The PyReshaper User's Manual

(Version 0.9.0)

Kevin Paul[1]

[1] kpaul@ucar.edu

Table of Contents

What is it?

The PyReshaper is a tool for converting NetCDF time-slice formatted files into time-series format. It is written in Python as an easy-to-install package consisting of 4 Python modules.

Requirements

The PyReshaper is built upon 2 third-party Python packages, which separately depend upon other packages, as indicated below.

- PyNIO (v1.4.1)
  - numpy (v1.4)
  - NetCDF
- mpi4py (v1.3)
  - A dynamic/shared library installation of MPI or MPI-2

No thorough testing has been done to show whether earlier versions of these dependencies will work with the PyReshaper. The versions listed have been shown to work, and it is assumed that later versions will continue to work.

How can I get it?

The best way to obtain the PyReshaper code is to check it out from the UCAR subversion site, as shown below.

Unix Command Line:

```
$ svn co https://subversion.ucar.edu/asap/pyReshaper/tags/v0.9.0 pyReshaper
```

This is the most recent stable version of the source code. The trunk is also available for download, if you choose to have the most up-to-date version.

How do I set it up?

Installation

In this section, we describe how to install the PyReshaper package on a unix-like system. The procedure is similar for a Mac, but we have not tested the package on Windows.

As described in the previous section, first check out the source code from the subversion repository. On unix-like systems, the command is shown below.

Unix Command Line:

```
$ svn co https://subversion.ucar.edu/asap/pyReshaper/tags/v0.9.0 pyReshaper
```

Enter into the newly created directory.

Unix Command Line:

```
$ cd pyReshaper
```

The contents of the repository will look like the following.

Unix Command Line:

```
$ ls
apidoc/  doc/  Doxyfile  doxypy.py  pyreshaper/  README  scripts/  setup.py
tests/
```

To install in package, type the following command from this directory.

Unix Command Line:

```
$ python setup.py install --user
```

If you are a system administrator, you can leave off the '`--user`' option, and the package will be installed in `/usr/local`, by default. Alternatively, you may specify your own installation root directory with the '`--prefix`' option.

Generating the API Documentation

If you are a developer, you may find the Doxygen-generated API documentation helpful in understanding the design and functionality of the PyReshaper code. To generate this documentation, you must have Doxygen available and installed. If you do, the API documentation can be easily generated with the following command from the top-level PyReshaper directory.

Unix Command Line:

```
$ doxygen Doxyfile
```

The API documentation will be placed in the `apidoc/html/` directory.

Before Using the PyReshaper

After the Pyreshaper package has been installed using the procedure above, you must add the installation site-packages directory to your PYTHONPATH. If you installed with the '`--user`' option, this means adding the '`$HOME/.local/lib/python2.X/site-packages`' directory to your PYTHONPATH. If you specified a different '`--prefix`' option, then you must point to that prefix directory. For bash users, this is done with the following command.

Unix Command Line:

```
$ export PYTHONPATH=$PYTHONPATH:$PREFIX/lib/python2.X/site-packages
```

where the `$PREFIX` is the root installation directory used when installing the PyReshaper package (`$HOME/.local/` if using the '`--user`' option), and the value of '`X`' will correspond to the version of Python used to install the PyReshaper package.

If you want to use the command-line interface to the PyReshaper, you must also add the PyReshaper executables directory to your PATH. Like for the PYTHONPATH, this can be done with the following command.

Unix Command Line:

```
$ export PATH=$PATH:$PREFIX/bin
```

How do I use it?

Some General Concepts

Before we describe the various ways you can use the PyReshaper, we must describe more about what, precisely, the PyReshaper is designed to do.

As we've already mentioned, the PyReshaper is designed to convert a set of NetCDF files from *time-slice* format to *time-series* format. This statement contains a number of assumptions that pertain to the time-slice (input) data, which we list below.

1. Each time-slice NetCDF file has multiple *time-dependent* variables inside it, but can have many time-independent variables inside it, as well.
2. Each time-slice NetCDF file contains data for times that do not overlap with each other. (That is, each time-slice NetCDF file can contain data spanning a number of simulation time steps. However, the span of time contained in one time slice cannot overlap the span of time in another time-slice.)
3. Every time-slice NetCDF file contains the same time-dependent variables, just at differing times.

Similarly, there are a number of assumptions made about the *time-series* data produced by the PyReshaper conversion process.

1. By default, every time-dependent variable will be written to its own time-series NetCDF file.
2. Any time-dependent variables that should be included in every time-series file (e.g., such as 'time' itself), instead of getting their own time-series file, must be specified by name.
3. Every time-independent variable that appears in the time-slice files will be written to every time-series file.
4. Every time-series file written by the PyReshaper will span the total range of time spanned by all time-slice files specified.
5. Every time-series file will be named with the same *prefix* and *suffix*, according to the rule:

```
time_series_filename = prefix + variable_name + suffix
```

   where the `variable_name` is the name of the time-dependent variable associated with that time-series file.

It is important to understand the implications of the last assumption on the list above. Namely, it is important to note what this assumption means in terms of NetCDF file-naming conventions. It is common for the file-name to contain information that pertains to the time-sampling frequency of the data in the file, or the range of time spanned by the time-series file, or any number of other things. *To conform to such naming conventions, it may be required that the total set of time-slice files that the user which to convert to time-series be given to the PyReshaper in multiple subsets, or chunks.* Throughout this manual, we will refer to such "chunks' as **streams**. As such, every single PyReshaper operation is designed to act on a **single stream**.

Using the PyReshaper from within Python

Obviously, one of the advantages of writing the PyReshaper in Python is that it is easy to import features (modules) of the PyReshaper into your own Python code, as you might link your own software tools to an external third-party library. The library API for the PyReshaper is designed to be simple and light-weight, making it easy to use in your own Python tools or scripts.

Single-Stream Usage

Below, we show an example of how to use the PyReshaper from within Python to convert a **single stream** from time-slice format to time-series format.

Python Source: Single-Stream Example

```
from pyreshaper import specification, reshaper

# Create a Specifier object (that defined a single stream to be converted)
specifier = specification.create_specifier()

# Specify the input needed to perform the PyReshaper conversion
specifier.input_file_list = [ '/path/to/infile1.nc',
                              '/path/to/infile2.nc', ...]
specifier.netcdf_format = 'netcdf4c'
specifier.output_file_prefix = '/path/to/outfile_prefix.'
specifier.output_file_suffix = '.000101-001012.nc'
specifier.time_variant_metadata = ['time', 'time_bounds', ...]

# Create the Reshaper object
rshpr = reshaper.create_reshaper(specifier, serial=False, verbosity=1)
# Run the conversion (slice-to-series) process
rshpr.convert()

# Print timing diagnostics
rshpr.print_diagnostics()
```

In the above example, it is important to understand the input given to the PyReshaper. Namely, all of the input for this single stream is contained by a single instantiation of a `Specifier` object (the code for which is defined in the `specification` module). We will describe each attribute of the Specifier object below.

`Specifier` Object Attributes

`input_file_list`	This specifies a list of input (time-slice) file paths that all conform to the input file assumptions (described above). The list of input files need not be time-ordered, as the PyReshaper will order them appropriately. (This means that this list can easily be generated by using filename globs.) In the example above, each file path is full and absolute, for safety's sake.
`netcdf_format`	This is a string specifying what NetCDF format will be used to write the output (time-series) files. In the above example, NetCDF4 with level-1 compression is requested. Acceptable Options are: `'netcdf'`: NetCDF3 `'netcdf4'`: NetCDF4 uncompressed `'netcdf4c'`: NetCDF4 compressed (level 1)
`output_file_prefix`	This is a string specifying the common output (time-series) filename prefix. It is assumed that each time-series file will be named according to the rule: `filename = prefix + variable_name + suffix` It is important to understand, as in the example above, that the prefix can include the full, absolute path information for the output (time-series) files.
`output_file_suffix`	This is a string specifying the common output (time-series) filename suffix. It is assumed that each time-series file will be named according to the above rule.
`time_variant_metadata`	This specifies a list of variable names corresponding to variables that should be written to *every* output (time-series) NetCDF file.

Even though the PyReshaper is designed to work on a single stream at a time, multiple streams can be defined as input to the PyReshaper. When running the PyReshaper with multiple stream, multiple `Specifier` objects must be created, one for each stream.

Multiple Stream Usage

In the example below, we show one way to define a multiple stream PyReshaper run.

Python Source: Multiple-Stream Example

```
from pyreshaper import specification, reshaper

# Assuming all data defining each stream is contained
# in a list called 'streams'
specifiers = {}
for stream in streams:
      specifier = specification.create_specifier()

      # Define the Pyreshaper input for this stream
      specifier.input_file_list = stream.input_file_list
      specifier.netcdf_format = stream.netcdf_format
      specifier.output_file_prefix = stream.output_file_prefix
      specifier.output_file_suffix = stream.output_file_suffix
      specifier.time_variant_metadata = stream.time_variant_metadata

      # Append this Specifier to the dictionary of specifiers
      specifiers[stream.name] = specifier

# Create the Reshaper object
rshpr = reshaper.create_reshaper(specifiers, serial=False, verbosity=1)

# Run the conversion (slice-to-series) process
rshpr.convert()

# Print timing diagnostics
rshpr.print_diagnostics()
```

In the above example, we assume the properly formatted data (like the data shown in the single-stream example above) is contained in the list called '`streams`'. In addition to the data needed by each `Specifier` (i.e., the data defining each stream), this example assumes that a name has been given to each stream, contained in the attribute '`stream.name`'. Each

`Specifier` is then contained in a dictionary with keys corresponding to the stream name and values corresponding to the stream `Specifier`. This name will be used when printing diagnostic information during the `convert()` and `print_diagnostics()` operations of the PyReshaper.

Alternatively, the `specifiers` object (in the above example) can be a Python list, instead of a Python dictionary. If this is the case, the list of `Specifier` objects will be converted to a dictionary, with the keys of the dictionary corresponding to the list index (i.e., an integer).

Arguments to the `create_reshaper()` Function

In both examples above, the `Reshaper` object (`rshpr`) is created by passing the single `Specifier` object, list of `Specifier` objects, or dictionary of named `Specifier` objects, to the function `create_reshaper()`. This function returns a `Reshaper` object that has the functions `convert()` and `print_diagnostics()` that perform the time-slice to time-series conversion step and print useful timing diagnostics, respectively.

Additionally, the `create_reshaper()` function takes the parameter '`serial`', which can be `True` or `False`, indicating whether the `Reshaper convert()` step should be done in serial (`True`) or parallel (`False`). By default, parallel operation is assumed if this parameter is not specified.

The `create_reshaper()` function also takes the parameter '`verbosity`', which specified what level of output (to stdout) will be produced during the `convert()` step. Currently, there are only three (3) verbosity levels:

1. **`verbosity = 0`**: This means that no output will be produced unless specifically requested (i.e., by calling the `print_diagnostics()` function).
2. **`verbosity = 1`**: This means that only output that would be produced by the head rank of a parallel process will be generated.
3. **`verbosity = 2`**: This means that all output from all processors will be generated, but any output that is the same on all processors will only be generated once.

By setting the verbosity parameter in the `create_reshaper()` function to a value of 2 or above will result in the greatest amount of output.

Arguments to the `convert()` Function

While not shown in the above examples, there is an argument to the `convert()` function of the `Reshaper` object called '`output_limit`'. This argument sets an integer limit on the number of time-series files generated during the `convert()` operation (per processor). This can be useful for debugging purposes, as it can greatly reduce the length of time consumed in the `convert()` function. (A value of 0 indicates no limit, or all output file will be generated.)

Using the PyReshaper from the Unix Command-Line

While the most flexible way of using the PyReshaper is from within Python, as described above, it is also possible to run the PyReshaper from the command-line. In this section, we describe how to use the Python script `slice2series`, which provides a command-line interface (CLI) to the PyReshaper. (This script will be installed in the `$PREFIX/bin` directory, where `PREFIX` is the installation root directory.)

Below is an example of how to use the PyReshaper CLI, `slice2series`, for a serial run.

Unix Command-Line Interface: Serial Example

```
$ slice2series --serial \
               --netcdf_format='netcdf4c' \
               --output_prefix='/path/to/outfile_prefix.' \
               --output_suffix='000101-001012.nc' \
               -m 'time' -m 'time_bounds' \
               /path/to/infiles/*.nc
```

In this example, you will note that we have specified each time-dependent metadata variable name with its own '`-m`' option. (In this case, there are only 2, '`time`' and '`time_bounds`'.) We have also specified the list of input (time-slice) files using a wildcard, that the Unix shell fills in with a list of all filenames that match this pattern. (In this case, it is all files with the '`.nc`' file extension in the directory '`/path/to/infiles`'.) These command-line options and arguments specify all of the same input passed to the `Specifier` objects in the previous examples.

For parallel operation, one must launch the slice2series script from the appropriate MPI launcher. On the Yellowstone system (`yellowstone.ucar.edu`), this is done with the following command.

Unix Command-Line Interface: Yellowstone Parallel Example

```
$ mpirun.lsf slice2series \
               --netcdf_format='netcdf4c' \
               --output_prefix='/path/to/outfile_prefix.' \
               --output_suffix='000101-001012.nc' \
               -m 'time' -m 'time_bounds' \
               /path/to/infiles/*.nc
```

In the above example, this will launch the `slice2series` script into the MPI environment already created by either a request for an interactive session or from an LSF submission script.

Additional Arguments to the `slice2series` Script

While the basic options shown in the previous two (2) examples above are sufficient for most purposes, two additional options are available. The '`--verbosity`' option can be used to set the verbosity level, just like the `verbosity` argument to the `create_reshaper()` function described in the previous sections. Additionally, the '`--limit`' command-line option can be used to set the `output_limit` argument of the `Reshaper convert()` function, also described in the previous sections.