autorelease Documentation

Release 0.4.1.dev0

David W.H. Swenson

Mar 15, 2023

Contents

Autorelease is several things at once. It is:

- a set of Travis configs that can be imported to automate a cautious approach to releasing on PyPI
- a suite of tests to ensure that the Travis configs are behaving as desired
- a setup.py that can be used across many projects after simple customization
- a “one true version” approach to keep your version string in the fewest places possible
- useful scripts to handle things related to automating releases, such as automatically drafting release notes based on GitHub PR tags

Each of these things is individually useful, but the parts work together to make something greater than their sum.

CHAPTER 1

Philosophy

Part of running a software project involves managing the code repository. This leads to issues of maintaining old code while simultaneously developing new code, and avoiding dumping experimental code on user who just want something that is stable. Version control systems, such as `git`, go a long way toward making this easier, but we developers still have to set (and follow) some rules in order for this to work. Here's the approach that I've been using on some of my own projects.

First, let's consider different types of branches that need to be maintained. The main distinction between different types of branch are whether they are released or not. There are

I think it is useful to speak in terms of invariants – what should be true whenever you look at these branches? This way the branches become a sort of contract with our users, and they know what to expect.

The `HEAD` of release branches, both support and stable, should always be released versions of the code. This means that they will have version numbers like 1.2 or 1.3.1. Internally, I use a flag `IS_RELEASED` to mark whether

As an example, let's say that I've just released code at 2.4.1, and I'm still maintaining one support branch (based on release 1.3.1). However, I'm also already developing

Branch	released	base version	branch name
support	True	release version (e.g., 1.3.1)	`1.x`
stable	True	most recent release (e.g., 2.4.1)	`stable`
development	False	next planned version (e.g., 2.5)	`master`
future	False	next major version (e.g., 3.0)	`3.0-dev`

Other people prefer to have the development branch called `dev` and the stable branch called `master`; my view is that most people who go work with a clone of the repository will want the cutting edge of development, so we put that in `master`. But both approaches are completely reasonable.

Note that it is still possible to add features to the `1.x` support branch and create a release 1.4. In fact, you could choose to have several support branches: instead of only one called `1.x`, you could have a `1.3` and a `1.4`. Personally, I'm only only likely to have one support branch at any time, although in theory you might have several.

So if this settles the invariants, the next question is, how do I do this in practice? How do I maintain these invariants?

This is where it becomes useful to have a step-by-step guide. The most common process just involves the stable and development branches, which in this case are `stable` and `master`. The process for doing a new release is:

1. Make a branch off of `master` for the new release. Set `IS_RELEASED = True`, and fix the version number anywhere else (e.g., docs).
2. Make a pull request against `stable`. This will include all the changes since the last release. Merge when ready.
3. Make a branch from `stable`, update to the next (dev) version and set `IS_RELEASED = False`. Make and merge a PR of that into `master`.

At this point, the repository is ready. The current state of `stable` can be made into a GitHub release, and that tag can be checked out to build and deploy. Note that, at all times, the HEAD of `stable` was a release (or about to made into a release) and the HEAD of `master` was never a release. This means that we maintained our invariant.

Now, what about the other two types of branches? The future branches and the support branches?

Future branches are easy: just regularly merge changes from the development branch into the future branch. Maintaining support branches is a little more complicated, but not too much.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search