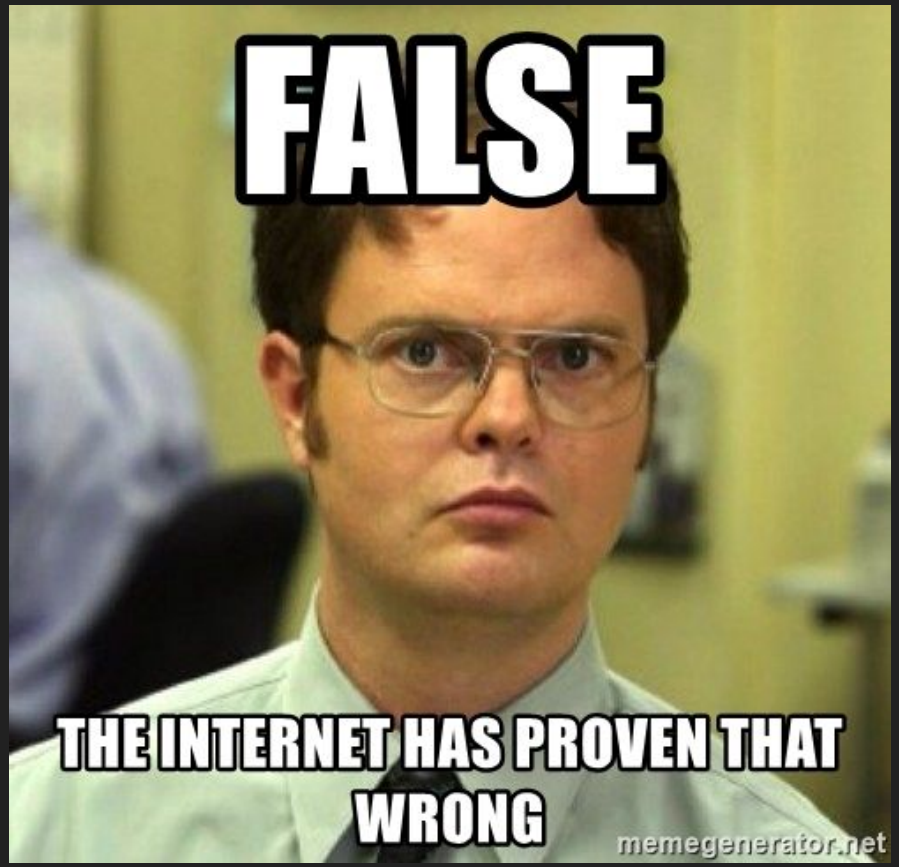
Releasing Software

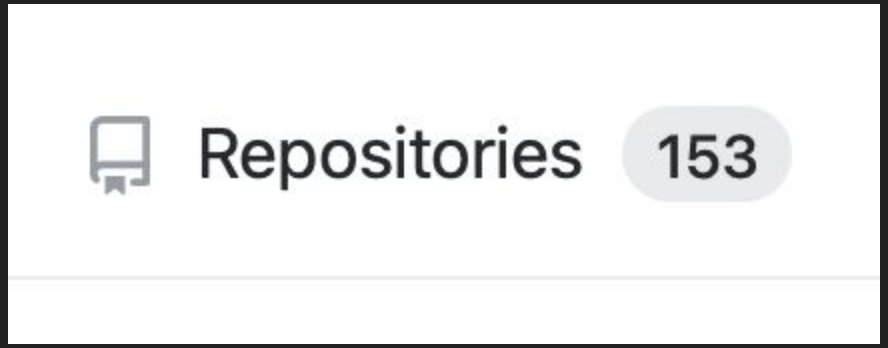
George Lesica - Wheeler Lab

Software is meant to be used by people to do things.


FALSE
THE INTERNET HAS PROVEN THAT WRONG
memegenerator.net

Seriously, though.

Unfortunately, releasing software is hard, and boring.


Repositories
153

But it’s fun when people use our software!

Challenges

Or, what's so hard about this?

Discovery

Installation

Usage

Development

Discovery

We put our software where our users are.


Our software...

Methods and Tools

- Publishing
  - Publish tools even without research
  - Discuss tools in academic and non-academic publications
- GitHub
  - Repo descriptions and topics
  - Explanatory READMEs
- Evangelize
  - Look for use cases
  - Seek out collaborations

But just who are these “user” people?

Think in archetypes.

Archetypes

A microbiologist who writes code, but is self-taught and unfamiliar with modern programming languages like Rust.

A software engineer working in bioinformatics who has spent significant time writing Python code, but primarily in a non-academic setting.

A time-constrained CS graduate student who is conversant in basic genetics, but lacks a deep understanding of microbiology.

A systems administrator who is asked to install and maintain software for a cluster, but knows little about biology and almost never writes code beyond shell scripts.

Decide who to focus on, and provide them with reasonable solutions.

Installation

Installation should be easy, but also reliable.

Different Software: Different Principles

Libraries

- Integrate into package ecosystems
  - Don’t do your own thing, even if the dominant thing is terrible
  - Decide what to support and do it well
- Use semantic versions
  - Automated dependency resolution tools are a thing, play nice with them
  - Release often, don’t let bug fixes wallow on the main branch, unreleased

Applications

- Integrate into package ecosystems to the extent that it serves users
  - No need for a Mac package if the software is intended for clusters
  - No need for an .deb or .rpm package if the software is targeted at technical people
  - Consider static linking to split the difference
- Clearly enumerate dependencies and supported versions
- Provide release notes

Specific Recommendations

Python

Libraries

- Publish to PyPI
  - Flit (preferred)
  - Poetry
- Considering publishing to Conda

Applications

- Minimize third-party dependencies (but don’t go overboard)
- Consider publishing to PyPI
- Consider publishing to Conda
- Consider publish Docker images
- Linux packages (.deb and .rpm) are a lot of work

Rust

Libraries

- Publish a Crate
- Go easy on the third-party dependencies

Applications

- Consider static linking
- Consider publishing a Crate
- Consider publishing to Conda

C / C++

Libraries

- Apparently, you just want to watch the world burn... nice
- Use CMake unless you can’t for some reason
- Use Autotools if you can’t use CMake
- Remember that the GCC on your machine isn’t the only compiler in existence

Applications

- Minimize build dependencies
- Use CMake
- Consider publishing to Conda
- Consider publishing a Docker image
- Remember that the GCC on your machine isn’t the only compiler in existence

Don’t try to do everything for everyone, let the user help!

Usage

Using your software should be easy for your target users, and possible for everyone else.

High-level Suggestions

Libraries

- Provide API documentation
- Provide meaningful examples
- Take advantage of ecosystem tools
- API documentation should cover “what” and “why”, but NOT “how”
  - No one cares (let’s be honest)
  - If they do care, they can read the code
- Design APIs to be consistent
  - Naming
  - Parameter order
  - Types

Applications

- Always provide “--help”
- Man pages are great, but these days the web is probably better
- Provide meaningful examples
- Thoroughly describe input and output formats
  - For standard formats like FASTA, explain what the file should (or will) contain

Specific Recommendations

Python

Libraries

- Provide docstrings
- Use Read The Docs

Applications

- Use Read The Docs
- Provide examples in your documentation

Rust

Libraries

- Provide doc comments
- Use rustdoc and docs.rs

Applications

- Use github.io to host a simple web site with explanation and examples

C / C++

Libraries

- Provide doc comments
- Use Doxygen or something similar for API docs
- Consider Read The Docs for hosting
- Use github.io otherwise

Applications

- Use github.io to host a simple web site with explanation and examples

Development

At the end of the day, a developer is just a particular kind of user.

General Recommendations

Provide an automated test suite - not every developer will be familiar with every part of your code, test suites provide confidence and peace of mind.

API documentation and code comments can provide important context for new developers.

Use a formatter and linter to enforce code style.

Use good variable, function, and class names.

Set up continuous integration to run your quality checks (tests, linter, formatter).

API Documentation

Bad

```
# Calculate the score for a row.
def calculate_score(row, target):
```

```
# A single sequence.
class Sequence:
```

Good

```
# Calculate the score for the given
# row based on the CrossMatch score
# algorithm.
def calculate_score(row, target)
```

```
# A single sequence that uses a
# particular alphabet and encodes
# blank (unknown) nucleotides.
class Sequence:
```

Code Comments

Bad

```
# Iterate through positions and add
# up all values greater than q.
for i in positions:
  if i > q:
    total += i


# Reset the counter
row_index = 0
```

Good

```
# Sum positions that are above the
# critical threshold (Smith, 2019).
for i in positions:
  if i > q:
    total += i


# Start over from the top of the
# column
row_index = 0
```

Style Considerations

- Names
  - `calculate_offset`
  - `calc_offset`
  - `calculateOffset`
  - `calcOffset`
- Order
  - `find_match(values, regex)`
  - `find_match(regex, values)`
  - `values.find_match(regex)`
  - `regex.find_match(values)`
- Control Flow
  - `for d in data:`
  - `for i in range(len(data)):`
  - `[f(d) for d in data]`
  - `map(f, data)`
- Paradigm
  - `Matrix.apply(f)`
  - `apply(M, f)`
  - `sorted(values)`
  - `values.sort()`

Names

Bad

```
value = sequence_scores[i]

for i in range(len(scores)):

file = open(...)

columns = len(matrix)

def process_data(...)
```

Good

```
score = sequence_scores[i]

for score_index in range(len(scores)):

sequence_file = open(...)

column_count = len(matrix)

# literally anything else
```

Specific Recommendations

Python

- Tests - pytest
- Formatter - black
- Linter - flake8 or pylint
- Type checking - mypy

Rust

The compiler handles most checks and the Cargo tool includes a test runner and code formatter.

C / C++

- Tests - ctest (built into CMake)
- Formatter - clang-format
- Linter - clang-tidy

Wrapping Up

Releasing software is a balancing act.

Doing it well is worthwhile for us, users, and developers.

Questions?