Software Release Guide (CWI CI Group)

Jan-Willem Buurlage, Allard Hendriksen

November 13, 2018

Contents

1 Introduction

This document consists of two parts. In the first part, we suggest useful tools, a good project structure, and some further guidelines, for sharing your software projects. In some sense, it is a checklist for your software package to make sure that it is easy to use for others, yet allows you to keep the freedom to develop your software further. The second part, *selected topics*, is a how-to guide for common tasks that you may need to perform while working on software.

These notes are not intended to teach you the basics of programming, source control, or for example using Linux. Instead, it is about everything

you need *after* you become a capable programmer, when you start producing software that becomes useful for others.

One question we should answer before diving in deep, is *why?* Indeed, why should we put so much effort in our software. We are, after all, scientists – not software developers! We believe there are many reasons why we *should* take care in sharing our scientific software, some of them we list below.

- **Collaborations within the group**. Although everyone in the CI CWI group are individual researchers, our work has a lot in common. Most of us work on algorithms, target the same kind of data set, and often face the same problems. For example, more often than not, an issue that you are facing has already been tackled by someone before. Sharing our solutions means not having to reinvent the wheel. Furthermore, someone else their expertise may push your research further. Whether this means a new application of a neural network structure, combining algorithms together for even better reconstructions, or simply using a method you developed for the reconstruction of data recorded in our lab.

- **Collaborations with other groups.** The CI CWI group takes a unique position in research – somewhere in between the hardcore mathematics, and the more experimental side of imaging. With our software, we can help bridge the gap between experimentalists (who are often still using what we consider ancient methods), and algorithm developers (who can be out of touch with the demands and requirements of real-world imaging experiments), by working together with both sides of this spectrum.

- **Publicity**. Putting out software that is being used by others gives a lot of exposure to yourself as a researcher, and the group as a whole. For example, ASTRA has been an important part of our public identity.

- **Real-world impact of your research.** Unfortunately, writing a beautiful article outlining all the ins and outs of your method does not mean that it will also be picked up and used by others. Time is scarce for everyone in science, and anything beyond the most trivial algorithm will most likely not be reimplemented outside our own research group. Publishing software alongside an article makes it *much* easier for others to use and apply your methods.

- **Scientific output.** Slowly but surely, software itself is being recognized as scientific output. There are many journals and conferences

dedicated to scientific software.

It can be intimidating to share your software with colleagues, let alone to release it to the public. We always want to ***clean up the code, and fix this last thing*** before we share it. However, especially since very few of us (if any) are trained as software engineers, it is not always clear what it means to develop good, or *clean*, software. This means that a lot of software is kept private until it is done, and regrettably this often means it is kept private indefinitely.

Hopefully, after reading the next section, you will be fully equipped to turn your software projects into clear, clean and usable packages. Our intention is to ease the transition of your project from something that is written primarily for yourself, to something others can also use.

2 Anatomy of a software project

Let us assume that you have developed an image reconstruction algorithm. You have designed your numerical experiments, and implemented them in Python (perhaps on top of the ASTRA toolbox).

In the most extreme case, the result is a single script. This script contains for example some classes, functions and some code that applies them to a data set.

Here, we will transform this script into a software package that is installable and usable by others, and maintainable and testable by you, the developer. We will take the following steps, which we will explain in detail later. (1) **Update project structure**. A software project consists of more than just the code. In should also include additional files, containing additional information for the user. (2) **Add documentation**. Since you developed the software, you know exactly how it works (and how it doesn't). Unfortunately, your users cannot read your mind. Minimally, you should provide some basic information on how to install and use your software, as well as what your users are allowed to do with it. (3) **Add files for distribution**. Before you can release your software into the wild, you need to add build files, and license. (4) **Release the software**. With a good structure, documentation and distribution files in place – we are ready for release. We add release notes to the project, and assign a version to the software. (5) **Further development**. Unfortunately, having users also means having some responsibility. Luckily, with the setup we recommend, you can continue developing your software without frustrating your users by following a small set of rules.

We will first discuss the steps, and it is not written to *following along as you read*, but rather to give an overview of what a minimal software package consists of. After this initial discussion, we will provide you with more detailed information on how to realize these steps for your software, as most of the steps can be automated.

2.1 Update project structure

First, we need to convert our code into a clean and predictable structure. The recommended structure depends mainly on the programming language used. We only consider Python software for now, but we will discuss other languages later. Our hypothetical software package will be called `hsp`, for hypothetical software package.

We will first set up our directory as follows:

```
example
   run_on_data.py
hsp
    hsp.py
    __init__.py
```

Here, `hsp.py` contains functions and classes, while `run_on_data.py` contains code applying them to data. Note in bigger projects you can split `hsp.py` further into multiple files, grouping them together by functionality. The file `__init__.py` ties them together. There can also be multiple examples.

With this structure, we can use `import hsp` from scripts residing at our root folder. Later we will set our package up so that you can run this import statement from anywhere.

2.2 Add documentation

There are two classes of documentation. First, there is usually a readme file called `README.md`, containing some basic information on the project. Second, detailed documentation is provided on a separate webpage (hosted online or shipped alongside the software). We will discuss a good way to set up initial documentation.

The readme file is comparable to the metadata and abstract of an article. It should provide a potential user with all the information they need to decide whether to use your software. Like writing good abstracts, writing good readme documents is an art form. At the minimum, it should contain

the following. (1) The name of your software package, together with a short (one or two sentences) summary of what it does. (2) Information on how to install the software. If it is a software library, it is usually a good idea to show a short example of code written on top of the library. (3) Information on the authors, the license, how to contribute, and potentially what article to cite if people use your software.

The more detailed documentation can contain the README as a landing page, but typically takes a more *tutorial-like* approach to describe the software. Usually, this means explaining how to do some example task with the software, or explaining the example code that is shipped alongside the library. The example code itself can also be considered documentation, and can usually be a stripped down version of the code you use yourself for e.g. numerical experiments. Furthermore, the documentation should provide an API overview, documenting all the classes and functions you expose to the user. This overview can be generated automatically from your code, using function signatures and optional *docstrings* that you may have written. For Python projects, we recommend using Sphinx to generate the documentation.

2.3 Add files for distribution

Next, you should allow yourself and your users to use your package by a simple `import` statement. For this, you can include a `setup.py` for `setuptools`, allowing your users to do a `pip install`. You can also add support for `conda` by including some basic `.yaml` files.

You should to include a `LICENSE.md` file letting our users know what they are allowed to do with our software. Typically, software written by our group is released under the GPL.

2.4 Release the software

Releasing your software does not only mean that it is made available publicly. It means to provide a fixed, frozen version of the code that has no known bugs, and to assign it a version number. An initial version number can be `v0.1.0`.

This is an important service to your potential users, because it gives them a predictable state of your code to fall back on. If you decide to further develop your code, you can do this freely without worrying about breaking your users' codes, because they can depend on this fixed release.

This brings us to another important file, the release notes, which are

contained in a file `CHANGELOG.md`. For the initial release, this file does not much information. Maybe a notice that it is the first public release, and the date at which it is published. However, as you add, amend, and change your software, you may want to release a next version, say `v0.2.0`. In the release notes, you can let the user know what has changed, and they should be able to see immediately if they should expect any difficulties when upgrading to the new version.

We recommend using GitHub to publish your software. GitHub has a good mechanism for managing releases. Simply click on the *releases* button, and create a new release. Provide a tag, name, and some basic information – and GitHub will automatically mark the current state of the software as a new release. Next, if it is a Python project, you can upload the new release to Anaconda and/or PyPI.

2.5 Further development

Now that your software is out in the public, making big changes to your software probably means breaking your users code. As a courtesy to your users, whenever you add, change or remove something add a line to your change log describing what is new.

When releasing your new software, choose a new version number according to semantic versioning. In brief, when choosing a new number `vX.Y.Z` increment `Z` if you are fixing incorrect behaviour, `Y` if you are adding new functionality but user code should not break, and `X` if you make major changes that can break user code. This gives your users a lot of information on the new release simply by looking at the version number: they can always safely upgrade to your new release if `X` is unchanged, and should check what's new when either `X` or `Y` is increased.

2.6 Putting theory to practice: developing a Python a software package

Instead of performing all the steps outlined above manually, the easiest way to set up your project correctly is by using the `cookiecutter` template we provide.

2.6.1 Set up software project structure

First, install `cookiecutter`.

```
pip install -U cookiecutter
```

We provide a basic template for Python software packages. You can use this template to set up a new directory that contains all the files that were discussed in the previous sections, as well as a mock software project. You can then copy your code into the Python files that is generated by the template (for details on how to do this, see below). First, run the following command, and answer the questions that are asked.

```
cookiecutter https://github.com/cicwi/cookiecutter-cicwi.git
```

This will generate a directory with the name of your project. Next, you need to set up a new GitHub repository. We assume you have made such a repository, and that it resides at `https://github.com/username/project`, and that the name of your project is `hsp`. Next we set up Git.

```
cd hsp # replace with project name
git init
git remote add github https://github.com/username/project
```

To use the template, we require the installation of some dependencies and tools. We assume that you are inside a `conda` environment. You can install all the dependencies by running the following command.

```
make install_dev install
```

2.6.2 Adding your code

Now that everything is set up, we are ready to copy your code into the project. Typically, the code you want to share takes the form of a library. In particular, there are some classes and functions that others can make use of. Simply copy these classes and functions to `hsp/hsp.py`. This file contains the code for your library.

It is a good idea to add a minimal example of how to use your library. This should somehow represent the core functionality of the software and should not be more than a couple of lines of code. For this, you can edit the file `examples/getting_started.py`. Your application code, that for example applies your library to data, can probably also be converted to an interesting example. You can add more scripts to the `examples/` directory showing more advanced usage of your library.

2.6.3 Creating an initial release

Creating an initial release of your software consists of a number of simple steps. First, you should build the documentation by running:

```
make docs
```

and next add the files in `docs/` to your repository. Edit the file `CHANGELOG.md`, and add a release date to `v0.1.0` (change the YYYY-MM-DD).

Next, we will publish the release on GitHub. Push all your changes. Next, go to your GitHub repository page, and click on the button saying **0 releases**, and next on **create a new release**. You are asked to provide three things

- The *tag version* should be v0.1.0.
- The *Release title* can be 'Initial release'.
- What to set as the description for this initial release is up to you, it can for example be one or two sentences about your project.

It may be a good idea to mark your project as a pre-release until your are confident that it is widely usable. Finally, click **Public release**. GitHub will now create a new *tag* (but not a new commit) permanently marking this version of your code as `v0.1.0`. Optionally, you can immediately pull this tag to your local repository:

```
git pull --tags
```

Next, we create a `conda` package:

```
make conda_package
```

If you have set up your account for Anaconda uploads, you can now upload the package, see the tail of the output of the previous command for instructions. This will provide potential users with an easy way to install your software.

2.6.4 Further development

As time progresses, you will most likely continue the development of your software package. We recommend to release a minimal version of your software early, and to release updates often. Updates to software packages come in roughly three forms (with changes ranges from small to big). (1) **patches** are fixes of previously released functionality that do not work according to specifications. (2) **minor** updates add additional functionality, but old programs written on top of your software should still function. (3) **major** updates are those that can break existing code, by changing or removing behaviour of a previous release.

As a service to your users, it is important to maintain `CHANGELOG.md`. Here, you document the main changes you have made since the previous release. This is done in the *Unreleased* section, that after a while may look like this:

```
## [Unreleased]
### Added
- Add a function ‘hsp.myfunction‘.
- Add another function ‘hsp.myotherfunction‘.

### Removed
- Remove obsolete function ‘hsp.myoldfunction‘.

### Changed
- Change default value for argument ‘arg‘ of ‘hsp.fn‘ to ‘None‘.
```

This helps make releasing new version in the future much easier. If you decide to make a new version (again, this can already be done after a single patch: it is important not to hoard new functionality!) then it is like the initial release with the following changes. (1) you need to decide on a new version number `vX.Y.Z`. This is done according to the biggest change you have made. If this is a **patch**, `Z` is increased by one compared to the previous version, if it is a **minor** `Y` is increased by one, otherwise `X` is increased by one. This can be done automatically with any one of the following three commands:

```
bumpversion patch # increase Z
bumpversion minor # increase Y
bumpversion major # increase X
```

Next, we update `CHANGELOG.md`. Change `[Unreleased]` to `[X.Y.Z] - YYYY-MM-DD`. Add a line at the end of the document that looks like this

```
[X.Y.Z]: https://github.com/username/project/compare/vA.B.C...vX.Y.Z
```

W here `vA.B.C` is the version number of the previous release. This allows users to click on the version number in the CHANGELOG and get an overview of all the code changes since the previous release.

The remaining steps are as with the initial release. For a release *Description* you can simply copy the section of the CHANGELOG corresponding to the new version.