contribution-guide.org

Release

January 01, 2017

Contents

CHAPTER 1

About

This document provides a set of best practices for open source contributions - bug reports, code submissions / pull requests, etc.

For the most part, these guidelines apply equally to *any* project regardless of programming language or topic. Where applicable, we outline where individual projects/languages may have additional requirements.

Naturally, this document is itself open source, and we encourage feedback & suggestions for improvement.

1.1 Sources

Currently this document draws from the contribution documentation for a handful of related Python open source projects: Fabric, Invoke, Paramiko, etc.

It's expected that over time we will incorporate additional material from related attempts at consolidating this type of info. We'll update with a list here when that happens.

CHAPTER 2

Submitting bugs

2.1 Due diligence

Before submitting a bug, please do the following:

- Perform **basic troubleshooting** steps:
  - **Make sure you're on the latest version.** If you're not on the most recent version, your problem may have been solved already! Upgrading is always the best first step.
  - **Try older versions.** If you're already *on* the latest release, try rolling back a few minor versions (e.g. if on 1.7, try 1.5 or 1.6) and see if the problem goes away. This will help the devs narrow down when the problem first arose in the commit log.
  - **Try switching up dependency versions.** If the software in question has dependencies (other libraries, etc) try upgrading/downgrading those as well.
- **Search the project's bug/issue tracker** to make sure it's not a known issue.
- If you don't find a pre-existing issue, consider **checking with the mailing list and/or IRC channel** in case the problem is non-bug-related.

2.2 What to put in your bug report

Make sure your report gets the attention it deserves: bug reports with missing information may be ignored or punted back to you, delaying a fix. The below constitutes a bare minimum; more info is almost always better:

- **What version of the core programming language interpreter/compiler are you using?** For example, if it's a Python project, are you using Python 2.7.3? Python 3.3.1? PyPy 2.0?
- **What operating system are you on?** Windows? (Vista? 7? 32-bit? 64-bit?) Mac OS X? (10.7.4? 10.9.0?) Linux? (Which distro? Which version of that distro? 32 or 64 bits?) Again, more detail is better.
- **Which version or versions of the software are you using?** Ideally, you followed the advice above and have ruled out (or verified that the problem exists in) a few different versions.
- **How can the developers recreate the bug on their end?** If possible, include a copy of your code, the command you used to invoke it, and the full output of your run (if applicable.)
  - A common tactic is to pare down your code until a simple (but still bug-causing) "base case" remains. Not only can this help you identify problems which aren't real bugs, but it means the developer can get to fixing the bug faster.

CHAPTER 3

Contributing changes

3.1 Version control branching

- Always **make a new branch** for your work, no matter how small. This makes it easy for others to take just that one set of changes from your repository, in case you have multiple unrelated changes floating around.
  - A corollary: **don't submit unrelated changes in the same branch/pull request**! The maintainer shouldn't have to reject your awesome bugfix because the feature you put in with it needs more review.
- **Base your new branch off of the appropriate branch** on the main repository:
  - **Bug fixes** should be based on the branch named after the **oldest supported release line** the bug affects.
    - E.g. if a feature was introduced in 1.1, the latest release line is 1.3, and a bug is found in that feature - make your branch based on 1.1. The maintainer will then forward-port it to 1.3 and master.
    - Bug fixes requiring large changes to the code or which have a chance of being otherwise disruptive, may need to base off of **master** instead. This is a judgement call – ask the devs!
  - **New features** should branch off of **the 'master' branch**.
    - Note that depending on how long it takes for the dev team to merge your patch, the copy of `master` you worked off of may get out of date! If you find yourself 'bumping' a pull request that's been sidelined for a while, **make sure you rebase or merge to latest master** to ensure a speedier resolution.

3.2 Code formatting

- **Follow the style you see used in the primary repository**! Consistency with the rest of the project always trumps other considerations. It doesn't matter if you have your own style or if the rest of the code breaks with the greater community - just follow along.
- Python projects usually follow the PEP-8 guidelines (though many have minor deviations depending on the lead maintainers' preferences.)

3.3 Documentation isn't optional

It's not! Patches without documentation will be returned to sender. By "documentation" we mean:

- **Docstrings** (for Python; or API-doc-friendly comments for other languages) must be created or updated for public API functions/methods/etc. (This step is optional for some bugfixes.)

  - Don't forget to include versionadded/versionchanged ReST directives at the bottom of any new or changed Python docstrings!
    * Use `versionadded` for truly new API members – new methods, functions, classes or modules.
    * Use `versionchanged` when adding/removing new function/method arguments, or whenever behavior changes.
- New features should ideally include updates to **prose documentation**, including useful example code snippets.
- All submissions should have a **changelog entry** crediting the contributor and/or any individuals instrumental in identifying the problem.

3.4 Tests aren't optional

Any bugfix that doesn't include a test proving the existence of the bug being fixed, may be suspect. Ditto for new features that can't prove they actually work.

We've found that test-first development really helps make features better architected and identifies potential edge cases earlier instead of later. Writing tests before the implementation is strongly encouraged.

3.5 Full example

Here's an example workflow for a project `theproject` hosted on Github, which is currently in version 1.3.x. Your username is `yourname` and you're submitting a basic bugfix. (This workflow only changes slightly if the project is hosted at Bitbucket, self-hosted, or etc.)

3.5.1 Preparing your Fork

1. Hit 'fork' on Github, creating e.g. `yourname/theproject`.
2. Clone your project: `git clone git@github.com:yourname/theproject`.
3. Create a branch: `cd theproject; git checkout -b foo-the-bars 1.3`.

3.5.2 Making your Changes

1. Add changelog entry crediting yourself.
2. Write tests expecting the correct/fixed functionality; make sure they fail.
3. Hack, hack, hack.
4. Run tests again, making sure they pass.
5. Commit your changes: `git commit -m "Foo the bars"`

3.5.3 Creating Pull Requests

1. Push your commit to get it back up to your fork: `git push origin HEAD`
2. Visit Github, click handy "Pull request" button that it will make upon noticing your new branch.
3. In the description field, write down issue number (if submitting code fixing an existing issue) or describe the issue + your fix (if submitting a wholly new bugfix).

4. Hit ‘submit’! And please be patient - the maintainers will get to you when they can.