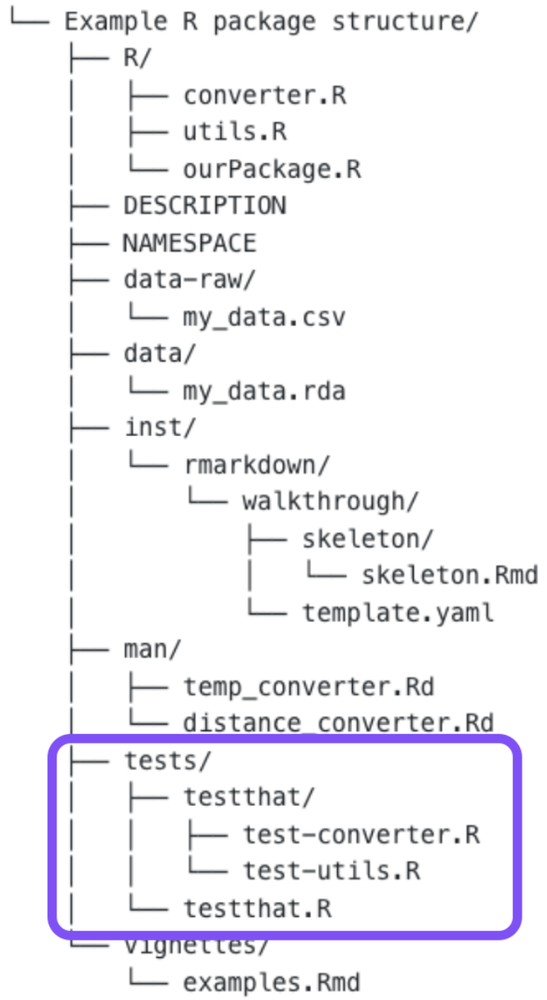
Unit tests

DEVELOPING R PACKAGES

Jasmin Ludolf
Content Developer

R package structure



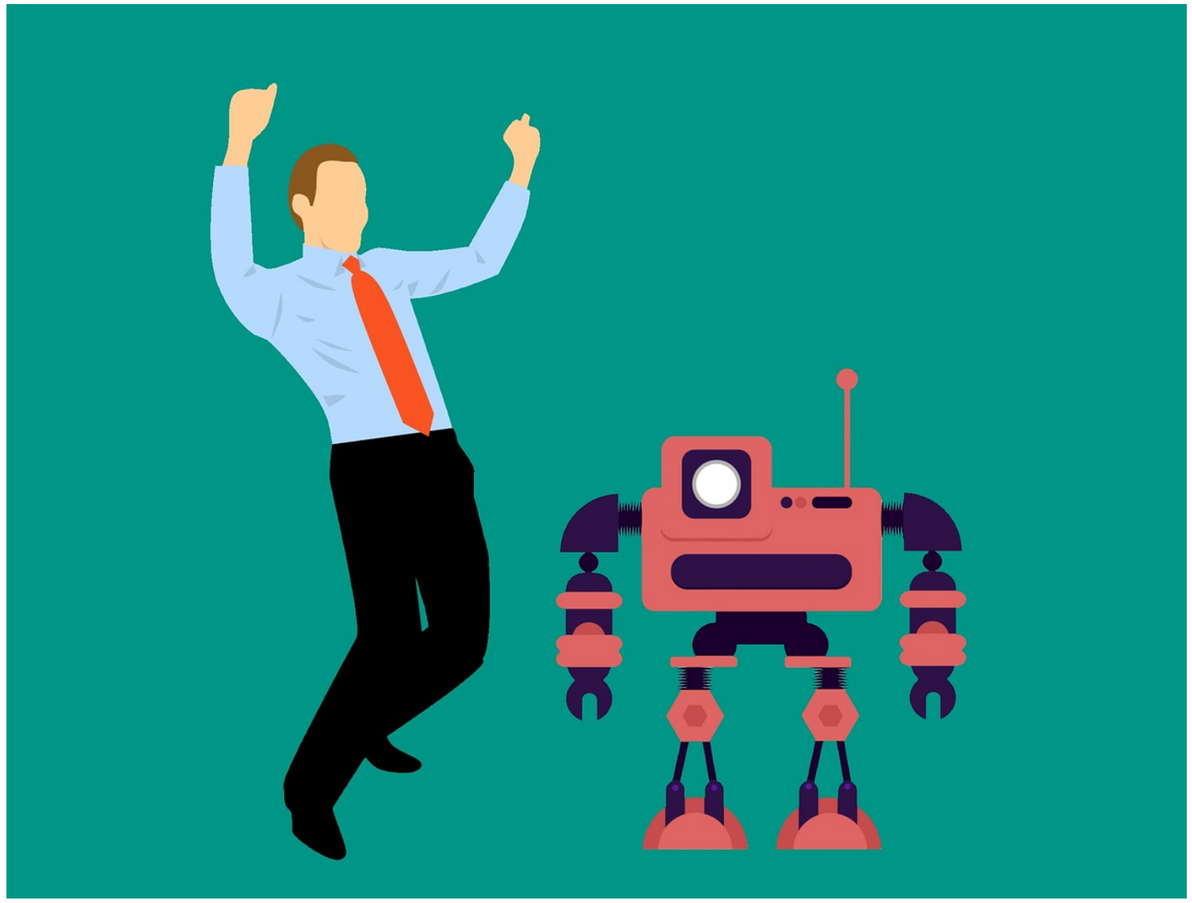
Tests

Testing your code works

- Documentation helps our users
- And helps the package author with development and maintenance
- Manually checking functions can be difficult

What are unit tests?

- Automated and run all at once
- Verify functionality of package units
- Ensure expected behavior and correct output
- Catch bugs early
- Support clean and testable code, supporting collaboration



Structuring unit testing in R

- Testing framework with `usethis` + `testthat`

```
library(usethis)
use_testthat()
```

- `testthat` package: simple and intuitive syntax for writing unit tests

`use_testthat()` output :

```
✔ Adding 'testthat' to Suggests field in DESCRIPTION
✔ Adding '3' to Config/testthat/edition
✔ Creating 'tests/testthat/'
✔ Writing 'tests/testthat.R'
● Call `use_test()` to initialize a basic test file and open it for editing.
```

Creating test template files

- Match names of `.R` files in `R` directory to names in `tests/testthat`
  - `R/temp_converter.R`
  - `R/time_converter.R`
- Corresponding files in `tests/testthat`
  - `tests/testthat/test-temp_converter.R`
  - `tests/testthat/test-time_converter.R`

A function for creating these files

```
use_test(name = "temp_converter")
use_test(name = "time_converter")
```

✔ Writing 'tests/testthat/test-temp_converter.R'
● Modify 'tests/testthat/test-temp_converter.R'

✔ Writing 'tests/testthat/test-time_converter.R'
● Modify 'tests/testthat/test-time_converter.R'

Let's practice!

DEVELOPING R PACKAGES

Exploring expect statements

DEVELOPING R PACKAGES

Jasmin Ludolf
Content Developer

Most common expect statements

- Expect statements validate R functions, or units

In `testthat` package:

- `expect_equal()` : equality with small tolerance allowed
- `expect_identical()` : exact equality
- `expect_output()` : matching text from `object` call

- `expect_error()` : error from `object` call
  - Errors stop code execution; warnings don't
- `expect_warning()` : warning produced by `object` call

From example to unit test

```
#' @examples
#' # Convert 32F to C
#' temp_converter(32, "Fahrenheit", "Celsius")
```

```
[1] 0
```

In the `tests/testthat/test-temp_converter.R` file:

```
library(testthat)
expect_equal(
  object = temp_converter(32, "Fahrenheit", "Celsius"),
  expected = 0
)
```

Expecting identical and expecting equal

`expect_identical()`

```
expect_identical(sqrt(3) ^ 2, 3)
```

```
Error: sqrt(3)^2 (`actual`) not
identical to 3 (`expected`).

  `actual`: 2.9999999999999996
`expected`: 3.0000000000000000
```

`expect_equal()`

- Use `expect_equal()` when comparing numeric values

```
expect_equal(sqrt(3) ^ 2, 3)
```

- `expect_equal()`
  - Allows for small differences
  - Has a `tolerance` argument, default usually good

Expecting output

```
expect_output(
    print("Testing R Packages is fun"),
   "funk"
)
```

```
Error: `print\("Testing R Packages is
fun"\)` does not match "funk".
Actual value: "\[1\] "Testing R
Packages is fun""
```

OUTPUT

Expecting warning

```
expect_warning(
  temp_converter(-40,
                 "Celsius",
                 "Celsius")
)
```

Expecting error

```
expect_error(
  temp_converter(300,
                 "Kelvin",
                 "Fahrenheit")
)
```

Recap

- Each expectation should test an aspect of a function
- When combined together, 100% of function aspects should be covered by unit tests

Let's practice!

DEVELOPING R PACKAGES

Storing expectations as unit tests and running tests

DEVELOPING R PACKAGES

Jasmin Ludolf
Content Developer

datacamp

Workflow

1. Create examples to show how our function works
2. Write expectations to check function behavior
3. Group expectations that test common aspect into a unit test
4. Run all unit tests in a file
5. Run all unit tests throughout a package

Grouping similar function expectations together

```
expect_equal(temp_converter(32, "Fahrenheit", "Celsius"), 0)
expect_warning(temp_converter(-40, "Celsius", "Celsius"))
expect_error(temp_converter(300, "Kelvin", "Fahrenheit"))
```

```
library(testthat)
test_that("Conversion from F to C and C to F works", {
  expect_equal(temp_converter(32, "Fahrenheit", "Celsius"), 0)
  expect_warning(temp_converter(-40, "Celsius", "Celsius"))
  expect_error(temp_converter(300, "Kelvin", "Fahrenheit"))
})
```

```
Test passed 🎉
```

What if tests failed?

```
library(testthat)
test_that("Conversion from F to C and C to F works", {
      expect_equal(temp_converter(32,  "Fahrenheit", "Celsius")), 1)
      expect_warning(temp_converter(-40), "Celsius", "Fahrenheit")
      expect_error(temp_converter(25))
})
```

```
X Conversion from F to C and C to F works
-- 1. Error: `temp_converter(32,  "Fahrenheit", "Celsius")` not equal to 1
(@test_conversion.R:2)
--
-- 2. Error: `temp_converter(-40)` did not throw the expected warning.
(@test_conversion.R:3)
--
temp_converter(25) did not throw an error.
```

Running all tests in a file

- To run all of the tests in a file, use `test_file()`

```
library(testthat)
test_file("test-temp_conversion.R")
```

```
Conversion from F to C and C to F works
```

Run tests on the examples

```
#' @examples
#' # Convert 25 degrees Celsius to Fahrenheit
#' temp_converter(25, unit_from = "Celsius", unit_to = "Fahrenheit")
#' # Convert 100 degrees Fahrenheit to Celsius
#' temp_converter(100, unit_from = "Fahrenheit", unit_to = "Celsius")
```

- `roxygenize()` generates a help file for `temp_converter` in `man` directory

```
test_example("man/temp_converter.Rd")
```

Test passed 🎉

You gotta run 'em all!

- Use `test_file()` if you want to focus on a particular file
- Don't have to run `test_file()` on each testing file each time
- Can instead run `test_package()` to run them all!

```
test_package("unitConverter")
```

Let's practice!

DEVELOPING R PACKAGES

Final steps

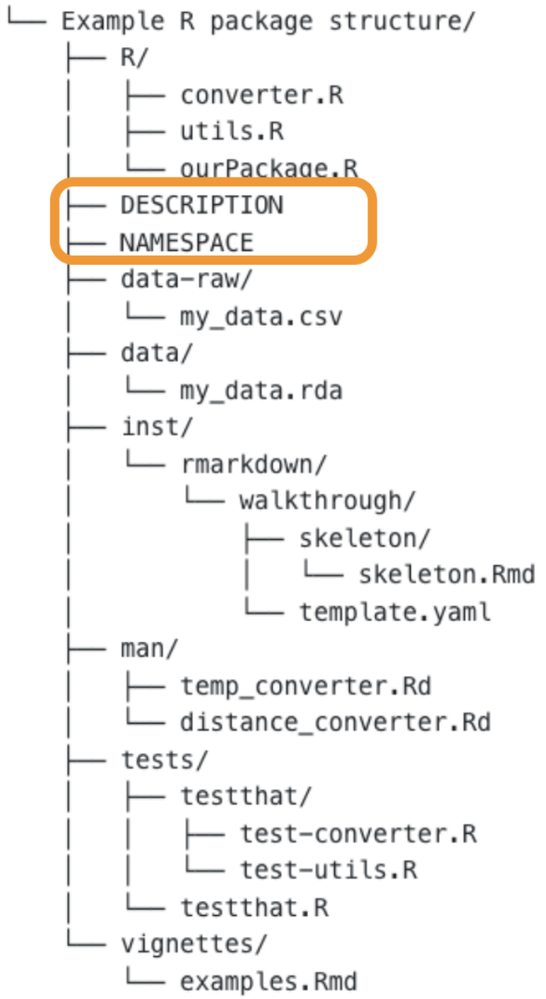
DEVELOPING R PACKAGES

Jasmin Ludolf
Content Developer

datacamp

R package structure

Supporting Files



Package versions

Example: `tidyr` `1.3.0`

- Format: `<major>.<minor>.<patch>.<dev>`
- Default: `0.0.0.9000`
- `9000` : package in development

Package version incrementing guidelines

Format: `<major>.<minor>.<patch>.<dev>`

- `dev` : rarely changed, remove for release
- `patch` : changed for bug fixes
  - More bugs before functionality change
- `minor` : small additional functionality added to package before initial release
  - `patch` number reset to `0`
- `major` : package considered stable to release
  - From `1.5.3` to `2.0.0` : old code could break

- Not in development: `0.0.0.9000` -> `0.0.0`
- One bug fixed: `0.0.1`
- One more bug fixed: `0.0.2`
- Additional functionality: `0.1.0`

- Stable for release: `1.0.0`
- Big changes: `2.0.0`

Update title, author, version, and description

```
Package: unitConverter
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).
```

```
Package: unitConverter
Title: Unit Conversion Utilities for Distance,
    Time, Weight, and Temperature
Version: 0.1.0
Authors@R:
    person("Jasmin", "Ludolf", ,
           "myemail@example.com",
           role = c("aut", "cre"))
Description: The `unitConverter` package provides
    a collection of utility functions for
    converting distance, time, weight, and
    temperature values. It offers seamless
    conversion between various units within each
    category, allowing users to easily transform
    measurement data.
```

Documenting a data file

```
use_r("temperature_data")
```

Edit `R/temperature_data.R`:

```
#' Temperature values and units
#'
#' Temperature values and corresponding unit (Celsius, Fahrenheit, or Kelvin)
#' @format Data frame with two columns and 1000 rows
#' \describe{
#' \item{value}{Numeric temperature value.}
#' \item{unit}{Temperature unit.}
#' }
#' @examples
#'   temperature_data
"temperature_data"
```

```
roxygenize()
```

What does devtools::check() look for?

`devtools::check()`

- Can the package be installed
- `DESCRIPTION` information is correct
- Package dependencies
- Syntax errors in code
- Complete documentation
- Successful tests running
- Successful vignettes building

Running devtools::check()

Let's practice!

DEVELOPING R PACKAGES

Congratulations!

DEVELOPING R PACKAGES

Jasmin Ludolf
Content Developer

Chapter 1 review

- R package structure: directories, `DESCRIPTION`, `NAMESPACE`
  - `usethis::create_package()`
- Including data in the package
  - `usethis::use_data()`
- Writing R functions for package development
  - `dump()`
- Installing and testing the package locally
  - `devtools::install()`

Chapter 2 review

- Compare packages vs. scripts
- Choose informative package name
- Check package name availability on CRAN
  - `available::available()`
- Explore different license options
  - `usethis::use_mit_license()`
- Load package components and check packages
  - `devtools::load_all()` and `devtools::check()`

Chapter 3 review

- Help file components and exported functions
- Importance of examples in documentation
- Creating function examples with `roxygen2`
  - `roxygen2::roxygenize()`
- Understanding purpose of vignettes
- Browsing and assessing vignettes
  - `browseVignettes()`
- Designing and building vignettes
  - `usethis::use_vignette()` and `devtools::build_vignettes()`

Chapter 4 review

- Recognizing and creating unit tests
  - `usethis::use_testthat()` and `usethis::use_test()`
- Exploring expect statements for testing
  - `testthat::expect_equal()` and `testthat::expect_error()`
- Organizing and running unit tests
  - `testthat::test_that()`
- Updating package metadata and documentation
- Discussing package versioning
- Running a package check
  - `devtools::check()`

Next steps

- Sharing your package online/with others
- Using `devtools::release()` for package release to CRAN
- Implementing continuous integration for your package

Hooray!

DEVELOPING R PACKAGES