openstatsguide's Checklist for good statistical software packages

Developers Value Tests For Software Longevity

Alessandro Gasparini, Daniel Sabanés Bové, Nils Penard, Audrey Yeo Te-ying

Scope

We encourage developers of statistical software packages to follow this minimum set of good practices around:

> "**D**ocumentation, **V**ignettes, **T**ests, **F**unctions, **S**tyle, **L**ife cycle"

These keywords can be easily remembered with the mnemonic bridge sentence:

> "**D**evelopers **V**alue **T**ests **F**or **S**oftware **L**ongevity"

While the recommendations are rather generic, we focus on functional programming languages and give links to implementations in R, Python and Julia.

This guide primarily addresses developers of statistical packages. Users interested in assessing the quality of existing statistical packages will find complementary "validation" focused resources valuable, as listed in References.

Documentation

… is important for both users and developers to understand all objects in your package, without reading and interpreting the underlying source code.

1. Use in-line comments to generate their corresponding documentation. {roxygen2}
2. Do also **document internal functions and classes**.
3. Add **code comments for ambiguous or complex pieces of internal code**.

Functions

… should be short, simple and enforce argument types with assertions.

1. **Write short functions** for a single and well defined purpose, with few arguments.
2. Use type hints declaring which arguments expect which type of input. {roxytypes}
3. **Enforce types and other expected properties** of function arguments with assertions. {`checkmate`}

Vignettes

… provide a comprehensive and long-form overview of your package's functionality from a user point of view.

1. **Provide an introduction vignette** that opens up the package to new users.
2. Include deep dive vignettes, including describing statistical methodology.
3. Host your vignettes on a dedicated website. {`pkgdown`}

Style

… should be language idiomatic and enforced by style checks.

1. Use language idiomatic code and **follow the "clean code" rules**.
2. Use a formatting tool to automate code formatting. {`styler`}
3. **Use a style checking tool** to enforce a consistent and readable code style. {`lintr`}

Test

… are a fundamental safety net and development tool to ensure that your package works as expected, as you develop and use it.

1. **Write unit tests** for all functions/ classes in your package ("white box" testing).
2. **Write functional tests** for your user API ("black box" testing). {testthat}
3. In addition, **ensure good coverage** of your code with your tests as a final check. {covr}

Lifecycle

… management is crucial to build up your user base in a sustainable way. Life cycle management is simplified by reducing dependencies, and should comprise a central code repository.

1. Reduce dependencies to simplify maintenance of your own package and depend on other packages that you trust and deem stable enough.
2. **Track dependencies and pin their versions** to produce consistent results and behaviours by using more system level that serves as a source of truth for all packages developers.
3. Maintain the change log and deprecate functionality before deleting it.
4. Use a **central repository** for version control and publication of a **contributing guide** to enable community contributions.