Setting up R development environment on github

Ronald (Ryy) Glenn Thomas

2024-12-08

Table of contents

1 Introduction

Figure 1: purrr

Its often the case that a data scientist needs to share an R function with a co-worker or a students. The post describes a step by step methodology for wrapping the function in a package and sharing it either on github or CRAN.

By way of overview we want to create a directory (or repository) that contains the package contents. The base elements of the package are the DESCRIPTION file, the NAMESPACE file, the R directory, the tests directory, and the man directory. Other files such as the README.md, the LICENSE file, and the .gitignore file are optional but recommended.

Start by using the various tools in the `usethis` package to streamline the process.

Start with `usethis::create_package("~/dev/my_package")` to create the package directory and the DESCRIPTION and NAMESPACE files. Assuming the package will be named `my_package` and your development directory is `~/dev`.

```
install.packages("devtools")
install.packages("usethis")
library(usethis)
library(devtools)
usethis::create_package("~/dev/my_package")
```

This creates the following directory structure.

```
julia dev/my_package   tree --charset=ascii
.
|-- DESCRIPTION
|-- NAMESPACE
`-- R

julia dev/my_package   more DESCRIPTION
Package: my_package
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person("First", "Last", , "first.last@example.com", role = c("aut", "cre"),
           comment = c(ORCID = "YOUR-ORCID-ID"))
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to pick a
    license
Encoding: UTF-8
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.3.2
```

Next copy the R file containing the function to the `R` directory and add a `#'` roxygen comment block to the top of the file. Then call `devtools::document()` to generate the `man` directory containing the help page.

```
devtools::document()
```

At this point the directory structure looks like this.

```
julia dev/my_package   tree --charset=ascii
.
|-- DESCRIPTION   # package Metadata
|-- NAMESPACE     # Exports and imports declarations
|-- R             # R functions
|   `-- my_package.R
|-- man           # Documentation for the functions
    `-- my_package.Rd
```

The next step is to set up testing.

```
usethis::use_testthat()
```

```
call inside R
usethis::use_test("my_package")
```

This open an editor. Enter the unit tests using the test_that function.

```
# Test: Empty dataframe error
test_that("t2f throws an error for empty dataframe", {
  empty_df <- data.frame()
  expect_error(my_package(empty_df, filename = "empty_table"), "`df` must not be empty")
})
```

Set up a new repository on github.

```
git init
git add .
git commit -m "Initial commit"
```

```
usethis::use_github()
usethis::use_gpl3_license("R.G.Thomas")
usethis::use_readme_md()
usethis::use_code_of_conduct("rgthomas@ucsd.edu")
usethis::use_tidy_contributing()
```

Add each dependency (e.g. `kableExtra`) (e.g. kableExtra) `(e.g. kableExtra)`

```
usethis::use_package("kableExtra", type = "Imports")
```

Finally do a full check using `devtools::check()`. This reflects the checks that CRAN will perform when you submit the package.

```
devtools::build()
devtools::install()
devtools::test()
devtools::check()
```

2 Debugging workflow

```
git checkout -b fix-bug
```

    Debug locally and isolate the issue.

- `Create a local branch (fix-bug) for the fix.`
- git checkout -b fix-bug
- `Make and test the changes.`
- Run devtools::test() to confirm all tests pass.
- Use devtools::check() to validate the package.
- git add .
- git commit -m "Fix issue with my_package function"
- git push
- Merge the branch into the main branch and clean up.
- git checkout main
- git merge fix-bug
- git branch -d fix-bug

- git push
- `Open a Pull Request.`
- `Update the version number`
- usethis::use_version("patch")
- and push the final changes.
- git push

3 Debugging Workflow from chatGPT

Follow these steps to debug and fix issues in your R package:

1. **Debug Locally**
   - Isolate the issue using R debugging tools like `browser()`, `traceback()`, or `debug()`.
2. **Create a Local Git Branch**
   - Create a branch for the fix to isolate your changes:

   ```
   git checkout -b fix-bug
   ```
3. **Make and Test Changes**
   - Modify your code to fix the issue and add or update unit tests as needed.
   - Run tests to confirm functionality:

   ```
   devtools::test()  # Confirm all tests pass
   devtools::check() # Validate the package complies with CRAN standards
   ```
4. **Commit Your Changes**
   - Stage and commit your changes:

   ```
   git add .
   git commit -m "Fix issue with my_package function"
   ```
5. **Push the Branch**

- Push the branch to GitHub for collaboration or to prepare for merging:

```
git push origin fix-bug
```

6. **Open a Pull Request**

   - Open a Pull Request (PR) on GitHub to merge the fix into the main branch. Include a clear description of the changes.

7. **Merge and Clean Up**

   - After review and approval, merge the branch into the main branch:

```
git checkout main
git merge fix-bug
```

   - Delete the branch locally and remotely:

```
git branch -d fix-bug
git push origin --delete fix-bug
```

8. **Test the Main Branch**

   - Ensure the main branch passes all tests:

```
devtools::test()  # Confirm functionality
devtools::check() # Validate compliance
```

9. **Update the Version Number**

   - Increment the package version using `usethis::use_version()`:

```
usethis::use_version("patch")  # Use "patch", "minor", or "major"
```

   - Commit and push the version update:

```
git add DESCRIPTION
git commit -m "Bump version to 1.0.1"
git push
```