Software Engineering Workshop Exercises

Thomas Stainer (UKAEA)

August 14, 2018

1 Setup

For the sake of these exercises we require Python3, pip3 and git installed.
Once you have these dependencies, we should install pytest-cov to run our tests and get the code coverage.

```
[1206][tom:~/SWE18/] pip3 install --user pytest-cov
```

The code for the exercises is hosted on GitHub and mirrored on GitLab. These are publicly available and will stay available for at least one year after this course.

The site address for each is listed below:
GitHub: `https://github.com/thomasms/SEW18`
GitLab: `https://gitlab.com/thomasms/SEW18`

2 Exercise 1

This is the introductory example to get to grips with unit testing.
Checkout branch 'exercise1' for this case.

A python module, named 'numerical.py', and a corresponding unit test module, namely 'numericaltest.py' should exist in the subdirectory 'exercise/1'.

```
[1206][tom:~/SWE18/exercises/1] ls
    numerical.py numericaltest.py
```

Two public functions have been implemented which simply aim to check the value passed to it is a float (isfloat) and to get the value (getfloat). Although not very exciting it is a good introduction and is actually a useful function when parsing Fortran written files, like ENDF format files, since Fortran allows for a strange number format as 4.5676-307, meaning 4.5676×10^{307}. Python does not recognise this as a float and can then cause errors when trying to parse files with Fortran style formatting.

Can you implement the two unit tests to try and cover all the major possible inputs and see if the implementation works?

What code coverage did you get?
Hint: code coverage can be retrieved by running the following command in the base directory.

```
[1206][tom:~/SWE18/] pytest --cov=numerical exercises/1/numericaltest.py
```

An example solution is given in the branch 'exercise1_sol'.

3 Exercise 2

This exercise builds on the first example, showing the advantages of unit testing by catching a bug in our code.
Checkout branch 'exercise2' for this case.

The same module (numerical.py) and test module (numericaltest.py) should exist in the subdirectory 'exercise/2'.

```
[1206][tom:~/SWE18/exercises/2] ls
    numerical.py numericaltest.py
```

The implementation is the same as in exercise 1 but now we have added some extra test cases, which we think should pass but actually cause our tests to fail.

Can you fix the implementation to correct the bug and make the tests pass?

Did you get 97% code coverage? Can you add a case that gets 100% code coverage.
Hint: code coverage can be retrieved by running the following command in the base directory.

```
[1206][tom:~/SWE18/] pytest --cov=numerical exercises/2/numericaltest.py
```

An example solution is given in the branch 'exercise2_sol'.

4 Exercise 3

Using the previous exercise module we will setup a CI system using GitLab. For this we first need to setup our own repository on GitLab to enable using their CI system.

A template *.gitlab-ci.yml* file is given in the 'setup_ci' branch of the main repository, as below.

```
variables:
  GET_SOURCES_ATTEMPTS: 3


mybuild:
    image: fispact/ubuntu:18.04_gfortran_7
    script:
    - echo "Running an empty CI"
    tags:
    - docker
```

If you commit this to your repository, does it trigger a new build?
Can you alter this to now run our numericaltest.py?
Can you add a badge to your readme showing the build status and code coverage?

The solution for this can be found on branch 'setup_ci_sol'.

Additional: It is useful to test on multiple operating systems using different versions of python. This can do some smoke testing, for example does it work on Windows, is it Python2 compatible? An example of how to do this is shown on branch 'ci_build_template'. On this branch a template is used to avoid repetition of code and we can easily add new environments.

5 Exercise 4

We will now restructure the code to separate our tests from our source, with the aim to create a package.
In the previous exercises we had a flat hierarchy with our tests and source at the top level, this is not advisable and does not help when the project grows in size. It is better to introduce a structure for our code. A common and recommended approach is to have the following structure:

```
- mypackage
    - .gitignore
    - .gitlab-ci.yml
    - LICENSE
    - README
    - setup.py
    - mypackage
        - __init__.py
        - module1.py
```

```
        - module2.py
        ...
    - tests
        - testsuite.py
        - module1tests.py
        - module2tests.py
        ...
    - examples
        - example1.py
        - example2.py
        ...
```

The code on the branch 'package' uses this structure with a package name of sewpy. The modules remain the same but we have now packaged them up, making it easier to distribute and use.

To install the package we can use the setup.py file by running the following in the base directory.

```
pip3 install .
```

Then it is trivial to import the exposed modules via

```
import sewpy as sp

a = sp.getfloat("4.5-9")
...
```

This works because we have exposed our modules via the __init__.py file inside the package. Looking inside you will see the imports declared.

```
from sewpy.numerical import getfloat, isfloat
```

Are you able to install the package locally and run the examples?

6 Exercise 5

We will now use Test Driven Developement (TDD) to write a ENDFreader module.
ENDF file format readers are commonly needed for many nuclear data problems and analyses. It seems everyone writes there own, so I we will continue that trend.
Some skeleton classes have been written for certain ENDF record types: text, cont, head, list. However, the implementation to parse these has been omitted. We have define some test cases already using the interface defined in the module,

showing how we expect it to behave. Without the implementation, the test suite should produce some failures. Our challenge is to implement the parse methods to pass the tests.
The code for this exercise is on branch 'endfreader' of the main repository. I hope you notice that the current build is failing for this branch because it is lacking the implementation required to pass the tests.

Can you implement the *readLine* and *read* methods for the ENDFTextRecord, ENDFContRecord, ENDFHeadRecord and ENDFListRecord classes to make the tests pass?

The solution for this can be found on branch 'endfreader_sol'.

7 Exercise 6

We will now attempt something different to touch upon regression testing. For this exercise we will need to work in brach 'regtests'.
A script exists which is located in the *sewpy/tools* directory, named *sewpyconvert.py*. This is a basic script that attempts to solve a Bateman style equation given a matrix and vector. It takes an input file of the form below and produces an output file. It has not been unit tested so we attempt to setup regression tests first to make sure we don't break existing functionality if we make some changes.

```
initial_time
0.0

final_time
1e3

timestep
1.0

inventory
2
200.0
5.0

matrix
2
-1.0, 0.0
1.0, 0.0
```

This exercise aims to illustrate how to setup some basic regression tests and how to incorporate this into our CI.

A shell script exists (sorry windows users - can you make a similar bat script?) to run the regression tests locally. It is encouraged to run our tests locally first before committing them, so we need to do this easily, hence the need for a script. This script runs the solver for 2 different cases (input files) and then compares the results using the standard diff command. More complex systems will have a better framework for regression testing, but a simple diff will work for now, as don't worry about tolerances for this example.

Running the script will fail since the reference output files do not exist. Can you create the reference output files and add them to the ref directory?

Does the script now pass?

Can you add this to the CI also?

Can check it works by changing the code to break the regression tests?