Research Software Engineering with Python - MPHYG001
Coursework 1 Report

Oscar Bennett
Student Number: 14087294

December 17, 2015

1 Project Overview

This report accompanies the submission of the first coursework assignment of the UCL module MPHYG001. The work described here involved turning a piece of code into the form of package to be released. The specific function of the code provided for this project is to plot a graph of the amount of 'greenness' between two chosen cities.

In this first section I describe the details of how I carried out the project. I describe the folder structure of the completed package, the version control strategy, the command line entry point, the way the package was prepared for installation on other computers, and the way automated unit tests were implemented to ensure each part of the program worked as expected. In the later sections I discuss some problems I encountered during the project, reflect on the pros and cons of going through the process of packaging a program in this manner, and finally consider the important steps to take in order to build up a community of users.

1.1 Project Structure

The folder structure along with the files included in the package are laid out in Figure 1.

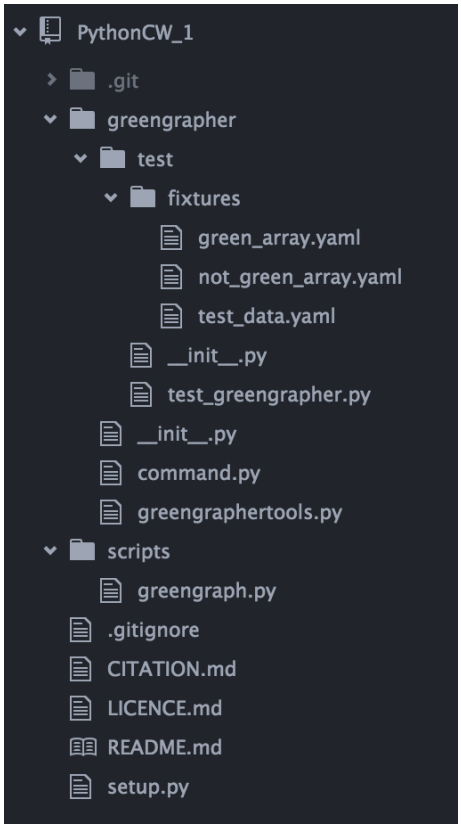

Figure 1: Layout of folders and files within the package

A README file, CITATION file, and LICENCE file were included in the package's root folder. The README file included a description of the package, how to install it and how to use it. The CITATION file described how to cite my work if someone wanted to use my package in published work. The LICENCE file contained the details of my chosen licence (The MIT Licence in this case).

1.2 Git

Throughout the development of this project Git version control was used along with a private GitHub repository at https://github.com. Regular commits were made during the course of completing the project accompanied by informative comments. The .git folder containing all this information has been submitted along with the code.

During the course of completing the project I generally used a 'master' Git branch to point to the latest stable version of the project and a 'develop' Git branch to develop and test new extensions of the code. I regularly pushed to the remote GitHub repository in order to keep my work backed up. I also used the Issue tracking capabilities on GitHub to keep track of which parts of the assignment still needed to be completed and which parts of the code were not yet working correctly.

1.3 Command line entry point

The command line entry point after installation of the package is provided by running the script greengraph.py (seen in the scripts folder in Figure 1). Arguments can be passed to the function which control: which two cities are looked at; the number of greenness samples (or steps) taken between the two cities; and the name of the file that the graph will be saved to. The help message that is displayed by the program illustrating its use is shown in Figure 2.

```
>> greengraph.py [-h] [--begin BEGIN] [--end END] [--steps STEPS] [--out OUT]

A program which displays a graph of the amount of green along a straight line
between two cities on Google Maps

optional arguments:
  -h, --help                show this help message and exit

  --begin BEGIN, -b BEGIN   City to start measuring from

  --end END, -e END         City to finish measuring at

  --steps STEPS, -s STEPS   Number of equal steps between cities to show greenness. Default is 20.

  --out OUT, -o OUT         Name of output file to save graph to
```

Figure 2: The help message printed by the program which demonstrates how to run the program and pass arguments to it

1.4 Packaging for installation

The package was setup for easy installation on other computers (using something like pip). This was achieved by structuring the project in a standard way (see Figure 1) as well as including a setup.py file in the root folder of the package which contains instructions for a computer on how to correctly install the package as well as which external packages the code needs in order to run. The contents of the setup.py file is shown in Figure 3.

setup.py

```
1  from setuptools import setup, find_packages
2
3  setup(
4      name = 'greengrapher',
5      version = '1.0',
6      packages = find_packages(exclude = ['*test']),
7      scripts = ['scripts/greengraph.py'],
8      install_requires = ['numpy','matplotlib','geopy','requests','nose']
9  )
```

Figure 3: Contents of the setup.py file

1.5 Tests

An array of automated tests of the code were implemented and placed in the file test_greengrapher.py. These tests load data from files in the fixtures folder in order to test each part of the program in turn under different conditions. I used YAML as my parser. The tests can be run quickly and conveniently using the nose testing

framework. Simply typing 'nosetests' into the command line when in the root directory of the package will run all the tests.

Thirteen unit tests were implemented which all together test each class and method contained within the package in turn. The tests are run under different conditions (the case when the map found has no green pixels, and the case when it contains *only* green pixels). A test is also performed to ensure the program raises a value error if an inappropriate 'step' value is passed to the program.

The program, when running under normal conditions, needs to interact with the internet to obtain the necessary mapping information to complete its task. In order to make testing the program possible without an internet connection I used mocks and mock patches to prevent the program interacting with the internet during the tests. The mocks simulate the information passing to and from the internet and the program. An example of one of the tests is shown in Figure 4.

```
def test_construct_Map():
    #Tests the Map class constructor with a mocks to prevent interaction with the internet
    with patch.object(requests,'get') as mock_get:
        with patch.object(img,'imread') as mock_imread:
            mock_map = greengraphertools.Map(111,222)

        mock_get.assert_called_with(
        "http://maps.googleapis.com/maps/api/staticmap?",
        params = {
        'sensor':'false',
        'zoom':10,
        'size':'400x400',
        'center':'111,222',
        'style':'feature:all|element:labels|visibility:off',
        'maptype':'satellite'})
```

Figure 4: An example test. This is testing the constructor of the Map class, ensuring that the correct information is passed to the requests.get function when an instance of the Map class is created. Mocks are used to prevent the program actually calling the requests.get function and prevent the program throwing an error when an actual map image is not returned and passed to the img.imread function.

2 Problems encountered

The main difficulty I encountered during this project related to designing good tests for the code. The best way of testing a program is not always immediately obvious and involves trying to anticipate all the important ways some code might potentially break (or be broken by someone else!). I tried to design tests that would ensure that the program would function as expected assuming all the tests were passed.

3 Pros and Cons of preparing work for release

This project demonstrated to me the significant amount of work involved in preparing a program to be released as a package. The large amount of work involved in the process was a drawback. However although it involved a lot of work I found the process was actually a very helpful way of forcing me to adopt some healthy coding habits. It ensured that I structured my work well, wrote robust code, and implemented good automated tests designed to prevent future changes from introducing bugs (changes perhaps suggested by outside users sending pull-requests on GitHub).

The use of a setup.py file along with programs like pip convey a significant advantage when distributing work because they ensure that program dependancies are met and managed in systematic and automated way on other computers. Doing this manually would be very fiddly and would almost certainly lead to problems when other people tried to install and use your work.

The PyPI package index is a catalogue of all open source python packages. Placing a new package in PyPI makes it very easy for others to download and install your work (simply by typing "pip install your_package"

into their command line). Placing your work in a convenient place like this encourages the dissemination of your work and increases the chance that your work will have a greater impact on the scientific community.

4 Steps to build a community of users

When you write and release a piece of software you ideally want it to be found and used by a wide range of other people in order for the work to have an impact in your field. Additionally, if a large number of people start using your code, they can sometimes start improving or extending it for you! There are ways that you can make this more likely to happen.

Publishing your work on the PyPI package index and on GitHub is a good way to put your work in a place where it can be easily found and downloaded by others. Releasing the software open source with a permissive licence increases the likelihood of people adopting it into their own work as well. Responding quickly and helpfully to queries about your code from other users will encourage them to continue to use it and tell others about it. People sometimes suggest improvements or extensions that would make the software more useful. Responding to these suggestions and implementing the changes into the code can make your work attractive/interesting to a wider base of users. Publishing scientific work that uses and cites your software package is another good way of raising awareness of your work and encouraging other people to use it.