MPHYG001

Packaging Greengraph

Simon Stiebellehner (ucabsti)

January 2017

1 Usage of command line entry point

Greengraph can be invoked from the command line. It requires two positional arguments, which are two geographical points (i.e. begin and end). Since a default value of 25 is set for steps, this is an optional argument. Furthermore, users can specify a location (i.e. path, optional argument) to which the output graph is saved. If the user chooses not to specify a path, the graph is displayed using matplotlib but not automatically saved.

```
usage: greengraph [-h] [-s STEPS] [-p PATH] begin end

Generates a graph that displays the number of green pixels per step between
two geographical locations.

positional arguments:
  begin        Enter start location, e.g. 'London'.
  end          Enter location of target destination, e.g. 'Cambridge'.

optional arguments:
  -h, --help   show this help message and exit
  -s STEPS     Steps between begin and end, e.g. '10' (default = 25).
  -p PATH      If specified, graph is saved to location specified by the path,
               e.g. '/home/user/graph.png'. Otherwise, graph is only displayed
               but not auto-saved.
```

2 Problems encountered in completing your work

In the course of the assignment, four major problems were encountered:

2.1 Testing: Mocking/Patching

In general, designing suitable and effective test cases was challenging. Also, building an initial understanding of how mocks and patches are used to test code was difficult. However, this was essential for preventing tests from accessing the Internet and for designing mocks that return suitable values as inputs for other functions.

2.2 Command Line Interface

In the beginning it seemed as if building a working command line interface was a rather trivial task. However, after implementing a first version it quickly turned out that some things did not work as expected. Despite the fact that the package could be installed flawlessly from github (using pip) and local sources, it was not accessible on my computer through the command line (‘command not found’). After considerable time spent doing research on this issue and debugging, I found that the root cause of this was inconsistent naming of one of my files.

2.3 BytesIO/StringIO

As I am using Python 3.5, using StringIO the way it was coded in the assignment code did not work. For this reason I had to modify the original code by replacing the StringIO function through BytesIO in most cases.

2.4 Google API limiting get requests

Briefly after starting the assignment the Google API began to block my get requests (too many requests in too short time). However, signing up for an API key and using it in the get requests solved that problem.

3 Advantages and costs involved in ...

3.1 ... preparing work for release

Preparing work for release means a lot of additional effort compared to writing code for personal use only. First, since others will (hopefully) be using the code, it should be readable and well-documented. Also, the package should be designed in a way so that it can easily be installed using a package manager such as pip. However, this requires additional effort such as writing a setup file. Furthermore, assuming that the author does not want to publish a faulty or little robust package, thorough testing must be implemented. Besides coding work, the author must also decide on miscellaneous things such as licensing, where to publish the package and perhaps how to build a community.

However, there are considerable advantages of undergoing these preparatory steps. Despite the fact that many developers do not enjoy documenting and testing code thoroughly, it is proven that this does not only improve the quality and robustness of the software, but it can also save tremendous amounts of time when implementing changes in future.

I believe that the benefits of preparing software for release mostly compensate for the additional required effort. If one plans to use software for a longer period of time, good documentation and proper tests will sooner or later save a lot of time and effort. Also, after completing this work an eventual publishing of the software is only a small step.

It is important to note that publishing code is no one-way street, but it will lead to receiving feedback that helps the author to improve his code and his coding skills. Besides that, a published package might be cited in scientific publications and a community might even evolve around it.

3.2 ... use of package managers like pip

Before the existence of package managers, users had to manually download and move the right files to the right locations. This was a difficult, tedious and error-prone task. However, fortunately package managers such as 'pip' for Python were developed. Package managers make installing software packages fast and easy. This might sound straightforward, however, the impact this has had on the adoption rate and development of open source programming languages is significant. If the installation of software packages was still as difficult as in pre-package-manager times, there would be considerably less motivation for developers to publish packages and for users to download them. Furthermore, since package managers like pip take care of the installation, this enables developers to program largely platform-independently.

3.3 ... package indexes like PyPI

Package indexes, such as PyPI for Python or CRAN for R are software repositories, which both host and link to software packages that can be submitted by practically anyone. The goal of package indexes is to provide a central repository for software packages to ultimately simplify package installation. Together with package managers package indexes make installing packages very easy. For instance, whereas remotely stored Python packages that are not on PyPI (which is the default package index of pip) must be installed specifying a URL, packages listed on PyPI can be installed using three words, e.g.: 'pip install numpy'.

4 Steps to build a community of users

Naturally, the first step in building a community is **making the code available online** (e.g. github). Since the goal is to make others work with it, the code should be well documented and readable. Moreover, explicitly stating under which **license** the code can be used is essential for giving both the author and (potential) collaborators legal certainty.

As soon as the basic functionality is implemented and tested the code should be **packaged**. This eases the installation process by enabling others to install it using, for example in the case of Python, 'pip'. Easy installation allows more users to use the package, hence, speeds up distribution and boosts awareness for the project. However, only hosting the project on github limits visibility. To overcome this problem the package should be submitted and published on a large platform such as the package index **PyPI**. This also has the advantage of easing installation even further (knowledge of github repository/URL not required).

One should be aware that just publishing a package does not automatically attract contributors. In fact, at least until a community of notable size has grown, the author should **keep the package up-to-date** and **promote** it. No matter how useful the package is, if it's outdated or nobody even knows about it the emergence of a community is unlikely.

Furthermore, when first collaborators have joined the project, an **issue tracking system** should be set up to facilitate efficient collaboration.