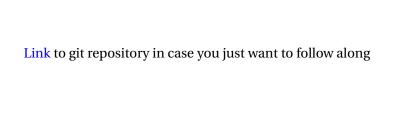
Python Projects and Unit Testing

Structuring Python projects and efficiently testing code with Continuous Integration

Hitham Hassan

IPPP, Dept. of Physics Durham University

22/02/2021



Why Test Code?

- Following from the debugging talk half of your time programming is debugging.
- Want to make sure that sensible output is given for sensible and insensible input.
- Important that all code meets quality standards before deployment.
 - Does the code use the conventions of the programming language (e.g. style guides)?
 - How much of the code can we account for the behaviour of (code coverage)?
 - ▶ Does the code behave as intended?
- Frameworks now exist that allow for automated code testing that makes this far less cumbersome.

Topics Covered Today

- 1. How to set up and structure Python projects.
- 2. Exporting Python scripts as packages in your local environment.
- 3. Comprehensive unit testing in Python.
- 4. Continuous Integration (CI) with *GitHub Actions* to automate the testing procedure and notify you where issues arise.

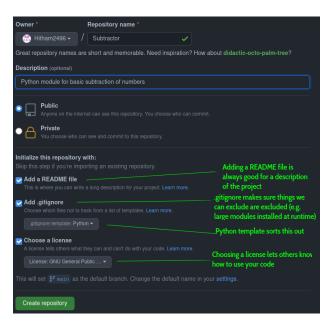
IMPORTANT: These principles can be applied to *any* project in *any* language (within reason) - Python makes for a soft introduction to the topic in this talk.

Today's task: write a 'difference' function diff(x, y) that subtracts two numbers¹.

¹I'm being deliberately vague here

Python Project Structure

Python Project Structure - Create Git Repository



Python Project Structure - Directory Format

Firstly, clone your repository, for me it clones into a directory called Subtractor.

Create the following sets of directories in 'Subtractor' (or your equivalent)

Subtractor

- Subtractor/src this is a directory for our source code and modules
 - Subtractor/src/subtractor this is the name I chose for the module we write today
- **Subtractor/tests** this is for the tests we want to execute

If you create more modules, place them in their own directories in **Subtractor/src**

Python Project Structure - Our Script

#!/usr/bin/env python

Create a Python script e.g. diff_calc.py in Subtractor/src/subtractor. These are the contents of mine:

```
def diff(x, y):
    """
    Finds the difference between
    two numbers.
    """
    return x -y
```

- ▶ In the same directory create a blank file called __init__.py
- ► This tells Python that we're building a module that we (will) want to import elsewhere in the project.

Exporting Packages

Exporting Packages - Endless Configuration Files

- To export our code as a module in our local environment, python requires many configuration files, all of these are to be created in **Subtractor**:
 - pyproject.toml this specifies which build tools will be used to create the module.
 - setup.py this will be execute the necessary methods to set up the module compilation.
 - setup.cfg configuration file for setup.py
 - requirements.txt other modules we require (e.g. numPy) and their specific versions, we don't need any for this project.
 - requirements_dev.txt other modules we require for developing (i.e. testing in out case) and their specific versions, we do need this!
- ► The PyPI documentation provides in-depth guidelines for these summaries in backup slides.

Interlude - What are we doing again?

- We're trying to get python to run our code in our local environment - this makes using modules that we've written simpler without hardcoding any directories.
- ► In **Subtractor**, run pip install -e .
- This uses pip to install the code specified in the configuration files such that we can import it into our code like a normal python package - anywhere.
- Try it! Open a python console anywhere and run:

```
>>> from subtractor.diff_calc import diff
>>> diff(10,2)
8
```

► The-e flag in the pip command ensures that we install in 'editable' mode, i.e. suited for developing.

Unit Testing in Python

Unit Testing - Sensible Output

What should diff return in the following cases:

- ▶ diff(0.1, 0.3)?
- ▶ diff(1, 0.7)?
- ▶ diff(3, 2)?

What about:

- ▶ diff("hi",6)?
- ▶ diff(4)?

We can want to ensure that the output is sensible - this includes sensible error messages where the input is insensible.

Unit Testing - Sensible Output

- For diff(0.1, 0.3), the Python subtraction operator allows us to return negative floats (the output is sensible).
- Similarly the subtraction operator allows us to subtract floats and integers (even mixed) so diff(1, 0,7) and diff(3, 2) work.
- Try diff("hi", 6) in the python console you had open is the error you get very helpful? We can do better.
- ▶ What about diff (4)? Do we want an error in this case²? We can specify a default value for e.g. y.

²we can still raise an error, this isn't so clear cut

Unit Testing - Let's upgrade diff_calc.py

```
#!/usr/bin/env python
def diff(x, y=0):
    Finds the difference between
    two numbers x and y, default
    value of y is 0.
    11 11 11
    if not (isinstance(x, int) or isinstance(x, float)):
        raise ValueError(
                "diff accepts only float or int arguments.")
    if not (isinstance(y, int) or isinstance(y, float)):
        raise ValueError(
                "diff accepts only float or int arguments.")
    return x -y
```

Unit Testing - unittest Library

- We specified necessary components for the tests in our configuration file - let's write a test for diff_calc.py.
- Convention: in tests create a python script called test_diff_calc.py. Also create a blank __init__.py file in the same place.
- We will use Python's in-built unittest package.
- ► To do this we create a *derived class* of the unittest. TestCase object this will create an object that contains all of the different tests we want to run.
- ► These tests will be run by calling the main() method in unittest.

Unit Testing - test_diff_calc.py

```
import unittest
from subtractor.diff_calc import diff
class TestSubtract(unittest.TestCase):
    def test_something(self):
    def test_something_else(self):
if __name__ == "__main__":
    unittest.main(verbosity=2)
```

Unit Testing - unittest Library

- ► The scope for testing in Python is huge, would encourage looking up what you can do.
- For our purposes the test functions we use will be void and undecorated, we will make use of the simple assert methods available in unittest
 - assert(some_bool) the test will pass iff some_bool==True
 - assertEqual(x, y) the test will pass iff x==y
 - assertRaises(some_exception, some_method, *args) the test will pass iff some_method(*args) raises
 some_exception
- Once we have made our tests (look at the example on GitHub if you want inspiration), we can run them by executing pytest in Subtractor.

Unit Testing - What Could Go Wrong?

- Since we are only subtracting two numbers, it's hard to see what could go wrong. However if you execute the example on GitHub you should get an error.
- The verbose option in the test file gives some comprehensive output, the key lines are:

- Python uses floating point numbers which to high precision can lead to inaccuracies. Try 1.2 - 0.2 in a Python console!

Unit Testing - Upgrade diff_calc.py Again

```
#!/usr/bin/env python
def diff(x, y=0, TOL=9):
    11 11 11
    Finds the difference between
    two numbers x and y, default
    value of y is 0.
    11 11 11
    if not (isinstance(x, int) or isinstance(x, float)):
        raise ValueError(
                 "diff accepts only float or int arguments.")
    if not (isinstance(y, int) or isinstance(y, float)):
        raise ValueError(
                "diff accepts only float or int arguments.")
    return round(x -y, TOL)
```

Unit Testing - Style Guides and Linter

- ▶ PEP8 is seen as the most universal Python style guide; it has many rules regarding how code is laid out (e.g. whitespace, comments, variable names, etc.).
- Rather than manually checking against the conditions required to accord with the style guide we can use a *linter*.
- flake8 is a Python linter, running flake8 somefile.py will tell you where the style guide is not followed in somefile.py.
- Running it on our module shows that we have a whitespace issue in our return statement, adding a space: x -y to x - y fixes the issue.

Continuous Integration

Continuous Integration

- Manually running these tests every time is cumbersome if we have many tests and source files we may forget to run on all of them.
- We can use Continuous Integration (CI) to build and test our project every time we make a small change.
- GitHub Actions makes this very convenient allows us to create a workflow that executes certain tasks (e.g. building, testing, linting ...) in an order and configuration we specify.
- Uses YAML configuration scripts lots of templates available!
- Go to Actions -> Continuous Integration -> {choose whatever you like!}

Continuous Integration

- For our script, we are treating it as a module so choose *Python package*, looking at the script we see that it builds our project, tests all modules and lints all scripts!
- We can customise certain aspects:
 - do we want to run this script whenever we commit to any branch, or just main?
 - which versions of Python do we want to test with? Which operating systems?
 - we can add e.g. requirements_dev.txt to the script in case we have other test libraries that are needed.
- When we commit a change to the main branch (for instance), a workflow will be sent to the queue where it will wait until a compatible *runner* is free to run it.
- You can inspect the output of these workflows to see where the errors occur - our linter and rounding error would be picked up here!

Continuous Integration and Delivery - Ideas

- For Python we can do many things build applications, build modules, ...
- We can equally organise a Continuous Delivery (CD) system something that publishes or deploys our work in some way.
- We can use CD (which works similarly through GitHub Actions) to publish our package to PyPI so that anyone can install it with pip!
- Can provide sensitive information e.g. login info for PyPI that can be stored securely in *GitHub Secrets* for use in these workflows.
- ► In practice, using API (Application Programming Interface) keys is preferred over keeping passwords stored PyPI grants the option to create these for all your projects!

COntinuous Integration and Delivery - Ideas

The scope of CI/CD extends *well* beyond testing Python modules, here are just a few off-the-cuff examples:

- Build and deploy a website.
- Build a paper with LaTeX and host it online.
- Execute the same code in a docker container to mimic an environment that is difficult to set up correctly by hand (this is really useful for HEP software).

What will you do?

Backup

Some Notes on Security

- ▶ Brief important note with PyPI anyone can upload code to be installed via pip so make sure you never use pip with sudo!
- Similarly, if you want to provide e.g. login details for a CD system you can upload these to GitHub secrets where they are securely kept.
- It is better practice where available to use API (Application Programming Interface) keys for authentication.
- ► For Python packages, you can generate these online in PyPI for each project you create and add them to GitHub secrets.

Exporting Packages - pyproject.toml

► This file contains instructions for which build tools to use and is fairly standard - will be applicable as is to most basic projects. These are the contents:

```
[build-system]
requires = ["setuptools>=42.0", "wheel"]
build-backend = "setuptools.build_meta"
```

► The build tools (e.g. pip and build) used by Python require this file to understand how the module is supposed to be built.

Exporting Packages - setup.py

- Previously, *dynamic* configuration was the favoured approach
 all the setup commands are executed in setup.py.
- ► This can be problematic can lead to security issues.
- This is one reason why you should **NOT** use pip with sudo
- Now we use static configuration, we provide the settings for setup.py in a configuration file. setup.py looks like:

```
from setuptools import setup
if __name__ == "__main__":
    setup()
```

Exporting Packages - setup.py - Common Issue

- ► If you are having permission issues, specify the -user option when installing your module.
- ► To enable this compatibility you may need to modify your setup.py such that it looks like:

```
from setuptools import setup
import site
import sys

site.ENABLE_USER_SITE = "--user" in sys.argv[1:]

if __name__ == "__main__":
    setup()
```

Exporting Packages - setup.cfg

- This is the configuration file for setup.py
- ▶ The format is simple, we have three main sections:
 - [metadata] basic information about the module e.g. name, author, version, license, description, etc.
 - [options] options that need to be set for correct installation, e.g. which packages need to be build, what packages the install requires, where the packages are stored, etc.
 - [options.extras_require] extra requirements for development, for our purposes this is where we will list the modules needed for testing.
- ▶ Please base this on the version available on the repo, altering names etc. where necessary.