Testing with testthat

Lecture 06

Dr. Colin Rundel

Package checking

`R CMD check` - What it does

`R CMD check` is CRAN's comprehensive quality control system that runs dozens of checks:

- **Package structure** - Correct directories, required files (DESCRIPTION, NAMESPACE)
- **Code syntax** - All R code parses without errors
- **Documentation** - All functions documented, all examples run
- **Dependencies** - All used packages exist and in DESCRIPTION
- **Tests** - All test files execute without errors
- **CRAN policy compliance** - Follows all submission guidelines (written and unwritten)

devtools::check() vs R CMD check

`devtools::check()` is a convenient wrapper around `R CMD check`:

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

```
R CMD build .
R CMD check packagename_1.0.0.tar.gz
```

Benefits of `devtools::check()`:

- Automatically handles building and checking
- Better integrated with RStudio workflow
- Cleaner output formatting
- Automatically installs package first

Interpreting check output

`R CMD check` produces three levels of issues:

- **ERROR** 🔴 - Must be fixed before CRAN submission
- **WARNING** 🟡 - Should be fixed (CRAN may reject)
- **NOTE**  - Optional improvements (CRAN usually accepts)

Anything flagged must be addressed in the CRAN submission process.

While the check is running issues are shown inline, *and* summarized at the end.

GitHub Actions for continuous checking

Set up automated checking with:

```
usethis::use_github_action("check-standard")
```

This creates `.github/workflows/R-CMD-check.yaml` that runs checks on:

- **Latest R** on macOS, Windows, Linux (Ubuntu)
- **Previous R** and **R-devel** on Linux (Ubuntu)

Package testing

Basic test structure

Package tests live in `tests/`,

- Any R scripts found in the folder will be run when Checking the package (not Building)
- Generally "tests" are considered a failure if an error is thrown, but warnings are also tracked
- Testing is possible via base R but it is not recommended (See Writing R Extensions)
- There is functionality for comparing test outputs to expected results, but it has limited functionality
- Note that R CMD check also runs all documentation examples (unless explicitly tagged with dont run) - which is also used for basic testing (does the code run without error)


test
that
www.rstudio.com

testthat fundamentals

testthat is the most widely used testing framework for R packages with excellent RStudio integration.

A project can be initialized to use testthat via,

```
1 usethis::use_testthat(3)  # Use latest edition
```

This creates the following files and directories:

- `tests/testthat.R` - Entry point for R CMD check
- `tests/testthat/` - Directory for test files
- Adds testthat to `DESCRIPTION`'s `Suggests` field

Editions are a way for newer versions of testthat to maintain backwards compatibility, Edition 3 is the latest version

testthat project structure

```
mypackage/
├── R/
│   └── utils.R
├── tests/
│   ├── testthat.R
│   └── testthat/
│       └── test-utils.R
└── DESCRIPTION
```

Test file naming:

testthat script structure

Tests are hierarchically organized:

- **File** - Collection of related tests
- **Test** - Group of related expectations (`test_that()`)
- **Expectation** - Single assertion (`expect_equal()`, `expect_error()`)

```
1 test_that("`+` works correctly", {
2   expect_equal(`+`(2, 3), 5)
3   expect_equal(`+`(0, 0), 0)
4   expect_type(`+`(1, 1), "double")
5   expect_type(`+`(1L, 1L), "integer")
6 })
```

```
Test passed 🥳
```

Running tests

There are multiple ways to execute your package's tests:

During development:

- `devtools::test()` - Run all tests
- `devtools::test_file("tests/testthat/test-utils.R")` - Run one file
- **Ctrl/Cmd+Shift+T** (RStudio) - Run all tests
- **Ctrl/Cmd+T** (RStudio) - Run tests for current file

From command line:

- `R CMD check` - Runs tests as part of package check
- `Rscript -e "devtools::test()"` - In scripts/CI

Core expectation functions

testthat provides many expectation functions for different scenarios:

Equality and identity

```
expect_equal(actual, expected)      # Equal within tolerance
expect_identical(actual, expected) # Exactly identical
expect_true(x)                      # Exactly TRUE
expect_false(x)                     # Exactly FALSE
```

Types and classes:

```
expect_type(x, "double")            # Storage type
expect_s3_class(df, "data.frame")  # S3 class
```

Conditions:

```
expect_error(code, regexp = "...")    # Throws error (optiona
expect_warning(code, regexp = "...")  # Throws warning
expect_message(code, regexp = "...")  # Prints message
```

expect_equal() vs expect_identical()

Understanding the difference is important:

```
test_that("equality vs identity", {
  # These pass - expect_equal has tolerance for floating point
  expect_equal(0.1 + 0.2, 0.3)
  expect_equal(1L, 1.0)             # Integer vs double
  expect_true(0.2+0.2 == 0.4)

  # These fail - expect_identical requires exact match
  expect_identical(0.1 + 0.2, 0.3) # FALSE due to floating point
  expect_identical(1L, 1.0)        # FALSE, different types
  expect_true(0.1+0.2 == 0.3)      # FALSE due to floating point
})
```

```
── Failure: equality vs identity ──────────────────
0.1 + 0.2 not identical to 0.3.
Objects equal but not identical

── Failure: equality vs identity ──────────────────
1L not identical to 1.
Objects equal but not identical

── Failure: equality vs identity ──────────────────
0.1 + 0.2 == 0.3 is not TRUE
```

```
`actual`:   FALSE
`expected`: TRUE
```

```
Error:
! Test failed
```

Testing function outputs

```
calculate_mean_ci = function(x, conf_level = 0.95) {
  if (length(x) == 0)
    stop("Cannot calculate CI for empty vector")
  if (any(is.na(x)))
    stop("Missing values not allowed")

  n = length(x)
  mean_x = mean(x)
  se = sd(x) / sqrt(n)
  t_val = qt((1 + conf_level) / 2, df = n - 1)

  c(lower = mean_x - t_val * se, upper = mean_x + t_val * se)
}
```

Example tests

```
test_that("calculate_mean_ci works correctly", {
  # Test normal case
  result = calculate_mean_ci(c(1, 2, 3, 4, 5))
  expect_type(result, "double")
  expect_length(result, 2)
  expect_named(result, c("lower", "upper"))
  expect_true(result["lower"] < result["upper"])

  # Test with known values
  expect_equal(
    calculate_mean_ci(c(0, 0, 0)),
    c(lower = 0, upper = 0)
  )

  # Test confidence level parameter
  ci_95 = calculate_mean_ci(c(1, 2, 3), conf_level = 0.95)
  ci_99 = calculate_mean_ci(c(1, 2, 3), conf_level = 0.99)
  expect_true(ci_99["upper"] - ci_99["lower"] > ci_95["upper"] - ci_95["lower"])
})
```

```
Test passed 🥳
```

Testing error conditions

It is important to test that your functions fail appropriately,

```
test_that("calculate_mean_ci handles edge cases", {
  # Empty vector should error
  expect_error(calculate_mean_ci(numeric(0)), "Cannot calculate CI for empty vector")

  # Missing values should error
  expect_error(calculate_mean_ci(c(1, 2, NA)), "Missing values not allowed")

  # Invalid confidence level should error (if we add validation)
  expect_error(calculate_mean_ci(1:5, conf_level = 1.5), "conf_level must be between 0 and 1"

  # Single value (edge case to think about)
  expect_error(calculate_mean_ci(5))  # Or should this work?
})
```

```
── Warning: calculate_mean_ci handles edge cases ──
NaNs produced
Backtrace:
    ▆
 1. ├─testthat::expect_error(...)
 2. │ └─testthat:::quasi_capture(...)
 3. │   ├─testthat (local) .capture(...)
 4. │   │ └─base::withCallingHandlers(...)
 5. │   └─rlang::eval_bare(quo_get_expr(.quo), quo_get_env(.quo))
 6. └─global calculate_mean_ci(1:5, conf_level = 1.5)
 7.   └─stats::qt((1 + conf_level)/2, df = n - 1)
```

```
── Failure: calculate_mean_ci handles edge cases ──
`calculate_mean_ci(1:5, conf_level = 1.5)` did not throw an error.

── Warning: calculate_mean_ci handles edge cases ──
NaNs produced
Backtrace:
    ▆
 1. ├─testthat::expect_error(calculate_mean_ci(5))
 2. │ └─testthat:::quasi_capture(...)
 3. │   ├─testthat (local) .capture(...)
Error:
! Test failed
```

Testing for errors

Testing for errors is important, but `expect_error()` can be dangerous if you don't check the output. All that the expectation tells you is that *some* error was thrown, not that it was the *right* error.

```
calculate_discount = function(price, discount_percent) {
  if (price < 0) stop("Price cannot be negative")
  if (discount_percent > 100) stop("Discount cannot exceed 100%")

  price * (1 - discount_pct / 100)  # Bug: wrong variable name
}

test_that("demonstrates why checking error messages matters", {
  # ✗ passes but for the wrong reason!
  expect_error(calculate_discount(100, 150))
  # ✓ This correctly tests the price validation
  expect_error(calculate_discount(-50, 10), "Price cannot be negative")
})
```

```
Test passed 🎉
```

Skipping tests

Skip tests when certain conditions aren't met:

```
test_that("database connection works", {
  skip_if_not_installed("RPostgreSQL")
  skip_if(Sys.getenv("TEST_DB_URL") == "", "Database URL not set")
  skip_on_cran()  # Skip on CRAN (for tests that take too long)
  skip_on_ci()    # Skip on continuous integration

  # Your database tests here...
})

test_that("internet-dependent test", {
  skip_if_offline()

  # Test that requires internet connection
  result = download_data("https://example.com/api")
  expect_type(result, "list")
})
```

Snapshot tests

Snapshot tests capture the output of your functions and compare against previously saved results:

- **First run**: Snapshot is created and saved
- **Subsequent runs**: Current output compared against saved snapshot
- **When output changes**: Test fails, you review and accept/reject the change

Snapshot tests are best for::

- Error messages and warnings
- Complex data structure outputs
- Printed output from functions
- Any output where exact specification is difficult

expect_snapshot() for output

Test printed output and messages:

```
print_summary = function(data) {
  cat("Data summary:\n")
  cat("Rows:", nrow(data), "\n")
  cat("Columns:", ncol(data), "\n")
  cat("Column names:", paste(names(data), collapse = ", "), "\n")
}

test_that("print_summary produces consistent output", {
  df = data.frame(x = 1:3, y = letters[1:3])

  expect_snapshot({
    print_summary(df)
  })
})
```

Snapshot output

Creates `tests/testthat/_snaps/test-print_summary.md`:

```
# print_summary produces consistent output

    Code
      print_summary(df)
    Output
      Data summary:
      Rows: 3
      Columns: 2
      Column names: x, y
```

Managing snapshots

Accepting changes

```
# Run this to accept all snapshot chan
snapshot_accept()

# Or accept a specific test file:
snapshot_accept("test-myfunction.R")
```

Reviewing changes

```
# Run this to review all snapshot chan
snapshot_review()

# Or review a specific test file:
snapshot_review("test-myfunction.R")
```

Some best practices:

- Review snapshot changes carefully in code review
- Don’t commit snapshot updates without understanding why they changed
- Use descriptive test names for easier snapshot identification

Why testing matters

Testing is a fundamental part of creating reliable, maintainable R packages (and code in general):

- **Catch bugs early** - Find problems before they reach users
- **Document behavior** - Tests serve as executable specifications
- **Prevent regressions** - Ensure new changes don't break existing functionality
- **Enable refactoring** - Change implementation with confidence

Testing as documentation

Well-written tests serve multiple purposes:

```
test_that("mean() behaves as expected", {
  # Basic usage - compute arithmetic mean
  expect_equal(mean(c(1, 2, 3)), 2)

  # Missing values cause NA by default
  expect_true(is.na(mean(c(1, 2, NA))))

  # na.rm = TRUE removes missing values before calculation
  expect_equal(mean(c(1, 2, NA), na.rm = TRUE), 1.5)

  # Empty vector returns NA with warning
  expect_warning(result <- mean(numeric(0)))
  expect_true(is.na(result))
})
```

Tests make your intentions clear to future maintainers (including yourself!)

Test-Driven Development

The TDD cycle: Red-Green-Refactor

Test-Driven Development follows a simple cycle:

1. 🔴 **Red**: Write a failing test for the functionality you want to implement
2. 🟢 **Green**: Write the minimal code to make the test pass
3. 🔵 **Refactor**: Clean up the code while keeping tests green
4. **Repeat**: Move on to the next piece of functionality

This approach ensures:

- You only write code that's actually needed
- Every line of code is covered by tests
- Your design is driven by actual usage

TDD example

Let's implement a `is_palindrome()` function using TDD:

Step 1 - Write the test(s) first

```
1 test_that("is_palindrome works correctly", {
2   expect_true(is_palindrome(c(1, 2, 3, 2, 1)))
3   expect_true(is_palindrome(c("a", "b", "a")))
4   expect_false(is_palindrome(c(1, 2, 3)))
5   expect_true(is_palindrome(c(5)))  # Single element
6   expect_true(is_palindrome(numeric(0)))  # Empty vector
7 })
```

```
── Error: is_palindrome works correctly ────────────
Error in `is_palindrome(c(1, 2, 3, 2, 1))`: could not find function "is_palindrome"
Backtrace:
    ▆
 1. └─testthat::expect_true(is_palindrome(c(1, 2, 3, 2, 1)))
 2.   └─testthat::quasi_label(enquo(object), label, arg = "object")
 3.     └─rlang::eval_bare(expr, quo_get_env(quo))

Error:
! Test failed
```

Step 2

Write minimal code to pass:

```
is_palindrome = function(x) {
  all(x == rev(x))
}
```

Which we then check with our existing tests:

```
test_that("is_palindrome works correctly", {
  expect_true(is_palindrome(c(1, 2, 3, 2, 1)))
  expect_true(is_palindrome(c("a", "b", "a")))
  expect_false(is_palindrome(c(1, 2, 3)))
  expect_true(is_palindrome(c(5)))  # Single element
  expect_true(is_palindrome(numeric(0)))  # Empty vector
})
```

```
Test passed 🥇
```

Step 3: Refactor

We can consider a slightly improved implementation:

```
1 is_palindrome = function(x) {
2   identical(x, rev(x))
3 }
```

Which we again verify with the tests:

```
1 test_that("is_palindrome works correctly", {
2   expect_true(is_palindrome(c(1, 2, 3, 2, 1)))
3   expect_true(is_palindrome(c("a", "b", "a")))
4   expect_false(is_palindrome(c(1, 2, 3)))
5   expect_true(is_palindrome(c(5)))  # Single element
6   expect_true(is_palindrome(numeric(0)))  # Empty vector
7 })
```

```
Test passed 🥇
```

Step 4: Repeat

We can consider additional functionality, such as input validation by expanding our tests:

```
test_that("is_palindrome errors for non-atomic input", {
  expect_error(is_palindrome(list(1, 2, 1)))
})
```

```
── Failure: is_palindrome errors for non-atomic input ──
`is_palindrome(list(1, 2, 1))` did not throw an error.

Error:
! Test failed
```

```
is_palindrome = function(x) {
  stopifnot("Input must be an atomic vector" = is.atomic(x))
  identical(x, rev(x))
}
```

```
test_that("is_palindrome errors for non-atomic input", {
  expect_error(is_palindrome(list(1, 2, 1)))
})
```

```
Test passed 🎉
```

TDD in the real world

In practice, TDD may not be followed strictly, but the principles remain valuable:

- Tests should guide your design and implementation
- Tests should not be an after thought once your code is “done”
- Refactoring is easier and safer with a solid test suite
- Writing tests 2nd can lead to missing edge cases / faulty assumptions

Why Packages?

Benefits of packages

Organizing your projects as a package provides many advantages:

- Benefit from the existing infrastructure for package development
- Easier to share and distribute your code (dependencies, installation, documentation, etc.)
- Easier to bundle and document data sets
- Better support for testing and documentation
- Tends to lead to better organized, modular code and overall better design

Packages and LLMs

We will go into this more on Monday, but packages are also a great way to structure your code to work with LLMs:

- Proscribed structure makes it easier for the LLMs to understand your codebase
- Better context management
- Better grounding and easier interation through tests and checks