Enhancing R package quality with testthat

Yaoxiang Li

2025-01-20

The importance of unit testing in R package development

Why Testing Matters:

- **Catch bugs early**: Testing prevents regression by ensuring changes don't break existing functionality.
- **Encourage modularity**: Writing tests leads to modular code that is easier to debug and maintain.
- **Enable collaboration**: Comprehensive tests make it easier for contributors to understand and extend the package.
- **Support continuous integration**: Tests are essential for CI pipelines that ensure your package is always in a deploy-able state.

testthat Basics and core features

Key Functions:

1. `test_that(description, code)`: Organizes individual tests.
2. Expectations:

- **Basic Expectations**:
  - `expect_equal()`, `expect_identical()`, `expect_true()`, `expect_false()`.
- **Error and Warning Testing**:
  - `expect_error()`, `expect_warning()`, `expect_message()`.

Example: Testing mathematical operations.

```
library(testthat)

test_that("basic_arithmetic_operations_work_correctly", {
  expect_equal(1 + 1, 2)
  expect_identical(2 * 3, 6)
  expect_true(is.numeric(10 / 2))
})
```

Skipping Tests: Useful when external resources are unavailable.

```
test_that("test_skipped_when_internet_is_unavailable", {
  skip_if_offline()
  expect_error(httr::GET("https://some-api-url.com"))
})
```

Organizing your tests for maintainability

Structure:

- Use a `tests/testthat` directory for organization.
- Naming convention: Align test file names with function names (e.g., `test_function.R`).

Special Files:

- `setup.R`: Code executed before running tests.
- `teardown.R`: Code executed after tests complete.
- `helper-*.R`: Functions shared across multiple tests.

Practical scenarios with examples

Testing Data Validation: - Check if a function properly validates inputs.

```
test_that("Function rejects invalid inputs", {
  expect_error(my_function(NULL), "Input cannot be NULL")
  expect_error(my_function("invalid_string"), "Input must be numeric")
})
```

Testing Random Outputs: - Use fixed seeds to ensure reproducibility.

```
set.seed(42)
random_output <- sample(1:10, 5)

test_that("Random output is consistent with fixed seed", {
  expect_equal(random_output, c(9, 2, 6, 7, 5))
})
```

Applying testthat to the medrxivr Package

Applications of Unit Testing in medrxivr

The medrxivr package demonstrates the importance of comprehensive testing for ensuring the reliability and reproducibility of bioinformatics tools. Here are some applications from its test suite:

Example of API Testing

Objective: Validate data integrity and ensure consistent output when accessing external APIs.

Example:

```
library(testthat)
test_that("check_data_inputs_return_the_same_number_of_results", {
  skip_on_cran()
  skip_if_offline() # Skips tests when offline or API is unavailable
  mx1 <- mx_search(
    data = mx_snapshot("6c4056d2cccd6031d92ee4269b1785c6ec4d555b"),
    query = "dementia", from_date = "2019-01-01", to_date = "2020-01-01")
  mx2 <- mx_search(
    data = mx_api_content(
      from_date = "2019-01-01", to_date = "2020-01-01", include_info = TRUE
    ), query = "dementia")
  expect_equal(nrow(mx1), nrow(mx2))
})
```

File download and export testing

- **Objective**:
  - Tests validate naming schemes (e.g., ID-based and DOI-based).
  - Includes status updates and error handling for missing files.
- **Example**:

```
library(testthat)
mx_result <- data.frame(
  link_pdf = "https://www.medrxiv.org/content/10.1101/19003301v4.full.pdf",
  ID = "271", doi = "10.1101/19003301")
test_that("naming_of_downloaded_pdfs", {
  skip_on_cran()
  skip_if_offline()
  tmpdir <- tempdir()
  mx_download(mx_result, tmpdir, name = "ID")
  expect_true(file.exists(paste0(tmpdir, "/271.pdf")))
})
```

Syntax and query validation

- **Objective**: Test parsing and transformation of complex queries.
  - Ensures consistent query transformations (e.g., capitalization, wildcard handling).
  - Verifies expected behavior for logical operators like `NEAR`.
- **Example**:

```
library(testthat)

test_that("syntax_operators", {
  expect_true(grepl(mx_caps("ncov"), c("NCOV", "ncov", "NcOv")))
  expect_false(grepl(mx_caps("Test test"), "test  test"))
})
```

Test Fixtures

- Create temporary files or environments for isolated tests.

```r
with_temp_file <- function(code) {
  temp_file <- tempfile()
  on.exit(unlink(temp_file), add = TRUE)
  code(temp_file)
}

test_that("Temporary file handling", {
  with_temp_file(function(temp_file) {
    writeLines("Hello, world!", temp_file)
    expect_true(file.exists(temp_file))
    expect_equal(readLines(temp_file), "Hello, world!")
  })
})
```

Custom Expectations

- Create reusable expectations to simplify complex checks.

```r
expect_multiple_of <- function(x, multiple) {
  if (x %% multiple != 0) {
    stop(sprintf("%s is not a multiple of %s", x, multiple))
  }
  invisible(TRUE)
}

test_that("Custom expectation works", {
  expect_multiple_of(10, 5)
  expect_error(expect_multiple_of(10, 3), "10 is not a multiple of 3")
})
```

Example of using snapshot

```
test_that("Data frame snapshot remains consistent", {
  expect_snapshot_value(generate_dataframe(), style = "json2") # Compatible st
})

test_that("Snapshot with large data", {
  set.seed(123) # Ensure reproducibility
  large_df <- data.frame(id = 1:1000, value = rnorm(1000))
  expect_snapshot_value(large_df, style = "json2")
})
```

Snapshot Directory Customization

Snapshot testing is a powerful feature in `testthat` that helps validate the state of complex outputs. The ability to capture and compare object states over time ensures stability in package behavior. Additionally, customizing snapshot directories through monkey patching enhances testing workflows.

Customizing Snapshot Directories

By default, snapshot files are saved in `tests/testthat/_snaps/`. However, it is sometimes necessary to customize this directory without modifying the `testthat` source code. This can be achieved by dynamically modifying internal behavior at runtime.

Example: Modifying Snapshot Directory with Custom Patching

```
cus_ss_dir <- function(function_name, new_directory,
old_directory = "_snaps") {
  target_function <- getFromNamespace(function_name, ns = "testthat")
  function_body <- deparse(body(target_function))
  updated_body <- gsub(sprintf('"%s"', old_directory),
  sprintf('"%s"', new_directory), function_body, fixed = TRUE)
  body(target_function) <- parse(text = paste(updated_body, collapse = "\n"))
  assignInNamespace(function_name, target_function, ns = "testthat")
}
cus_ss_dir("test_files_reporter", "_snapshots")
cus_ss_dir("snapshot_meta", "_snapshots")
```

Pros: - Allows snapshots to be saved in `tests/testthat/_snapshots/` or any other preferred directory. - Works seamlessly with existing `expect_snapshot()` and `expect_snapshot_file()` functions.

Additional Customization for Relative Paths

For portability across environments, avoid absolute paths. Instead, use relative paths:

```
cus_ss_dir("test_files_reporter", "../../custom_snaps")
cus_ss_dir("snapshot_meta", "../../custom_snaps")
```

Cons: - This approach relies on internal `testthat` structures, which may change in future releases (Use tools like `renv` to manage dependencies and ensure compatibility). - Avoid hardcoding absolute paths to maintain cross-platform compatibility.

Best Practices for New Contributors

1. **Write tests alongside code**: Don't leave testing as an afterthought.
2. **Start small**: Focus on simple expectations before moving to complex scenarios.
3. **Use readable descriptions**: Ensure test descriptions clearly convey the intent.
4. **Review coverage**: Use tools like `covr` (https://covr.r-lib.org/) to identify untested areas.
5. **Check for edge cases**: Test boundary values, empty inputs, and unusual conditions.