MPHYSG001: Packaging Greengraph

Padraic Calpin
PHDCA33

January 10, 2016

Command Line Interface

```
usage: greengraph [-h] -t START -f END [-o FILENAME]
 [--save-maps] [-s STEPS]

A tool for calculating the amount of green between two places, given
by the place name or a coordinate pair lat,lon.

optional arguments:
  -h, --help            show this help message and exit
  -t START, --to START  Starting location
  -f END, --from END    Ending location
  -o FILENAME, --out FILENAME
                        Output filename
  --save-maps           Save the amount of green at each step
  -s STEPS, --steps STEPS
                        Number of steps
```

Listing 1: Help message for the greengraph command.

Problems Encountered

The main difficulties I had with this assignment were regarding the tests.

Making the code installable with setup.py and creating a command line interface were relatively straightforward, but I found that coming up with effective tests required careful thought about what each function should be doing.

Learning how to effectively trace function calls with Mock and patch was also difficult to get used to, especially when it came to writing a test for the command line interface, which included 10 patched function calls, some of which had to pass data to each other.

In particular, I found it hard to write tests that I felt were *effective*, and not just making sure the code doesn't crash. Coming up with fixtures to test, in particular, was difficult, and this code could probably benefit from using fixture files to more comprehensively test with a range of inputs. That said, I do feel that the tests I've written here do a good job ensuring the code works as advertised.

Discussion of Open Source Practices

For python, the tools required to release a project are very mature; while greengraph is a relatively simple package, it only required a small glance at the setuptools documentation and a single definition to make it installable. Combined with mature portals like `github.com` and the PyPI index, the direct cost of preparing software for release itself seems quite minimal.

Instead, what could be considered the 'cost' of the Open Source model is in lost revenue potential. When, for example, a LabView machine can sell for £$\mathcal{O}(10^3)$, this is understandable. However, there is a misconception that being Open Source somehow *forbids* making money from software, which is expressly not true; the GPL, for example, clearly states

"When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish)..."

and indeed, several Open Source companies exist which create revenue from offering support, consultancy, or additional services related to the software they release. The GPL is by no means the only Open Source license, either, and it is possible to retain a close hand on intellectual property rights while still open-sourcing code.

The advantages of Open Source are, I feel, significant. Making code available opens it up to review, suggestions, and contributions by and from the community, helping a project to grow and mature. This is why I chose the GPL for this project; I think all software can benefit from being open and reviewable.

The advantages of package managers and repositories are also, I think, significant, and far outweigh any associated costs in terms of management time (which solutions like virtualenv help to mitigate.) By making it easy to find and install useful libraries and manage dependencies, these package managers allow projects to leverage existing, well maintained resources, to achieve more. It's significant that package managers have been implemented for all manner of software, from programming languages (such as PyPI) to frameworks (such as Carthage or npm) to entire operating systems (Homebrew).

Building a community for a project

To develop a community of users, a package needs two things:

1) Documentation

To be useful, the code requires good documentation that allows others to make the most of the software itself. This can be done either using internal tools such as docstrings in Python, writing external documentation, or making use of features like Wikis/readthedocs provided by portals like Github and PyPI.

2) Feedback

When users find errors in the code, or want to request or add additional functionality, there must be a mechanism to do so. Again, this can be achieved painlessly using the Issues feature on Github, or by hosting your own issue tracker.