UNIVERSITY COLLEGE LONDON

EPSRC CENTRE FOR DOCTORAL TRAINING

DELIVERING QUANTUM TECHNOLOGIES

MPHYG001 First Continuous Assessment: Packaging Greengraph

Author:
Gareth Siôn JONES

Student Number:
16070299

January 6, 2017

1 Introduction

The program '*greengraph*' generates a graph of the proportion of green pixels in a series of satellite images between two points. A task was assigned to take this code and turn it into a package ready for release, meeting minimum software engineering requirements. The requirements of this work are as follows:

- Break the code up into appropriate files, and an appropriate folder structure
- Employ git version control
- Add a command line entry point
- Package ready for PIP installation
- Create automated tests for each method and class, using mocks to avoid interacting with the Internet

This report details the process undertaken to achieve these goals.

1.1 Folder Structure

The supplied code was given as one continuous program. It was separated into individual classes, and unit tests were written for each class. The tree of this project is shown below:

```
|    CITATION.md
|    LICENSE.md
|    README.md
|    setup.py
|    USAGE.md
|
|---greengraph
|    |    command.py
|    |    graph.py
|    |    map.py
|    |    __init__.py
|    |
|    |---test
|         |    test_graph.py
|         |    test_map.py
|         |    __init__.py
|         |
|         |---fixtures
|                   journey.yaml
|                   location_sequence.yaml
|                   london_green.npy
|                   london_sat.png
|
|---scripts
          greengraph
```

1.2 Git Version Control

The project was uploaded to a personal git hub account, and updated and committed regularly. A folder containing the git submission is included with this submission.

2 Command Line Entry Point

One of the objectives of this project was to allow a user to run the program from the command line, using the following example command:

greengraph –start London –end Cardiff –steps 50 –out graph.png

Terms preceded with '–' are optional arguments. If the user does not provide these arguments, the execution will default to the above arguments.

In order to achieve this, *command.py* was created, which uses Python's argparse library to parse user input command line arguments:

```python
#!/usr/bin/env python
from matplotlib import pyplot as plt
from argparse import ArgumentParser
from .graph import Greengraph

def process():
    parser = ArgumentParser(description = "Generate a graph showing the ammount " +
            "of green space between two geographical locations by determining" +
            " the number of green pixels in a satellite image")
    parser.add_argument('--start', '-s', default="London", required=False,
            help='Starting location')
    parser.add_argument('--end', '-d', default="Cardiff", required=False,
            help='End Location')
    parser.add_argument('--steps', '-p', default="50", required=False,
            help='Number of steps between the start and end locations')
    parser.add_argument('--out', '-o', default="graph.png", required=False,
            help='Output file name')

    arguments = parser.parse_args()

    mygraph=Greengraph(arguments.start, arguments.end)
    data = mygraph.green_between(arguments.steps)

    fig = plt.plot(data)
    fig = plt.savefig(arguments.out)

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

This function passes the command line arguments as arguments to methods in the Greengraph class, and plots the resulting graph.

The entry point to the program was implemented using the setuptools library in the *setup.py* file:

```python
#!/usr/bin/env python

from setuptools import setup, find_packages

setup(
    name = "greengraph",
    version = "0.1.0",
    packages = find_packages(exclude=['*test']),
    scripts = ['scripts/greengraph'],
    install_requires = ['numpy', 'geopy', 'matplotlib', 'requests', 'argparse']
)
```

The setup file searches for the setup script *greengraph*, which ultimately allows the program to run from the command line:

```
from greengraph.command import process
process()
```

Once a user runs the setup file, he can run greengraph from the command line.

3 Problems Encountered

The primary difficulty faced was, during the testing of the Map class, in the show_green method, self is not passed as an argument to the method, and the global variable *array* is not defined. Modifications were made such that self was passed, and array was changed to *np.array*. However, show_green is not used in the program.

One of the major difficulties faced was developing on two different computers, one running python 2.7 and the other running python 3.5. The updated python does not have full backwards compatibility with it's predecessor, and so at several points attempting to use the mock library became difficult, with python 3.5 throwing TypeErrors.

A further difficulty came when initially attempting to use the suggesting command line arguments syntax from the assignment brief. 'from' is a keyword in python, and so had to be modified to 'start'.

4 Advantages and Disadvantages of Preparing Work for Release, and the Use of Package Managers and Package Indexes

Although preparing work for release to be installed using package managers can be a cumbersome and time intensive process, it does have many advantages. Package indexes are repositories in which one can find many well defined, useful libraries, removing the need to develop code which has already been developed by another party. These libraries have to conform to a certain standard, and so their method of installation is uniform across all libraries. It is possible to install libraries using the package index alone, but it is often more convenient to use a package manager, which will offer tools for both the installation and management of the package. In addition, these package managers maintains all of the install dependencies, meaning that if a package requires other libraries to work, the package manager will deal with this autonomously.

Although both package managers and indexes are useful, they do come at a cost. They are language dependent, meaning that if a project includes or depends upon code in multiple languages, extra difficulties will arise. This can be solved using certain package managers like Conda, but these only work in particular scenarios. Solutions include using package managers such as Conda but this only works for particular scenarios. Some packages are also operating system dependent, meaning that one may need to maintain code for multiple package indexes and managers.

5 Building a Community of Users

There are several steps which could be taken to build a community of contributors and users of a library.

One such step is by using a version control managing system such as GitHub. This allows the code to be easily shared to a wide number of users, whom can both use and contribute to the library. Version control allows developers to keep track of working versions whilst bugs are being addressed and updates applied.

In order to inspire confidence in other developers to join a project, there needs to be at least one consistent lead contributor to the project, who acts as a point of contact to interested parties and maintains the project. If the project is allowed to fall into a state of not being maintained, other users will feel less inclined to join. Implementing good coding practices, such as designing for testability, will further reassure the community of the integrity of the project.

Another step that can be taken to ensure a community of users for a project develops is the use of open, creative commons licenses. These allow other users access to code, freely distributed, but also ensures that the developers get credit for their work.