pygromacs Documentation

Release 0.1

Petter Johansson

January 09, 2015

Contents

Contents:

CHAPTER 1

Software Development Toolbox Project

This program was (partly) written for the Software Development Toolbox course given by PDC at KTH, Fall of 2014.

Author Petter Johansson

Email petter.johansson@scilifelab.se

Source https://github.com/pjohansson/pygromacs

1.1 Project Abstract

The motivation of this project was partly to switch *pygromacs* to a test driven development model, and partly to add documentation using Sphinx and *Read the Docs*.

1.2 Moving to test driven development

I started out by choosing *PyTest* as a test suite. A function was added to setup.py to make it easy to run a local test:

```
python setup.py test
```

Later on, coverage reporting was added using a PyTest plug-in. Finally, *Travis CI* [1] and *Coveralls* [2] integration was added, mostly to try the services out.

Using test driven development, the existing class `MdpFile` in `pygromacs/gmxfiles.py` for handling a Gromacs MDP input file with the following functionality was modified:

- Reading input file parameters, values, and optional comments
- Adding, removing and modifying values
- Inspecting the read parameters through searching
- Saving the file to a new path, backing up any existing file at path

Tests were implemented for all present functionality, and changes made to existing functionality were made in a write-tests-first work flow.

[1] https://travis-ci.org/pjohansson/pygromacs
[2] https://coveralls.io/r/pjohansson/pygromacs

1.2.1 Reflections on the switch

Prior to the project the test suite PyUnit had been used to create a single test, but at that point the idea was to write some tests after implementing features to make sure that they worked correctly. I had no experience with actual test driven development.

After trying out the method of writing tests first I'm of the mind that it works very well. As far as I can tell right now it has the following very nice features:

- It forces me to define the functionality of functions before writing them.
- As I write I have easy access to feedback on whether functionality is working properly.
- Since every feature has a test written for it I get constant assurance that features keep working as I modify code.

To me these are all very useful features. By forcing myself to write a test for a single functional addition and then writing its implementation I'm always writing against a simple, very specific target. It's a clear difference from how I've written code in the past: I used to work against a very loosely defined idea of what a function or class should do. The implementation and features could change at will, sometimes several times during the coding of them.

Since I'm not a trained programmer I've had difficulties with actually following the advice of writing a software specification before starting to code, but in some sense writing the tests first has forced me to at least think of how I want results structured before heading into the coding process. That has been a very interesting and useful experience. As a result, my code is leaning more towards doing smaller tasks, but doing them well and returning a well defined result once finished. I am sure that it has reduced the complexity of my programming.

1.2.2 The utility of tests

While having to write a test specification first when doing test driven development has been hugely advantageous to me, the tests themselves feel just as important in many ways. Setting up the initial test and some edge cases that need to be passed helps keeping the actual code clean when programming, but also gives me something akin to a progress report during the implementation. At all times I know what remains to be done until the function is fully featured.

Perhaps most important, though, is the fact that the tests are continuously checked when editing the code. Since unit tests are implemented for hopefully every present functionality, the test suite instantly raises an error if the same result is not produced anymore. This gives me control over what functionality breaks when modifying code, making it much easier to return to old functions and modify them when needed, without the risk of breaking the program in various, possibly obscure ways. I am safe in the knowledge that if my modified code passes the tests, I can move on instead of having to remember which parts of the code are interacting with the modifications. For example, this was put to the test in commit 2078e30 where I made a printing function not return its output, ran the test suite, and moved on when no issues were found.

Making a change to some functionality, then, is more or less as simple as changing the test specification, either modifying the expected controls or adding new ones as required, then changing the function to the new spec. If this breaks another test, either make sure that the spec is not contradicting itself, or return to the function to fix it. This work flow is very comfortable to me.

1.2.3 Writing the tests

It took some time to get used to writing tests in a simple way and still get good coverage. I have tried to write them in very modular ways, with every functionality getting their separate tests, in which I individually also check edge cases. In `test_get_option()` of `pygromacs/tests/test_mdp.py` I write the tests in order of verifying that some options read from a file correspond to ones that I enter manually(`'nsteps'`, `'Tcoupl'`), then add a test for an option that is present in the file but not set (`'include'`), an option that is *not* present in the file (`'not-a-parameter'`) and finally two cases where I try to find an option by entering an integer (10) and boolean (True) instead of a string, to ensure that those return nothing instead of raising a Type Error.

Trying to think of possible edge and error cases for tests is challenging, and again has forced me to think of how my program should handle them even before starting to program. In the same file, the test `test_comment()` sets up quite a few variations on how a comment can be entered for a parameter option, and defines how the end result should appear as. It is the best example of how I used the tests to create the function specification, as I returned to it after the initial pass and rewrote the tests to produce better looking results. The edge cases present in the test put me in line of how to modify the code.

1.3 Documentation using Sphinx

Adding documentation to the project has been a lot of fun using the suggested tools of the course. While taking some time to set up it has shown me not only how simple it is to create fully covered project web sites, but as with tests it has helped me to think my functions through before writing them.

For the project, documentation was added to all functions and classes in `pygromacs/gmxfiles.py` and generated using Sphinx. The module documentation is available at the Read the Docs web page.

1.3.1 Module documentation

Grasping the syntax of .rst files was simple enough for the most part, and a pleasant surprise in how easy it is to use. It took some time to figure out how to link module functions and attributes when writing function documentation, but once the syntax was down it was a breeze to use the *autodoc* tools to parse docstrings. Moving on I plan to create separate pages with usage documentation that is not generated from docstrings, but written in an easy-to-read-manner, but it's comforting to know that it's easy to get both without a lot of extra effort.

1.3.2 Enforcing better code

As with trying out test driven development, properly documenting my functions actually helped me improve the code quality of the project. I noticed that trying to explain the function in a way that's easy to read means that the function itself should be reasonably simple, and not throw unexpected returns or behaviour.

I set out to make sure that every function has a well documented input and output, which means that both have to be on well defined forms. As with test assertions this made me consider up-front what I wanted the function to do, and draft the result in text. To use the same example as before, when writing the documentation for a function that printed the file contents, I thought that the function also returning the output was too ugly to put into text since there's no reason for that return value to be there (other than its current debugging purpose). In commit 2078e30 I revert this behaviour, informed by having written a concise docstring for the function.

1.3.3 Writing prettier docstrings

I was not quite happy with writing docstrings in the ReST markup format, since they became somewhat hard to read when browsing the source. A search on the Internet led me to the Sphinx extension *Napoleon* [3] which parses docstrings written in Numpy or Google format, both of which are easy to read as plain text. In commit b221fb9 I move all docstrings to this format.

A big advantage of this style is that attributes, input and output are clearly labelled and that their Types are encouraged to be a part of any argument. As with the return values as described above this makes me think of how to parse input data when designing the function documentation, which helps me avoid duck typing and encourage that different types of data are used to warp functions in strange ways. Which I have a tendency to do when not keeping myself on a tight leash.

[3] http://sphinxcontrib-napoleon.readthedocs.org/en/latest/

1.4 Final reflections

For me this project has been very useful. My program suite can now easily read and modify .mdp files for Gromacs and further modifications will be well documented. Getting familiar with tests is making me more confident when coding, since I know that I should not be able to break things without noticing. And seeing hands-on that it's easy to write legible documentation and setting up a web page for it was also neat, although I'm probably the only one who will ever read it.

Most interesting though, is that both trying out test driven development as a structure, and writing proper documentation is forcing me to write simpler code. My code will mostly be avoiding functions that change things outside of their scope, or implement weird and undocumented state changes. I'm trying as much as possible to instead write small, simple functions that do only a few things and have well defined input and output.

1.5 Footnotes

pygromacs package

2.1 Submodules

2.2 pygromacs.gmxfiles module

class `pygromacs.gmxfiles.`**`MdpFile`**(*path=''*)

Bases: `builtins.object`

Container for MDP files.

Parameters **path** (*str, optional*) – Read from file at this path

`path`

Path to the last-read file. Used as default by `save()` when writing changes to disk.

`lines`

This is an ordered list of `MdpOption` objects, containing the parameters, values, and comments which together make up a file. It is a list to keep a read file as close to the original as possible when modifying it.

`options`

This is a dictionary of parameters, linking to objects in `lines`. Used internally to quickly access any parameter of that list and thus file.

class **`MdpOption`**(*parameter='', value='', comment='', index=None*)

Bases: `builtins.object`

Container for an MDP option.

Parameters

- **parameter** (*str*) – A parameter,
- **value** (*str*) – its value
- **comment** (*str*) – and comment
- **index** (*int*) – Index of option in `MdpFile.lines`

`print`(*comment=True*)

Print option as a line.

Uses a standard MDP format. Use `comment` to print or ignore a comment.

`MdpFile.`**`get_option`**(*parameter*)

Return the value of a parameter.

Parameters **parameter** (*str*) – A parameter

Returns The parameter value, empty if not found

Return type str

`MdpFile.`**`print`**`(`*comment=True*`)`

Print the current file.

Parameters **comment** (*bool, optional*) – Print or ignore comments

`MdpFile.`**`print_option`**`(`*parameter*`)`

Print a parameter, its value and comment.

`MdpFile.`**`read`**`(`*path*`)`

Read an MDP file at `path`.

Updates `path` to given value. Parameters and lines are stored in `lines` and `options`.

`MdpFile.`**`remove_option`**`(`*parameter*`)`

Remove a parameter from the file.

`MdpFile.`**`save`**`(`*path='‘, verbose=True, ext='mdp'*`)`

Save current MDP file.

The written content is set in `lines`.

Parameters

- **path** (*str, optional*) – Write file to this path (default: `path`)
- **verbose** (*bool, optional*) – Print information about save
- **ext** (*str, optional*) – Use this file extension (default: ‘mdp’)

`MdpFile.`**`search`**`(`*parameter*`)`

Search for a parameter in the file.

Prints any matching option and its value.

Returns Number of options found

Return type int

`MdpFile.`**`set_comment`**`(`*parameter, comment*`)`

Add a comment to a parameter.

`MdpFile.`**`set_option`**`(`*parameter, value, comment='‘*`)`

Set a parameter value.

If the parameter is not set the option is appended to end of `lines`.

Parameters

- **parameter** (*str*) – A parameter to set,
- **value** (*str*) – its new value
- **comment** (*str, optional*) – and comment

class `pygromacs.gmxfiles.`**`Topol`**`(`***kwargs*`)`

Bases: `builtins.object`

2.3 pygromacs.utils module

`pygromacs.utils.`**`prepare_path`**`(`*path, verbose=True*`)`

Prepare a path for writing.

Creates required directories and backs up any conflicting file.

Parameters

- **path** (*str*) – Path to file
- **verbose** (*bool, optional*) – Whether or not to print information about a performed backup

Returns The path to a backed up file, empty if no backup was taken

Return type str

2.4 Module contents

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index