DaSL Contributor Guide

Scott Chamberlain

Table of contents

Welcome

> **Warning**
>
> This guide is under construction - content changes often.

This book is a resource for the community of people using, contributing to, and maintaining software at the Fred Hutch Cancer Center Data Science Lab (DaSL).

The book covers important aspects of software development, including how to get involved in software as a user or contributor, code style, code review, package documentation, and more. It includes both internal facing guidelines as well as for any contributions from folks other than DaSL staff.

- Contributing: Chapter 1
- Style: Chapter 2
- Code review: Chapter 3
- Package documentation: Chapter 4
- Package maintenance: Chapter 5
- Security: Chapter 6

DaSL primarily develops software in the R and Python programming languages. Where necessary, we'll discuss a topic with respect to both languages.

Inspiration

Inspiration for this guide is taken in part from:

- Tidyverse Style Guide
- rOpenSci Dev Guide

1 Contributing

This chapter is intended to direct you to the place you want to be, whether you want to get help, use software, contribute to software, and more. Reminder: the scope of this guide is software maintained by DaSL in the WILDS GitHub organization. If you'd like help with your own software schedule a house call.

1.1 Communication Channels

- Slack: Fred Hutch folks can chat with other community members and DaSL and Fred Hutch SciComp staff in the Fred Hutch Data Slack. Join our Slack here https://hutchdatascience.org/joinslack/
- GitHub Issues: Each GitHub repository has an issues tab where you can ask questions, propose a feature, and more. See below for more details.
- Are there others?

1.2 Software

This section concerns any software created in the GitHub repositories under the WILDS GitHub organization.

Find the thing you would like to do below and follow its instructions.

Question: If you have a question you can ask in the Slack linked above, or open an issue in any of the GitHub repositories.

Bug: If you want to report a bug, open an issue in the appropriate GitHub repository.

Feature: If you want to request a feature, open an issue in the appropriate GitHub repository.

Contribute code/docs: If you want to contribute to software - whether code, documentation or something else - open an issue in the appropriate GitHub repository to discuss, then open a pull request to make your contribution.

Code of Conduct: DaSL GitHub repositories should have their own code of conduct - likely some version of the Contributor Covenant. Refer to the COC in the repository for specific guidance.

2 Style

What is code style? It has to do with how you organize your code. The end goal of styling your code is that your code is more consistent. Styling code by definition has to be opinionated. You may not agree with every decision made in a style guide, but having style be done automatically for you gives you more time to think about the important decisions in your code.

If you always write code by yourself and you're the only one that will ever look at it, that's one thing. But that's rarely the case, especially if you consider that yourself 6 months or 5 years from now is a user that is keenly interested in readable, consistent code. Readable, consistent code will be especially appreciated by other users and contributors.

2.1 R

2.1.1 Package styler

The styler package allows you to interactively format your code according to the tidyverse style guide.

The lintr package does automated checks of your code according to the style guide.

2.1.2 IDE and Text editor support

RStudio supports styler via the Addins drop down; see the RStudio User Guide.

Support for `styler` in other editors is provided via the R languageserver:

- VSCode: vscode-R
- Atom: atom-ide-r
- Sublime Text: R-IDE
- Vim/NeoVim: LanguageClient-neovim

See the R languageserver GitHub repository for more information on using the R language-server.

2.1.3 Command line/Terminal/R

In the getwilds/makefiles repo we have an R package Makefile template with three make targets for styling package code: `lint_package`, `style_file`, and `style_package`. With that Makefile in the root of your package you can run `lint_package` to check for any problems, and run `style_file` or `style_package` to fix any problems. You can also just run the R code in those make targets in a terminal or within R.

2.1.4 GitHub Actions

To get setup with GitHub Actions and `lintr` and `styler`, first install `lintr` if you don't have it:

```
if (!requireNamespace("pak", quietly=TRUE)) {
  install.packages("pak")
}
pak::pkg_install("lintr")
```

And then run:

```
lintr::use_lintr()
```

To create a configuration file for `lintr`. In the file created (`.lintr`) you can set custom congifuration as needed for your package (see the lintr vignette).

Next, run:

```
usethis::use_github_action("lint")
```

which creates a file `.github/workflows/lint.yaml` in your package to run `lintr` checks on your package. This action only reports problems and does not fix them for you. If you want to have any problems fixed automatically and committed to your main branch, run:

```
usethis::use_github_action("style")
```

Note that the above action will create commits all with the same message: `"Style code (GHA)"`.

We think in most cases it makes sense to only use the `lint` action and not he `style` action, but you're free to use either or both.

2.2 Python

2.2.1 Package Ruff

There are a number of different options for styling/formatting Python code, including Black, Flake8, isort, Ruff, and more. We recommend using `Ruff` as it's extremely fast and encompasses all the things that the other tools do, and more.

2.2.2 IDE and Text editor support

- VSCode: Ruff VS Code Extension
- Sublime Text: via ruff-lsp
- Vim/NeoVim: via ruff-lsp

Ruff supports the Language Server Protocol via the ruff-lsp.

2.2.3 Command line/Terminal

In the getwilds/makefiles repo we have a Python package Makefile template with two make targets for styling package code: `lint-fix` and `lint-check`. With that Makefile in the root of your package you can run `lint-check` to check for any problems, and `lint-fix` to fix any problems. You can also just run the command line tools in a terminal (e.g., `ruff check .`).

2.2.4 GitHub Actions

There's a few different options for `Ruff` for GitHub Actions. See Ruff docs for details.

3 Code review

Code review ….

3.1 DaSL interal guidelines

still discussing this internally …

3.2 Community guidelines

We have code review guidance for labs at https://hutchdatascience.org/code_review/. The site contains higher level code review guidance broken down by lab roles, including lab leader, lab manager and lab developer.

If you want lower level code review guidance the Advanced Reproducibility in Cancer Informatics course has two chapters that will be helpful:

- Engaging in Code Review - as an author
- Engaging in Code Review - as a reviewer

4 Documentation

All formally supported DaSL R and Python packages should have package documentation.

4.1 R

We use the pkgdown package to create documentation for R packages, and host it on GitHub Pages. To get started with `pkgdown`, in R within the root of your package run usethis::use_pkgdown_github_pages() - it will set up a `_pkgdown.yml` file in the root of your repo used to configure `pkgdown`, and add a `.github/workflows/publish.yml` file to build the package documentation on each push, pull request, or release. See pkgdown documentation for details on configuring documentation.

After pushing the above changes up to your repository, go to the Actions tab and you should see the new `publish` action running. It will build and then deploy the rendered package docs. The URL for your docs will vary depending on the GitHub organization your repository lives within. For WILDS, the base URL for now is `https://getwilds.github.io`. If your pacakge was in the WILDS org (at `https://github.com/getwilds`) your package (named `mypkg`) docs would live at `https://getwilds.github.io/mypkg`.

You can also build `pkgdown` docs locally - after running `usethis::use_pkgdown()` or `usethis::use_pkgdown_github_pages()` - by running `pkgdown::build_site()`. If you run `build_site()` within RStudio it should open up your site in your default browser, but may not do so if you run in a terminal. You can open the site in your browser by navigating to and opening the file `docs/index.html` within your repo.

4.1.1 DaSL pkgdown template

We are planning to have a DaSL specific `pkgdown` "package template" (see pkgdown docs for what this means) - but it's not ready to use yet. When it is ready, you will be able to specify our template like:

```
template:
  package: dasltemplate
```

For now just use the default theme that `pkgdown` provides.

4.2 Python

Just like there's a variety of ways to do packing in Python there's a variety of documentation options. Two of the well known options are:

- Sphinx - been around longer, uses reStructuredText
- MkDocs - newcomer, used Markdown

For either of the options above, they can be hosted in many places, including GitHub Pages and ReadTheDocs.

`Sphinx` and `MkDocs` are less automatic relative to `pkgdown`, so just be prepared for a bit more manual work.

4.3 Guidelines

- README: This is most likely the first place potential users will interact with your package. Make sure the README clearly states what the package does, and how to get started.
- Examples: All user facing functions should have examples. Make sure to be careful about how examples are run if there's any sensitive data or connections to remote services.
- (anything else?)

5 Package maintenance

5.1 Package versioning

There is a detailed discussion of versioning R packages in the lifecycle chapter of the R Packages book by Hadley Wickham and Jenny Bryan. Please follow that chapter in general for versioning of R and Python packages within DaSL. To make it easier to grok, below are some of the highlights, and some exceptions to that chapter.

5.1.1 Package version numbers

There's quite a bit of nuance - and surprises - to package version numbers - see the Package version number section for details. For example, using the `utils::package_version()` function, which parses package version strings into S3 classes, we get a suprising result:

```
"2.0" > "10.0"
#> [1] TRUE
package_version("2.0") > package_version("10.0")
#> [1] FALSE
```

With that example, please do think about your package versions before setting them.

5.1.1.1 DaSL Conventions

Following the Tidyverse package version conventions, DaSL packages will use the following conventions (see the link for more details):

- Always use `.` as the separator, never use `-`.
- A released version number consists of three components, `<major>.<minor>.<patch>`
- While a package is in between releases, there is a fourth component: the development version, starting at 9000 (e.g., `0.2.2.9000`), and incrementing from there until the package has another release at which point return to three components.

5.1.2 Ignore these

We are not following or enforcing any rules about changes at the function/class/etc level below the package level. For example, the R Packages book talks about using the `lifecycle` package to deal with function changes.

5.2 Package releases

In general follow the Releasing to CRAN chapter in the book R Packages for R, and the Releasing and versioning chapter in the book Python Packages for Python. Those chapters don't have to be followed to the letter, but in general they provide really good guidance.

There are a few aspects of releases we are opinionated about and would like DaSL R and Python packages to follow:

- Follow our versioning guidelines above
- Git tag released versions, and push the tag to GitHub
- Add the associated NEWS/Changelog items to a release associated with the tag on GitHub

6 Security

xxx