f90nml Documentation

Release 1.2

Marshall Ward

Jun 26, 2020

Contents

A Python module and command line tool for parsing Fortran namelist files

CHAPTER 1

Documentation

The complete documentation for `f90nml` is available from Read The Docs.

http://f90nml.readthedocs.org/en/latest/

CHAPTER 2

About f90nml

`f90nml` is a Python module and command line tool that provides a simple interface for the reading, writing, and modifying Fortran namelist files.

A namelist file is parsed and converted into an `Namelist` object, which behaves like a standard Python `dict`. Values are converted from Fortran data types to equivalent primitive Python types.

The command line tool `f90nml` can be used to modify individual values inside of a shell environment. It can also be used to convert the data between namelists and other configuration formats. JSON and YAML formats are currently supported.

CHAPTER 3

Quick usage guide

To read a namelist file `sample.nml` which contains the following namelists:

```
&config_nml
   input = 'wind.nc'
   steps = 864
   layout = 8, 16
   visc = 1.0e-4
   use_biharmonic = .false.
/
```

we would use the following script:

```
import f90nml
nml = f90nml.read('sample.nml')
```

which would would point `nml` to the following `dict`:

```
nml = {
    'config_nml': {
        'input': 'wind.nc',
        'steps': 864,
        'layout': [8, 16],
        'visc': 0.0001,
        'use_biharmonic': False
    }
}
```

File objects can also be used as inputs:

```
with open('sample.nml') as nml_file:
    nml = f90nml.read(nml_file)
```

To modify one of the values, say `steps`, and save the output, just manipulate the `nml` contents and write to disk using the `write` function:

```
nml['config_nml']['steps'] = 432
nml.write('new_sample.nml')
```

Namelists can also be saved to file objects:

```
with open('target.nml') as nml_file:
    nml.write(nml_file)
```

To modify a namelist but preserve its comments and formatting, create a namelist patch and apply it to a target file using the `patch` function:

```
patch_nml = {'config_nml': {'visc': 1e-6}}
f90nml.patch('sample.nml', patch_nml, 'new_sample.nml')
```

3.1 Command line interface

A command line tool is provided to manipulate namelist files within the shell:

```
$ f90nml config.nml -g config_nml -v steps=432
```

```
&config_nml
    input = 'wind.nc'
    steps = 432
    layout = 8, 16
    visc = 1.0e-4
    use_biharmonic = .false.
/
```

See the documentation for details.

CHAPTER 4

Installation

`f90nml` is available on PyPI and can be installed via pip:

```
$ pip install f90nml
```

The latest version of `f90nml` can be installed from source:

```
$ git clone https://github.com/marshallward/f90nml.git
$ cd f90nml
$ pip install .
```

4.1 Package distribution

`f90nml` is not distributed through any official packaging tools, but it is available on Arch Linux via the AUR:

```
$ git clone https://aur.archlinux.org/python-f90nml.git
$ cd python-f90nml
$ makepkg -sri
```

Volunteers are welcome to submit and maintain `f90nml` on other distributions.

4.2 Local install

Users without install privileges can append the `--user` flag to `pip` from the top `f90nml` directory:

```
$ pip install --user .
```

If pip is not available, then `setup.py` can still be used:

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

When using `setup.py` locally, some users have reported that `--prefix=` may need to be appended to the command:

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

4.3 YAML support

The command line tool offers support for conversion between namelists and YAML formatted output. If PyYAML is already installed, then no other steps are required. To require YAML support, install the `yaml` extras package:

```
$ pip install f90nml[yaml]
```

To install as a user:

```
$ pip install --user .[yaml]
```

CHAPTER 5

Contributing to `f90nml`

Users are welcome to submit bug reports, feature requests, and code contributions to this project through GitHub. More information is available in the Contributing guidelines.

CHAPTER 6

Contents

6.1 Usage

The basic API is available for working with conventional namelist files and performing simple operations, such as reading or modifying values.

Users who require more control over parsing and namelist output formatting should create objects which can be controlled using the properties described below.

6.1.1 Basic API

`f90nml.`**`read`**(*nml_path*)

Parse a Fortran namelist file and return its contents.

File object usage:

```
>>> with open(nml_path) as nml_file:
>>>     nml = f90nml.read(nml_file)
```

File path usage:

```
>>> nml = f90nml.read(nml_path)
```

This function is equivalent to the `read` function of the `Parser` object.

```
>>> parser = f90nml.Parser()
>>> nml = parser.read(nml_file)
```

`f90nml.`**`write`**(*nml*, *nml_path*, *force=False*, *sort=False*)

Save a namelist to disk using either a file object or its file path.

File object usage:

```
>>> with open(nml_path, 'w') as nml_file:
>>>     f90nml.write(nml, nml_file)
```

File path usage:

```
>>> f90nml.write(nml, 'data.nml')
```

This function is equivalent to the `write` function of the `Namelist` object `nml`.

```
>>> nml.write('data.nml')
```

By default, `write` will not overwrite an existing file. To override this, use the `force` flag.

```
>>> nml.write('data.nml', force=True)
```

To alphabetically sort the `Namelist` keys, use the `sort` flag.

```
>>> nml.write('data.nml', sort=True)
```

`f90nml.`**`patch`**(*nml_path*, *nml_patch*, *out_path=None*)

Create a new namelist based on an input namelist and reference dict.

```
>>> f90nml.patch('data.nml', nml_patch, 'patched_data.nml')
```

This function is equivalent to the `read` function of the `Parser` object with the patch output arguments.

```
>>> parser = f90nml.Parser()
>>> nml = parser.read('data.nml', nml_patch, 'patched_data.nml')
```

A patched namelist file will retain any formatting or comments from the original namelist file. Any modified values will be formatted based on the settings of the `Namelist` object.

6.1.2 Classes

class `f90nml.parser.`**Parser**

Fortran namelist parser.

read(*nml_fname*, *nml_patch_in=None*, *patch_fname=None*)

Parse a Fortran namelist file and store the contents.

```
>>> parser = f90nml.Parser()
>>> data_nml = parser.read('data.nml')
```

reads(*nml_string*)

Parse a namelist string and return an equivalent Namelist object.

```
>>> parser = f90nml.Parser()
>>> data_nml = parser.reads('&data_nml x=1 y=2 /')
```

comment_tokens

String of single-character comment tokens in the namelist.

Type `str`

Default `'!'`

Some Fortran programs will introduce alternative comment tokens (e.g. #) for internal preprocessing.

If you need to support these tokens, create a `Parser` object and set the comment token as follows:

```
>>> parser = f90nml.Parser()
>>> parser.comment_tokens += '#'
>>> nml = parser.read('sample.nml')
```

Be aware that this is non-standard Fortran and could mangle any strings using the # characters. Characters inside string delimiters should be protected, however.

default_start_index

Assumed starting index for a vector.

Type `int`

Default 1

Since Fortran allows users to set an arbitrary start index, it is not always possible to assign an index to values when no index range has been provided.

For example, in the namelist `idx.nml` shown below, the index of the values in the second assignment are ambiguous and depend on the implicit starting index.

```
&idx_nml
    v(3:5) = 3, 4, 5
    v = 1, 2
/
```

The indices of the second entry in `v` are ambiguous. The result for different values of `default_start_index` are shown below.

```
>>> parser = f90nml.Parser()
>>> parser.default_start_index = 1
>>> nml = parser.read('idx.nml')
>>> nml['idx_nml']['v']
[1, 2, 3, 4, 5]
```

```
>>> parser.default_start_index = 0
>>> nml = parser.read('idx.nml')
>>> nml['idx_nml']['v']
[1, 2, None, 3, 4, 5]
```

global_start_index

Define an explicit start index for all vectors.

Type `int`, `None`

Default `None`

When set to `None`, vectors are assumed to start at the lowest specified index. If no index appears in the namelist, then `default_start_index` is used.

When `global_start_index` is set, then all vectors will be created using this starting index.

For the namelist file `idx.nml` shown below,

```
&idx_nml
   v(3:5) = 3, 4, 5
/
```

the following Python code behaves as shown below.

```
>>> parser = f90nml.Parser()
>>> nml = parser.read('idx.nml')
>>> nml['idx_nml']['v']
[3, 4, 5]
```

```
>>> parser.global_start_index = 1
>>> nml = parser.read('idx.nml')
>>> nml['idx_nml']['v']
[None, None, 3, 4, 5]
```

Currently, this property expects a scalar, and applies this value to all dimensions.

row_major
Read multidimensional arrays in row-major format.

Type bool

Default False

Multidimensional array data contiguity is preserved by default, so that column-major Fortran data is represented as row-major Python list of lists.

The row_major flag will reorder the data to preserve the index rules between Fortran to Python, but the data will be converted to row-major form (with respect to Fortran).

sparse_arrays
Store unset rows of multidimensional arrays as empty lists.

Type bool

Default False

Enabling this flag will replace rows of unset values with empty lists, and will also not pad any existing rows when other rows are expanded.

This is not a true sparse representation, but rather is slightly more sparse than the default dense array representation.

strict_logical
Use strict rules for parsing logical data value parsing.

Type bool

Default True

The strict_logical flag will limit the parsing of non-delimited logical strings as logical values. The default value is True.

When strict_logical is enabled, only .true., .t., true, and t are interpreted as True, and only .false., .f., false, and f are interpreted as false.

When strict_logical is disabled, any value starting with .t or t is interpreted as True, while any string starting with .f or f is interpreted as False, as described in the language standard. However, it can interfere with namelists which contain non-delimited strings.

class f90nml.namelist.**Namelist**(*default_start_index=None*[, *items*])
Representation of Fortran namelist in a Python environment.

Namelists can be initialised as empty or with a pre-defined *dict* of *items*. If an explicit default start index is required for *items*, then it can be initialised with the *default_start_index* input argument.

In addition to the standard methods supported by *dict*, several additional methods and properties are provided for working with Fortran namelists.

groups()
Return an iterator that spans values with group and variable names.

Elements of the iterator consist of a tuple containing two values. The first is internal tuple containing the current namelist group and its variable name. The second element of the returned tuple is the value associated with the current group and variable.

patch(*nml_patch*)
Update the namelist from another partial or full namelist.

This is different from the intrinsic *update()* method, which replaces a namelist section. Rather, it updates the values within a section.

todict(*complex_tuple=False*)
Return a dict equivalent to the namelist.

Since Fortran variables and names cannot start with the _ character, any keys starting with this token denote metadata, such as starting index.

The `complex_tuple` flag is used to convert complex data into an equivalent 2-tuple, with metadata stored to flag the variable as complex. This is primarily used to facilitate the storage of the namelist into an equivalent format which does not support complex numbers, such as JSON or YAML.

write(*nml_path*, *force=False*, *sort=False*)
Write Namelist to a Fortran 90 namelist file.

```
>>> nml = f90nml.read('input.nml')
>>> nml.write('out.nml')
```

column_width
Set the maximum number of characters per line of the namelist file.

Type `int`

Default 72

Tokens longer than `column_width` are allowed to extend past this limit.

default_start_index
Set the default start index for vectors with no explicit index.

Type `int`, `None`

Default `None`

When the *default_start_index* is set, all vectors without an explicit start index are assumed to begin with *default_start_index*. This index is shown when printing the namelist output.

If set to *None*, then no start index is assumed and is left as implicit for any vectors undefined in *start_index*.

end_comma
Append commas to the end of namelist variable entries.

Type `bool`

Default `False`

Fortran will generally disregard any commas separating variable assignments, and the default behaviour is to omit these commas from the output. Enabling this flag will append commas at the end of the line for each variable assignment.

false_repr
Set the string representation of logical false values.

Type `str`

Default `'.false.'`

This is equivalent to the first element of `logical_repr`.

`float_format`
Set the namelist floating point format.

Type `str`

Default `''`

The property sets the format string for floating point numbers, following the format expected by the Python `format()` function.

`indent`
Set the whitespace indentation of namelist entries.

Type `int`, `str`

Default `' '` (four spaces)

This can be set to an integer, denoting the number of spaces, or to an explicit whitespace character, such as a tab (`\t`).

`index_spacing`
Apply a space between indexes of multidimensional vectors.

Type `bool`

Default `False`

`logical_repr`
Set the string representation of logical values.

Type `dict`

Default `{False: '.false.', True: '.true.'}`

There are multiple valid representations of True and False values in Fortran. This property sets the preferred representation in the namelist output.

The properties `true_repr` and `false_repr` are also provided as interfaces to the elements of `logical_repr`.

`start_index`
Set the starting index for each vector in the namelist.

Type `dict`

Default `{}`

`start_index` is stored as a dict which contains the starting index for each vector saved in the namelist. For the namelist `vec.nml` shown below,

```
&vec_nml
    a = 1, 2, 3
    b(0:2) = 0, 1, 2
    c(3:5) = 3, 4, 5
    d(:,:) = 1, 2, 3, 4
/
```

the `start_index` contents are

```
>>> import f90nml
>>> nml = f90nml.read('vec.nml')
>>> nml['vec_nml'].start_index
{'b': [0], 'c': [3], 'd': [None, None]}
```

The starting index of `a` is absent from `start_index`, since its starting index is unknown and its values cannot be assigned without referring to the corresponding Fortran source.

true_repr
Set the string representation of logical true values.

Type `str`

Default `.true.`

This is equivalent to the second element of `logical_repr`.

uppercase
Print group and variable names in uppercase.

Type `bool`

Default `False`

6.2 Command line interface

`f90nml` includes a command line tool which can be used to modify namelist variables inside of a shell environment. It can also be used to convert data between namelists and the JSON and YAML formats.

6.2.1 Options

-f FORMAT, --format FORMAT specify the output format (json, yaml, or nml)

-g GROUP, --group GROUP specify namelist group to modify. When absent, the first group is used

-h, --help display this help and exit

-p, --patch modify the existing namelist as a patch

-v EXPR, --variable EXPR specify the namelist variable to add or modify, followed by the new value. Expressions are of the form "VARIABLE=VALUE"

--version output version information and exit

6.2.2 Examples

The examples below use the namelist file `config.nml` with the following data.

```
&config_nml
input = 'wind.nc'
steps = 864
layout = 8, 16      ! (X, Y)
visc = 1e-4         ! m2 s-1
use_biharmonic = .false.
/
```

To display the formatted output of a namelist:

```
$ f90nml config.nml
```

```
&config_nml
    input = 'wind.nc'
    steps = 864
    layout = 8, 16
    visc = 0.0001
    use_biharmonic = .false.
/
```

To modify one of the values or add a new variable:

```
$ f90nml -g config_nml -v steps=432 config.nml
```

```
&config_nml
    input = 'wind.nc'
    steps = 432
    layout = 8, 16
    visc = 0.0001
    use_biharmonic = .false.
/
```

Multiple variables can be set with separate flags or separated by commas:

```
$ f90nml -g config_nml -v steps=432,date='19960101' config.nml
```

```
$ f90nml -g config_nml -v steps=432 -v date='19960101' config.nml
```

```
&config_nml
    input = 'wind.nc'
    steps = 432
    layout = 8, 16
    visc = 0.0001
    use_biharmonic = .false.
    date = 19960101
/
```

Spaces should not be used when assigning values.

When the namelist group is unspecified, the first group is assumed:

```
$ f90nml -v steps=432 config.nml
```

```
f90nml: warning: Assuming variables are in group 'config_nml'.
&config_nml
    input = 'wind.nc'
    steps = 432
    layout = 8, 16
    visc = 0.0001
    use_biharmonic = .false.
/
```

To save the modified namelist to a new file, say `out.nml`:

```
$ f90nml -v steps=432 config.nml out.nml
```

To patch the existing file and preserve comments:

```
$ f90nml -g config_nml -v steps=432 -p config.nml
```

```
&config_nml
input = 'wind.nc'
steps = 432
layout = 8, 16      ! (X, Y)
visc = 1e-4         ! m2 s-1
use_biharmonic = .false.
/
```

To convert the output to JSON format:

```
$ f90nml -g config_nml -v steps=432 config.nml -f json
```

```
{
    "config_nml": {
        "input": "wind.nc",
        "steps": 432,
        "layout": [
            8,
            16
        ],
        "visc": 0.0001,
        "use_biharmonic": false
    }
}
```

Output format is also inferred from the output extension.

```
$ f90nml -g config_nml -v steps=432 config.nml out.json
```

```
$ f90nml -g config_nml -v steps=432 config.nml out.yaml
```

JSON and YAML can also act as input files. Format is assumed by extension.

```
$ f90nml out.json
```

```
&config_nml
    input = 'wind.nc'
    layout = 8, 16
    steps = 864
    use_biharmonic = .false.
    visc = 0.0001
/
```

6.3 Notes

6.3.1 Data types

Fortran namelists do not contain any information about data type, which is determined by the target variables of the Fortran executable. `f90nml` infers the data type based on the value, but not all cases can be explicitly resolved.

`f90nml` tests values as one of each data type in the order listed below:

- Integer
- Floating point
- Complex floating point
- Logical (boolean)
- String

Strings act as a fallback type. If a value cannot be matched to any other value, then it is interpreted as a string.

In order to get the best results from `f90nml`, it is best to follow these guidelines:

- All strings should be enclosed by string delimiters (`'`, `"`).
- Floating point values should use decimals (`.`) or E notation.
- Array indices should be explicit
- Array values should be separated by commas (`,`)

6.3.2 Derived Types

User-defined types are saved as a nested hierarchy of `dicts`. For example, the following namelist

```
&dtype_nml
   a%b%c = 1
/
```

would be saved as the following `Namelist`:

```
nml = {
   'dtype_nml': {
      'a': {
         'b': {
            'c': 1
         }
      }
   }
}
```

6.3.3 Indexing

The indexing of a vector is defined in the Fortran source file, and a namelist can produce unexpected results if the starting index is implicit or unspecified. For example, the namelist below

```
&idx_nml
   v(1:2) = 5, 5
/
```

will assign values to different indices of `v` depending on its starting index, as in the examples below.

```
integer, dimension :: v(1:4)  ! Read as v = (5, 5, -, -)
integer, dimension :: v(0:3)  ! Read as v = (-, 5, 5, -)
```

Without explicit knowledge of the starting index, it is not possible to unambiguously represent the vector in Python.

In most cases, `f90nml` will internally assume a 1-based indexing, and will only output the values explicitly listed in the namelist file. If no index is provided, then `f90nml` will not add indices to the record.

However, `f90nml` can still provide some level of control over the starting index of a vector. The starting index can be explicitly set using various properties defined in the `Parser` and `Namelist` objects. For more information, see the class API.

6.4 Contributing to `f90nml`

`f90nml` development is driven by user feedback, and your contributions help to find bugs, add features, and improve performance. This is a small guide to help those who wish to contribute.

6.4.1 Development Portal

Code development is currently hosted at GitHub. Issues, feature requests, and code contributions are currently handled there.

https://github.com/marshallward/f90nml

6.4.2 Reporting errors

Any errors or general problems can be reported on GitHub's Issue tracker:

https://github.com/marshallward/f90nml/issues

The quickest way resolve a problem is to go through the following steps:

- Have I tested this on the latest GitHub (`master`) version?
- Have I provided a sample code block which reproduces the error? Have I tested the code block?
- Have I included the necessary input or output files?

  Sometimes a file attachment is required, since uncommon whitespace or Unicode characters may be missing from a standard cut-and-paste of the file.
- Have I provided a backtrace from the error?

Usually this is enough information to reproduce and resolve the problem. In some cases, we may need more information about your system, including the following:

- Your `f90nml` version:

```
>>> import f90nml
>>> f90nml.__version__
'1.1'
```

- The version and build of python:

```
>>> import sys
>>> print(sys.version)
3.6.8 (default, Apr 25 2019, 21:02:35)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-36)]
```

- Your operating system (Linux, OS X, Windows, etc.), and your distribution if relevant.

While more information can help, the most important step is to report the problem, and any missing information can be provided over the course of the discussion.

6.4.3 Feature requests

Feature requests are welcome, and can be submitted as Issues in the GitHub tracker.

https://github.com/marshallward/f90nml/issues

When preparing a feature request, consider providing the following information:

- What problem is this feature trying to solve?
- Is it solvable using Python intrinsics? How is it currently handled in similar modules?
- Does the feature current exist in similar modules (JSON, YAML, etc.)?
- Can you provide an example code block demonstrating the feature?
- For `Namelist` changes, is it compatible with the `dict` parent class?
- Does this feature require any new dependencies (e.g. NumPy)?

As a self-supported project, time is often limited and feature requests are often a lower priority than bug fixes or ongoing work, but a strong case can help to prioritize feature development.

6.4.4 Contributing to f90nml

Fixes and features are very welcome to `f90nml`, and are greatly encouraged. Much of the functionality of the `Parser` and `Namelist` classes have been either requested or contributed by other users.

If you are concerned that a project may not be suitable or may conflict with ongoing work, then feel free to submit a feature request with comment noting that you are happy to provide the feature.

Feature should be sent as pull requests via GitHub, specifically to the `master` branch, which acts as the main development branch.

Explicit patches via email are also welcome, although they are a bit more work to process.

When preparing a pull request, consider the following advice:

- Commit logs should be long-form. Don't use `commit -m "Added a feature!"`; instead provide a multiline description of your changes.

  Single line commits are acceptable for very minor changes, such as whitespace.

- Commit messages should generally try to be standalone and ought to avoid references to explicit GitHub content, such as issue numbers or usernames.

- Code changes must pass existing tests:

```
$ python setup.py test
```

- Code changes ought to satisfy PEP8, including line length limit. The following should raise no warnings:

```
$ pycodestyle f90nml
```

  `pycodestyle` is available on PyPI via pip:

```
$ pip install pycodestyle
```

- Providing a test case for your example would be greatly appreciated. See `tests/test_f90nml.py` for examples.

- Features should generally only depend on the standard library. Any features requiring an external dependency should only be enabled when the dependency is available.

In practice, contributions are rare and it's not difficult to sort out these issues inside of the pull request.

6.4.5 Have Fun

Most importantly, remember that it is more important to contribute than to follow the rules and never contribute. Issues can be sorted out in real time, and everything can be amended in the world of software.

CHAPTER 7

Licensing

`f90nml` is distributed under the Apache 2.0 License.

CHAPTER 8

Contact

Marshall Ward <f90nml@marshallward.org>

Index