# Setting up an R development environment on github

RG Thomas

2025-01-02

## Table of contents

## 0.1 Introduction

Figure 1: purrr

Its often the case that a data scientist needs to share an R function with a co-worker or a student. This post describes a step by step methodology for wrapping the function in a package, that includes a number of support files, and sharing it either via github or CRAN. This may seem like overkill but over time a not having to address the many technical issues that can arise when sharing a function in a more ad hoc manner may be appreciated.

## 0.2 The Problem/Data

The end goal of this terminal is to create a directory (or repository) that contains the package contents. The top level 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 a `README.md`, a `LICENSE` file are optional but recommended. The `DESCRIPTION` file contains metadata about the package such as the package name, the version number, the author, and the license. The `NAMESPACE` file contains the export and import declarations. The `R` directory contains the R functions. The `tests` directory contains the unit tests.

### 0.2.1 Step 1: Initial Repo Setup

Start by using the various helpful tools in the `devtools` and `usethis` packages to facilitate the repository building process.

Open `R` from the shell prompt in your development directory. and run the command `usethis::create_package("my_package")` to create the package directory and the `DESCRIPTION` and `NAMESPACE` files. Assuming the package will be named `my_package`.

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

This creates the following directory structure.

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

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 use the `usethis` package tools to generate repository support files.

```
usethis::use_git()
usethis::use_github()
use_gpl_license(version = 3, include_future = TRUE)
usethis::use_readme_md()
usethis::use_code_of_conduct("rgthomas@ucsd.edu")
usethis::use_tidy_contributing()
```

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"
```

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()
```

# 1 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

## 2 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
```

## 2.1 Reproducibility

```
# Print session info for reproducibility
sessionInfo()
```

```
R version 4.4.2 (2024-10-31)
Platform: aarch64-apple-darwin20
Running under: macOS Sequoia 15.2

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LA

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices datasets  utils     methods   base

other attached packages:
 [1] here_1.0.1         shiny_1.9.1        styler_1.10.3      quarto_1.4.4       pacman_0.5.1
[11] dplyr_1.1.4        purrr_1.0.2        readr_2.1.5        tidyr_1.3.1        tibble_3.2.1
[21] janitor_2.2.0      datapasta_3.1.0    ggthemes_5.1.0     conflicted_1.2.0   DT_0.33

loaded via a namespace (and not attached):
 [1] tidyselect_1.2.1  viridisLite_0.4.2 R.utils_2.12.3    fastmap_1.2.0     promises_1.3.2
[11] processx_3.8.4    magrittr_2.0.3    compiler_4.4.2     rlang_1.1.4       tools_4.4.2
[21] miniUI_0.1.1.1    R.cache_0.16.0    withr_3.0.2        R.oo_1.27.0       grid_4.4.2
[31] cli_3.6.3         generics_0.1.3    remotes_2.5.0      rstudioapi_0.17.1 tzdb_0.4.0
[41] hms_1.1.3         visdat_0.6.0      systemfonts_1.1.0  glue_1.8.0        ps_1.8.1
[51] htmltools_0.5.8.1 R6_2.5.1          rprojroot_2.0.4    evaluate_1.0.1    R.methodsS3_1.8.2
[61] xfun_0.49         fs_1.6.5          pkgconfig_2.0.3
```

## 2.2 Next Steps

- Suggest areas for further exploration

- Mention potential improvements
- Invite reader engagement

### 2.3 References

- Cite your sources
- Link to relevant documentation
- Credit other contributors

---

## 3 Debugging Errors with `devtools::test()`

This guide provides a step-by-step process to debug errors occurring during `devtools::test()` for R packages.

---

### 3.1 Step 1: Understand the `devtools::test()` Workflow

1. **What It Does**:
    - Runs all test scripts in the `tests/testthat/` directory using the **testthat** package.
    - Reports errors, warnings, and failed assertions.
2. **Output Format**:
    - Lists the file and test name where the error occurred.
    - Provides the expected and actual outputs (if applicable).
    - Shows the error message and traceback for debugging.

---

### 3.2 Step 2: Interpret the Error Message

Run `devtools::test()` in your R console or IDE terminal and note the output. Focus on:

1. **Test File and Test Name**:
   - Example:

     ```
     Failure (test-zzlongplot.R:31:3): parse_formula handles grouping and faceting variab
     result$y (`actual`) not equal to "y" (`expected`).
     ```

   - This indicates:
     - The error is in `test-zzlongplot.R`.
     - It occurred in the test named `parse_formula handles grouping and faceting variables`.
2. **Error Details**:
   - Example:

     ```
     `actual`:   "y ~ x | group"
     `expected`: "y"
     ```

   - The test expected `result$y` to be `"y"`, but the actual output was `"y ~ x | group"`.
3. **Traceback**:
   - The traceback provides a stack of function calls leading to the error. Use `traceback()` immediately after the test failure to get additional context.

---

### 3.3 Step 3: Isolate the Problem

1. **Run the Test Manually**:
   - Extract the failing test from the test file and run it manually:

```
library(testthat)
source("path/to/zzlongplot/R/parse_formula.R")
result <- parse_formula(y ~ x | group ~ facet_y + facet_x)
expect_equal(result$y, "y")
```

- This helps verify if the error occurs outside the testing framework.

2. **Verify Input Data**:
   - Ensure that the inputs to the function are as expected. For example, check if the formula passed to `parse_formula` matches the expected format.
3. **Use Debugging Tools**:
   - Insert `browser()` into the failing function to step through its execution:

```
parse_formula <- function(formula) {
  browser()
  # Function logic...
}
```

   - When the code pauses at `browser()`, inspect the environment using:

```
ls()
print(formula)
```

---

## 3.4 Step 4: Debugging Common Errors

### 3.4.1 Error 1: Assertion Failures

Example:

```
result$y (`actual`) not equal to "y" (`expected`).
```

**Steps**: 1. Check the test code: - Verify if the `expect_equal()` or similar assertion accurately reflects the intended behavior. - Example: `r expect_equal(result$y, "y")` - If the test expectation is wrong, update it to match the correct behavior.

2. Check the function output:
    - Run the function manually with the same inputs and inspect its output:

```
parse_formula(y ~ x | group ~ facet_y + facet_x)
```

3. Fix the function logic:
    - If the function output is incorrect, debug the function implementation.

---

### 3.4.2 Error 2: Unexpected Errors or Warnings

Example:

```
Error: Input must be a formula object
```

**Steps**: 1. Reproduce the Error: - Identify the exact input that triggers the error.

2. Add Input Validation:
    - Validate inputs at the start of the function to catch issues early.

```
if (!inherits(formula, "formula")) {
  stop("Input must be a formula object")
}
```

3. Check for Edge Cases:
    - Test the function with edge cases, such as missing or malformed inputs.

---

#### 3.4.3 Error 3: Missing Dependencies

Example:

```
Error: could not find function "mutate"
```

**Steps**: 1. Check Imports: - Ensure the missing function's package is listed under `Imports` in `DESCRIPTION`.

2. Explicitly Load Dependencies:
   - Use the `::` operator to call functions explicitly:

     ```
     dplyr::mutate(...)
     ```

3. Add `requireNamespace()`:
   - Dynamically load namespaces if not attached:

     ```
     if (!requireNamespace("dplyr", quietly = TRUE)) {
       stop("dplyr is required for this function.")
     }
     ```

---

### 3.5 Step 5: Add Debugging Output

1. **Print Debugging Information**:
   - Add `print()` or `cat()` statements to inspect variables:

     ```
     cat("Parsed y:", y_var, "\n")
     ```

2. **Use Logging**:
   - Use a logging package like **logger** to add structured debug messages.

---

### 3.6 Step 6: Rerun Tests

After making changes: 1. Save the modified code and test files.
2. Re-run the tests: `r devtools::test()`

---

### 3.7 Step 7: Finalize Fixes

1. **Remove Debugging Code**:
   - Remove `browser()`, `print()`, and other debug artifacts.
2. **Re-run Full Checks**:
   - Run `devtools::check()` to ensure the package passes all CRAN checks.

---

### 3.8 Example Workflow for Debugging

1. Identify the failing test:
   ```
   devtools::test()
   ```
2. Manually isolate and debug the test:
   ```
   result <- parse_formula(y ~ x | group ~ facet_y + facet_x)
   print(result)
   ```
3. Add debugging statements:
   ```
   parse_formula <- function(formula) {
     print(formula)
     browser()
     # Logic...
   }
   ```
4. Fix the issue and re-test:

```
devtools::test()
```

5. Remove debugging artifacts and run final checks:

```
devtools::check()
```

---

## 3.9 Conclusion

Debugging `devtools::test()` errors involves interpreting error messages, isolating failing tests, and systematically diagnosing issues in your code. With these steps, you can identify and fix problems effectively.

---