=str, required=False, default='Landuse_Land_HRU'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='Soil_Land_HRU'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='Veg_Land_HRU')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

ravenpy.config.emulators.sacsma module

class ravenpy.config.emulators.sacsma.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: *HRUs*

HRUs command for GR4J.

Pydantic is able to automatically detect if an HRU is Land or Lake if *hru_type* is provided.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.emulators.sacsma.LandHRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

root: Sequence[LandHRU]

class ravenpy.config.emulators.sacsma.**LandHRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = 'FOREST'*, *veg_class: str = 'FOREST'*, *soil_profile: str = 'DEFAULT_P'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Bases: *HRU*

aquifer_profile: str

land_use_class: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='FOREST'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='DEFAULT_P'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='FOREST')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

soil_profile: str

terrain_class: str

veg_class: str

class ravenpy.config.emulators.sacsma.**SACSMA**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: Params = Params(X01=Variable('X01'), X02=Variable('X02'), X03=Variable('X03'), X04=Variable('X04'), X05=Variable('X05'), X06=Variable('X06'), X07=Variable('X07'), X08=Variable('X08'), X09=Variable('X09'), X10=Variable('X10'), X11=Variable('X11'), X12=Variable('X12'), X13=Variable('X13'), X14=Variable('X14'), X15=Variable('X15'), X16=Variable('X16'), X17=Variable('X17'), X18=Variable('X18'), X19=Variable('X19'), X20=Variable('X20'), X21=Variable('X21'))*, *SoilClasses: SoilClasses = [{'name': 'UZT'}, {'name': 'UZF'}, {'name': 'LZT'}, {'name': 'LZFP'}, {'name': 'LZFS'}, {'name': 'ADIM'}, {'name': 'GW'}]*, *SoilProfiles: SoilProfiles = [{'name': 'LAKE'}, {'name': 'DEFAULT_P', 'soil_classes': ('UZT', 'UZF', 'LZT', 'LZFP', 'LZFS', 'ADIM', 'GW'), 'thicknesses': (Variable('X04'), Variable('X05'), Variable('X06'), Variable('X08'), Variable('X07'), 100, 100)}]*, *VegetationClasses: VegetationClasses = [{'max_ht': 4, 'max_lai': 5, 'max_leaf_cond': 5, 'name': 'FOREST'}]*, *LandUseClasses: LandUseClasses = [LandUseClass(name='FOREST', impermeable_frac=Variable('X13'), forest_coverage=1.0)]*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList = {'parameters': ('POROSITY', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'BASEFLOW_COEFF', 'UNAVAIL_FRAC'), 'pl': [ParameterList(name='[DEFAULT]', values=(1.0, Variable('X11'), Variable('X10'), Variable('X09'), 0.0, 0.0)), ParameterList(name='UZF', values=(1.0, Variable('X11'), Variable('X10'), Variable('X09'), Variable('X03'), 0.0)), ParameterList(name='LZFP', values=(1.0, Variable('X11'), Variable('X10'), Variable('X09'), Variable('X01'), Variable('X16'))), ParameterList(name='LZFS', values=(1.0, Variable('X11'), Variable('X10'), Variable('X09'), Variable('X02'), 0.0))]}*, *LandUseParameterList: LandUseParameterList = {'parameters': ('GAMMA_SHAPE', 'GAMMA_SCALE', 'MELT_FACTOR', 'STREAM_FRACTION', 'MAX_SAT_AREA_FRAC', 'BF_LOSS_FRACTION'), 'pl': [ParameterList(name='[DEFAULT]', values=(Variable('X18'), Variable('X19'), Variable('X17'), Variable('X15'), Variable('X14'), Variable('X12'))]}*, *VegetationParameterList: VegetationParameterList = {'parameters': ('RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT'), 'pl':*

Bases: *Config*

Sacramento - Soil Moisture Accounting model

References

Sorooshian, S., Duan, Q., and Gupta, V. K. (1993), Calibration of rainfall-runoff models: Application of global optimization to the Sacramento Soil Moisture Accounting Model, Water Resour. Res., 29(4), 1185– 1194, doi:10.1029/92WR02617.

calendar: Calendar

catchment_route: CatchmentRoute

evaporation: Evaporation

hru_state_variable_table: *HRUStateVariableTable*

hrus: HRUs

hydrologic_processes: Sequence[*Process* | Conditional]

land_use_classes: *LandUseClasses*

land_use_parameter_list: *LandUseParameterList*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow],
NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow',
alias_priority=2, validate_default=False), 'assimilated_state':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState],
NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState',
alias_priority=2, validate_default=False), 'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'basin_state_variables':
FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False,
default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2,
validate_default=False), 'calendar': FieldInfo(annotation=Calendar, required=False,
default='PROLEPTIC_GREGORIAN', alias='Calendar', alias_priority=2),
'catchment_route': FieldInfo(annotation=CatchmentRoute, required=False,
default='ROUTE_GAMMA_CONVOLUTION', alias='CatchmentRouting', alias_priority=2),
'channel_profile':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile],
NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile',
alias_priority=2, validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode': FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage': FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration': FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date': FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'enkf_mode':
FieldInfo(annotation=Union[EnKFMode, NoneType], required=False,
default_factory=<lambda>, alias='EnKFMode', alias_priority=2,
validate_default=False), 'ensemble_mode': FieldInfo(annotation=Union[EnsembleMode,
NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode',
alias_priority=2, validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Evaporation, required=False, default=<Evaporation.OUDIN:
'PET_OUDIN'>, alias='Evaporation', alias_priority=2), 'extra_rvt_filename':
FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>,
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

netcdf_attribute: Dict[str, str]

params: Params

potential_melt_method: PotentialMeltMethod

rain_snow_fraction: RainSnowFraction

routing: Routing

soil_classes: *SoilClasses*

soil_model: *SoilModel*

soil_parameter_list: *SoilParameterList*

soil_profiles: *SoilProfiles*

sub_basins: *SubBasins*

time_step: float | str

vegetation_classes: *VegetationClasses*

vegetation_parameter_list: *VegetationParameterList*

write_netcdf_format: bool

Submodules

ravenpy.config.base module

class ravenpy.config.base.**Command**

Bases: _Command

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class ravenpy.config.base.**FlatCommand**

Bases: Command

Only used to discriminate Commands that should not be nested.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class ravenpy.config.base.**GenericParameterList**(*, *Parameters: Sequence[str]*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

Bases: Command

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Sequence[str], required=True, alias='Parameters', alias_priority=2, description='Parameter names'), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

num_values_equal_num_names()

Check that the length of the parameter list equals the number of given parameter names.

parameters: Sequence[str]

pl: Sequence[ParameterList]

set_default_units()

units: Sequence[str] | None

class ravenpy.config.base.**LineCommand**

Bases: FlatCommand

A non-nested Command on a single line.

:CommandName {field_1} {field_2} … {field_n}

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

to_rv()

Return Raven configuration string.

class ravenpy.config.base.**ListCommand**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootModel, _Command

Use so that commands with __root__: Sequence[Command] behave like a list.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[Any], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[Any]

class ravenpy.config.base.**ParameterList**(*, *name: str = ''*, *values: Sequence[Variable | Expression | float | None | str] = ()*)

Bases: Record

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'name': FieldInfo(annotation=str, required=False, default=''), 'values': FieldInfo(annotation=Sequence[Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType, str]], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

classmethod no_none_in_default(*data*)

Make sure that no values are None for the [DEFAULT] record.

values: Sequence[Variable | Expression | float | None | str]

class ravenpy.config.base.**Params**

Bases: object

class ravenpy.config.base.**RV**

Bases: Command

Base class for RV configuration objects.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class ravenpy.config.base.**Record**

Bases: _Record

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class ravenpy.config.base.**RootCommand**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootModel, _Command

Generic Command for root models.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Any, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Any

class ravenpy.config.base.**RootRecord**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootModel, _Record

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=~RootModelRootType, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class ravenpy.config.base.**SymConfig**

Bases: object

arbitrary_types_allowed = True

ravenpy.config.base.**encoder**(*v: dict*) → dict

Return string representation of objects in dictionary.

This is meant to be applied to BaseModel attributes that either have an *alias* defined, or have a *root* attribute. The objective is to avoid creating *Command* objects for every configuration option:

- bool: ':{cmd}n' if obj else ''
- dict: ':{cmd} {key} {value}'
- enum: ':{cmd} {obj.value}'
- Command: obj.to_rv()
- Sequence: complicated
- Any other: ':{cmd} {obj}'

ravenpy.config.base.**optfield**(***kwds*)

Shortcut to create an optional field with an alias.

ravenpy.config.base.**parse_symbolic**(*value*, ***kwds*)

Inject values of symbolic variables into object and return object.

Note that parsing the output of *model_dump* can cause problems because there is not always enough information in the dictionary to recreate the correct model.

ravenpy.config.commands module

class ravenpy.config.commands.**AssimilateStreamflow**(*, *sb_id: str*)

Bases: LineCommand

Subbasin ID to assimilate streamflow for.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'sb_id': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

sb_id: str

class ravenpy.config.commands.**AssimilatedState**(*, *state: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | Literal['STREAMFLOW']*, *group: str*)

Bases: LineCommand

group: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'group': FieldInfo(annotation=str, required=True), 'state': FieldInfo(annotation=Union[Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'], Literal['STREAMFLOW']], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

state: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | Literal['STREAMFLOW']

class ravenpy.config.commands.**BasinIndex**(*, *sb_id: int = 1*, *name: str = 'watershed'*, *ChannelStorage: float = 0.0*, *RivuletStorage: float = 0.0*, *Qout: Sequence[float] = (1.0, 0.0, 0.0)*, *Qlat: Sequence[float] | None = None*, *Qin: Sequence[float] | None = None*)

Bases: Command

Initial conditions for a flow segment.

channel_storage: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'channel_storage': FieldInfo(annotation=float, required=False, default=0.0, alias='ChannelStorage', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default='watershed'), 'qin': FieldInfo(annotation=Union[Sequence[float], NoneType], required=False, default=None, alias='Qin', alias_priority=2), 'qlat': FieldInfo(annotation=Union[Sequence[float], NoneType], required=False, default=None, alias='Qlat', alias_priority=2), 'qout': FieldInfo(annotation=Sequence[float], required=False, default=(1.0, 0.0, 0.0), alias='Qout', alias_priority=2), 'rivulet_storage': FieldInfo(annotation=float, required=False, default=0.0, alias='RivuletStorage', alias_priority=2), 'sb_id': FieldInfo(annotation=int, required=False, default=1)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

classmethod parse(*s*)

qin: Sequence[float] | None

qlat: Sequence[float] | None

qout: Sequence[float]

rivulet_storage: float

sb_id: int

class ravenpy.config.commands.**BasinStateVariables**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.BasinIndex], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod parse(*sol*)

root: Sequence[*BasinIndex*]

class ravenpy.config.commands.**ChannelProfile**(*, *name: str = 'chn_XXX'*, *bed_slope: float = 0*, *survey_points: Tuple[Tuple[float, float], ...] = ()*, *roughness_zones: Tuple[Tuple[float, float], ...] = ()*)

Bases: FlatCommand

ChannelProfile command (RVP).

bed_slope: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'bed_slope': FieldInfo(annotation=float, required=False, default=0), 'name': FieldInfo(annotation=str, required=False, default='chn_XXX'), 'roughness_zones': FieldInfo(annotation=Tuple[Tuple[float, float], ...], required=False, default=()), 'survey_points': FieldInfo(annotation=Tuple[Tuple[float, float], ...], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

roughness_zones: Tuple[Tuple[float, float], ...]

survey_points: Tuple[Tuple[float, float], ...]

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**CustomOutput**(*, *time_per: Literal['DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS']*, *stat: Literal['AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES']*, *variable: str*, *space_agg: Literal['BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED']*, *filename: str = ''*)

Bases: LineCommand

Create custom output file to track a single variable, parameter or forcing function over time at a number of basins, HRUs, or across the watershed.

Parameters

- **time_per** (*{'DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS'}*) – Time period.
- **stat** (*{'AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES', 'HISTOGRAM [min] [max] [# bins]'*) – Statistic reported for each time interval.
- **variable** (*str*) – Variable or parameter name. Consult the Raven documentation for the list of allowed names.
- **space_agg** (*{'BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED'}*) – Spatial evaluation domain.
- **filename** (*str*) – Output file name. Defaults to something approximately like *<run name>_<variable>_<time_per>_<stat>_<space_agg>.nc*

filename: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'filename': FieldInfo(annotation=str, required=False, default=''), 'space_agg': FieldInfo(annotation=Literal['BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED'], required=True), 'stat': FieldInfo(annotation=Literal['AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES'], required=True), 'time_per': FieldInfo(annotation=Literal['DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS'], required=True), 'variable': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

space_agg: Literal['BY_BASIN', 'BY_HRU', 'BY_HRU_GROUP', 'BY_SB_GROUP', 'ENTIRE_WATERSHED']

stat: Literal['AVERAGE', 'MAXIMUM', 'MINIMUM', 'RANGE', 'MEDIAN', 'QUARTILES']

time_per: Literal['DAILY', 'MONTHLY', 'YEARLY', 'WATER_YEARLY', 'CONTINUOUS']

variable: str

class ravenpy.config.commands.**Data**(*, *data_type: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] = ''*, *units: str = ''*, *ReadFromNetCDF:* ReadFromNetCDF)

Bases: FlatCommand

data_type: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR']

classmethod from_nc(*fn*, *data_type*, *station_idx=1*, *alt_names=()*, ***kwds*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data_type': FieldInfo(annotation=Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], required=False, default=''), 'read_from_netcdf': FieldInfo(annotation=ReadFromNetCDF, required=True, alias='ReadFromNetCDF', alias_priority=2), 'units': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

read_from_netcdf: *ReadFromNetCDF*

units: **str**

class ravenpy.config.commands.**EnsembleMode**(*, *mode: Literal['ENSEMBLE_ENKF'] = 'ENSEMBLE_ENKF'*, *n: int*)

Bases: LineCommand

mode: **Literal['ENSEMBLE_ENKF']**

model_computed_fields: **ClassVar[dict[str, ComputedFieldInfo]] = {}**

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: **ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}**

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: **ClassVar[dict[str, FieldInfo]] = {'mode': FieldInfo(annotation=Literal['ENSEMBLE_ENKF'], required=False, default='ENSEMBLE_ENKF'), 'n': FieldInfo(annotation=int, required=True, description='Number of members')}**

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

n: **int**

class ravenpy.config.commands.**EvaluationPeriod**(*, *name: str*, *start: date*, *end: date*)

Bases: LineCommand

:EvaluationPeriod [period_name] [start yyyy-mm-dd] [end yyyy-mm-dd]

end: **date**

model_computed_fields: **ClassVar[dict[str, ComputedFieldInfo]] = {}**

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: **ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}**

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'end': FieldInfo(annotation=date, required=True), 'name': FieldInfo(annotation=str, required=True), 'start': FieldInfo(annotation=date, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

start: date

class ravenpy.config.commands.**ForcingPerturbation**(*, *forcing: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR']*, *dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']*, *p1: float*, *p2: float*, *adj: Literal['ADDITIVE', 'MULTIPLICATIVE']*, *hru_grp: str = ''*)

Bases: LineCommand

adj: Literal['ADDITIVE', 'MULTIPLICATIVE']

dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']

forcing: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR']

hru_grp: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'adj': FieldInfo(annotation=Literal['ADDITIVE', 'MULTIPLICATIVE'], required=True), 'dist': FieldInfo(annotation=Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA'], required=True), 'forcing': FieldInfo(annotation=Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], required=True), 'hru_grp': FieldInfo(annotation=str, required=False, default=''), 'p1': FieldInfo(annotation=float, required=True), 'p2': FieldInfo(annotation=float, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

p1: float

p2: float

class ravenpy.config.commands.**Gauge**(*, *name: str = 'default'*, *Latitude: float*, *Longitude: float*, *Elevation: float | None = None*, *RainCorrection: Variable | Expression | float | None = None*, *SnowCorrection: Variable | Expression | float | None = None*, *MonthlyAveEvaporation: Sequence | None = None*, *MonthlyAveTemperature: Sequence | None = None*, *MonthlyMinTemperature: Sequence | None = None*, *MonthlyMaxTemperature: Sequence | None = None*, *Data: Sequence[Data] | None = None*)

Bases: FlatCommand

classmethod confirm_monthly(*v*)

data: Sequence[Data] | None

property ds: Dataset

Return xarray Dataset with forcing variables keyed by Raven forcing names.

elevation: float | None

classmethod from_nc(*fn: str | Path | Sequence[Path]*, *data_type: Sequence[str] | None = None*, *station_idx: int = 1*, *alt_names: Dict[str, str] | None = None*, *mon_ave: bool = False*, *data_kwds: Dict[str, Any] | None = None*, *engine: str = 'h5netcdf'*, ***kwds*) → *Gauge*

Return Gauge instance with configuration options inferred from the netCDF itself.

Parameters

- **fn** (*str or Path or Sequence[Path]*) – NetCDF file path or paths.
- **data_type** (*Sequence[str], optional*) – Raven data types to extract from netCDF files, e.g. 'PRECIP', 'AVE_TEMP'. The algorithm tries to find all forcings in each file until one is found, then it stops searching for it in the following files.
- **station_idx** (*int*) – Index along station dimension. Starts at 1. Should be the same for all netCDF files.

- **alt_names** (*dict*) – Alternative variable names keyed by data type. Use this if variables do not correspond to CF standard defaults.
- **mon_ave** (*bool*) – If True, compute the monthly average.
- **data_kwds** (*dict[options.Forcings, dict[str, str]]*) – Additional *:Data* parameters keyed by forcing type and station id. Overrides inferred parameters. Use keyword "ALL" to pass parameters to all variables.
- **engine** (*{"h5netcdf", "netcdf4", "pydap"}*) – The engine used to open the dataset. Default is 'h5netcdf'.
- ****kwds** – Additional arguments for Gauge.

Returns
Gauge instance.

Return type
Gauge

latitude: float

longitude: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.Data], NoneType], required=False, default=None, alias='Data', alias_priority=2), 'elevation': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='Elevation', alias_priority=2), 'latitude': FieldInfo(annotation=float, required=True, alias='Latitude', alias_priority=2), 'longitude': FieldInfo(annotation=float, required=True, alias='Longitude', alias_priority=2), 'monthly_ave_evaporation': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyAveEvaporation', alias_priority=2), 'monthly_ave_temperature': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyAveTemperature', alias_priority=2), 'monthly_max_temperature': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyMaxTemperature', alias_priority=2), 'monthly_min_temperature': FieldInfo(annotation=Union[Sequence, NoneType], required=False, default=None, alias='MonthlyMinTemperature', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default='default'), 'rain_correction': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=None, alias='RainCorrection', alias_priority=2, description='Rain correction'), 'snow_correction': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=None, alias='SnowCorrection', alias_priority=2, description='Snow correction')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

monthly_ave_evaporation: Sequence | None

monthly_ave_temperature: Sequence | None

monthly_max_temperature: Sequence | None

monthly_min_temperature: Sequence | None

name: str

rain_correction: Variable | Expression | float | None

snow_correction: Variable | Expression | float | None

class ravenpy.config.commands.**GridWeights**(*, *NumberHRUs: int = 1*, *NumberGridCells: int = 1*, *data: Sequence[GWRecord] = (GWRecord(root=(1, 0, 1.0)),)*)

Bases: Command

GridWeights command.

Notes

command can be embedded in both a *GriddedForcing* or a *StationForcing*.

The default is to have a single cell that covers an entire single HRU, with a weight of 1.

class GWRecord(*root: RootModelRootType = PydanticUndefined*)

Bases: RootRecord

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Tuple[int, int, float], required=False, default=(1, 0, 1.0))}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Tuple[int, int, float]

data: Sequence[*GWRecord*]

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Sequence[ravenpy.config.commands.GridWeights.GWRecord], required=False, default=(GWRecord(root=(1, 0, 1.0)),)), 'number_grid_cells': FieldInfo(annotation=int, required=False, default=1, alias='NumberGridCells', alias_priority=2), 'number_hrus': FieldInfo(annotation=int, required=False, default=1, alias='NumberHRUs', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

number_grid_cells: int

number_hrus: int

classmethod parse(*s*)

class ravenpy.config.commands.**GriddedForcing**(*, *FileNameNC: Url | Path*, *VarNameNC: str*, *DimNamesNC: Sequence[str]*, *station_idx: int | None = None*, *TimeShift: float | None = None*, *LinearTransform: LinearTransform | None = None*, *Deaccumulate: bool | None = None*, *LatitudeVarNameNC: str | None = None*, *LongitudeVarNameNC: str | None = None*, *ElevationVarNameNC: str | None = None*, *name: str = ''*, *ForcingType: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] | None = None*, *GridWeights: GridWeights | RedirectToFile = GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),))*)

Bases: *ReadFromNetCDF*

GriddedForcing command (RVT).

classmethod check_dims(*v*)

forcing_type: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] | None

grid_weights: *GridWeights* | *RedirectToFile*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'deaccumulate': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None, alias='Deaccumulate', alias_priority=2), 'dim_names_nc': FieldInfo(annotation=Sequence[str], required=True, alias='DimNamesNC', alias_priority=2), 'elevation_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='ElevationVarNameNC', alias_priority=2), 'file_name_nc': FieldInfo(annotation=Union[Annotated[pydantic_core._pydantic_core.Url, UrlConstraints(max_length=2083, allowed_schemes=['http', 'https'], host_required=None, default_host=None, default_port=None, default_path=None)], Path], required=True, alias='FileNameNC', alias_priority=2, description='NetCDF file name.'), 'forcing_type': FieldInfo(annotation=Union[Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], NoneType], required=False, default=None, alias='ForcingType', alias_priority=2), 'grid_weights': FieldInfo(annotation=Union[GridWeights, RedirectToFile], required=False, default=GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),)), alias='GridWeights', alias_priority=2), 'latitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LatitudeVarNameNC', alias_priority=2), 'linear_transform': FieldInfo(annotation=Union[LinearTransform, NoneType], required=False, default=None, alias='LinearTransform', alias_priority=2), 'longitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LongitudeVarNameNC', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default=''), 'station_idx': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'time_shift': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='TimeShift', alias_priority=2, description='Time stamp shift in days.'), 'var_name_nc': FieldInfo(annotation=str, required=True, alias='VarNameNC', alias_priority=2, description='NetCDF variable name.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

station_idx: int | None

class ravenpy.config.commands.**HRU**(*, *hru_id: int = 1*, *area: Variable | Expression | float | None = 0*, *elevation: float = 0*, *latitude: float = 0*, *longitude: float = 0*, *subbasin_id: int = 1*, *land_use_class: str = '[NONE]'*, *veg_class: str = '[NONE]'*, *soil_profile: str = '[NONE]'*, *aquifer_profile: str = '[NONE]'*, *terrain_class: str = '[NONE]'*, *slope: float = 0.0*, *aspect: float = 0.0*, *hru_type: str | None = None*)

Bases: Record

Record to populate :HRUs command internal table (RVH).

aquifer_profile: str

area: Variable | Expression | float | None

aspect: float

elevation: float

hru_id: int

hru_type: str | None

land_use_class: str

latitude: float

longitude: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'aquifer_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'area': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0), 'aspect': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0), Le(le=360)]), 'elevation': FieldInfo(annotation=float, required=False, default=0), 'hru_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'hru_type': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'land_use_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'latitude': FieldInfo(annotation=float, required=False, default=0), 'longitude': FieldInfo(annotation=float, required=False, default=0), 'slope': FieldInfo(annotation=float, required=False, default=0.0, metadata=[Ge(ge=0)]), 'soil_profile': FieldInfo(annotation=str, required=False, default='[NONE]'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)]), 'terrain_class': FieldInfo(annotation=str, required=False, default='[NONE]'), 'veg_class': FieldInfo(annotation=str, required=False, default='[NONE]')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

slope: float

soil_profile: str

subbasin_id: int

terrain_class: str

veg_class: str

class ravenpy.config.commands.**HRUGroup**(*, *name: str*, *groups: _Rec*)

Bases: FlatCommand

groups: _Rec

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'groups': FieldInfo(annotation=HRUGroup._Rec, required=True), 'name': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

class ravenpy.config.commands.**HRUState**(*, *hru_id: int = 1*, *data: Dict[str, Variable | Expression | float | None] = None*)

Bases: Record

data: Dict[str, Variable | Expression | float | None]

hru_id: int

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data': FieldInfo(annotation=Dict[str, Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=False, default_factory=dict), 'hru_id': FieldInfo(annotation=int, required=False, default=1)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod parse(*sol*, *names=None*)

class ravenpy.config.commands.**HRUStateVariableTable**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

Table of HRU state variables.

If the HRUState include different attributes, the states will be modified to include all attributes.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.HRUState], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

classmethod parse(*sol: str*)

root: Sequence[*HRUState*]

set_attributes()

class ravenpy.config.commands.**HRUs**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

HRUs command (RVH).

classmethod ignore_unrecognized_hrus(*values*)

Ignore HRUs with unrecognized hru_type.

HRUs are ignored only if all allowed HRU classes define *hru_type*, and if the values passed include it.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.HRU], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*HRU*]

ravenpy.config.commands.**LU**

alias of *LandUseClass*

class ravenpy.config.commands.**LandUseClass**(*, *name: str = ''*, *impermeable_frac: Variable | Expression | float | None = 0.0*, *forest_coverage: Variable | Expression | float | None = 0.0*)

Bases: Record

forest_coverage: Variable | Expression | float | None

impermeable_frac: Variable | Expression | float | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'forest_coverage': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'impermeable_frac': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'name': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

class ravenpy.config.commands.**LandUseClasses**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.LandUseClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*LandUseClass*]

class ravenpy.config.commands.**LandUseParameterList**(*, *Parameters: Sequence[Literal['FOREST_COVERAGE', 'IMPERMEABLE_FRAC', 'ROUGHNESS', 'FOREST_SPARSENESS', 'DEP_MAX', 'MAX_DEP_AREA_FRAC', 'DD_MELT_TEMP', 'MELT_FACTOR', 'DD_REFREEZE_TEMP', 'MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_AGGRADATION', 'SNOW_PATCH_LIMIT', 'HBV_MELT_FOR_CORR', 'HBV_MELT_ASP_CORR', 'GLAC_STORAGE_COEFF', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'HBV_GLACIER_AG', 'CC_DECAY_COEFF', 'SCS_CN', 'SCS_IA_FRACTION', 'PARTITION_COEFF', 'MAX_SAT_AREA_FRAC', 'B_EXP', 'ABST_PERCENT', 'DEP_MAX_FLOW', 'DEP_N', 'DEP_SEEP_K', 'DEP_K', 'DEP_THRESHOLD', 'PDMROF_B', 'PONDED_EXP', 'OW_PET_CORR', 'LAKE_PET_CORR', 'LAKE_REL_COEFF', 'FOREST_PET_CORR', 'GAMMA_SCALE', 'GAMMA_SHAPE', 'GAMMA_SCALE2', 'GAMMA_SHAPE2', 'HMETS_RUNOFF_COEFF', 'AET_COEFF', 'GR4J_X4', 'UBC_ICEPT_FACTOR', 'STREAM_FRACTION', 'BF_LOSS_FRACTION']] | None = None*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

Bases: GenericParameterList

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['FOREST_COVERAGE', 'IMPERMEABLE_FRAC', 'ROUGHNESS', 'FOREST_SPARSENESS', 'DEP_MAX', 'MAX_DEP_AREA_FRAC', 'DD_MELT_TEMP', 'MELT_FACTOR', 'DD_REFREEZE_TEMP', 'MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_AGGRADATION', 'SNOW_PATCH_LIMIT', 'HBV_MELT_FOR_CORR', 'HBV_MELT_ASP_CORR', 'GLAC_STORAGE_COEFF', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'HBV_GLACIER_AG', 'CC_DECAY_COEFF', 'SCS_CN', 'SCS_IA_FRACTION', 'PARTITION_COEFF', 'MAX_SAT_AREA_FRAC', 'B_EXP', 'ABST_PERCENT', 'DEP_MAX_FLOW', 'DEP_N', 'DEP_SEEP_K', 'DEP_K', 'DEP_THRESHOLD', 'PDMROF_B', 'PONDED_EXP', 'OW_PET_CORR', 'LAKE_PET_CORR', 'LAKE_REL_COEFF', 'FOREST_PET_CORR', 'GAMMA_SCALE', 'GAMMA_SHAPE', 'GAMMA_SCALE2', 'GAMMA_SHAPE2', 'HMETS_RUNOFF_COEFF', 'AET_COEFF', 'GR4J_X4', 'UBC_ICEPT_FACTOR', 'STREAM_FRACTION', 'BF_LOSS_FRACTION']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['FOREST_COVERAGE', 'IMPERMEABLE_FRAC', 'ROUGHNESS', 'FOREST_SPARSENESS', 'DEP_MAX', 'MAX_DEP_AREA_FRAC', 'DD_MELT_TEMP', 'MELT_FACTOR', 'DD_REFREEZE_TEMP', 'MIN_MELT_FACTOR', 'MAX_MELT_FACTOR', 'REFREEZE_FACTOR', 'REFREEZE_EXP', 'DD_AGGRADATION', 'SNOW_PATCH_LIMIT', 'HBV_MELT_FOR_CORR', 'HBV_MELT_ASP_CORR', 'GLAC_STORAGE_COEFF', 'HBV_MELT_GLACIER_CORR', 'HBV_GLACIER_KMIN', 'HBV_GLACIER_AG', 'CC_DECAY_COEFF', 'SCS_CN', 'SCS_IA_FRACTION', 'PARTITION_COEFF', 'MAX_SAT_AREA_FRAC', 'B_EXP', 'ABST_PERCENT', 'DEP_MAX_FLOW', 'DEP_N', 'DEP_SEEP_K', 'DEP_K', 'DEP_THRESHOLD', 'PDMROF_B', 'PONDED_EXP', 'OW_PET_CORR', 'LAKE_PET_CORR', 'LAKE_REL_COEFF', 'FOREST_PET_CORR', 'GAMMA_SCALE', 'GAMMA_SHAPE', 'GAMMA_SCALE2', 'GAMMA_SHAPE2', 'HMETS_RUNOFF_COEFF', 'AET_COEFF', 'GR4J_X4', 'UBC_ICEPT_FACTOR', 'STREAM_FRACTION', 'BF_LOSS_FRACTION']] | None

class ravenpy.config.commands.**LinearTransform**(*, *scale: float = 1*, *offset: float = 0*)

Bases: Command

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'offset': FieldInfo(annotation=float, required=False, default=0), 'scale': FieldInfo(annotation=float, required=False, default=1)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

offset: float

scale: float

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**ObservationData**(*, *data_type: Literal['HYDROGRAPH'] = 'HYDROGRAPH'*, *units: str = ''*, *ReadFromNetCDF: ReadFromNetCDF*, *uid: str = '1'*)

Bases: *Data*

data_type: Literal['HYDROGRAPH']

classmethod from_nc(*fn*, *station_idx: int = 1*, *alt_names=()*, *engine='h5netcdf'*, ***kwds*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'data_type': FieldInfo(annotation=Literal['HYDROGRAPH'], required=False, default='HYDROGRAPH'), 'read_from_netcdf': FieldInfo(annotation=ReadFromNetCDF, required=True, alias='ReadFromNetCDF', alias_priority=2), 'uid': FieldInfo(annotation=str, required=False, default='1'), 'units': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

uid: str

class ravenpy.config.commands.**ObservationErrorModel**(*, *state: Literal['STREAMFLOW']*, *dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']*, *p1: float*, *p2: float*, *adj: Literal['ADDITIVE', 'MULTIPLICATIVE']*)

Bases: LineCommand

adj: Literal['ADDITIVE', 'MULTIPLICATIVE']

dist: Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'adj': FieldInfo(annotation=Literal['ADDITIVE', 'MULTIPLICATIVE'], required=True), 'dist': FieldInfo(annotation=Literal['DIST_UNIFORM', 'DIST_NORMAL', 'DIST_GAMMA'], required=True), 'p1': FieldInfo(annotation=float, required=True), 'p2': FieldInfo(annotation=float, required=True), 'state': FieldInfo(annotation=Literal['STREAMFLOW'], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

p1: float

p2: float

state: Literal['STREAMFLOW']

class ravenpy.config.commands.**Process**(**, algo: str = 'RAVEN_DEFAULT', source: str | None = None, to: Sequence[str] = ()*)

Bases: Command

Process type embedded in HydrologicProcesses command.

See processes.py for list of processes.

algo: str

classmethod is_list(*v*)

classmethod is_source_state_variable(*v: str*)

classmethod is_to_state_variable(*v: Sequence*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=str, required=False, default='RAVEN_DEFAULT'), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

source: str | None

to: Sequence[str]

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**RainSnowTransition**(*, *temp: Variable | Expression | float | None*, *delta: Variable | Expression | float | None*)

Bases: Command

Specify the range of temperatures over which there will be a rain/snow mix when partitioning total precipitation into rain and snow components.

delta: Variable | Expression | float | None

Range [C].

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'delta': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True), 'temp': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

temp: Variable | Expression | float | None

Midpoint of the temperature range [C].

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**ReadFromNetCDF**(*, *FileNameNC: Url | Path*, *VarNameNC: str*, *DimNamesNC: Sequence[str]*, *StationIdx: int = 1*, *TimeShift: float | None = None*, *LinearTransform: LinearTransform | None = None*, *Deaccumulate: bool | None = None*, *LatitudeVarNameNC: str | None = None*, *LongitudeVarNameNC: str | None = None*, *ElevationVarNameNC: str | None = None*)

Bases: FlatCommand

property da: DataArray

Return DataArray from configuration.

deaccumulate: bool | None

dim_names_nc: Sequence[str]

elevation_var_name_nc: str | None

file_name_nc: Url | Path

classmethod from_nc(*fn*, *data_type*, *station_idx=1*, *alt_names=()*, *engine='h5netcdf'*, ***kwds*)

Instantiate class from netCDF dataset.

latitude_var_name_nc: str | None

linear_transform: *LinearTransform* | None

longitude_var_name_nc: str | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'deaccumulate': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None, alias='Deaccumulate', alias_priority=2), 'dim_names_nc': FieldInfo(annotation=Sequence[str], required=True, alias='DimNamesNC', alias_priority=2), 'elevation_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='ElevationVarNameNC', alias_priority=2), 'file_name_nc': FieldInfo(annotation=Union[Annotated[pydantic_core._pydantic_core.Url, UrlConstraints(max_length=2083, allowed_schemes=['http', 'https'], host_required=None, default_host=None, default_port=None, default_path=None)], Path], required=True, alias='FileNameNC', alias_priority=2, description='NetCDF file name.'), 'latitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LatitudeVarNameNC', alias_priority=2), 'linear_transform': FieldInfo(annotation=Union[LinearTransform, NoneType], required=False, default=None, alias='LinearTransform', alias_priority=2), 'longitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LongitudeVarNameNC', alias_priority=2), 'station_idx': FieldInfo(annotation=int, required=False, default=1, alias='StationIdx', alias_priority=2, description='NetCDF index along station dimension. Starts at 1.'), 'time_shift': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='TimeShift', alias_priority=2, description='Time stamp shift in days.'), 'var_name_nc': FieldInfo(annotation=str, required=True, alias='VarNameNC', alias_priority=2, description='NetCDF variable name.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

classmethod reorder_time(*v*)

TODO: Return dimensions as x, y, t. Currently only puts time at the end.

station_idx: int

time_shift: float | None

var_name_nc: str

class ravenpy.config.commands.**RedirectToFile**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootCommand

RedirectToFile command (RVT).

Notes

For the moment, this command can only be used in the context of a *GriddedForcingCommand* or a *StationForcingCommand*, as a *grid_weights* field replacement when inlining is not desired.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Path, required=True, metadata=[PathType(path_type='file')])}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Path

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**Reservoir**(*, *name: str = 'Lake_XXX'*, *SubBasinID: int = 0*, *HRUID: int = 0*, *Type: str = 'RESROUTE_STANDARD'*, *WeirCoefficient: float = 0*, *CrestWidth: float = 0*, *MaxDepth: float = 0*, *LakeArea: float = 0*)

Bases: FlatCommand

Reservoir command (RVH).

crest_width: float

hru_id: int

lake_area: float

max_depth: float

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'crest_width': FieldInfo(annotation=float, required=False, default=0, alias='CrestWidth', alias_priority=2), 'hru_id': FieldInfo(annotation=int, required=False, default=0, alias='HRUID', alias_priority=2), 'lake_area': FieldInfo(annotation=float, required=False, default=0, alias='LakeArea', alias_priority=2, description='Lake area in m2'), 'max_depth': FieldInfo(annotation=float, required=False, default=0, alias='MaxDepth', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default='Lake_XXX'), 'subbasin_id': FieldInfo(annotation=int, required=False, default=0, alias='SubBasinID', alias_priority=2), 'type': FieldInfo(annotation=str, required=False, default='RESROUTE_STANDARD', alias='Type', alias_priority=2), 'weir_coefficient': FieldInfo(annotation=float, required=False, default=0, alias='WeirCoefficient', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

subbasin_id: int

type: str

weir_coefficient: float

ravenpy.config.commands.**SB**

alias of *SubBasin*

class ravenpy.config.commands.**SBGroupPropertyMultiplier**(*, *group_name: str*, *parameter_name: str*, *mult: float*)

Bases: LineCommand

group_name: str

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'group_name': FieldInfo(annotation=str, required=True), 'mult': FieldInfo(annotation=float, required=True), 'parameter_name': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

mult: float

parameter_name: str

ravenpy.config.commands.**SC**

alias of *SoilClass*

ravenpy.config.commands.**SP**

alias of *SoilProfile*

class ravenpy.config.commands.**SeasonalRelativeHeight**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands._MonthlyRecord], required=False, default=(<ravenpy.config.commands._MonthlyRecord object>,))}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[_MonthlyRecord]

class ravenpy.config.commands.**SeasonalRelativeLAI**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands._MonthlyRecord], required=False, default=(<ravenpy.config.commands._MonthlyRecord object>,))}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[_MonthlyRecord]

class ravenpy.config.commands.**SoilClasses**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

class SoilClass(**, name: str, mineral: Tuple[float, float, float] | None = None, organic: float | None = None*)

Bases: Record

mineral: Tuple[float, float, float] | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'mineral': FieldInfo(annotation=Union[Tuple[float, float, float], NoneType], required=False, default=None), 'name': FieldInfo(annotation=str, required=True), 'organic': FieldInfo(annotation=Union[float, NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

organic: float | None

classmethod validate_mineral(*v*)

Assert sum of mineral fraction is 1.

classmethod validate_mineral_pct(*v: Tuple*)

classmethod validate_organic_pct(*v: float*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.SoilClasses.SoilClass], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*SoilClass*]

class ravenpy.config.commands.**SoilModel**(*root: RootModelRootType = PydanticUndefined*)

Bases: RootCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=int, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: int

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**SoilParameterList**(*, *Parameters: Sequence[Literal['SAND_CON', 'CLAY_CON', 'SILT_CON', 'ORG_CON', 'POROSITY', 'STONE_FRAC', 'SAT_WILT', 'FIELD_CAPACITY', 'BULK_DENSITY', 'HYDRAUL_COND', 'CLAPP_B', 'CLAPP N,CLAPP M', 'SAT_RES', 'AIR_ENTRY_PRESSURE', 'WILTING_PRESSURE', 'HEAT_CAPACITY', 'THERMAL_COND', 'WETTING_FRONT_PSI', 'EVAP_RES_FC', 'SHUTTLEWORTH_B', 'ALBEDO_WET', 'ALBEDO_DRY', 'VIZ_ZMIN', 'VIC_ZMAX', 'VIC ALPHA', 'VIC_EVAP_GAMMA', 'MAX_PERC_RATE', 'PERC_N', 'PERC_COEFF', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'UNAVAIL_FRAC', 'HBV_BETA', 'MAX_BASEFLOW_RATE', 'BASEFLOW_N', 'BASEFLOW_COEFF', 'BASEFLOW_COEFF2', 'BASEFLOW_THRESH', 'BF_LOSS_FRACTION', 'STORAGE_THRESHOLD', 'MAX_CAP_RISE_RATE', 'MAX_INTERFLOW_RATE', 'INTERFLOW_COEF', 'UBC_EVAL_SOIL_DEF', 'UBC_INFIL_SOIL_DEF', 'GR4J_X2', 'GR4J_X3', 'B_EXP', 'PET_CORRECTION']] | None = None*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

Bases: GenericParameterList

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['SAND_CON', 'CLAY_CON', 'SILT_CON', 'ORG_CON', 'POROSITY', 'STONE_FRAC', 'SAT_WILT', 'FIELD_CAPACITY', 'BULK_DENSITY', 'HYDRAUL_COND', 'CLAPP_B', 'CLAPP N,CLAPP M', 'SAT_RES', 'AIR_ENTRY_PRESSURE', 'WILTING_PRESSURE', 'HEAT_CAPACITY', 'THERMAL_COND', 'WETTING_FRONT_PSI', 'EVAP_RES_FC', 'SHUTTLEWORTH_B', 'ALBEDO_WET', 'ALBEDO_DRY', 'VIZ_ZMIN', 'VIC_ZMAX', 'VIC ALPHA', 'VIC_EVAP_GAMMA', 'MAX_PERC_RATE', 'PERC_N', 'PERC_COEFF', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'UNAVAIL_FRAC', 'HBV_BETA', 'MAX_BASEFLOW_RATE', 'BASEFLOW_N', 'BASEFLOW_COEFF', 'BASEFLOW_COEFF2', 'BASEFLOW_THRESH', 'BF_LOSS_FRACTION', 'STORAGE_THRESHOLD', 'MAX_CAP_RISE_RATE', 'MAX_INTERFLOW_RATE', 'INTERFLOW_COEF', 'UBC_EVAL_SOIL_DEF', 'UBC_INFIL_SOIL_DEF', 'GR4J_X2', 'GR4J_X3', 'B_EXP', 'PET_CORRECTION']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['SAND_CON', 'CLAY_CON', 'SILT_CON', 'ORG_CON', 'POROSITY', 'STONE_FRAC', 'SAT_WILT', 'FIELD_CAPACITY', 'BULK_DENSITY', 'HYDRAUL_COND', 'CLAPP_B', 'CLAPP N,CLAPP M', 'SAT_RES', 'AIR_ENTRY_PRESSURE', 'WILTING_PRESSURE', 'HEAT_CAPACITY', 'THERMAL_COND', 'WETTING_FRONT_PSI', 'EVAP_RES_FC', 'SHUTTLEWORTH_B', 'ALBEDO_WET', 'ALBEDO_DRY', 'VIZ_ZMIN', 'VIC_ZMAX', 'VIC ALPHA', 'VIC_EVAP_GAMMA', 'MAX_PERC_RATE', 'PERC_N', 'PERC_COEFF', 'SAC_PERC_ALPHA', 'SAC_PERC_EXPON', 'SAC_PERC_PFREE', 'UNAVAIL_FRAC', 'HBV_BETA', 'MAX_BASEFLOW_RATE', 'BASEFLOW_N', 'BASEFLOW_COEFF', 'BASEFLOW_COEFF2', 'BASEFLOW_THRESH', 'BF_LOSS_FRACTION', 'STORAGE_THRESHOLD', 'MAX_CAP_RISE_RATE', 'MAX_INTERFLOW_RATE', 'INTERFLOW_COEF', 'UBC_EVAL_SOIL_DEF', 'UBC_INFIL_SOIL_DEF', 'GR4J_X2', 'GR4J_X3', 'B_EXP', 'PET_CORRECTION']] | None

class ravenpy.config.commands.**SoilProfile**(*, *name: str = ''*, *soil_classes: Sequence[str] = ()*, *thicknesses: Sequence[Variable | Expression | float | None] = ()*)

Bases: Record

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'name': FieldInfo(annotation=str, required=False, default=''), 'soil_classes': FieldInfo(annotation=Sequence[str], required=False, default=()), 'thicknesses': FieldInfo(annotation=Sequence[Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

soil_classes: Sequence[str]

thicknesses: Sequence[Variable | Expression | float | None]

class ravenpy.config.commands.**SoilProfiles**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.SoilProfile], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

root: Sequence[*SoilProfile*]

class ravenpy.config.commands.**StationForcing**(*, *FileNameNC: Url | Path*, *VarNameNC: str*, *DimNamesNC: Sequence[str]*, *station_idx: int | None = None*, *TimeShift: float | None = None*, *LinearTransform: LinearTransform | None = None*, *Deaccumulate: bool | None = None*, *LatitudeVarNameNC: str | None = None*, *LongitudeVarNameNC: str | None = None*, *ElevationVarNameNC: str | None = None*, *name: str = ''*, *ForcingType: Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'] | None = None*, *GridWeights: GridWeights | RedirectToFile = GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),))*)

Bases: *GriddedForcing*

StationForcing command (RVT).

classmethod check_dims(*v*)

classmethod from_nc(*fn*, *data_type*, *alt_names=()*, ***kwds*)

Instantiate class from netCDF dataset.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'deaccumulate': FieldInfo(annotation=Union[bool, NoneType], required=False, default=None, alias='Deaccumulate', alias_priority=2), 'dim_names_nc': FieldInfo(annotation=Sequence[str], required=True, alias='DimNamesNC', alias_priority=2), 'elevation_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='ElevationVarNameNC', alias_priority=2), 'file_name_nc': FieldInfo(annotation=Union[Annotated[pydantic_core._pydantic_core.Url, UrlConstraints(max_length=2083, allowed_schemes=['http', 'https'], host_required=None, default_host=None, default_port=None, default_path=None)], Path], required=True, alias='FileNameNC', alias_priority=2, description='NetCDF file name.'), 'forcing_type': FieldInfo(annotation=Union[Literal['PRECIP', 'PRECIP_DAILY_AVE', 'PRECIP_5DAY', 'SNOW_FRAC', 'SNOWFALL', 'RAINFALL', 'RECHARGE', 'TEMP_AVE', 'TEMP_DAILY_AVE', 'TEMP_MIN', 'TEMP_DAILY_MIN', 'TEMP_MAX', 'TEMP_DAILY_MAX', 'TEMP_MONTH_MAX', 'TEMP_MONTH_MIN', 'TEMP_MONTH_AVE', 'TEMP_AVE_UNC', 'TEMP_MAX_UNC', 'TEMP_MIN_UNC', 'AIR_DENS', 'AIR_PRES', 'REL_HUMIDITY', 'ET_RADIA', 'SHORTWAVE', 'SW_RADIA', 'SW_RADIA_NET', 'LW_RADIA_NET', 'LW_INCOMING', 'CLOUD_COVER', 'DAY_LENGTH', 'DAY_ANGLE', 'WIND_VEL', 'PET', 'OW_PET', 'PET_MONTH_AVE', 'POTENTIAL_MELT', 'SUBDAILY_CORR'], NoneType], required=False, default=None, alias='ForcingType', alias_priority=2), 'grid_weights': FieldInfo(annotation=Union[GridWeights, RedirectToFile], required=False, default=GridWeights(number_hrus=1, number_grid_cells=1, data=(GWRecord(root=(1, 0, 1.0)),)), alias='GridWeights', alias_priority=2), 'latitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LatitudeVarNameNC', alias_priority=2), 'linear_transform': FieldInfo(annotation=Union[LinearTransform, NoneType], required=False, default=None, alias='LinearTransform', alias_priority=2), 'longitude_var_name_nc': FieldInfo(annotation=Union[str, NoneType], required=False, default=None, alias='LongitudeVarNameNC', alias_priority=2), 'name': FieldInfo(annotation=str, required=False, default=''), 'station_idx': FieldInfo(annotation=Union[int, NoneType], required=False, default=None), 'time_shift': FieldInfo(annotation=Union[float, NoneType], required=False, default=None, alias='TimeShift', alias_priority=2, description='Time stamp shift in days.'), 'var_name_nc': FieldInfo(annotation=str, required=True, alias='VarNameNC', alias_priority=2, description='NetCDF variable name.')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class ravenpy.config.commands.**SubBasin**(***, *subbasin_id: int = 1*, *name: str = 'sub_001'*, *downstream_id: int = -1*, *profile: str = 'NONE'*, *reach_length: float = 0*, *gauged: bool = True*, *gauge_id: str | None = ''*)

Bases: `Record`

Record to populate RVH :SubBasins command internal table.

downstream_id: int

gauge_id: str | None

gauged: bool

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'downstream_id': FieldInfo(annotation=int, required=False, default=-1), 'gauge_id': FieldInfo(annotation=Union[str, NoneType], required=False, default=''), 'gauged': FieldInfo(annotation=bool, required=False, default=True), 'name': FieldInfo(annotation=str, required=False, default='sub_001'), 'profile': FieldInfo(annotation=str, required=False, default='NONE'), 'reach_length': FieldInfo(annotation=float, required=False, default=0), 'subbasin_id': FieldInfo(annotation=int, required=False, default=1, metadata=[Gt(gt=0)])}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

profile: str

reach_length: float

subbasin_id: int

class ravenpy.config.commands.**SubBasinGroup**(*, *name: str = ''*, *sb_ids: Sequence[int] = ()*)

Bases: FlatCommand

SubBasinGroup command (RVH).

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'name': FieldInfo(annotation=str, required=False, default=''), 'sb_ids': FieldInfo(annotation=Sequence[int], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

sb_ids: Sequence[int]

to_rv()

Return Raven configuration string.

class ravenpy.config.commands.**SubBasinProperties**(*, *Parameters: Sequence[Literal['TIME_TO_PEAK', 'TIME_CONC', 'TIME_LAG', 'NUM_RESERVOIRS', 'RES_CONSTANT', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'Q_REFERENCE', 'MANNINGS_N', 'SLOPE', 'DIFFUSIVITY', 'CELERITY', 'RAIN_CORR', 'SNOW_CORR']] | None = None*, *records: Sequence[SubBasinProperty] | None = None*)

Bases: Command

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['TIME_TO_PEAK', 'TIME_CONC', 'TIME_LAG', 'NUM_RESERVOIRS', 'RES_CONSTANT', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'Q_REFERENCE', 'MANNINGS_N', 'SLOPE', 'DIFFUSIVITY', 'CELERITY', 'RAIN_CORR', 'SNOW_CORR']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'records': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.SubBasinProperty], NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['TIME_TO_PEAK', 'TIME_CONC', 'TIME_LAG', 'NUM_RESERVOIRS', 'RES_CONSTANT', 'GAMMA_SHAPE', 'GAMMA_SCALE', 'Q_REFERENCE', 'MANNINGS_N', 'SLOPE', 'DIFFUSIVITY', 'CELERITY', 'RAIN_CORR', 'SNOW_CORR']] | None

records: Sequence[*SubBasinProperty*] | None

class ravenpy.config.commands.**SubBasinProperty**(*, *sb_id: str*, *values: Sequence[Variable | Expression | float | None]*)

Bases: Record

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'coerce_numbers_to_str': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'sb_id': FieldInfo(annotation=str, required=True), 'values': FieldInfo(annotation=Sequence[Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

sb_id: str

values: Sequence[Variable | Expression | float | None]

class ravenpy.config.commands.**SubBasins**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

SubBasins command (RVH).

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.SubBasin], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*SubBasin*]

ravenpy.config.commands.**TC**

alias of *TerrainClass*

class ravenpy.config.commands.**TerrainClass**(*, *name: str*, *hillslope_length: Variable | Expression | float | None*, *drainage_density: Variable | Expression | float | None*, *topmodel_lambda: Variable | Expression | float | None = None*)

Bases: Record

drainage_density: Variable | Expression | float | None

hillslope_length: Variable | Expression | float | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'drainage_density': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True), 'hillslope_length': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=True), 'name': FieldInfo(annotation=str, required=True), 'topmodel_lambda': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=None)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

topmodel_lambda: Variable | Expression | float | None

class ravenpy.config.commands.**TerrainClasses**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.TerrainClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*TerrainClass*]

ravenpy.config.commands.**VC**

alias of *VegetationClass*

class ravenpy.config.commands.**VegetationClass**(*, *name: str = ''*, *max_ht: Variable | Expression | float | None = 0.0*, *max_lai: Variable | Expression | float | None = 0.0*, *max_leaf_cond: Variable | Expression | float | None = 0.0*)

Bases: Record

max_ht: Variable | Expression | float | None

max_lai: Variable | Expression | float | None

max_leaf_cond: Variable | Expression | float | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'max_ht': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'max_lai': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'max_leaf_cond': FieldInfo(annotation=Union[Variable, Expression, float, NoneType], required=False, default=0.0), 'name': FieldInfo(annotation=str, required=False, default='')}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: str

class ravenpy.config.commands.**VegetationClasses**(*root: RootModelRootType = PydanticUndefined*)

Bases: ListCommand

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'root': FieldInfo(annotation=Sequence[ravenpy.config.commands.VegetationClass], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

root: Sequence[*VegetationClass*]

class ravenpy.config.commands.**VegetationParameterList**(*, *Parameters: Sequence[Literal['MAX_HEIGHT', 'MAX_LEAF_COND', 'MAX_LAI', 'SVF_EXTINCTION', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT', 'RAIN_ICEPT_FACT', 'SNOW_ICEPT_FACT', 'SAI_HT_RATIO', 'TRUNK_FRACTION', 'STEMFLOW_FRAC', 'ALBEDO', 'ALBEDO_WET', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'ROOT_EXTINCT', 'MAX_ROOT_LENGTH', 'MIN_RESISTIVITY', 'XYLEM_FRAC', 'ROOTRADIUS', 'PSI_CRITICAL', 'DRIP_PROPORTION', 'MAX_INTERCEPT_RATE', 'CHU_MATURITY', 'VEG_DIAM', 'VEG_MBETA', 'VEG_DENSPET_VEG_CORR', 'TFRAIN', 'TFSNOW', 'RELATIVE_HT', 'RELATIVE_LAI', 'CAP_LAI_RATIO', 'SNOCAP_LAI_RATIO']] | None = None*, *Units: Sequence[str] | None = None*, *pl: Sequence[ParameterList]*)

Bases: GenericParameterList

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'parameters': FieldInfo(annotation=Union[Sequence[Literal['MAX_HEIGHT', 'MAX_LEAF_COND', 'MAX_LAI', 'SVF_EXTINCTION', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT', 'RAIN_ICEPT_FACT', 'SNOW_ICEPT_FACT', 'SAI_HT_RATIO', 'TRUNK_FRACTION', 'STEMFLOW_FRAC', 'ALBEDO', 'ALBEDO_WET', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'ROOT_EXTINCT', 'MAX_ROOT_LENGTH', 'MIN_RESISTIVITY', 'XYLEM_FRAC', 'ROOTRADIUS', 'PSI_CRITICAL', 'DRIP_PROPORTION', 'MAX_INTERCEPT_RATE', 'CHU_MATURITY', 'VEG_DIAM', 'VEG_MBETA', 'VEG_DENSPET_VEG_CORR', 'TFRAIN', 'TFSNOW', 'RELATIVE_HT', 'RELATIVE_LAI', 'CAP_LAI_RATIO', 'SNOCAP_LAI_RATIO']], NoneType], required=False, default=None, alias='Parameters', alias_priority=2), 'pl': FieldInfo(annotation=Sequence[ravenpy.config.base.ParameterList], required=True), 'units': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default=None, alias='Units', alias_priority=2)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

parameters: Sequence[Literal['MAX_HEIGHT', 'MAX_LEAF_COND', 'MAX_LAI', 'SVF_EXTINCTION', 'RAIN_ICEPT_PCT', 'SNOW_ICEPT_PCT', 'RAIN_ICEPT_FACT', 'SNOW_ICEPT_FACT', 'SAI_HT_RATIO', 'TRUNK_FRACTION', 'STEMFLOW_FRAC', 'ALBEDO', 'ALBEDO_WET', 'MAX_CAPACITY', 'MAX_SNOW_CAPACITY', 'ROOT_EXTINCT', 'MAX_ROOT_LENGTH', 'MIN_RESISTIVITY', 'XYLEM_FRAC', 'ROOTRADIUS', 'PSI_CRITICAL', 'DRIP_PROPORTION', 'MAX_INTERCEPT_RATE', 'CHU_MATURITY', 'VEG_DIAM', 'VEG_MBETA', 'VEG_DENSPET_VEG_CORR', 'TFRAIN', 'TFSNOW', 'RELATIVE_HT', 'RELATIVE_LAI', 'CAP_LAI_RATIO', 'SNOCAP_LAI_RATIO']] | None

ravenpy.config.conventions module

ravenpy.config.defaults module

ravenpy.config.defaults.**default_nc_attrs**()

Return default NetCDF global attributes.

ravenpy.config.defaults.**nc_attrs**(*cls*, *val*)

Ensure default netCDF attributes are present

ravenpy.config.options module

class ravenpy.config.options.**AirPressureMethod**(*value*)

Bases: Enum

An enumeration.

BASIC = 'AIRPRESS_BASIC'

CONST = 'AIRPRESS_CONST'

DATA = 'AIRPRESS_DATA'

UBC = 'AIRPRESS_UBC'

class ravenpy.config.options.**Calendar**(*value*)

Bases: Enum

An enumeration.

ALL_LEAP = 'ALL_LEAP'

GREGORIAN = 'GREGORIAN'

JULIAN = 'JULIAN'

NOLEAP = 'NOLEAP'

PROLEPTIC_GREGORIAN = 'PROLEPTIC_GREGORIAN'

STANDARD = 'STANDARD'

class ravenpy.config.options.**CatchmentRoute**(*value*)

Bases: Enum

:CatchmentRoute

DELAYED_FIRST_ORDER = 'ROUTE_DELAYED_FIRST_ORDER'

DUMP = 'ROUTE_DUMP'

EXP = 'ROUTE_EXPONENTIAL'

EXPONENTIAL_UH = 'EXPONENTIAL_UH'

GAMMA = 'ROUTE_GAMMA_CONVOLUTION'

RESERVOIR = 'ROUTE_RESERVOIR_SERIES'

TRI = 'ROUTE_TRI_CONVOLUTION'

TRIANGULAR_UH = 'TRIANGULAR_UH'

TRI_CONVOLUTION = 'TRI_CONVOLUTION'

class ravenpy.config.options.**CloudCoverMethod**(*value*)

Bases: Enum

An enumeration.

DATA = 'CLOUDCOV_DATA'

NONE = 'CLOUDCOV_NONE'

UBC = 'CLOUDCOV_UBC'

class ravenpy.config.options.**EnKFMode**(*value*)

Bases: Enum

An enumeration.

CLOSED_LOOP = 'ENKF_CLOSED_LOOP'

FORECAST = 'ENKF_FORECAST'

OPEN_FORECAST = 'ENKF_OPEN_FORECAST'

OPEN_LOOP = 'ENKF_OPEN_LOOP'

SPINUP = 'ENKF_SPINUP'

class ravenpy.config.options.**EvaluationMetrics**(*value*)

Bases: Enum

An enumeration.

ABSERR = 'ABSERR'

ABSMAX = 'ABSMAX'

DIAG_SPEARMAN = 'DIAG_SPEARMAN'

KLING_GUPTA = 'KLING_GUPTA'

LOG_NASH = 'LOG_NASH'

NASH_SUTCLIFFE = 'NASH_SUTCLIFFE'

NSC = 'NSC'

PCT_BIAS = 'PCT_BIAS'

PDIFF = 'PDIFF'

RCOEFF = 'RCOEFF'

RMSE = 'RMSE'

TMVOL = 'TMVOL'

class ravenpy.config.options.**Evaporation**(*value*)

Bases: Enum

An enumeration.

CONSTANT = 'PET_CONSTANT'

DATA = 'PET_DATA'

FROMMONTHLY = 'PET_FROMMONTHLY'

HAMON_1961 = 'PET_HAMON_1961'

HARGREAVES = 'PET_HARGREAVES'

HARGREAVES_1985 = 'PET_HARGREAVES_1985'

MAKKINK_1957 = 'PET_MAKKINK_1957'

MOHYSE = 'PET_MOHYSE'

MONTHLY_FACTOR = 'PET_MONTHLY_FACTOR'

OUDIN = 'PET_OUDIN'

PENMAN_COMBINATION = 'PET_PENMAN_COMBINATION'

PENMAN_MONTEITH = 'PET_PENMAN_MONTEITH'

PRIESTLEY_TAYLOR = 'PET_PRIESTLEY_TAYLOR'

TURC_1961 = 'PET_TURC_1961'

VAP_DEFICIT = 'PET_VAPDEFICIT'

class ravenpy.config.options.**Interpolation**(*value*)

Bases: Enum

An enumeration.

AVERAGE_ALL = 'INTERP_AVERAGE_ALL'

FROM_FILE = 'INTERP_FROM_FILE'

INVERSE_DISTANCE = 'INTERP_INVERSE_DISTANCE'

NEAREST_NEIGHBOR = 'INTERP_NEAREST_NEIGHBOR'

class ravenpy.config.options.**LWIncomingMethod**(*value*)

Bases: Enum

An enumeration.

DATA = 'LW_INC_DATA'

DEFAULT = 'LW_INC_DEFAULT'

DINGMAN = 'LW_INC_DINGMAN'

SICART = 'LW_INC_SICART'

SKYVIEW = 'LW_INC_SKYVIEW'

class ravenpy.config.options.**LWRadiationMethod**(*value*)

Bases: Enum

An enumeration.

DATA = 'LW_RAD_DATA'

DEFAULT = 'LW_RAD_DEFAULT'

UBCWM = 'LW_RAD_UBC'

class ravenpy.config.options.**MonthlyInterpolationMethod**(*value*)

Bases: Enum

An enumeration.

LINEAR_21 = 'MONTHINT_LINEAR_21'

LINEAR_FOM = 'MONTHINT_LINEAR_FOM'

LINEAR_MID = 'MONTHINT_LINEAR_MID'

UNIFORM = 'MONTHINT_UNIFORM'

class ravenpy.config.options.**OroPETCorrect**(*value*)

Bases: Enum

An enumeration.

HBV = 'OROCORR_HBV'

NONE = 'OROCORR_NONE'

SIMPLELAPSE = 'OROCORR_SIMPLELAPSE'

class ravenpy.config.options.**OroPrecipCorrect**(*value*)

Bases: Enum

An enumeration.

HBV = 'OROCORR_HBV'

NONE = 'OROCORR_NONE'

SIMPLELAPSE = 'OROCORR_SIMPLELAPSE'

UBC = 'OROCORR_UBC'

class ravenpy.config.options.**OroTempCorrect**(*value*)

Bases: Enum

An enumeration.

HBV = 'OROCORR_HBV'

NONE = 'OROCORR_NONE'

SIMPLELAPSE = 'OROCORR_SIMPLELAPSE'

UBC = 'OROCORR_UBC'

UBC2 = 'OROCORR_UBC_2'

class ravenpy.config.options.**PotentialMeltMethod**(*value*)

Bases: Enum

:PotentialMelt algorithms

DATA = 'POTMELT_DATA'

DEGREE_DAY = 'POTMELT_DEGREE_DAY'

EB = 'POTMELT_EB'

HBV = 'POTMELT_HBV'

HMETS = 'POTMELT_HMETS'

NONE = 'POTMELT_NONE'

RESTRICTED = 'POTMELT_RESTRICTED'

UBC = 'POTMELT_UBC'

USACE = 'POTMELT_USACE'

class ravenpy.config.options.**PrecipIceptFract**(*value*)

Bases: Enum

EXPLAI = 'PRECIP_ICEPT_EXPLAI'

HEDSTROM = 'PRECIP_ICEPT_HEDSTROM'

LAI = 'PRECIP_ICEPT_LAI'

NONE = 'PRECIP_ICEPT_NONE'

USER = 'PRECIP_ICEPT_USER'

class ravenpy.config.options.**Precipitation**(*value*)

Bases: Enum

An enumeration.

DEFAULT = 'PRECIP_RAVEN'

class ravenpy.config.options.**RainSnowFraction**(*value*)

Bases: Enum

An enumeration.

DATA = 'RAINSNOW_DATA'

DINGMAN = 'RAINSNOW_DINGMAN'

HARDER = 'RAINSNOW_HARDER'

HBV = 'RAINSNOW_HBV'

HSPF = 'RAINSNOW_HSPF'

SNTHERM89 = 'RAINSNOW_SNTHERM89'

UBC = 'RAINSNOW_UBC'

WANG = 'RAINSNOW_WANG'

class ravenpy.config.options.**RelativeHumidityMethod**(*value*)

Bases: Enum

An enumeration.

CONSTANT = 'RELHUM_CONSTANT'

CORR = 'RELHUM_CORR'

DATA = 'RELHUM_DATA'

MINDEWPT = 'RELHUM_MINDEWPT'

WINDVEL = 'WINDVEL_CORR'

class ravenpy.config.options.**Routing**(*value*)

Bases: Enum

An enumeration.

DIFFUSIVE_WAVE = 'ROUTE_DIFFUSIVE_WAVE'

HYDROLOGIC = 'ROUTE_HYDROLOGIC'

MUSKINGUM = 'MUSKINGUM'

NONE = 'ROUTE_NONE'

PLUG_FLOW = 'ROUTE_PLUG_FLOW'

STORAGE_COEFF = 'ROUTE_STORAGE_COEFF'

class ravenpy.config.options.**SWCanopyCorrect**(*value*)

Bases: Enum

An enumeration.

DYNAMIC = 'SW_CANOPY_CORR_DYNAMIC'

NONE = 'SW_CANOPY_CORR_NONE'

STATIC = 'SW_CANOPY_CORR_STATIC'

UBC = 'SW_CANOPY_CORR_UBC'

class ravenpy.config.options.**SWCloudCorrect**(*value*)

Bases: Enum

An enumeration.

ANNANDALE = 'SW_CLOUD_CORR_ANNANDALE'

DINGMAN = 'SW_CLOUD_CORR_DINGMAN'

NONE = 'SW_CLOUD_CORR_NONE'

UBC = 'SW_CLOUD_CORR_UBCWM'

class ravenpy.config.options.**SWRadiationMethod**(*value*)

Bases: Enum

An enumeration.

DATA = 'SW_RAD_DATA'

DEFAULT = 'SW_RAD_DEFAULT'

UBCWM = 'SW_RAD_UBCWM'

class ravenpy.config.options.**SoilModel**(*value*)

Bases: Enum

An enumeration.

MULTILAYER = 'SOIL_MULTILAYER'

ONE_LAYER = 'SOIL_ONE_LAYER'

TWO_LAYER = 'SOIL_TWO_LAYER'

class ravenpy.config.options.**SubdailyMethod**(*value*)

Bases: Enum

An enumeration.

NONE = 'SUBDAILY_NONE'

SIMPLE = 'SUBDAILY_SIMPLE'

UBC = 'SUBDAILY_UBC'

class ravenpy.config.options.**WindspeedMethod**(*value*)

Bases: Enum

An enumeration.

CONSTANT = 'WINDVEL_CONSTANT'

DATA = 'WINDVEL_DATA'

UBC = 'WINDVEL_UBC'

ravenpy.config.parsers module

ravenpy.config.parsers.**output_files**(*run_name: str*, *path: Path*)

Return path to each output file if it exists.

ravenpy.config.parsers.**parse_diagnostics**(*fn: Path*)

Return dictionary of performance metrics.

ravenpy.config.parsers.**parse_nc**(*fn: Path*)

Open netCDF dataset with xarray if the path is valid, otherwise return None.

ravenpy.config.parsers.**parse_outputs**(*run_name: str = None*, *outputdir: [<class 'str'>, <class 'pathlib.Path'>] = None*)

Parse outputs from model execution.

Parameters

- **run_name** (*str*) – RunName value identifying model outputs.
- **outputdir** (*str or Path*) – Path to model output directory. Current directory if None.

Returns

Dictionary holding model outputs:

- hydrograph: xarray.Dataset
- storage: xarray.Dataset
- solution: Dict[str, Command]
- diagnostics: Dict[str, list]

Return type
dict

Notes

Values are set to None if no file is found.

ravenpy.config.parsers.**parse_raven_messages**(*path*)

Parse Raven_errors and extract the messages, structured by types.

ravenpy.config.parsers.**parse_solution**(*fn: Path*, *calendar: str = 'PROLEPTIC_GREGORIAN'*)

Return command objects from the model output *solution.rvc*.

ravenpy.config.processes module

class ravenpy.config.processes.**Abstraction**(***, *algo: Literal['ABST_PERCENTAGE', 'ABST_FILL', 'ABST_SCS', 'ABST_PDMROF']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['ABST_PERCENTAGE', 'ABST_FILL', 'ABST_SCS', 'ABST_PDMROF']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['ABST_PERCENTAGE', 'ABST_FILL', 'ABST_SCS', 'ABST_PDMROF'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Baseflow**(*, *algo: Literal['BASE_LINEAR', 'BASE_LINEAR_CONSTRAIN', 'BASE_LINEAR_ANALYTIC', 'BASE_POWER_LAW', 'BASE_CONSTANT', 'BASE_VIC', 'BASE_THRESH_POWER', 'BASE_THRESH_STOR', 'BASE_GR4J', 'BASE_TOPMODEL', 'BASE_SACRAMENTO']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['BASE_LINEAR', 'BASE_LINEAR_CONSTRAIN', 'BASE_LINEAR_ANALYTIC', 'BASE_POWER_LAW', 'BASE_CONSTANT', 'BASE_VIC', 'BASE_THRESH_POWER', 'BASE_THRESH_STOR', 'BASE_GR4J', 'BASE_TOPMODEL', 'BASE_SACRAMENTO']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['BASE_LINEAR', 'BASE_LINEAR_CONSTRAIN', 'BASE_LINEAR_ANALYTIC', 'BASE_POWER_LAW', 'BASE_CONSTANT', 'BASE_VIC', 'BASE_THRESH_POWER', 'BASE_THRESH_STOR', 'BASE_GR4J', 'BASE_TOPMODEL', 'BASE_SACRAMENTO'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**CanopyDrip**(*, *algo: Literal['CANDRIP_RUTTER', 'CANDRIP_SLOWDRAIN']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['CANDRIP_RUTTER', 'CANDRIP_SLOWDRAIN']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['CANDRIP_RUTTER', 'CANDRIP_SLOWDRAIN'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**CanopyEvaporation**(*, *algo: Literal['CANEVP_RUTTER', 'CANEVP_MAXIMUM', 'CANEVP_ALL']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['CANEVP_RUTTER', 'CANEVP_MAXIMUM', 'CANEVP_ALL']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['CANEVP_RUTTER', 'CANEVP_MAXIMUM', 'CANEVP_ALL'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**CanopySublimation**(*, *algo: Literal['CANEVP_ALL', 'CANEVP_MAXIMUM', 'CANSUBLIM_ALL', 'CANSUBLIM_MAXIMUM']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['CANEVP_ALL', 'CANEVP_MAXIMUM', 'CANSUBLIM_ALL', 'CANSUBLIM_MAXIMUM']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['CANEVP_ALL', 'CANEVP_MAXIMUM', 'CANSUBLIM_ALL', 'CANSUBLIM_MAXIMUM'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**CapilaryRise**(**, algo: Literal['CRISE_HBV'], source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['CRISE_HBV']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['CRISE_HBV'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**CapillaryRise**(**, algo: Literal['RISE_HBV', 'CRISE_HBV'], source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['RISE_HBV', 'CRISE_HBV']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['RISE_HBV', 'CRISE_HBV'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Conditional**(*, *kind: Literal['HRU_TYPE', 'LAND_CLASS', 'HRU_GROUP']*, *op: Literal['IS', 'IS_NOT']*, *value: str*)

Bases: Command

Conditional statement

kind: Literal['HRU_TYPE', 'LAND_CLASS', 'HRU_GROUP']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'kind': FieldInfo(annotation=Literal['HRU_TYPE', 'LAND_CLASS', 'HRU_GROUP'], required=True), 'op': FieldInfo(annotation=Literal['IS', 'IS_NOT'], required=True), 'value': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*__context: Any*) → None

This function is meant to behave like a BaseModel method to initialise private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **__context** – The context.

op: Literal['IS', 'IS_NOT']

to_rv()

Return Raven configuration string.

value: str

class ravenpy.config.processes.**Convolve**(*, *algo: Literal['CONVOL_GR4J_1', 'CONVOL_GR4J_2', 'CONVOL_GAMMA', 'CONVOL_GAMMA_2']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['CONVOL_GR4J_1', 'CONVOL_GR4J_2', 'CONVOL_GAMMA', 'CONVOL_GAMMA_2']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['CONVOL_GR4J_1', 'CONVOL_GR4J_2', 'CONVOL_GAMMA', 'CONVOL_GAMMA_2'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**CropHeatUnitEvolve**(*, *algo: Literal['CHU_ONTARIO']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['CHU_ONTARIO']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['CHU_ONTARIO'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**DepressionOverflow**(*, *algo: Literal['DFLOW_THRESHPOW', 'DFLOW_LINEAR']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['DFLOW_THRESHPOW', 'DFLOW_LINEAR']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['DFLOW_THRESHPOW', 'DFLOW_LINEAR'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Flush**(**, algo: Literal['RAVEN_DEFAULT'] = 'RAVEN_DEFAULT', source: str | None = None, to: Sequence[str] = (), p: float | None = None*)

Bases: *Process*

algo: Literal['RAVEN_DEFAULT']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['RAVEN_DEFAULT'], required=False, default='RAVEN_DEFAULT'), 'p': FieldInfo(annotation=Union[float, NoneType], required=False, default=None), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

p: float | None

class ravenpy.config.processes.**GlacierMelt**(**, algo: Literal['GMELT_SIMPLE_MELT', 'GMELT_HBV', 'GMELT_UBC'], source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['GMELT_SIMPLE_MELT', 'GMELT_HBV', 'GMELT_UBC']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['GMELT_SIMPLE_MELT', 'GMELT_HBV', 'GMELT_UBC'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**GlacierRelease**(*, *algo: Literal['GRELEASE_LINEAR', 'GRELEASE_HBV_EC']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['GRELEASE_LINEAR', 'GRELEASE_HBV_EC']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['GRELEASE_LINEAR', 'GRELEASE_HBV_EC'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Infiltration**(*, *algo: Literal['INF_RATIONAL', 'INF_SCS', 'INF_ALL_INFILTRATES', 'INF_GR4J', 'INF_GREEN_AMPT', 'INF_GA_SIMPLE', 'INF_UPSCALED_GREEN_AMPT', 'INF_HBV', 'INF_UBC', 'INF_VIC', 'INF_VIC_ARNO', 'INF_PRMS', 'INF_HMETS']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['INF_RATIONAL', 'INF_SCS', 'INF_ALL_INFILTRATES', 'INF_GR4J', 'INF_GREEN_AMPT', 'INF_GA_SIMPLE', 'INF_UPSCALED_GREEN_AMPT', 'INF_HBV', 'INF_UBC', 'INF_VIC', 'INF_VIC_ARNO', 'INF_PRMS', 'INF_HMETS']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['INF_RATIONAL', 'INF_SCS', 'INF_ALL_INFILTRATES', 'INF_GR4J', 'INF_GREEN_AMPT', 'INF_GA_SIMPLE', 'INF_UPSCALED_GREEN_AMPT', 'INF_HBV', 'INF_UBC', 'INF_VIC', 'INF_VIC_ARNO', 'INF_PRMS', 'INF_HMETS'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Interflow**(**, algo: Literal['INTERFLOW_PRMS']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['INTERFLOW_PRMS']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['INTERFLOW_PRMS'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**LakeEvaporation**(**, algo: Literal['LAKE_EVAP_BASIC']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['LAKE_EVAP_BASIC']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['LAKE_EVAP_BASIC'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**LakeFreeze**(**, algo: Literal['LFREEZE_BASIC', 'LFREEZE_THERMAL'], source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['LFREEZE_BASIC', 'LFREEZE_THERMAL']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['LFREEZE_BASIC', 'LFREEZE_THERMAL'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**LakeRelease**(**, algo: Literal['LAKEREL_LINEAR'], source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['LAKEREL_LINEAR']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['LAKEREL_LINEAR'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**LateralEquilibrate**(**, algo: str = 'RAVEN_DEFAULT', source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

Lateral equilibrate

Instantaneously equilibrates groundwater storage in basin HRUs.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=str, required=False, default='RAVEN_DEFAULT'), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**LateralFlush**(**, algo: str = 'RAVEN_DEFAULT', source: str | None = None, to: Sequence[str] = ()*)

Bases: *Process*

Lateral flush

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=str, required=False, default='RAVEN_DEFAULT'), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**OpenWaterEvaporation**(*, *algo: Literal['OPEN_WATER_EVAP', 'OPEN_WATER_RIPARIAN']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['OPEN_WATER_EVAP', 'OPEN_WATER_RIPARIAN']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['OPEN_WATER_EVAP', 'OPEN_WATER_RIPARIAN'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Overflow**(*, *algo: Literal['OVERFLOW_RAVEN', 'RAVEN_DEFAULT']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['OVERFLOW_RAVEN', 'RAVEN_DEFAULT']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['OVERFLOW_RAVEN', 'RAVEN_DEFAULT'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Percolation**(*, *algo: Literal['PERC_GAWSER', 'PERC_LINEAR', 'PERC_POWER_LAW', 'PERC_PRMS', 'PERC_SACRAMENTO', 'PERC_CONSTANT', 'PERC_GR4J', 'PERC_GR4JEXCH', 'PERC_GR4JEXCH2']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['PERC_GAWSER', 'PERC_LINEAR', 'PERC_POWER_LAW', 'PERC_PRMS', 'PERC_SACRAMENTO', 'PERC_CONSTANT', 'PERC_GR4J', 'PERC_GR4JEXCH', 'PERC_GR4JEXCH2']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['PERC_GAWSER', 'PERC_LINEAR', 'PERC_POWER_LAW', 'PERC_PRMS', 'PERC_SACRAMENTO', 'PERC_CONSTANT', 'PERC_GR4J', 'PERC_GR4JEXCH', 'PERC_GR4JEXCH2'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Precipitation**(*, *algo: Literal['PRECIP_RAVEN', 'RAVEN_DEFAULT'] = 'PRECIP_RAVEN'*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['PRECIP_RAVEN', 'RAVEN_DEFAULT']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['PRECIP_RAVEN', 'RAVEN_DEFAULT'], required=False, default='PRECIP_RAVEN'), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**ProcessGroup**(*, *p: Sequence[Process]*, *params: Sequence[Variable | Expression | float | None]*)

Bases: Command

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'p': FieldInfo(annotation=Sequence[ravenpy.config.commands.Process], required=True), 'params': FieldInfo(annotation=Sequence[Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

p: Sequence[*Process*]

params: Sequence[Variable | Expression | float | None]

to_rv()

Return Raven configuration string.

class ravenpy.config.processes.**Seepage**(*, *algo: Literal['SEEP_LINEAR']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SEEP_LINEAR']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SEEP_LINEAR'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SnowAlbedoEvolve**(*, *algo: Literal['SNOALB_UBC']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SNOALB_UBC']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SNOALB_UBC'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SnowBalance**(*, *algo: Literal['SNOBAL_SIMPLE_MELT', 'SNOBAL_COLD_CONTENT', 'SNOBAL_HBV', 'SNOBAL_TWO_LAYER', 'SNOBAL_CEMA_NIEGE', 'SNOBAL_HMETS', 'SNOWBAL_GAWSER', 'SNOWBAL_UBC']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SNOBAL_SIMPLE_MELT', 'SNOBAL_COLD_CONTENT', 'SNOBAL_HBV', 'SNOBAL_TWO_LAYER', 'SNOBAL_CEMA_NIEGE', 'SNOBAL_HMETS', 'SNOWBAL_GAWSER', 'SNOWBAL_UBC']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SNOBAL_SIMPLE_MELT', 'SNOBAL_COLD_CONTENT', 'SNOBAL_HBV', 'SNOBAL_TWO_LAYER', 'SNOBAL_CEMA_NIEGE', 'SNOBAL_HMETS', 'SNOWBAL_GAWSER', 'SNOWBAL_UBC'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SnowMelt**(*, *algo: Literal['MELT_POTMELT']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['MELT_POTMELT']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['MELT_POTMELT'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SnowRefreeze**(*, *algo: Literal['FREEZE_DEGREE_DAY']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['FREEZE_DEGREE_DAY']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['FREEZE_DEGREE_DAY'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SnowTempEvolve**(*, *algo: Literal['SNOTEMP_NEWTONS']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SNOTEMP_NEWTONS']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SNOTEMP_NEWTONS'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SoilBalance**(*, *algo: Literal['SOILBAL_SACSMA']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SOILBAL_SACSMA']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SOILBAL_SACSMA'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**SoilEvaporation**(*, *algo: Literal['SOILEVAP_VIC', 'SOILEVAP_HBV', 'SOILEVAP_HYPR', 'SOILEVAL_CHU', 'SOILEVAP_UBC', 'SOILEVAP_GR4J', 'SOILEVAP_TOPMODEL', 'SOILEVAP_SEQUEN', 'SOILEVAP_ROOT', 'SOILEVAP_ROOT_CONSTRAIN', 'SOILEVAP_ROOTFRAC', 'SOILEVAP_GAWSER', 'SOILEVAP_FEDERER', 'SOILEVAP_ALL', 'SOILEVAP_LINEAR', 'SOILEVAP_SACSMA', 'SOILEVAP_HYMOD2']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SOILEVAP_VIC', 'SOILEVAP_HBV', 'SOILEVAP_HYPR', 'SOILEVAL_CHU', 'SOILEVAP_UBC', 'SOILEVAP_GR4J', 'SOILEVAP_TOPMODEL', 'SOILEVAP_SEQUEN', 'SOILEVAP_ROOT', 'SOILEVAP_ROOT_CONSTRAIN', 'SOILEVAP_ROOTFRAC', 'SOILEVAP_GAWSER', 'SOILEVAP_FEDERER', 'SOILEVAP_ALL', 'SOILEVAP_LINEAR', 'SOILEVAP_SACSMA', 'SOILEVAP_HYMOD2']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SOILEVAP_VIC', 'SOILEVAP_HBV', 'SOILEVAP_HYPR', 'SOILEVAL_CHU', 'SOILEVAP_UBC', 'SOILEVAP_GR4J', 'SOILEVAP_TOPMODEL', 'SOILEVAP_SEQUEN', 'SOILEVAP_ROOT', 'SOILEVAP_ROOT_CONSTRAIN', 'SOILEVAP_ROOTFRAC', 'SOILEVAP_GAWSER', 'SOILEVAP_FEDERER', 'SOILEVAP_ALL', 'SOILEVAP_LINEAR', 'SOILEVAP_SACSMA', 'SOILEVAP_HYMOD2'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

class ravenpy.config.processes.**Split**(*, *algo: str = 'RAVEN_DEFAULT'*, *source: str | None = None*, *to: Tuple[str, str]*, *p: float = None*)

Bases: *Process*

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=str, required=False, default='RAVEN_DEFAULT'), 'p': FieldInfo(annotation=float, required=False, default=None), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Tuple[str, str], required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

p: float

to: Tuple[str, str]

class ravenpy.config.processes.**Sublimation**(**, algo: Literal['SUBLIM_SVERDRUP', 'SUBLIM_KUZMIN', 'SUBLIM_CENTRAL_SIERRA', 'SUBLIM_PSBM', 'SUBLIM_WILLIAMS']*, *source: str | None = None*, *to: Sequence[str] = ()*)

Bases: *Process*

algo: Literal['SUBLIM_SVERDRUP', 'SUBLIM_KUZMIN', 'SUBLIM_CENTRAL_SIERRA', 'SUBLIM_PSBM', 'SUBLIM_WILLIAMS']

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'algo': FieldInfo(annotation=Literal['SUBLIM_SVERDRUP', 'SUBLIM_KUZMIN', 'SUBLIM_CENTRAL_SIERRA', 'SUBLIM_PSBM', 'SUBLIM_WILLIAMS'], required=True), 'source': FieldInfo(annotation=Union[str, NoneType], required=False, default=None), 'to': FieldInfo(annotation=Sequence[str], required=False, default=())}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

model_post_init(*_ModelMetaclass__context: Any*) → None

We need to both initialize private attributes and call the user-defined model_post_init method.

ravenpy.config.rvs module

class ravenpy.config.rvs.**Config**(*, *EnKFMode: EnKFMode | None = None*, *WindowSize: int | None = None*, *SolutionRunName: str | None = None*, *ExtraRVTFilename: str | None = None*, *OutputDirectoryFormat: str | Path | None = None*, *ForecastRVTFilename: str | None = None*, *TruncateHindcasts: bool | None = None*, *ForcingPerturbation: Sequence[ForcingPerturbation] | None = None*, *AssimilatedState: Sequence[AssimilatedState] | None = None*, *AssimilateStreamflow: Sequence[AssimilateStreamflow] | None = None*, *ObservationErrorModel: Sequence[ObservationErrorModel] | None = None*, *params: Any = None*, *SoilClasses: SoilClasses | None = None*, *SoilProfiles: SoilProfiles | None = None*, *VegetationClasses: VegetationClasses | None = None*, *LandUseClasses: LandUseClasses | None = None*, *TerrainClasses: TerrainClasses | None = None*, *SoilParameterList: SoilParameterList | None = None*, *LandUseParameterList: LandUseParameterList | None = None*, *VegetationParameterList: VegetationParameterList | None = None*, *ChannelProfile: Sequence[ChannelProfile] | None = None*, *GlobalParameter: Dict[str, Variable | Expression | float | None] | None = {}*, *RainSnowTransition: RainSnowTransition | None = None*, *SeasonalRelativeLAI: SeasonalRelativeLAI | None = None*, *SeasonalRelativeHeight: SeasonalRelativeHeight | None = None*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*, *SubBasins: SubBasins | None = None*, *SubBasinGroup: Sequence[SubBasinGroup] | None = None*, *SubBasinProperties: SubBasinProperties | None = None*, *SBGroupPropertyMultiplier: Sequence[SBGroupPropertyMultiplier] | None = None*, *HRUs: HRUs | None = None*, *HRUGroup: Sequence[HRUGroup] | None = None*, *Reservoirs: Sequence[Reservoir] | None = None*, *HRUStateVariableTable: HRUStateVariableTable | None = None*, *BasinStateVariables: BasinStateVariables | None = None*, *UniformInitialConditions: Dict[str, Variable | Expression | float | None] | None = None*, *SilentMode: bool | None = None*, *NoisyMode: bool | None = None*, *RunName: str | None = None*, *Calendar: Calendar | None = None*, *StartDate: date | datetime | datetime | None = None*, *AssimilationStartTime: date | datetime | datetime | None = None*, *EndDate: date | datetime | datetime | None = None*, *Duration: float | None = None*, *TimeStep: float | str | None = None*, *Routing: Routing | None = None*, *CatchmentRoute: CatchmentRoute | None = None*, *Evaporation: Evaporation | None = None*, *OW_Evaporation: Evaporation | None = None*, *SWRadiationMethod: SWRadiationMethod | None = None*, *SWCloudCorrect: SWCloudCorrect | None = None*, *SWCanopyCorrect: SWCanopyCorrect | None = None*, *LWRadiationMethod: LWRadiationMethod | None = None*, *WindspeedMethod: WindspeedMethod | None = None*, *RainSnowFraction: RainSnowFraction | None = None*, *PotentialMeltMethod: PotentialMeltMethod | None = None*, *OroTempCorrect: OroTempCorrect | None = None*, *OroPrecipCorrect: OroPrecipCorrect | None = None*, *OroPETCorrect: OroPETCorrect | None = None*, *CloudCoverMethod: CloudCoverMethod | None = None*, *PrecipIceptFract: PrecipIceptFract | None = None*, *SubdailyMethod: SubdailyMethod | None = None*, *MonthlyInterpolationMethod: MonthlyInterpolationMethod | None = None*, *SoilModel: SoilModel | None = None*, *TemperatureCorrection: bool | None = None*, *LakeStorage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH',*

Bases: *RVI*, *RVC*, *RVH*, *RVT*, *RVP*, *RVE*

duplicate(***kwds*)

Duplicate this model, changing the values given in the keywords.

header(*rv*)

Return the header to print at the top of each RV file.

property is_symbolic

Return True if configuration contains symbolic expressions.

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'assimilation_start_time': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2, validate_default=False), 'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'calendar': FieldInfo(annotation=Union[Calendar, NoneType], required=False, default_factory=<lambda>, alias='Calendar', alias_priority=2, validate_default=False), 'catchment_route': FieldInfo(annotation=Union[CatchmentRoute, NoneType], required=False, default_factory=<lambda>, alias='CatchmentRoute', alias_priority=2, validate_default=False), 'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'cloud_cover_method': FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False, default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2, validate_default=False), 'custom_output': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput], NoneType], required=False, default_factory=<lambda>, alias='CustomOutput', alias_priority=2, validate_default=False), 'debug_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DebugMode', alias_priority=2, validate_default=False), 'define_hru_groups': FieldInfo(annotation=Union[Sequence[str], NoneType], required=False, default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2, validate_default=False), 'deltares_fews_mode': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode', alias_priority=2, validate_default=False), 'direct_evaporation': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2, description='Rainfall is automatically reduced through evapotranspiration up to the limit of the calculated PET.', validate_default=False), 'dont_write_watershed_storage': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage', alias_priority=2, description='Do not write watershed storage variables to disk.', validate_default=False), 'duration': FieldInfo(annotation=Union[float, NoneType], required=False, default_factory=<lambda>, alias='Duration', alias_priority=2, validate_default=False), 'end_date': FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'ensemble_mode': FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False, default_factory=<lambda>, alias='EnsembleMode', alias_priority=2, validate_default=False), 'evaluation_metrics': FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics], NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics', alias_priority=2, validate_default=False), 'evaluation_period': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod], NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod', alias_priority=2, validate_default=False), 'evaporation': FieldInfo(annotation=Union[Evaporation, NoneType], required=False, default_factory=<lambda>, alias='Evaporation', alias_priority=2,

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

property rvc

property rve

property rvh

property rvi

property rvp

property rvt

set_params(*params: Dict | Sequence*) → *Config*

Return a new instance of Config with params frozen to their numerical values.

set_solution(*fn: Path*, *timestamp: bool = True*) → *Config*

Return a new instance of Config with hru, basin states and start date set from an existing solution.

Parameters

- **fn** (*Path*) – Path to solution file.
- **timestamp** (*bool*) – If False, ignore time stamp information in the solution. If True, the solution will set StartDate to the solution's timestamp.

Returns
Config with internal state set from the solution file.

Return type
Config

write_rv(*workdir: str | Path*, *modelname=None*, *overwrite=False*, *header=True*)

Write configuration files to disk.

Parameters

- **workdir** (*str, Path*) – A directory where rv files will be written to disk.
- **modelname** (*str*) – File name stem for rv files. If not given, defaults to *RunName* if set, otherwise *raven*.
- **overwrite** (*bool*) – If True, overwrite existing configuration files.
- **header** (*bool*) – If True, write a header at the top of each RV file.

zip(*workdir: str | Path*, *modelname=None*, *overwrite=False*)

Write configuration to zip file.

Parameters

- **workdir** (*Path, str*) – Path to zip archive storing RV files.
- **modelname** (*str*) – File name stem for rv files. If not given, defaults to *RunName* if set, otherwise *raven*.
- **overwrite** (*bool*) – If True, overwrite existing configuration zip file.

class ravenpy.config.rvs.**RVC**(*, *HRUStateVariableTable:* HRUStateVariableTable | *None* = *None*, *BasinStateVariables:* BasinStateVariables | *None* = *None*, *UniformInitialConditions: Dict[str, Variable | Expression | float | None] | None* = *None*)

Bases: RV

basin_state_variables: *BasinStateVariables* **| None**

hru_state_variable_table: *HRUStateVariableTable* **| None**

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'basin_state_variables': FieldInfo(annotation=Union[BasinStateVariables, NoneType], required=False, default_factory=<lambda>, alias='BasinStateVariables', alias_priority=2, validate_default=False), 'hru_state_variable_table': FieldInfo(annotation=Union[HRUStateVariableTable, NoneType], required=False, default_factory=<lambda>, alias='HRUStateVariableTable', alias_priority=2, validate_default=False), 'uniform_initial_conditions': FieldInfo(annotation=Union[Dict[str, Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], NoneType], required=False, default_factory=<lambda>, alias='UniformInitialConditions', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

uniform_initial_conditions: Dict[str, Variable | Expression | float | None] | None

class ravenpy.config.rvs.**RVE**(*, *EnKFMode: EnKFMode | None* = *None*, *WindowSize: int | None* = *None*, *SolutionRunName: str | None* = *None*, *ExtraRVTFilename: str | None* = *None*, *OutputDirectoryFormat: str | Path | None* = *None*, *ForecastRVTFilename: str | None* = *None*, *TruncateHindcasts: bool | None* = *None*, *ForcingPerturbation: Sequence[*ForcingPerturbation*] | None* = *None*, *AssimilatedState: Sequence[*AssimilatedState*] | None* = *None*, *AssimilateStreamflow: Sequence[*AssimilateStreamflow*] | None* = *None*, *ObservationErrorModel: Sequence[*ObservationErrorModel*] | None* = *None*)

Bases: RV

assimilate_streamflow: *Sequence[AssimilateStreamflow]* **| None**

assimilated_state: *Sequence[AssimilatedState]* **| None**

enkf_mode: *EnKFMode* **| None**

extra_rvt_filename: str | None

forcing_perturbation: *Sequence[ForcingPerturbation]* **| None**

forecast_rvt_filename: str | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'assimilate_streamflow': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilateStreamflow], NoneType], required=False, default_factory=<lambda>, alias='AssimilateStreamflow', alias_priority=2, validate_default=False), 'assimilated_state': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.AssimilatedState], NoneType], required=False, default_factory=<lambda>, alias='AssimilatedState', alias_priority=2, validate_default=False), 'enkf_mode': FieldInfo(annotation=Union[EnKFMode, NoneType], required=False, default_factory=<lambda>, alias='EnKFMode', alias_priority=2, validate_default=False), 'extra_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='ExtraRVTFilename', alias_priority=2, validate_default=False), 'forcing_perturbation': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ForcingPerturbation], NoneType], required=False, default_factory=<lambda>, alias='ForcingPerturbation', alias_priority=2, validate_default=False), 'forecast_rvt_filename': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='ForecastRVTFilename', alias_priority=2, validate_default=False), 'observation_error_model': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ObservationErrorModel], NoneType], required=False, default_factory=<lambda>, alias='ObservationErrorModel', alias_priority=2, validate_default=False), 'output_directory_format': FieldInfo(annotation=Union[str, Path, NoneType], required=False, default_factory=<lambda>, alias='OutputDirectoryFormat', alias_priority=2, validate_default=False), 'solution_run_name': FieldInfo(annotation=Union[str, NoneType], required=False, default_factory=<lambda>, alias='SolutionRunName', alias_priority=2, validate_default=False), 'truncate_hindcasts': FieldInfo(annotation=Union[bool, NoneType], required=False, default_factory=<lambda>, alias='TruncateHindcasts', alias_priority=2, validate_default=False), 'window_size': FieldInfo(annotation=Union[int, NoneType], required=False, default_factory=<lambda>, alias='WindowSize', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

observation_error_model: Sequence[*ObservationErrorModel*] | None

output_directory_format: str | Path | None

solution_run_name: str | None

truncate_hindcasts: bool | None

window_size: int | None

class ravenpy.config.rvs.**RVH**(*, *SubBasins:* SubBasins | *None* = *None*, *SubBasinGroup:* *Sequence[*SubBasinGroup*]* | *None* = *None*, *SubBasinProperties:* SubBasinProperties | *None* = *None*, *SBGroupPropertyMultiplier:* *Sequence[*SBGroupPropertyMultiplier*]* | *None* = *None*, *HRUs:* HRUs | *None* = *None*, *HRUGroup: Sequence[*HRUGroup*]* | *None* = *None*, *Reservoirs:* *Sequence[*Reservoir*]* | *None* = *None*)

Bases: RV

hru_group: Sequence[*HRUGroup*] | None

hrus: *HRUs* | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'hru_group': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.HRUGroup], NoneType], required=False, default_factory=<lambda>, alias='HRUGroup', alias_priority=2, validate_default=False), 'hrus': FieldInfo(annotation=Union[HRUs, NoneType], required=False, default_factory=<lambda>, alias='HRUs', alias_priority=2, validate_default=False), 'reservoirs': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.Reservoir], NoneType], required=False, default_factory=<lambda>, alias='Reservoirs', alias_priority=2, validate_default=False), 'sb_group_property_multiplier': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.SBGroupPropertyMultiplier], NoneType], required=False, default_factory=<lambda>, alias='SBGroupPropertyMultiplier', alias_priority=2, validate_default=False), 'sub_basin_group': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.SubBasinGroup], NoneType], required=False, default_factory=<lambda>, alias='SubBasinGroup', alias_priority=2, validate_default=False), 'sub_basin_properties': FieldInfo(annotation=Union[SubBasinProperties, NoneType], required=False, default_factory=<lambda>, alias='SubBasinProperties', alias_priority=2, validate_default=False), 'sub_basins': FieldInfo(annotation=Union[SubBasins, NoneType], required=False, default_factory=<lambda>, alias='SubBasins', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

reservoirs: Sequence[*Reservoir*] | None

sb_group_property_multiplier: Sequence[*SBGroupPropertyMultiplier*] | None

sub_basin_group: Sequence[*SubBasinGroup*] | None

sub_basin_properties: *SubBasinProperties* | None

sub_basins: *SubBasins* | None

class ravenpy.config.rvs.**RVI**(*, *SilentMode: bool | None = None*, *NoisyMode: bool | None = None*, *RunName: str | None = None*, *Calendar: Calendar | None = None*, *StartDate: date | datetime | datetime | None = None*, *AssimilationStartTime: date | datetime | datetime | None = None*, *EndDate: date | datetime | datetime | None = None*, *Duration: float | None = None*, *TimeStep: float | str | None = None*, *Routing: Routing | None = None*, *CatchmentRoute: CatchmentRoute | None = None*, *Evaporation: Evaporation | None = None*, *OW_Evaporation: Evaporation | None = None*, *SWRadiationMethod: SWRadiationMethod | None = None*, *SWCloudCorrect: SWCloudCorrect | None = None*, *SWCanopyCorrect: SWCanopyCorrect | None = None*, *LWRadiationMethod: LWRadiationMethod | None = None*, *WindspeedMethod: WindspeedMethod | None = None*, *RainSnowFraction: RainSnowFraction | None = None*, *PotentialMeltMethod: PotentialMeltMethod | None = None*, *OroTempCorrect: OroTempCorrect | None = None*, *OroPrecipCorrect: OroPrecipCorrect | None = None*, *OroPETCorrect: OroPETCorrect | None = None*, *CloudCoverMethod: CloudCoverMethod | None = None*, *PrecipIceptFract: PrecipIceptFract | None = None*, *SubdailyMethod: SubdailyMethod | None = None*, *MonthlyInterpolationMethod: MonthlyInterpolationMethod | None = None*, *SoilModel: SoilModel | None = None*, *TemperatureCorrection: bool | None = None*, *LakeStorage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | None = None*, *DefineHRUGroups: Sequence[str] | None = None*, *HydrologicProcesses: Sequence[Process | Conditional | ProcessGroup] | None = None*, *EvaluationMetrics: Sequence[EvaluationMetrics] | None = None*, *EvaluationPeriod: Sequence[EvaluationPeriod] | None = None*, *EnsembleMode: EnsembleMode | None = None*, *WriteNetcdfFormat: bool | None = None*, *NetCDFAttribute: Dict[str, str] | None = None*, *CustomOutput: Sequence[CustomOutput] | None = None*, *DirectEvaporation: bool | None = None*, *DeltaresFEWSMode: bool | None = None*, *DebugMode: bool | None = None*, *DontWriteWatershedStorage: bool | None = None*, *PavicsMode: bool | None = None*, *SuppressOutput: bool | None = None*, *WriteForcingFunctions: bool | None = None*, *WriteSubbasinFile: bool | None = None*, *WriteLocalFlows: bool | None = None*)

Bases: RV

assimilation_start_time: date | datetime | datetime | None

calendar: Calendar | None

catchment_route: CatchmentRoute | None

cloud_cover_method: CloudCoverMethod | None

custom_output: Sequence[*CustomOutput*] | None

classmethod dates2cf(*v*, *info*)

Convert dates to cftime dates.

debug_mode: bool | None

define_hru_groups: Sequence[str] | None

deltares_fews_mode: bool | None

direct_evaporation: bool | None

dont_write_watershed_storage: bool | None

duration: float | None

end_date: date | datetime | datetime | None

ensemble_mode: *EnsembleMode* | None

evaluation_metrics: Sequence[EvaluationMetrics] | None

evaluation_period: Sequence[*EvaluationPeriod*] | None

evaporation: Evaporation | None

hydrologic_processes: Sequence[*Process* | Conditional | ProcessGroup] | None

classmethod init_soil_model(*v*)

lake_storage: Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP', 'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY', 'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW', 'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR', 'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP', 'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE', 'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT', 'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'] | None

lw_radiation_method: LWRadiationMethod | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

```
model_fields:  ClassVar[dict[str, FieldInfo]] = {'assimilation_start_time':
FieldInfo(annotation=Union[date, datetime, datetime, NoneType], required=False,
default_factory=<lambda>, alias='AssimilationStartTime', alias_priority=2,
validate_default=False), 'calendar':  FieldInfo(annotation=Union[Calendar,
NoneType], required=False, default_factory=<lambda>, alias='Calendar',
alias_priority=2, validate_default=False), 'catchment_route':
FieldInfo(annotation=Union[CatchmentRoute, NoneType], required=False,
default_factory=<lambda>, alias='CatchmentRoute', alias_priority=2,
validate_default=False), 'cloud_cover_method':
FieldInfo(annotation=Union[CloudCoverMethod, NoneType], required=False,
default_factory=<lambda>, alias='CloudCoverMethod', alias_priority=2,
validate_default=False), 'custom_output':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.CustomOutput],
NoneType], required=False, default_factory=<lambda>, alias='CustomOutput',
alias_priority=2, validate_default=False), 'debug_mode':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DebugMode', alias_priority=2,
validate_default=False), 'define_hru_groups':
FieldInfo(annotation=Union[Sequence[str], NoneType], required=False,
default_factory=<lambda>, alias='DefineHRUGroups', alias_priority=2,
validate_default=False), 'deltares_fews_mode':  FieldInfo(annotation=Union[bool,
NoneType], required=False, default_factory=<lambda>, alias='DeltaresFEWSMode',
alias_priority=2, validate_default=False), 'direct_evaporation':
FieldInfo(annotation=Union[bool, NoneType], required=False,
default_factory=<lambda>, alias='DirectEvaporation', alias_priority=2,
description='Rainfall is automatically reduced through evapotranspiration up to the
limit of the calculated PET.', validate_default=False),
'dont_write_watershed_storage':  FieldInfo(annotation=Union[bool, NoneType],
required=False, default_factory=<lambda>, alias='DontWriteWatershedStorage',
alias_priority=2, description='Do not write watershed storage variables to disk.',
validate_default=False), 'duration':  FieldInfo(annotation=Union[float, NoneType],
required=False, default_factory=<lambda>, alias='Duration', alias_priority=2,
validate_default=False), 'end_date':  FieldInfo(annotation=Union[date, datetime,
datetime, NoneType], required=False, default_factory=<lambda>, alias='EndDate',
alias_priority=2, validate_default=False), 'ensemble_mode':
FieldInfo(annotation=Union[EnsembleMode, NoneType], required=False,
default_factory=<lambda>, alias='EnsembleMode', alias_priority=2,
validate_default=False), 'evaluation_metrics':
FieldInfo(annotation=Union[Sequence[ravenpy.config.options.EvaluationMetrics],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationMetrics',
alias_priority=2, validate_default=False), 'evaluation_period':
FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.EvaluationPeriod],
NoneType], required=False, default_factory=<lambda>, alias='EvaluationPeriod',
alias_priority=2, validate_default=False), 'evaporation':
FieldInfo(annotation=Union[Evaporation, NoneType], required=False,
default_factory=<lambda>, alias='Evaporation', alias_priority=2,
validate_default=False), 'hydrologic_processes':
FieldInfo(annotation=Union[Sequence[Union[ravenpy.config.commands.Process,
ravenpy.config.processes.Conditional, ravenpy.config.processes.ProcessGroup]],
NoneType], required=False, default_factory=<lambda>, alias='HydrologicProcesses',
alias_priority=2, validate_default=False), 'lake_storage':
FieldInfo(annotation=Union[Literal['SURFACE_WATER', 'ATMOSPHERE', 'ATMOS_PRECIP',
'PONDED_WATER', 'SOIL', 'SOIL[0]', 'SOIL[1]', 'SOIL[2]', 'GROUNDWATER', 'CANOPY',
'CANOPY_SNOW', 'TRUNK', 'ROOT', 'DEPRESSION', 'WETLAND', 'LAKE_STORAGE', 'SNOW',
'SNOW_LIQ', 'GLACIER', 'GLACIER_ICE', 'CONVOLUTION', 'CONV_STOR',
'SURFACE_WATER_TEMP', 'SNOW_TEMP', 'COLD_CONTENT', 'GLACIER_CC', 'SOIL_TEMP',
'CANOPY_TEMP', 'SNOW_DEPTH', 'PERMAFROST_DEPTH', 'SNOW_COVER', 'SNOW_AGE',
'SNOW_ALBEDO', 'CROP_HEAT_UNITS', 'CUM_INFIL', 'CUM_SNOWMELT', 'CONSTITUENT',
'CONSTITUENT_SRC', 'CONSTITUENT_SW', 'CONSTITUENT_SINK', 'MULTIPLE'], NoneType],
required=False, default_factory=<lambda>, alias='LakeStorage', alias_priority=2,
```

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

monthly_interpolation_method: MonthlyInterpolationMethod | None

netcdf_attribute: Dict[str, str] | None

noisy_mode: bool | None

oro_pet_correct: OroPETCorrect | None

oro_precip_correct: OroPrecipCorrect | None

oro_temp_correct: OroTempCorrect | None

ow_evaporation: Evaporation | None

pavics_mode: bool | None

potential_melt_method: PotentialMeltMethod | None

precip_icept_frac: PrecipIceptFract | None

rain_snow_fraction: RainSnowFraction | None

routing: Routing | None

run_name: str | None

silent_mode: bool | None

soil_model: *SoilModel* | None

start_date: date | datetime | datetime | None

subdaily_method: SubdailyMethod | None

suppress_output: bool | None

sw_canopy_correct: SWCanopyCorrect | None

sw_cloud_correct: SWCloudCorrect | None

sw_radiation_method: SWRadiationMethod | None

temperature_correction: bool | None

time_step: float | str | None

windspeed_method: WindspeedMethod | None

write_forcing_functions: bool | None

write_local_flows: bool | None

write_netcdf_format: bool | None

write_subbasin_file: bool | None

class ravenpy.config.rvs.RVP(*, *params: Any = None*, *SoilClasses:* SoilClasses | *None = None*, *SoilProfiles:* SoilProfiles | *None = None*, *VegetationClasses:* VegetationClasses | *None = None*, *LandUseClasses:* LandUseClasses | *None = None*, *TerrainClasses:* TerrainClasses | *None = None*, *SoilParameterList:* SoilParameterList | *None = None*, *LandUseParameterList:* LandUseParameterList | *None = None*, *VegetationParameterList:* VegetationParameterList | *None = None*, *ChannelProfile: Sequence[*ChannelProfile*] | None = None*, *GlobalParameter: Dict[str, Variable | Expression | float | None] | None = {}*, *RainSnowTransition:* RainSnowTransition | *None = None*, *SeasonalRelativeLAI:* SeasonalRelativeLAI | *None = None*, *SeasonalRelativeHeight:* SeasonalRelativeHeight | *None = None*)

Bases: RV

channel_profile: Sequence[*ChannelProfile*] | None

global_parameter: Dict[str, Variable | Expression | float | None] | None

land_use_classes: *LandUseClasses* | None

land_use_parameter_list: *LandUseParameterList* | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'channel_profile': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ChannelProfile], NoneType], required=False, default_factory=<lambda>, alias='ChannelProfile', alias_priority=2, validate_default=False), 'global_parameter': FieldInfo(annotation=Union[Dict[str, Union[pymbolic.primitives.Variable, pymbolic.primitives.Expression, float, NoneType]], NoneType], required=False, default={}, alias='GlobalParameter', alias_priority=2), 'land_use_classes': FieldInfo(annotation=Union[LandUseClasses, NoneType], required=False, default_factory=<lambda>, alias='LandUseClasses', alias_priority=2, validate_default=False), 'land_use_parameter_list': FieldInfo(annotation=Union[LandUseParameterList, NoneType], required=False, default_factory=<lambda>, alias='LandUseParameterList', alias_priority=2, validate_default=False), 'params': FieldInfo(annotation=Any, required=False, default=None), 'rain_snow_transition': FieldInfo(annotation=Union[RainSnowTransition, NoneType], required=False, default_factory=<lambda>, alias='RainSnowTransition', alias_priority=2, validate_default=False), 'seasonal_relative_height': FieldInfo(annotation=Union[SeasonalRelativeHeight, NoneType], required=False, default_factory=<lambda>, alias='SeasonalRelativeHeight', alias_priority=2, validate_default=False), 'seasonal_relative_lai': FieldInfo(annotation=Union[SeasonalRelativeLAI, NoneType], required=False, default_factory=<lambda>, alias='SeasonalRelativeLAI', alias_priority=2, validate_default=False), 'soil_classes': FieldInfo(annotation=Union[SoilClasses, NoneType], required=False, default_factory=<lambda>, alias='SoilClasses', alias_priority=2, validate_default=False), 'soil_parameter_list': FieldInfo(annotation=Union[SoilParameterList, NoneType], required=False, default_factory=<lambda>, alias='SoilParameterList', alias_priority=2, validate_default=False), 'soil_profiles': FieldInfo(annotation=Union[SoilProfiles, NoneType], required=False, default_factory=<lambda>, alias='SoilProfiles', alias_priority=2, validate_default=False), 'terrain_classes': FieldInfo(annotation=Union[TerrainClasses, NoneType], required=False, default_factory=<lambda>, alias='TerrainClasses', alias_priority=2, validate_default=False), 'vegetation_classes': FieldInfo(annotation=Union[VegetationClasses, NoneType], required=False, default_factory=<lambda>, alias='VegetationClasses', alias_priority=2, validate_default=False), 'vegetation_parameter_list': FieldInfo(annotation=Union[VegetationParameterList, NoneType], required=False, default_factory=<lambda>, alias='VegetationParameterList', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

params: Any

rain_snow_transition: *RainSnowTransition* | None

seasonal_relative_height: *SeasonalRelativeHeight* | None

seasonal_relative_lai: *SeasonalRelativeLAI* | None

soil_classes: *SoilClasses* | None

soil_parameter_list: *SoilParameterList* | None

soil_profiles: *SoilProfiles* | None

terrain_classes: *TerrainClasses* | None

vegetation_classes: *VegetationClasses* | None

vegetation_parameter_list: *VegetationParameterList* | None

class ravenpy.config.rvs.**RVT**(*, *Gauge: Sequence[Gauge] | None = None*, *StationForcing: Sequence[StationForcing] | None = None*, *GriddedForcing: Sequence[GriddedForcing] | None = None*, *ObservationData: Sequence[ObservationData] | None = None*)

Bases: RV

gauge: Sequence[*Gauge*] | None

gridded_forcing: Sequence[*GriddedForcing*] | None

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True, 'extra': 'forbid', 'populate_by_name': True, 'validate_assignment': True, 'validate_default': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'gauge': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.Gauge], NoneType], required=False, default_factory=<lambda>, alias='Gauge', alias_priority=2, validate_default=False), 'gridded_forcing': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.GriddedForcing], NoneType], required=False, default_factory=<lambda>, alias='GriddedForcing', alias_priority=2, validate_default=False), 'observation_data': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.ObservationData], NoneType], required=False, default_factory=<lambda>, alias='ObservationData', alias_priority=2, validate_default=False), 'station_forcing': FieldInfo(annotation=Union[Sequence[ravenpy.config.commands.StationForcing], NoneType], required=False, default_factory=<lambda>, alias='StationForcing', alias_priority=2, validate_default=False)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

observation_data: Sequence[*ObservationData*] | None

station_forcing: Sequence[*StationForcing*] | None

ravenpy.config.rvs.**is_symbolic**(*params: dict*) → bool

Return True if parameters include a symbolic variable.

ravenpy.config.utils module

ravenpy.config.utils.**filter_for**(*kls*, *attrs*, ***kwds*)

Return attributes that are fields of dataclass.

Notes

If attrs includes an attribute name and its Raven alias, e.g. *linear_transform* and *LinearTransform*, the latter will have priority.

ravenpy.config.utils.**get_annotations**(*a*)

Return all annotations inside [] or Union[...].

ravenpy.config.utils.**get_average_annual_runoff**(*nc_file_path: str | PathLike[str]*, *area_in_m2: float*, *time_dim: str = 'time'*, *obs_var: str = 'qobs'*, *na_value: int | float = -1.2345*)

Compute the average annual runoff from observed data.

ravenpy.config.utils.**nc_specs**(*fn: str | PathLike[str]*, *data_type: str*, *station_idx: int | None = None*, *alt_names: str | Sequence[str] | None = None*, *mon_ave: bool = False*, *engine: str = 'h5netcdf'*, *linear_transform=None*)

Extract specifications from netCDF file.

Parameters

- **fn** (*str, Path*) – NetCDF file path or DAP link.
- **data_type** (*str*) – Raven data type.
- **station_idx** (*int, optional*) – Index along station dimension. Starts at 1.
- **alt_names** (*str, list*) – Alternative variable names for data type if not the CF standard default.
- **mon_ave** (*bool*) – If True, compute the monthly average.
- **engine** (*str*) – The engine used to open the dataset. Default is 'h5netcdf'.

Return type
dict

Notes

Attributes: var_name_nc, dim_names_nc, units, linear_transform, latitude_var_name_nc, longitude_var_name_nc, elevation_var_name_nc latitude, longitude, elevation, name

ravenpy.extractors package

Submodules

ravenpy.extractors.forecasts module

ravenpy.extractors.forecasts.**get_CASPAR_dataset**(*climate_model: str*, *date: datetime*, *thredds: str = 'https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/'*, *directory: str = 'dodsC/birdhouse/disk2/caspar/daily/'*) → Tuple[Dataset, List[DatetimeIndex | Series | Timestamp | Any]]

Return CASPAR dataset.

Parameters

- **climate_model** (*str*) – Type of climate model, for now only "GEPS" is supported.
- **date** (*dt.datetime*) – The date of the forecast.
- **thredds** (*str*) – The thredds server url. Default: "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/"
- **directory** (*str*) – The directory on the thredds server where the data is stored. Default: "dodsC/birdhouse/disk2/caspar/daily/"

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.forecasts.**get_ECCC_dataset**(*climate_model: str*, *thredds: str = 'https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/'*, *directory: str = 'dodsC/datasets/forecasts/eccc_geps/'*) → Tuple[Dataset, List[DatetimeIndex | Series | Timestamp | Any]]

Return latest GEPS forecast dataset.

Parameters

- **climate_model** (*str*) – Type of climate model, for now only "GEPS" is supported.
- **thredds** (*str*) – The thredds server url. Default: "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/"
- **directory** (*str*) – The directory on the thredds server where the data is stored. Default: "dodsC/datasets/forecasts/eccc_geps/"

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.forecasts.**get_hindcast_day**(*region_coll: fiona.Collection*, *date*, *climate_model='GEPS'*)

Generate a forecast dataset that can be used to run raven.

Data comes from the CASPAR archive and must be aggregated such that each file contains forecast data for a single day, but for all forecast timesteps and all members.

The code takes the region shapefile, the forecast date required, and the climate_model to use, here GEPS by default, but eventually could be GEPS, GDPS, REPS or RDPS.

ravenpy.extractors.forecasts.**get_recent_ECCC_forecast**(*region_coll: fiona.Collection*, *climate_model: str = 'GEPS'*) → Dataset

Generate a forecast dataset that can be used to run raven.

Data comes from the ECCC datamart and collected daily. It is aggregated such that each file contains forecast data for a single day, but for all forecast timesteps and all members.

The code takes the region shapefile and the climate_model to use, here GEPS by default, but eventually could be GEPS, GDPS, REPS or RDPS.

Parameters

- **region_coll** (*fiona.Collection*) – The region vectors.
- **climate_model** (*str*) – Type of climate model, for now only "GEPS" is supported.

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.forecasts.**get_subsetted_forecast**(*region_coll: fiona.Collection*, *ds: Dataset*, *times: datetime | DataArray*, *is_caspar: bool*) → Dataset

Get Subsetted Forecast.

This function takes a dataset, a region and the time sampling array and returns the subsetted values for the given region and times.

Parameters

- **region_coll** (*fiona.Collection*) – The region vectors.
- **ds** (*xr.Dataset*) – The dataset containing the raw, worldwide forecast data
- **times** (*dt.datetime or xr.DataArray*) – The array of times required to do the forecast.
- **is_caspar** (*bool*) – True if the data comes from Caspar, false otherwise. Used to define lat/lon on rotated grid.

Returns
The forecast dataset.

Return type
xr.Dataset

ravenpy.extractors.routing_product module

class ravenpy.extractors.routing_product.**BasinMakerExtractor**(*df*, *hru_aspect_convention='GRASS'*, *routing_product_version='2.1'*)

Bases: object

This is a class to encapsulate the logic of converting the Routing Product into the required data structures to generate the RVH file format.

Parameters

- **df** (*GeoDataFrame*) – Sub-basin information.
- **hru_aspect_convention** (*{"GRASS", "ArcGIS"}*) – How sub-basin aspect is defined.
- **routing_product_version** (*{"2.1", "1.0"}*) – Version of the BasinMaker data.

HRU_ASPECT_CONVENTION = 'GRASS'

MANNING_DEFAULT = 0.035

MAX_RIVER_SLOPE = 1e-05

ROUTING_PRODUCT_VERSION = '2.1'

USE_LAKE_AS_GAUGE = False

USE_LAND_AS_GAUGE = False

USE_MANNING_COEFF = False

WEIR_COEFFICIENT = 0.6

extract(*hru_from_sb: bool = False*) → dict

Extract data from the Routing Product shapefile and return dictionaries that can be parsed into Raven Commands.

Parameters

hru_from_sb (*bool*) – If True, draw HRU information from subbasin information. This is likely to yield crude results.

Returns

"sub_basins"
Sequence of dictionaries with *SubBasin* attributes.

"sub_basin_group"
Sequence of dictionaries with *SubBasinGroup* attributes.

"reservoirs"
Sequence of dictionaries with *Reservoir* attributes.

"channel_profile"
Sequence of dictionaries with *ChannelProfile* attributes.

"hrus"
Sequence of dictionaries with *HRU* attributes.

Return type
dict

class ravenpy.extractors.routing_product.**GridWeightExtractor**(*input_file_path*, *routing_file_path*, *dim_names=('lon_dim', 'lat_dim')*, *var_names=('longitude', 'latitude')*, *routing_id_field='SubId'*, *netcdf_input_field='NetCDF_col'*, *gauge_ids=None*, *sub_ids=None*, *area_error_threshold=0.05*)

Bases: `object`

Class to extract grid weights.

Notes

The original version of this algorithm can be found at: https://github.com/julemai/GridWeightsGenerator

AREA_ERROR_THRESHOLD = 0.05

CRS_CAEA = 3573

CRS_LLDEG = 4326

DIM_NAMES = ('lon_dim', 'lat_dim')

NETCDF_INPUT_FIELD = 'NetCDF_col'

ROUTING_ID_FIELD = 'SubId'

VAR_NAMES = ('longitude', 'latitude')

extract() → dict

Return dictionary to create a GridWeights command.

ravenpy.extractors.routing_product.**open_shapefile**(*path: str | PathLike*)

Return GeoDataFrame from shapefile path.

ravenpy.extractors.routing_product.**upstream_from_coords**(*lon: float*, *lat: float*, *df: DataFrame | geopandas.GeoDataFrame*) → DataFrame | geopandas.GeoDataFrame

Return the sub-basins located upstream from outlet.

Parameters

- **lon** (*float*) – Longitude of outlet.
- **lat** (*float*) – Latitude of outlet.
- **df** (*pandas.DataFrame or geopandas.GeoDataFrame*) – Routing product.

Returns
Sub-basins located upstream from outlet.

Return type
pandas.DataFrame or geopandas.GeoDataFrame

ravenpy.extractors.routing_product.**upstream_from_id**(*fid: int*, *df: DataFrame | geopandas.GeoDataFrame*) → DataFrame | geopandas.GeoDataFrame

Return upstream sub-basins by evaluating the downstream networks.

Parameters

- **fid** (*int*) – feature ID of the downstream feature of interest.
- **df** (*pandas.DataFrame or geopandas.GeoDataFrame*) – A GeoDataframe comprising the watershed attributes.

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pandas.DataFrame or geopandas.GeoDataFrame

ravenpy.utilities package

Submodules

ravenpy.utilities.analysis module

ravenpy.utilities.analysis.**circular_mean_aspect**(*angles: ndarray*) → ndarray

Return the mean angular aspect based on circular arithmetic approach.

Parameters
angles (*np.ndarray*) – Array of aspect angles

Returns
Circular mean of aspect array.

Return type
np.ndarray

ravenpy.utilities.analysis.**dem_prop**(*dem: str | Path*, *geom: shapely.geometry.Polygon | shapely.geometry.MultiPolygon | List[shapely.geometry.Polygon | shapely.geometry.MultiPolygon] | None = None*, *directory: str | Path | None = None*) → dict

Return raster properties for each geometry.

This

Parameters

- **dem** (*str or Path*) – DEM raster in reprojected coordinates.
- **geom** (*Polygon or MultiPolygon or List[Polygon or MultiPolygon]*) – Geometry over which aggregate properties will be computed. If None compute properties over entire raster.
- **directory** (*str or Path*) – Folder to save the GDAL terrain analysis outputs.

Returns
Dictionary storing mean elevation [m], slope [deg] and aspect [deg].

Return type
dict

ravenpy.utilities.analysis.**gdal_aspect_analysis**(*dem: str | Path*, *set_output: str | Path | bool = False*, *flat_values_are_zero: bool = False*) → ndarray | osgeo.gdal.Dataset

Return the aspect of the terrain from the DEM.

The aspect is the compass direction of the steepest slope (0: North, 90: East, 180: South, 270: West).

Parameters

- **dem** (*str or Path*) – Path to file storing DEM.
- **set_output** (*str or Path or bool*) – If set to a valid filepath, will write to this path, otherwise will use an in-memory gdal.Dataset.
- **flat_values_are_zero** (*bool*) – Designate flat values with value zero. Default: -9999.

Returns
Aspect array.

Return type
np.ndarray

Notes

Ensure that the DEM is in a *projected coordinate*, not a geographic coordinate system, so that the horizontal scale is the same as the vertical scale (m).

ravenpy.utilities.analysis.**gdal_slope_analysis**(*dem: str | Path*, *set_output: str | Path | None = None*, *units: str = 'degree'*) → ndarray

Return the slope of the terrain from the DEM.

The slope is the magnitude of the gradient of the elevation.

Parameters

- **dem** (*str or Path*) – Path to file storing DEM.
- **set_output** (*str or Path, optional*) – If set to a valid filepath, will write to this path, otherwise will use an in-memory gdal.Dataset.
- **units** (*str*) – Slope units. Default: 'degree'.

Returns
Slope array.

Return type
np.ndarray

Notes

Ensure that the DEM is in a *projected coordinate*, not a geographic coordinate system, so that the horizontal scale is the same as the vertical scale (m).

ravenpy.utilities.analysis.**geom_prop**(*geom: shapely.geometry.Polygon | shapely.geometry.MultiPolygon | shapely.geometry.GeometryCollection*) → dict

Return a dictionary of geometry properties.

Parameters
geom (*Polygon or MultiPolygon or GeometryCollection*) – Geometry to analyze.

Returns
Dictionary storing polygon area, centroid location, perimeter and gravelius shape index.

Return type
dict

Notes

Some of the properties should be computed using an equal-area projection.

ravenpy.utilities.calibration module

class ravenpy.utilities.calibration.**SpotSetup**(*config: ~ravenpy.config.rvs.Config, low: ~ravenpy.config.base.Params | ~typing.Sequence, high: [<class 'ravenpy.config.base.Params'>, typing.Sequence], workdir: str | ~pathlib.Path | None = None*)

Bases: `object`

evaluation()

Return the observation.

Since Raven computes the objective function itself, we simply return a placeholder.

init_params(*low: ~ravenpy.config.base.Params | ~typing.Sequence, high: [<class 'ravenpy.config.base.Params'>, typing.Sequence]*)

objectivefunction(*evaluation*, *simulation*)

Return the objective function.

Note that we short-circuit the evaluation and simulation entries, since the objective function has already been computed by Raven.

parameters()

Return a random parameter combination.

simulation(*x*)

Run the model, but return a placeholder value instead of the model output.

ravenpy.utilities.checks module

Checks for various geospatial and IO conditions.

ravenpy.utilities.checks.**boundary_check**(**args: str | Path, max_y: int | float = 60, min_y: int | float = -60*) → None

Verify that boundaries do not exceed specific latitudes for geographic coordinate data. Emit a UserWarning if so.

Parameters

- ***args** (*Sequence of str or Path*) – str or Path to file(s)
- **max_y** (*int or float*) – Maximum value allowed for latitude. Default: 60.
- **min_y** (*int or float*) – Minimum value allowed for latitude. Default: -60.

ravenpy.utilities.checks.**feature_contains**(*point: Tuple[str | float | int, str | float | int] | shapely.geometry.Point, shp: str | Path | List[str | Path]*) → dict | bool

Return the first feature containing a location.

Parameters

- **point** (*tuple[Union[int, float, str], Union[str, float, int]], Point]*) – Geographic coordinates of a point (lon, lat) or a shapely Point.
- **shp** (*str or Path or list of str or Path*) – The path to the file storing the geometries.

Returns
The feature found.

Return type
dict or bool

Notes

This is really slow. Another approach is to use the *fiona.Collection.filter* method.

ravenpy.utilities.checks.**multipolygon_check**(*geom: shapely.geometry.GeometryCollection*) → None

Perform a check to verify a geometry is a MultiPolygon

Parameters
geom (*GeometryCollection*)

Return type
None

ravenpy.utilities.checks.**single_file_check**(*file_list: Sequence[str | Path]*) → Any

Return the first element of a file list. Raise an error if the list is empty or contains more than one element.

Parameters
file_list (*Sequence of str or Path*)

ravenpy.utilities.coords module

ravenpy.utilities.coords.**infer_scale_and_offset**(*da: DataArray*, *data_type: str*, *cumulative: bool = False*) → Tuple[float, float]

Return scale and offset parameters from data.

Infer scale and offset parameters describing the linear transformation from the units in file to Raven compliant units.

Parameters

- **da** (*xr.DataArray*) – Input data.
- **data_type** (*str*) – Raven data type, e.g. 'PRECIP', 'TEMP_AVE', etc.
- **cumulative** (*bool*) – Default: False.

Returns
Scale and offset parameters.

Return type
float, float

Notes

Does not work with accumulated variables.

ravenpy.utilities.coords.**param**(*model*)

Return a parameter coordinate.

Parameters
model (*str*) – Model name.

ravenpy.utilities.coords.**realization**(*n*)

Return a realization coordinate.

Parameters
n (*int*) – Size of the ensemble.

ravenpy.utilities.coords.**units_transform**(*source*, *target*, *context='hydro'*)

Return linear transform parameters to convert one unit to another.

If the target unit is given by $y = ax + b$, where x is the value of the source unit, then this function returns a, b.

Parameters

- **source** (*str, pint.Unit*) – Source unit string, pint-recognized.
- **target** (*str*) – Target unit string, pint-recognized.
- **context** (*str, optional*) – Context of unit conversion. Default: "hydro".

ravenpy.utilities.forecasting module

Created on Fri Jul 17 09:11:58 2020

@author: ets

ravenpy.utilities.forecasting.**climatology_esp**(*config*, *workdir: str | Path | None = None*, *years: List[int] | None = None*, *overwrite: bool = False*) → *EnsembleReader*

Ensemble Streamflow Prediction based on historical variability.

Run the model using forcing for different years. No model warm-up is performed by this function, make sure the initial states are consistent with the start date.

Parameters

- **config** (ravenpy.config.rvs.Config) – Model configuration.
- **years** (*List[int]*) – Years from which forcing time series will be drawn. If None, run for all years where forcing data is available.
- **workdir** (*str or Path*) – The path to rv files and model outputs. If None, create a temporary directory.
- **overwrite** (*bool*) – Whether to overwrite existing values or not. Default: False

Returns
Class facilitating the analysis of multiple Raven outputs.

Return type
EnsembleReader

ravenpy.utilities.forecasting.**compute_forecast_flood_risk**(*forecast: Dataset*, *flood_level: float*, *thredds: str = 'https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/'*) → Dataset

Returns the empirical exceedance probability for each forecast day based on a flood level threshold.

Parameters

- **forecast** (*xr.Dataset*) – Ensemble or deterministic streamflow forecast.
- **flood_level** (*float*) – Flood level threshold. Will be used to determine if forecasts exceed this specified flood threshold. Should be in the same units as the forecasted streamflow.
- **thredds** (*str*) – The thredds server url. Default: "https://pavics.ouranos.ca/twitcher/ows/proxy/thredds/"

Returns
Time series of probabilities of flood level exceedance.

Return type
xr.Dataset

ravenpy.utilities.forecasting.**ensemble_prediction**(*config*, *forecast: str | Path*, *ens_dim: str = 'member'*, *workdir=None*, *overwrite=True*, ***kwds*) → *EnsembleReader*

Ensemble Streamflow Prediction based on historical weather forecasts (CASPAR or other).

Run the model using forcing for different years. No model warm-up is performed by this function, make sure the initial states are consistent with the start date.

Parameters

- **config** (ravenpy.config.rvs.Config) – Model configuration.
- **forecast** (*str or Path*) – Forecast subsetted to the catchment location (.nc).
- **ens_dim** (*str*) – Name of dimension to iterate over.
- **workdir** (*str or Path*) – The path to rv files and model outputs. If None, create temporary directory.
- **overwrite** (*bool*) – Overwrite files when writing to disk.
- ****kwds** – Keywords for the *Gauge.from_nc* function.

Returns
Class facilitating the analysis of multiple Raven outputs.

Return type
EnsembleReader

ravenpy.utilities.forecasting.**hindcast_climatology_esp**(*config: Config*, *warm_up_duration: int*, *years: List[int] | None = None*, *hindcast_years: List[int] | None = None*, *workdir: str | Path | None = None*, *overwrite: bool = False*) → Dataset

Hindcast of Ensemble Prediction Streamflow.

This function runs an emulator initialized for each year in *hindcast_years*, using the forcing time series for each year in *years*. This allows an assessment of the performance of the ESP. The total number of simulations is given by *len(years) * len(hindcasts_years)*.

Parameters

- **config** (ravenpy.config.rvs.Config) – Model configuration. Initial states will be overwritten.
- **warm_up_duration** (*int*) – Number of days to run the model prior to the starting date to initialize the state variables.
- **workdir** (*Path*) – Work directory. If None, creates a temporary directory.
- **years** (*List[int]*) – Years from which forcing time series will be drawn. If None, run for all years where forcing data is available.
- **hindcast_years** (*List[int]*) – Years for which the model will be initialized and the *climatology_esp* function run. Defaults to all years when forcing data is available.
- **overwrite** (*bool*) – If True, overwrite existing files.

Returns
The array containing the (init, member, lead) dimensions ready for using in climpred. (qsim)

Return type
xarray.DataArray

Notes

The dataset output dimensions are

- *init*: hindcast issue date,
- *member*: ESP members of the hindcasting experiment,
- *lead*: number of lead days of the forecast.

ravenpy.utilities.forecasting.**hindcast_from_meteo_forecast**(*config*, *forecast: str | Path*, *ens_dim: str = 'member'*, *workdir=None*, *overwrite=True*, ***kwds*) → *EnsembleReader*

Ensemble Streamflow Prediction based on historical weather forecasts (CASPAR or other).

Run the model using forcing for different years. No model warm-up is performed by this function, make sure the initial states are consistent with the start date.

Parameters

- **config** (ravenpy.config.rvs.Config) – Model configuration.
- **forecast** (*str or Path*) – Forecast subsetted to the catchment location (.nc).
- **ens_dim** (*str*) – Name of dimension to iterate over.
- **workdir** (*str or Path*) – The path to rv files and model outputs. If None, create temporary directory.
- **overwrite** (*bool*) – Overwrite files when writing to disk.
- ****kwds** – Keywords for the *Gauge.from_nc* function.

Returns
Class facilitating the analysis of multiple Raven outputs.

Return type
EnsembleReader

ravenpy.utilities.forecasting.**to_climpred_hindcast_ensemble**(*hindcast: Dataset*, *observations: Dataset*) → climpred.HindcastEnsemble

Create a hindcasting object that can be used by the *climpred* toolbox for hindcast verification.

Parameters

- **hindcast** (*xarray.Dataset*) – The hindcasted streamflow data for a given period.
- **observations** (*xarray.Dataset*) – The streamflow observations that are used to verify the hindcasts.

Returns
The hindcast ensemble formatted to be used in climpred.

Return type
climpred.HindcastEnsemble

ravenpy.utilities.forecasting.**warm_up**(*config*, *duration: int*, *workdir: str | Path | None = None*, *overwrite: bool = False*) → *Config*

Run the model on a time series preceding the start date.

Parameters

- **config** (ravenpy.config.rvs.Config) – Model configuration.
- **duration** (*int*) – Number of days the warm-up simulation should last *before* the start date.
- **workdir** (*Path*) – Work directory.
- **overwrite** (*bool*) – If True, overwrite existing files.

Returns
Model configuration with initial state set by running the model prior to the start date.

Return type
ravenpy.config.rvs.Config

ravenpy.utilities.geo module

Tools for performing geospatial translations and transformations.

ravenpy.utilities.geo.**determine_upstream_ids**(*fid: str*, *df: DataFrame | geopandas.GeoDataFrame*, *basin_field: str | None = None*, *downstream_field: str | None = None*, *basin_family: str | None = None*) → DataFrame | geopandas.GeoDataFrame

Return a list of upstream features by evaluating the downstream networks.

Parameters

- **fid** (*str*) – feature ID of the downstream feature of interest.
- **df** (*pd.DataFrame*) – A Dataframe comprising the watershed attributes.
- **basin_field** (*str*) – The field used to determine the id of the basin according to hydro project.
- **downstream_field** (*str*) – The field identifying the downstream sub-basin for the hydro project.
- **basin_family** (*str, optional*) – Regional watershed code (For HydroBASINS dataset).

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pd.DataFrame

ravenpy.utilities.geo.**find_geometry_from_coord**(*lon: float*, *lat: float*, *df: geopandas.GeoDataFrame*) → geopandas.GeoDataFrame

Return the geometry containing the given coordinates.

lon
[float] Longitude.

lat
[float] Latitude.

df
[GeoDataFrame] Data.

Returns
Record whose geometry contains the point.

Return type
GeoDataFrame

ravenpy.utilities.geo.**generic_raster_clip**(*raster: str | Path*, *output: str | Path*, *geometry: shapely.geometry.Polygon | shapely.geometry.MultiPolygon | List[shapely.geometry.Polygon | shapely.geometry.MultiPolygon]*, *touches: bool = False*, *fill_with_nodata: bool = True*, *padded: bool = True*, *raster_compression: str = 'lzw'*) → None

Crop a raster file to a given geometry.

Parameters

- **raster** (*Union[str, Path]*) – Path to input raster.
- **output** (*Union[str, Path]*) – Path to output raster.
- **geometry** (*Union[Polygon, MultiPolygon, List[Union[Polygon, MultiPolygon]]]*) – Geometry defining the region to crop.
- **touches** (*bool*) – Whether to include cells that intersect the geometry or not. Default: True.
- **fill_with_nodata** (*bool*) – Whether to keep pixel values for regions outside of shape or set as nodata or not. Default: True.
- **padded** (*bool*) – Whether to add a half-pixel buffer to shape before masking or not. Default: True.
- **raster_compression** (*str*) – Level of data compression. Default: 'lzw'.

Return type
None

ravenpy.utilities.geo.**generic_raster_warp**(*raster: str | Path*, *output: str | Path*, *target_crs: str | dict | pyproj.CRS*, *raster_compression: str = 'lzw'*) → None

Reproject a raster file.

Parameters

- **raster** (*Union[str, Path]*) – Path to input raster.

- **output** (*Union[str, Path]*) – Path to output raster.
- **target_crs** (*str or dict*) – Target projection identifier.
- **raster_compression** (*str*) – Level of data compression. Default: 'lzw'.

Return type
None

ravenpy.utilities.geo.**generic_vector_reproject**(*vector: str | Path*, *projected: str | Path*, *source_crs: str | pyproj.CRS = 4326*, *target_crs: str | pyproj.CRS | None = None*) → None

Reproject all features and layers within a vector file and return a GeoJSON

Parameters

- **vector** (*Union[str, Path]*) – Path to a file containing a valid vector layer.
- **projected** (*Union[str, Path]*) – Path to a file to be written.
- **source_crs** (*Union[str, pyproj.crs.CRS]*) – CRS for the source geometry. Default: 4326.
- **target_crs** (*Union[str, pyproj.crs.CRS]*) – CRS for the target geometry.

Return type
None

ravenpy.utilities.geo.**geom_transform**(*geom: shapely.geometry.GeometryCollection | shapely.geometry.shape*, *source_crs: str | int | pyproj.CRS = 4326*, *target_crs: str | int | pyproj.CRS | None = None*) → shapely.geometry.GeometryCollection

Change the projection of a geometry.

Assuming a geometry's coordinates are in a *source_crs*, compute the new coordinates under the *target_crs*.

Parameters

- **geom** (*Union[GeometryCollection, shape]*) – Source geometry.
- **source_crs** (*Union[str, int, CRS]*) – Projection identifier (proj4) for the source geometry, e.g. '+proj=longlat +datum=WGS84 +no_defs'.
- **target_crs** (*Union[str, int, CRS]*) – Projection identifier (proj4) for the target geometry.

Returns
Reprojected geometry.

Return type
GeometryCollection

ravenpy.utilities.geoserver module

GeoServer interaction operations.

Working assumptions for this module: * Point coordinates are passed as shapely.geometry.Point instances. * BBox coordinates are passed as (lon1, lat1, lon2, lat2). * Shapes (polygons) are passed as shapely.geometry.shape parsable objects. * All functions that require a CRS have a CRS argument with a default set to WGS84. * GEOSERVER_URL points to the GeoServer instance hosting all files. * For legacy reasons, we also accept the *GEO_URL* environment variable.

TODO: Refactor to remove functions that are just 2-lines of code. For example, many function's logic essentially consists in creating the layer name. We could have a function that returns the layer name, and then other functions expect the layer name.

ravenpy.utilities.geoserver.**filter_hydro_routing_attributes_wfs**(*attribute: str | None = None*, *value: str | float | int | None = None*, *level: int = 12*, *lakes: str = '1km'*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → str

Return a URL that formats and returns a remote GetFeatures request from hydro routing dataset.

For geographic rasters, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **attribute** (*list*) – Attributes/fields to be queried.
- **value** (*str or int or float*) – The requested value for the attribute.
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
URL to the GeoJSON-encoded WFS response.

Return type
str

ravenpy.utilities.geoserver.**filter_hydrobasins_attributes_wfs**(*attribute: str*, *value: str | float | int*, *domain: str*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → str

Return a URL that formats and returns a remote GetFeatures request from the USGS HydroBASINS dataset.

For geographic raster grids, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **attribute** (*str*) – Attribute/field to be queried.
- **value** (*str or float or int*) – Value for attribute queried.
- **domain** (*{"na", "ar"}*) – The domain of the HydroBASINS data.

- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
URL to the GeoJSON-encoded WFS response.

Return type
str

ravenpy.utilities.geoserver.**get_hydro_routing_attributes_wfs**(*attribute: Sequence[str]*, *level: int = 12*, *lakes: str = '1km'*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → str

Return a URL that formats and returns a remote GetFeatures request from hydro routing dataset.

For geographic rasters, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **attribute** (*list*) – Attributes/fields to be queried.
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
URL to the GeoJSON-encoded WFS response.

Return type
str

ravenpy.utilities.geoserver.**get_hydro_routing_location_wfs**(*coordinates: Tuple[str | float | int, str | float | int]*, *lakes: str*, *level: int = 12*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → dict

Return features from the hydro routing data set using bounding box coordinates.

For geographic rasters, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **coordinates** (*Tuple[str or float or int, str or float or int]*) – Geographic coordinates of the bounding box (left, down, right, up).
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
A GeoJSON-derived dictionary of vector features (FeatureCollection).

Return type
dict

ravenpy.utilities.geoserver.**get_hydrobasins_location_wfs**(*coordinates: Tuple[str | float | int, str | float | int]*, *domain: str | None = None*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → Dict[str, str | int | float]

Return features from the USGS HydroBASINS data set using bounding box coordinates.

For geographic raster grids, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **coordinates** (*Tuple[str or float or int, str or float or int]*) – Geographic coordinates of the bounding box (left, down, right, up).
- **domain** (*{"na", "ar"}*) – The domain of the HydroBASINS data.
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
A GeoJSON-encoded vector feature.

Return type
dict

ravenpy.utilities.geoserver.**get_raster_wcs**(*coordinates: Iterable | Sequence[float | str]*, *geographic: bool = True*, *layer: str | None = None*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → bytes

Return a subset of a raster image from the local GeoServer via WCS 2.0.1 protocol.

For geographic raster grids, subsetting is based on WGS84 (Long, Lat) boundaries. If not geographic, subsetting based on projected coordinate system (Easting, Northing) boundaries.

Parameters

- **coordinates** (*Sequence of int or float or str*) – Geographic coordinates of the bounding box (left, down, right, up)
- **geographic** (*bool*) – If True, uses "Long" and "Lat" in WCS call. Otherwise, uses "E" and "N".
- **layer** (*str*) – Layer name of raster exposed on GeoServer instance, e.g. 'public:CEC_NALCMS_LandUse_2010'
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
A GeoTIFF array.

Return type
bytes

ravenpy.utilities.geoserver.**hydro_routing_upstream**(*fid: str | float | int*, *level: int = 12*, *lakes: str = '1km'*, *geoserver: str = 'https://pavics.ouranos.ca/geoserver/'*) → Series

Return a list of hydro routing features located upstream.

Parameters

- **fid** (*str or float or int*) – Basin feature ID code of the downstream feature.
- **level** (*int*) – Level of granularity requested for the lakes vector (range(7,13)). Default: 12.
- **lakes** (*{"1km", "all"}*) – Query the version of dataset with lakes under 1km in width removed ("1km") or return all lakes ("all").
- **geoserver** (*str*) – The address of the geoserver housing the layer to be queried. Default: https://pavics.ouranos.ca/geoserver/.

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pd.Series

ravenpy.utilities.geoserver.**hydrobasins_aggregate**(*gdf: DataFrame*) → DataFrame

Aggregate multiple HydroBASINS watersheds into a single geometry.

Parameters
gdf (*pd.DataFrame*) – Watershed attributes indexed by HYBAS_ID

Return type
pd.DataFrame

ravenpy.utilities.geoserver.**hydrobasins_upstream**(*feature: dict*, *domain: str*) → DataFrame

Return a list of HydroBASINS features located upstream.

Parameters

- **feature** (*dict*) – Basin feature attributes, including the fields ["HYBAS_ID", "NEXT_DOWN", "MAIN_BAS"].
- **domain** (*{"na", "ar"}*) – Domain of the feature, North America or Arctic.

Returns
Basins ids including *fid* and its upstream contributors.

Return type
pd.Series

ravenpy.utilities.geoserver.**select_hybas_domain**(*bbox: Tuple[int | float, int | float, int | float, int | float] | None = None*, *point: Tuple[int | float, int | float] | None = None*) → str

Provided a given coordinate or boundary box, return the domain name of the geographic region the coordinate is located within.

Parameters

- **bbox** (*Optional[Tuple[Union[float, int], Union[float, int], Union[float, int], Union[float, int]]]*) – Geographic coordinates of the bounding box (left, down, right, up).
- **point** (*Optional[Tuple[Union[float, int], Union[float, int]]]*) – Geographic coordinates of an intersecting point (lon, lat).

Returns
The domain that the coordinate falls within. Possible results: "na", "ar".

Return type
str

ravenpy.utilities.graphs module

Library to perform graphs for the streamflow time series analysis.

The following graphs can be plotted:

- hydrograph
- mean_annual_hydrograph
- spaghetti_annual_hydrograph

ravenpy.utilities.graphs.**forecast**(*file: str | Path*, *fcst_var: str = 'q_sim'*) → Figure

Create a graphic of the hydrograph for each forecast member.

Parameters

- **file** (*str or Path*) – Raven output file containing simulated streamflows.
- **fcst_var** (*str*) – Name of the streamflow variable.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.graphs.**hindcast**(*file: str | Path*, *fcst_var: str*, *qobs: str | Path*, *qobs_var: str*) → Figure

Create a graphic of the hydrograph for each hindcast member.

Parameters

- **file** (*str or Path*) – Raven output file containing simulated streamflows.
- **fcst_var** (*str*) – Name of the streamflow variable.
- **qobs** (*str or Path*) – Streamflow observation file, with times matching the hindcast.
- **qobs_var** (*str*) – Name of the streamflow observation variable.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.graphs.**hydrograph**(*file_list: Sequence[str | Path]*)

Create a graphic of the hydrograph for each model simulation.

Parameters
file_list (*Sequence of str or Path*) – Raven output files containing simulated streamflows.

ravenpy.utilities.graphs.**mean_annual_hydrograph**(*file_list: Sequence[str | Path]*)

Create a graphic of the mean hydrological cycle for each model simulation.

Parameters
file_list (*Sequence of str or Path*) – Raven output files containing simulated streamflows.

ravenpy.utilities.graphs.**spaghetti_annual_hydrograph**(*file: str | Path*)

Create a spaghetti plot of the mean hydrological cycle for one model simulations.

The mean simulation is also displayed.

Parameters
file (*str or Path*) – Raven output files containing simulated streamflows of one model.

ravenpy.utilities.graphs.**ts_fit_graph**(*ts: DataArray*, *params: DataArray*) → Figure

Create graphic showing a histogram of the data and the distribution fitted to it.

The graphic contains one panel per watershed.

Parameters

- **ts** (*xr.DataArray*) – Stream flow time series with dimensions (time, nbasins).
- **params** (*xr.DataArray*) – Fitted distribution parameters returned by *xclim.land.fit* indicator.

Returns
Figure showing a histogram and the parameterized pdf.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.graphs.**ts_graphs**(*file*, *trend=True*, *alpha=0.05*)

Create a figure with the statistics so one can see a trend in the data.

Graphs for time series statistics.

Parameters
file (*str or Path*) – xarray-compatible file containing streamflow statistics for one run.

ravenpy.utilities.io module

Tools for reading and writing geospatial data formats.

ravenpy.utilities.io.**address_append**(*address: str | Path*) → str

Format a URL/URI to be more easily read with libraries such as "rasterstats".

Parameters
address (*Union[str, Path]*) – URL/URI to a potential zip or tar file

Returns
URL/URI prefixed for archive type

Return type
str

ravenpy.utilities.io.**archive_sniffer**(*archives: str | Path | List[str | Path]*, *working_dir: str | Path | None = None*, *extensions: Sequence[str] | None = None*) → List[str | Path]

Return a list of locally unarchived files that match the desired extensions.

Parameters

- **archives** (*str or Path or list of str or Path*) – Archive location or list of archive locations.
- **working_dir** (*str or Path, optional*) – String or Path to a working location.
- **extensions** (*Sequence of str, optional*) – List of accepted extensions.

Returns
List of files with matching accepted extensions.

Return type
list of str or Path

ravenpy.utilities.io.**crs_sniffer**(**args: str | Path | Sequence[str | Path]*) → List[int | str] | str | int

Return the list of CRS found in files.

Parameters
args (*Union[str, Path, Sequence[Union[str, Path]]]*) – Path(s) to the file(s) to examine.

Returns
Returns either a list of CRSes or a single CRS definition, depending on the number of instances found.

Return type
Union[List[str], str]

ravenpy.utilities.io.**generic_extract_archive**(*resources: str | Path | List[bytes | str | Path]*, *output_dir: str | Path | None = None*) → List[str]

Extract archives (tar/zip) to a working directory.

Parameters

- **resources** (*str or Path or list of bytes or str or Path*) – List of archive files (if netCDF files are in list, they are passed and returned as well in the return).
- **output_dir** (*str or Path, optional*) – String or Path to a working location (default: temporary folder).

Returns
List of original or of extracted files.

Return type
list

ravenpy.utilities.io.**get_bbox**(*vector: str | Path*, *all_features: bool = True*) → Tuple[float, float, float, float]

Return bounding box of all features or the first feature in file.

Parameters

- **vector** (*str or Path*) – A path to file storing vector features.
- **all_features** (*bool*) – Return the bounding box for all features. Default: True.

Returns
Geographic coordinates of the bounding box (lon0, lat0, lon1, lat1).

Return type
float, float, float, float

ravenpy.utilities.io.**is_within_directory**(*directory: str | PathLike*, *target: str | PathLike*) → bool

ravenpy.utilities.io.**raster_datatype_sniffer**(*file: str | Path*) → str

Return the type of the raster stored in the file.

Parameters
file (*Union[str, Path]*) – Path to file.

Returns
rasterio datatype of array values

Return type
str

ravenpy.utilities.io.**safe_extract**(*tar: TarFile*, *path: str = '.'*, *members=None*, ***, *numeric_owner=False*) → None

ravenpy.utilities.mk_test module

Created on Wed Jul 29 09:16:06 2015 @author: Michael Schramm

ravenpy.utilities.mk_test.**check_num_samples**(*beta: float*, *delta: float*, *std_dev: float*, *alpha: float = 0.05*, *n: float = 4*, *num_iter: int = 1000*, *tol: float = 1e-06*, *num_cycles: int = 10000*, *m: int = 5*) → int | float

Check number of samples.

This function is an implementation of the "Calculation of Number of Samples Required to Detect a Trend" section written by Sat Kumar Tomer (satkumartomer@gmail.com) which can be found at: http://vsp.pnnl.gov/help/Vsample/Design_Trend_Mann_Kendall.htm As stated on the webpage in the URL above the method uses a Monte-Carlo simulation to determine the required number of points in time, n, to take a measurement in order to detect a linear trend for specified small probabilities that the MK test will make decision errors. If a non-linear trend is actually present, then the value of n computed by VSP is only an approximation to the correct n. If non-detects are expected in the resulting data, then the value of n computed by VSP is only an approximation to the correct n, and this approximation will tend to be less accurate as the number of non-detects increases.

Parameters

- **beta** (*float*) – Probability of falsely accepting the null hypothesis
- **delta** (*float*) – Change per sample period, i.e., the change that occurs between two adjacent sampling times
- **std_dev** (*float*) – Standard deviation of the sample points.
- **alpha** (*float*) – Significance level (0.05 default)
- **n** (*int*) – Initial number of sample points (4 default).
- **num_iter** (*int*) – Number of iterations of the Monte-Carlo simulation (1000 default).
- **tol** (*float*) – Tolerance level to decide if the predicted probability is close enough to the required statistical power value (1e-6 default).
- **num_cycles** (*int*) – Total number of cycles of the simulation. This is to ensure that the simulation does finish regardless of convergence or not (10000 default).
- **m** (*int*) – If the tolerance is too small then the simulation could continue to cycle through the same sample numbers over and over. This parameter determines how many cycles to look back. If the same number of samples has been determined m cycles ago then the simulation will stop.

Examples

```
>>> num_samples = check_num_samples(0.2, 1, 0.1)
```

ravenpy.utilities.mk_test.**mk_test_calc**(*x: ndarray*, *alpha: float = 0.05*) → Tuple[str, float, float, float] | None

Make test calculation.

This function is derived from code originally posted by Sat Kumar Tomer (satkumartomer@gmail.com) See also: http://vsp.pnnl.gov/help/Vsample/Design_Trend_Mann_Kendall.htm The purpose of the Mann-Kendall (MK) test (Mann 1945, Kendall 1975, Gilbert 1987) is to statistically assess if there is a monotonic upward or downward trend of the variable of interest over time. A monotonic upward (downward) trend means that the variable consistently increases (decreases) through time, but the trend may or may not be linear. The MK test can be used in place of a parametric linear regression analysis, which can be used to test if the slope of the

estimated linear regression line is different from zero. The regression analysis requires that the residuals from the fitted regression line be normally distributed; an assumption not required by the MK test, that is, the MK test is a non-parametric (distribution-free) test. Hirsch, Slack and Smith (1982, page 107) indicate that the MK test is best viewed as an exploratory analysis and is most appropriately used to identify stations where changes are significant or of large magnitude and to quantify these findings.

Parameters

- **x** (*np.array*) – a vector of data.
- **alpha** (*float*) – significance level (0.05 default)

Return type
str, float, float, float

Notes

trend: tells the trend (increasing, decreasing or no trend) h: True (if trend is present) or False (if trend is absence) p: p value of the significance test z: normalized test statistics

Examples

```
>>> x = np.random.rand(100)
>>> trend, h, p, z = mk_test(x, 0.05)
```

ravenpy.utilities.nb_graphs module

This module contains functions creating web-friendly interactive graphics using holoviews.

The graphic outputs are meant to be displayed in a notebook. In a console, use *hvplot.show(fig)* to render the figures.

ravenpy.utilities.nb_graphs.**hydrographs**(*ds: Dataset*)
Return a graphic showing the discharge simulations and observations.

ravenpy.utilities.nb_graphs.**mean_annual_hydrograph**(*ds: Dataset*)
Return a graphic showing the discharge simulations and observations.

ravenpy.utilities.nb_graphs.**spaghetti_annual_hydrograph**(*ds: Dataset*)
Create a spaghetti plot of the mean hydrological cycle for one model simulations.

ravenpy.utilities.nb_graphs.**ts_fit_graph**(*ts: DataArray*, *params: DataArray*) → Figure
Create graphic showing a histogram of the data and the distribution fitted to it.

The graphic contains one panel per watershed.

Parameters

- **ts** (*xr.DataArray*) – Stream flow time series with dimensions (time, nbasins).
- **params** (*xr.DataArray*) – Fitted distribution parameters returned by *xclim.land.fit* indicator.

Returns
Figure showing a histogram and the parameterized pdf.

Return type
matplotlib.pyplot.Figure

ravenpy.utilities.ravenio module

Tools for reading outputs and writing inputs for the Raven executable.

ravenpy.utilities.ravenio.**parse_configuration**(*fn*) → Dict[str, Any]

Parse Raven configuration file.

Returns a dictionary keyed by parameter name.

ravenpy.utilities.regionalization module

Tools for hydrological regionalization.

ravenpy.utilities.regionalization.**IDW**(*qsims: DataArray*, *dist: Series*) → DataArray

Inverse distance weighting.

Parameters

- **qsims** (*xr.DataArray*) – Ensemble of hydrogram stacked along the *members* dimension.
- **dist** (*pd.Series*) – Distance from catchment which generated each hydrogram to target catchment.

Returns
Inverse distance weighted average of ensemble.

Return type
xr.DataArray

ravenpy.utilities.regionalization.**distance**(*gauged: DataFrame*, *ungauged: Series*) → Series

Return geographic distance [km] between ungauged and database of gauged catchments.

Parameters

- **gauged** (*pd.DataFrame*) – Table containing columns for longitude and latitude of catchment's centroid.
- **ungauged** (*pd.Series*) – Coordinates of the ungauged catchment.

Return type
pd.Series

ravenpy.utilities.regionalization.**multiple_linear_regression**(*source: DataFrame*, *params: DataFrame*, *target: DataFrame*) → Tuple[List[Any], List[int]]

Multiple Linear Regression for model parameters over catchment properties.

Uses known catchment properties and model parameters to estimate model parameter over an ungauged catchment using its properties.

Parameters

- **source** (*pd.DataFrame*) – Properties of gauged catchments.
- **params** (*pd.DataFrame*) – Model parameters of gauged catchments.
- **target** (*pd.DataFrame*) – Properties of the ungauged catchment.

Returns
A named tuple of the estimated model parameters and the R2 of the linear regression.

Return type
list of Any, list of int

ravenpy.utilities.regionalization.**read_gauged_params**(*model*)

Return table of NASH-Sutcliffe Efficiency values and model parameters for North American catchments.

Returns

- *pd.DataFrame* – Nash-Sutcliffe Efficiency keyed by catchment ID.
- *pd.DataFrame* – Model parameters keyed by catchment ID.

ravenpy.utilities.regionalization.**read_gauged_properties**(*properties*) → DataFrame

Return table of gauged catchments properties over North America.

Returns
Catchment properties keyed by catchment ID.

Return type
pd.DataFrame

ravenpy.utilities.regionalization.**regionalization_params**(*method: str*, *gauged_params: DataFrame*, *gauged_properties: DataFrame*, *ungauged_properties: DataFrame*, *filtered_params: DataFrame*, *filtered_prop: DataFrame*) → List[float]

Return the model parameters to use for the regionalization.

Parameters

- **method** (*{'MLR', 'SP', 'PS', 'SP_IDW', 'PS_IDW', 'SP_IDW_RA', 'PS_IDW_RA'}*) – Name of the regionalization method to use.
- **gauged_params** (*pd.DataFrame*) – A DataFrame of parameters for donor catchments (size = number of donors)
- **gauged_properties** (*pd.DataFrame*) – A DataFrame of properties of the donor catchments (size = number of donors)
- **ungauged_properties** (*pd.DataFrame*) – A DataFrame of properties of the ungauged catchment (size = 1)
- **filtered_params** (*pd.DataFrame*) – A DataFrame of parameters of all filtered catchments (size = all catchments with NSE > min_NSE)
- **filtered_prop** (*pd.DataFrame*) – A DataFrame of properties of all filtered catchments (size = all catchments with NSE > min_NSE)

Returns
List of model parameters to be used for the regionalization.

Return type
list

ravenpy.utilities.regionalization.**regionalize**(*config: Config*, *method: str*, *nash: Series*, *params: DataFrame | None = None*, *props: DataFrame | None = None*, *target_props: Series | dict | None = None*, *size: int = 5*, *min_NSE: float = 0.6*, *workdir: str | Path | None = None*, *overwrite: bool = False*, ***kwds*) → Tuple[DataArray, Dataset]

Perform regionalization for catchment whose outlet is defined by coordinates.

Parameters

- **config** (ravenpy.config.rvs.Config) – Symbolic emulator configuration. Only GR4JCN, HMETS and Mohyse are supported.
- **method** (*{'MLR', 'SP', 'PS', 'SP_IDW', 'PS_IDW', 'SP_IDW_RA', 'PS_IDW_RA'}*) – Name of the regionalization method to use.
- **nash** (*pd.Series*) – NSE values for the parameters of gauged catchments.
- **params** (*pd.DataFrame*) – Model parameters of gauged catchments. Needed for all but MRL method.
- **props** (*pd.DataFrame*) – Properties of gauged catchments to be analyzed for the regionalization. Needed for MLR and RA methods.
- **target_props** (*pd.Series or dict*) – Properties of ungauged catchment. Needed for MLR and RA methods.
- **size** (*int*) – Number of catchments to use in the regionalization.
- **min_NSE** (*float*) – Minimum calibration NSE value required to be considered as a donor.
- **workdir** (*Union[str, Path]*) – Work directory. If None, a temporary directory will be created.
- **overwrite** (*bool*) – If True, existing files will be overwritten.
- ****kwds** – Model configuration parameters, including the forcing files (ts).

Returns

- *(qsim, ensemble)*
- **qsim** (*DataArray (time,)*) – Multi-donor averaged predicted streamflow.
- **ensemble** (*Dataset*) –

  q_sim
  [DataArray (realization, time)] Ensemble of members based on number of donors.

  parameter
  [DataArray (realization, param)] Parameters used to run the model.

ravenpy.utilities.regionalization.**similarity**(*gauged: DataFrame*, *ungauged: DataFrame*, *kind: str = 'ptp'*) → Series

Return similarity measure between gauged and ungauged catchments.

Parameters

- **gauged** (*pd.DataFrame*) – Gauged catchment properties.
- **ungauged** (*pd.DataFrame*) – Ungauged catchment properties
- **kind** (*{'ptp', 'std', 'iqr'}*) – Normalization method: peak to peak (maximum - minimum), standard deviation, inter-quartile range.

Return type
pd.Series

ravenpy.utilities.testdata module

Tools for searching for and acquiring test data.

ravenpy.utilities.testdata.**get_file**(*name: str | Path | Sequence[str | Path]*, *github_url: str = 'https://github.com/Ouranosinc/raven-testdata'*, *branch: str = 'master'*, *cache_dir: str | Path = '/home/docs/.cache/raven_testing_data'*) → Path | List[Path]

Return a file from an online GitHub-like repository. If a local copy is found then always use that to avoid network traffic.

Parameters

- **name** (*str or Path or Sequence of str or Path*) – Name of the file or list/tuple of names of files containing the dataset(s) including suffixes.
- **github_url** (*str*) – URL to GitHub repository where the data is stored.
- **branch** (*str*) – For GitHub-hosted files, the branch to download from. Default: "master".
- **cache_dir** (*str or Path*) – The directory in which to search for and write cached data.

Return type
Path or list of Path

ravenpy.utilities.testdata.**get_local_testdata**(*patterns: str | Sequence[str]*, *temp_folder: str | Path*, *branch: str = 'master'*, *_local_cache: str | Path = '/home/docs/.cache/raven_testing_data'*) → Path | List[Path]

Copy specific testdata from a default cache to a temporary folder.

Return files matching *pattern* in the default cache dir and move to a local temp folder.

Parameters

- **patterns** (*str or Sequence of str*) – Glob patterns, which must include the folder.
- **temp_folder** (*str or Path*) – Target folder to copy files and filetree to.
- **branch** (*str*) – For GitHub-hosted files, the branch to download from. Default: "master".
- **_local_cache** (*str or Path*) – Local cache of testing data.

Return type
Union[Path, List[Path]]

ravenpy.utilities.testdata.**open_dataset**(*name: str*, *suffix: str | None = None*, *dap_url: str | None = None*, *github_url: str = 'https://github.com/Ouranosinc/raven-testdata'*, *branch: str = 'master'*, *cache: bool = True*, *cache_dir: str | Path = '/home/docs/.cache/raven_testing_data'*, ***kwds*) → Dataset

Open a dataset from the online GitHub-like repository.

If a local copy is found then always use that to avoid network traffic.

Parameters

- **name** (*str*) – Name of the file containing the dataset. If no suffix is given, assumed to be netCDF ('.nc' is appended).
- **suffix** (*str, optional*) – If no suffix is given, assumed to be netCDF ('.nc' is appended). For no suffix, set "".

- **dap_url** (*str, optional*) – URL to OPeNDAP folder where the data is stored. If supplied, supersedes github_url.
- **github_url** (*str*) – URL to GitHub repository where the data is stored.
- **branch** (*str, optional*) – For GitHub-hosted files, the branch to download from.
- **cache** (*bool*) – If True, then cache data locally for use on subsequent calls.
- **cache_dir** (*str or Path*) – The directory in which to search for and write cached data.
- ****kwds** – For NetCDF files, keywords passed to xarray.open_dataset.

Return type
xr.Dataset

See also:

`xarray.open_dataset`

`ravenpy.utilities.testdata.`**query_folder**(*folder: str | None = None*, *pattern: str | None = None*, *github_url: str = 'https://github.com/Ouranosinc/raven-testdata'*, *branch: str = 'master'*) → List[str]

Lists the files available for retrieval from a remote git repository with get_file. If provided a folder name, will perform a globbing-like filtering operation for parent folders.

Parameters

- **folder** (*str, optional*) – Relative pathname of the sub-folder from the top-level.
- **pattern** (*str, optional*) – Regex pattern to identify a file.
- **github_url** (*str*) – URL to GitHub repository where the data is stored.
- **branch** (*str*) – For GitHub-hosted files, the branch to download from. Default: "master".

Return type
list of str

13.1.2 Submodules

13.1.3 ravenpy.__version__ module

13.1.4 ravenpy.ravenpy module

Main module.

class `ravenpy.ravenpy.`**Emulator**(*config: Config*, *workdir: str | PathLike | None = None*, *modelname: str | None = None*, *overwrite: bool = False*)

Bases: `object`

property config: *Config*

Read-only model configuration.

property modelname: str

File name stem of configuration files.

property output: *OutputReader*

Return simulation output object.

property output_path: Path | None

Path to model outputs.

resume(*timestamp: bool = True*) → *Config*

Return new model configuration using state variables from the end of the run.

timestamp: bool
If False, ignore time stamp information in the solution. If True, the solution will set StartDate to the solution's timestamp.

run(*overwrite: bool = False*) → *OutputReader*

Run the model. This will write RV files if not already done.

Parameters
overwrite (*bool*) – If True, overwrite existing files.

property workdir: Path

Path to RV files and output subdirectory.

class ravenpy.ravenpy.**EnsembleReader**(*, *run_name: str | None = None*, *paths: List[str | PathLike] | None = None*, *runs: List[OutputReader] | None = None*, *dim: str = 'member'*)

Bases: object

property files

property hydrograph

property storage

class ravenpy.ravenpy.**OutputReader**(*run_name: str | None = None*, *path: str | Path | None = None*)

Bases: object

property diagnostics: dict | None

Return model diagnostics.

property files: dict

Return paths to output files.

property hydrograph: Dataset

Return the hydrograph.

property messages: str | None

property path: Path

Path to output directory.

property solution: dict | None

Return solution file content.

property storage: Dataset

Return the storage variables.

exception ravenpy.ravenpy.**RavenError**

Bases: Exception

This is an error that is meant to be raised whenever a message of type "ERROR" is found in the Raven_errors.txt file resulting from a Raven (i.e. the C program) run.

exception ravenpy.ravenpy.**RavenWarning**

Bases: Warning

This is a warning corresponding to a message of type "WARNING" in the Raven_errors.txt file resulting from a Raven (i.e. the C program) run.

ravenpy.ravenpy.**run**(*modelname: str*, *configdir: str | Path*, *outputdir: str | Path | None = None*, *overwrite: bool = True*, *verbose: bool = False*) → Path

Run Raven given the path to an existing model configuration.

Parameters

- **modelname** (*str*) – Configuration files stem, i.e. the file name without extension.
- **configdir** (*Path or str*) – Path to configuration files directory.
- **outputdir** (*Path or str, optional*) – Path to model simulation output. If None, will write to configdir/output.
- **overwrite** (*bool*) – If True, overwrite existing files.
- **verbose** (*bool*) – If True, always display Raven warnings. If False, warnings will only be printed if an error occurs.

Returns
Path to model outputs.

Return type
Path

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z