contributing

Contributing guide for projects in the sphinx-toolbox organization.

Dominic Davis-Foster

Apr 25, 2024

Contents

Chapter

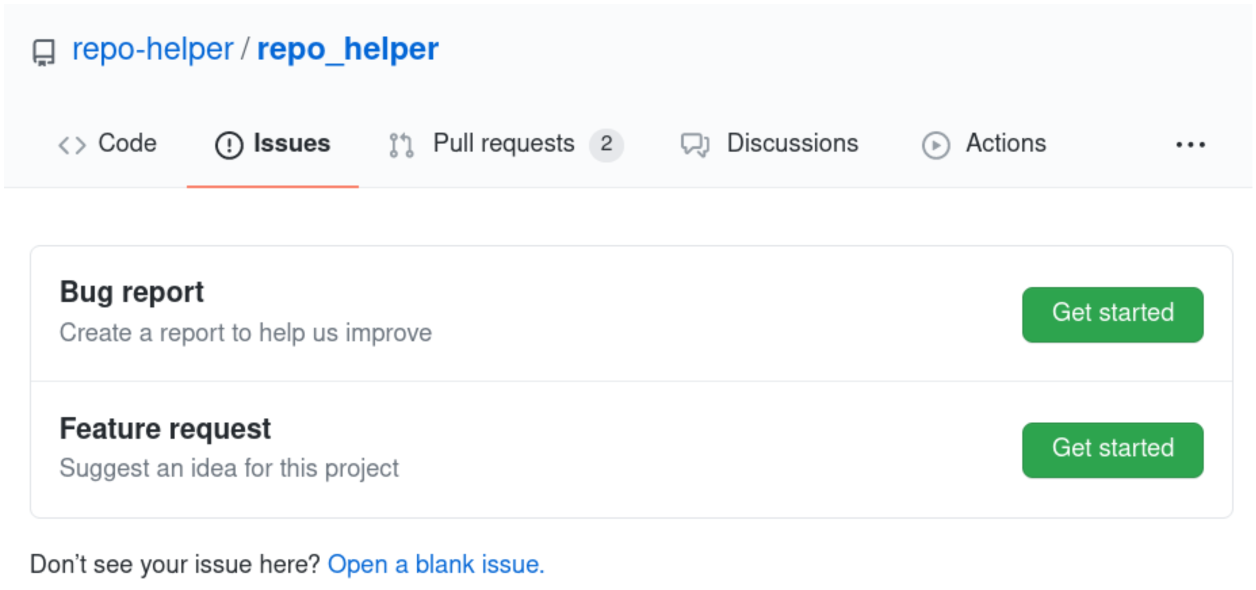
ONE

Getting Started

Each project's source code and issue tracker are hosted on GitHub. Before making your first contributing you will need a GitHub account. If you don't already have an account visit `https://github.com/join` .

1.1 Issues

Issues can be reported using the issue trackers on GitHub. Click the *Issues* tab at the top of the repository's page and then click the green *New* button. You'll be presented with a list of issue templates to choose from:



Bug report
Create a report to help us improve

Feature request
Suggest an idea for this project

Don't see your issue here? Open a blank issue.

Once you've chosen the template that best fits your issue you'll see a template like this:

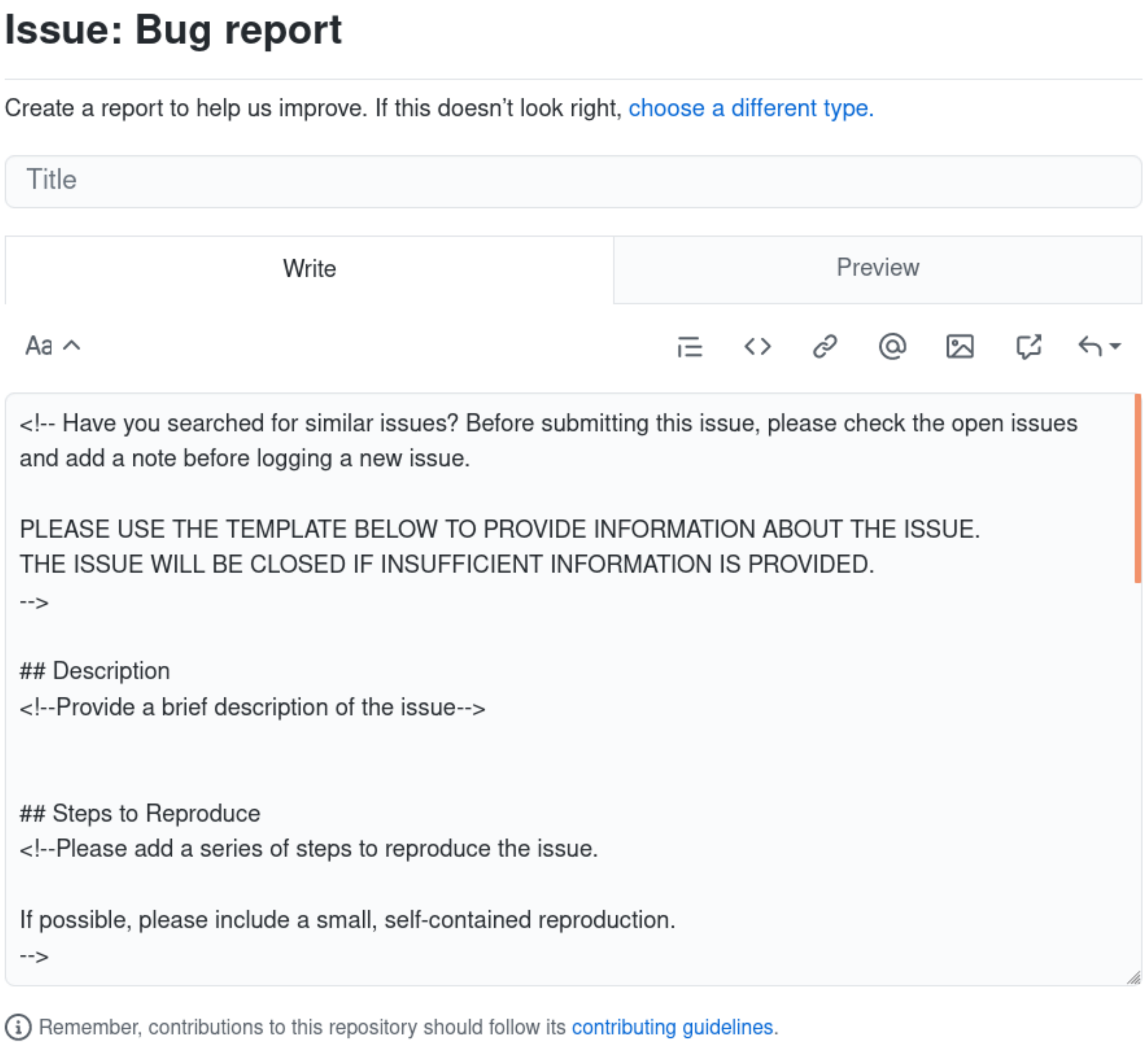
Feel free to remove any sections which you don't think are relevant.

1.2 Making Changes

Before making changes to the code you will need to fork the repository and clone your fork. GitHub have an excellent guide to this at `https://guides.github.com/activities/forking/`.

Once you have done this you'll need to set up your development environment. The recommended tool is pyproject-devenv, which will automatically install the project's runtime, build and test requirements:

```
$ cd <project_dir>
$ python3 -m pip install pyproject-devenv
$ pyproject-devenv
```


Issue: Bug report
Create a report to help us improve. If this doesn't look right, choose a different type.
Title
Write
Preview
Aa

```
<!-- Have you searched for similar issues? Before submitting this issue, please check the open issues
and add a note before logging a new issue.

PLEASE USE THE TEMPLATE BELOW TO PROVIDE INFORMATION ABOUT THE ISSUE.
THE ISSUE WILL BE CLOSED IF INSUFFICIENT INFORMATION IS PROVIDED.
-->

## Description
<!--Provide a brief description of the issue-->


## Steps to Reproduce
<!--Please add a series of steps to reproduce the issue.

If possible, please include a small, self-contained reproduction.
-->
```

Remember, contributions to this repository should follow its contributing guidelines.

This will create the virtualenv in the `venv` directory. Alternatively you can use *virtualenv* and install the dependencies with pip manually.

In either case, be sure to *activate the virtual environment*. With bash:

```
$ source venv/bin/ACTIVATE
```

```
(sphinx-toolbox) $
```

Style

formate is used for code formatting. It can be run manually via pre-commit:

```
$ pre-commit run formate -a
```

or, to run the complete autoformatting suite:

```
$ pre-commit run -a
```

The general formatting rules are based on **PEP 8**, but with some differences:

2.1 Indentation

Tabs are used instead of spaces.

2.2 Function signatures

Function signatures should be laid out like this:

```
def long_function_name(
		var_one: str,
		var_two: int,
		var_three: List[str],
		var_four: Dict[str, Any],
		) -> Iterable[Tuple[str, int]]:
	...
```

with each argument on its own line, and the function name and return annotatons on separate lines. Ensure to include the trailing comma after the last argument's annotation. If the signature is short enough it may be placed on one line:

```
def function_name(var_one: str, var_two: int) -> bool:
	...
```

Functions should always be type annotated. `*args` and `**kwargs` need not be annotated, but other use of `*` (e.g. `**children`) should be annotated. Annotations may be omitted in tests, but are strongly encouraged.

2.3 Type Annotations

String annotations should be avoided. Exceptions include `TypeVars` and where the object is imported using `with TYPE_CHECKING:` to resolve a circular dependency. **PEP 563** (`from __future__ import annotations`) must not be used.

Type hints should be valid with the earliest supported Python version for the project, typically Python 3.6. This includes the use of `typing.Dict[...]` etc. rather than `dict[...]`, and importing certain objects from typing-extensions.

2.4 Long `if` Statements

Long `if` statements should be formatted with `if (` on one line, the conditions on the following lines and the `):` on the final line. The operator should be at the start of the lines, not at the end.

Where possible such long statements should be avoided, for example by splitting it into multiple if statements or calculating the value in advance.

2.5 Long Sequences

Long sequences should be written with each element on its own line and a trailing comma after the last element:

```
# Bad
my_list = [
        1, 2, 3,
        4, 5, 6
        ]
```

```
# Good
my_list = [
        1,
        2,
        3,
        4,
        5,
        6,
        ]
```

2.6 Maximum Line Length

Limit all lines to a maximum of 110 characters. This also applies to docstrings, expect for long URLs in explicit hyperlink targets. Converting long, implicit targets to explicit targets improves the readability of the docstring. This also applies to reStructuredText files in the documentation.

If the summary line of a docstring must exceed 110 characters the line must be wrapped and the docstring marked with `# noqa: D400` immediately after the closing quotes.

2.7 Blank Lines

Blank lines must be used:

- After a file's top-level comments.
- Before a comment indicating a group of imports.
- After a group of imports.
- Immediately after a docstring.
- Immediately after a class, function or method signature if there is no docstring.

Blank lines must *not* be used:

- Between a class, function or method signature and its docstring.

2.8 Module Level Dunder Names

Module level "dunders" (i.e. names with two leading and two trailing underscores) such as `__all__`, `__author__`, `__version__`, etc. should be placed after the module docstring and any imports.

They should be in the following order:

- `__author__`
- `__copyright__`
- `__license__`
- `__version__`
- `__email__`
- <a blank line>
- `__all__`

`__all__` should always be included, even for modules with no public API. Only include the names of public objects. If the module has no public API write `__all__ = ()`. `TypeVars` may be included if it is necessary to document them, otherwise they should be omitted.

The flake8-dunder-all `pre-commit` hook can be used to automatically generate `__all__`, although its output should be checked by hand as it can sometimes produce odd results.

2.9 String Quotes

Strings should use double quotes where possible, except for single characters and empty strings which should use single quotes. Single quotes may be used where a double quote occurs within the string.

Triple double quotes (`"""`) must be used for docstrings.

2.10 Docstrings

The first line of the docstring must start a new line:

```
# Bad:
"""Return a foobang
"""
```

```
# Good:
"""
Return a foobang
"""
```

This rule is enforced even for short one-liner docstrings, although those should be rare.

Each docstring must document the function's parameters. For classes, the class docstring should document the arguments of `__init__` or `__new__` as appropriate.

`__init__` should not have a docstring. `__new__` may have a docstring in certain crcumstances, expecially for `namedtuples`.

2.11 Naming Style

Function and method names should be written in `lower_case_with_underscores`.

Class names should use `CamelCase`. When using acronyms, capitalize all the letters of the acronym. `HTTPServerError` is better than `HttpServerError`.

Private classes, functions and instance variables should be prefixed with a single underscore.

Chapter

THREE

Testing

tox is used to automate testing, with the tests themselves run with pytest.

Start by installing `tox` and tox-envlist if you don't already have them:

```
$ python3 -m pip install tox tox-envlist
```

To run tests for a specific Python version, such as Python 3.6:

```
$ tox -e py36
```

or, to test all Python versions:

```
$ tox -n test
```

3.1 Type Annotations

Type annotations are checked using mypy:

```
$ tox -e mypy
```

3.2 Code Quality

Run flake8 with `tox` to ensure your changes don't introduce any issues:

```
$ tox -e flake8
```

3.3 Coverage

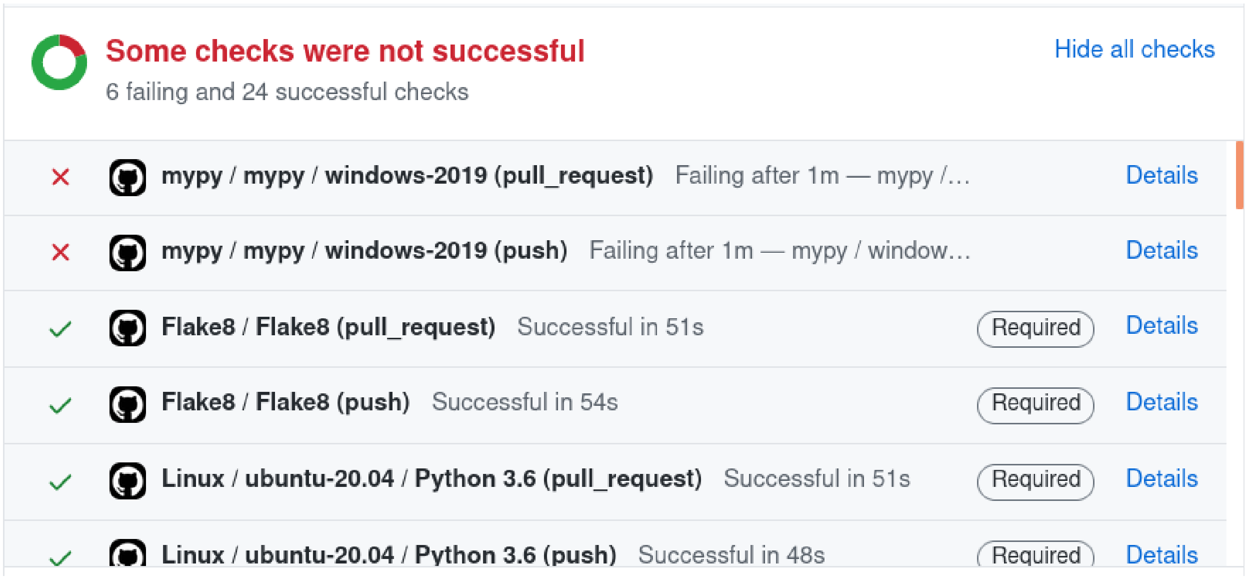
On POSIX systems with Firefox installed an HTML coverage report can be generated by running:

```
$ tox -n cov
```

Ensure your changes do not cause a significant decrease in the test coverage. If the coverage drops below the level set in `repo_helper.yml` (or 80% not specified) the tests will fail.

3.4 GitHub Actions

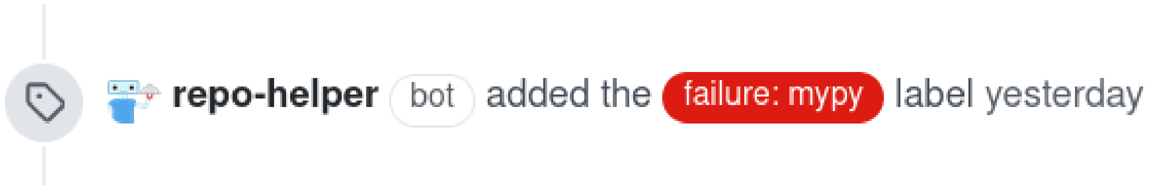
Tests are run on pushes to GitHub using GitHub Actions. You can see the results of these test at the bottom of the pull request page:



Some checks were not successful
6 failing and 24 successful checks
Hide all checks

✗ mypy / mypy / windows-2019 (pull_request) Failing after 1m — mypy /… Details
✗ mypy / mypy / windows-2019 (push) Failing after 1m — mypy / window… Details
✓ Flake8 / Flake8 (pull_request) Successful in 51s Required Details
✓ Flake8 / Flake8 (push) Successful in 54s Required Details
✓ Linux / ubuntu-20.04 / Python 3.6 (pull_request) Successful in 51s Required Details
✓ Linux / ubuntu-20.04 / Python 3.6 (push) Successful in 48s Required Details

A label will be added to the pull request automatically if the tests fail:



repo-helper bot added the failure: mypy label yesterday

This makes it easy to identify which tests are failing. Once the tests pass the label will be removed automatically.

If you are a first time contributor to a project manual approval is required for GitHub Actions to run. For more information see `https://github.blog/2021-04-22-github-actions-update-helping-maintainers-combat-bad-actors/`

Branch protection is used to ensure the following tests pass before merging pull requests:

- Tests on Windows and Linux for all CPython version between 3.6 and 3.9 supported by the project.
- Tests on Windows and Linux for PyPy 3.6, if supported by the project.
- mypy type checking on Windows and Linux.
- Flake8
- The documentation check, if the project has documentation.

If the project only supports Linux the tests on Windows will not run and are not required to merge the pull request.

Tests on macOS are optional as they take longer than other platforms. CPython 3.10 and PyPy 3.7 are considered experimental and will not block a pull request from being merged if they fail. However, you should still check the results of these runs to ensure your changes have not introduced any errors there.

You should check the *Files changed* tab of the pull request to see whether any issues have been identified. This can be due to syntax errors in the documentation source or issues identified by flake8 and Codefactor.

Your pull request may be commented on by Coveralls to report any changes to the code coverage.

Chapter

FOUR

Documentation

Most projects have documentation generated with Sphinx and hosted on ReadTheDocs.

The documentation source is located in the `doc-source` directory. A local copy of the documentation can be built with `tox`:

```
$ tox -e docs
```

The built documentation can be found in the `doc-soure/build/html` directory.

Chapter

FIVE

Building from source

The recommended way to build from source is via tox:

```
$ tox -e build
```

As well as building the sdist and wheel this will check them for common errors, such as missing files or potential rendering issues on PyPI.

The output files will be in the `dist` directory.

If you wish, you may also use pep517.build or another **PEP 517**-compatible build tool.

Chapter

SIX

Release Process

New releases are created using a combination of repo-helper and GitHub Actions. To start you will need `repo-helper` installed:

```
$ python3 -m pip install repo-helper --upgrade
```

Then ensure all changes have been committed or stashed:

```
$ git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

The create a release using repo-helper release:

```
$ repo-helper release minor
```

Repleace `minor` with `major` or `patch` as appropriate.

Once the release has been created locally, push to GitHub. Ensure you push the tag too:

```
$ git push
$ git push --tags
```

Once the tests pass GitHub Actions will take care of building and uploading to PyPI and Anaconda (if enabled for the project). You should keep an eye on the tests to ensure they pass.

The release will be automatcally copied to GitHub Releases within the next two days using OctoCheese.

Index