Unit testing & continuous integration

How to find bugs as soon as you create them

David A. Selby
R-thritis Computing Group
19 November 2021

Typically, we test code to ensure *its output meets our expectations*.

Great Expectations

```
is_plausible <- function(value, minimum = -Inf, maximum = Inf)
  value >= minimum & value <= maximum
```

What do you **expect** the output to be?

```
is_plausible(c(10, 37, -999), min = 0, max = 112)
```

```
# [1]  TRUE  TRUE FALSE
```

```
is_plausible("2", min = 0, max = 10)
```

```
# [1] FALSE
```

Why? Because `"10" < "2"` in lexical order

Unit testing

Unit testing in R

Unit testing is the process in which the smallest testable parts of source code are tested individually and independently, usually in an automated way.

- Formalises the testing of code
- Makes it easier to identify bugs when they are introduced
- Helps ensure that the code meets all necessary criteria
- Reassures the user that the code works correctly

Unit testing in R

testthat is an R package created by Hadley Wickham for the purpose of writing unit tests for R code. It is available on CRAN.

Other unit testing packages are available (but not covered here), e.g. **RUnit**, **testrmd**.

Unit testing with **testthat**

The **testthat** framework comprises three parts:

Expectations ←
: the core functions. They all have prefix `expect_`

Tests ←
: a series of expectations about one feature, wrapped in `test_that()`

Files ←
: containing a set of tests of related functionality

Unit testing with **testthat**

Expectations in **testthat** compare an object with a reference value or property. If they **do not match**, an error is thrown.

(If they do match, the tested object is returned, invisibly.)

```
> expect_equal
> expect_is
> expect_length
> expect_true
> expect_error
> ...
```

CENTRE FOR EPIDEMIOLOGY VERSUS ARTHRITIS

Unit testing with **testthat**

`expect_equal` compares a number with a reference value

```
hyp <- 3^2 + 4^2
expect_equal(hyp, 25) # Runs without error
expect_equal(hyp, 26) # Throws an error:
```

```
# Error: `hyp` not equal to 26.
# 1/1 mismatches
# [1] 25 - 26 == -1
```

```
expect_equal(sqrt(2), 1.41) # Throws an error:
```

```
# Error: sqrt(2) not equal to 1.41.
# 1/1 mismatches
# [1] 1.41 - 1.41 == 0.00421
```

```
expect_equal(sqrt(2), 1.41, tolerance = .01) # No error thrown
```

CENTRE FOR EPIDEMIOLOGY VERSUS ARTHRITIS

Unit testing with **testthat**

`expect_is` checks the data type of an object

```
val <- 43.7
expect_is(val, 'numeric')   # Runs without error
expect_is(val, 'character') # Produces an error:
```

```
# Error: `val` inherits from `'numeric'` not `'character'`.
```

`expect_length` checks the number of elements in a vector

```
expect_length(letters, 26) # Runs without error
expect_length(letters, 25) # Throws an error:
```

```
# Error: `letters` has length 26, not length 25.
```

CENTRE FOR EPIDEMIOLOGY VERSUS ARTHRITIS

Unit testing with **testthat**

Some expectation functions check against a (fixed) condition, rather than a custom reference value.

```
codes <- c(no = F, yes = T)
expect_true(codes['yes'])    # Runs without error
expect_true('yes')           # Throws an error:

# Error: "yes" is not TRUE
#
# `actual` is a character vector ('yes')
# `expected` is a logical vector (TRUE)

expect_named(codes)          # Runs without error
expect_false(as.logical(0))  # Runs without error
```

Unit testing with **testthat**

Other such functions include `expect_error`, which checks that a function call throws an error when evaluated.

Analogously, there are `expect_warning`, `expect_message`, etc.

```
expect_error(3.14 + 'hello')  # Runs without error
expect_error(3.14 < 'hello')  # Throws an error:
```

```
# Error: `3.14 < "hello"` did not throw an error.
```

Back to our example

```
expect_true( is_plausible(5, min = 0, max = 10) )   # No error
expect_false( is_plausible(-1, min = 0, max = 1) )  # No error
expect_true( is_plausible('2', min = 0, max = 10) ) # Error:
```

```
# Error: is_plausible("2", min = 0, max = 10) is not TRUE
#
# `actual`:   FALSE
# `expected`: TRUE
```

Now decide:

[1] Fix the code just enough to satisfy the expectation,

[2] Or: have a rethink?

Back to our example

Fix to pass the test (not recommended here; will cause `warnings`):

```
is_plausible <- function(value, minimum = -Inf, maximum = Inf) {
  value <- as.numeric(value)
  value >= minimum & value <= maximum
}
expect_true( is_plausible('2', min = 0, max = 10) ) # No error
```

Or rethink, by expecting stricter handling of inputs:

```
is_plausible <- function(value, minimum = -Inf, maximum = Inf) {
  stopifnot(is.numeric(value))
  value >= minimum & value <= maximum
}
expect_error( is_plausible('2', min = 0, max = 10) ) # No error
```

Writing tests

```
test_that('Scalar integers', {
  expect_true(is_plausible(5, min = 0, max = 10))
  expect_false(is_plausible(-1, min = 0, max = 1))
})
```

```
# Test passed
```

```
test_that('Integer vectors', {
  expect_equal(is_plausible(0:10, min = 0, max = 10),
               rep(TRUE, 11))
  expect_equal(is_plausible(-1:2, min = 0, max = 1),
               c(F, T, T, F))
})
```

```
# Test passed
```

Writing tests

```
test_that('Handle unusual or missing inputs', {
  expect_error(is_plausible('2', min = 0, max = 10))
  expect_error(is_plausible(min = 0, max = 10))
  expect_error(is_plausible(2, min = "0", max = 10))
  expect_error(is_plausible(2, min = 0, max = "10"))
})
```

```
# -- Failure (<text>:4:3): Handle unusual or missing inputs ----------------
# `is_plausible(2, min = "0", max = 10)` did not throw an error.
#
# -- Failure (<text>:5:3): Handle unusual or missing inputs ----------------
# `is_plausible(2, min = 0, max = "10")` did not throw an error.
```

Unit testing workflows

- Add a `tests/` folder to an R package
  - run tests with `<Ctrl> + <Shift> + T`
  - tests will also run during `R CMD CHECK`
- Or run tests locally in analysis folder:
  - `test_file()` / `test_dir()`
  - Re-test with every edit: `auto_test()`
- R Markdown 'test chunks' with `error=TRUE` or **testrmd**

Note: most documentation for **testthat** assumes you are writing a package. Future talk: how to do analysis as an R package...

Test-driven development

[1] Write a test **before** any other code.

[2] Check that the test **fails**.

[3] Write **enough code to make it pass**.

[4] Add another test and iterate steps 1–3.

[5] *Refactor* the code, while ensuring it passes all tests.

Continuous integration

Continuous integration

In software engineering, **continuous integration (CI)** is the practice of merging all changes to a central repository, and automatically rebuilding & testing the code after every change.

- Use version control to track/manage changes
- Use CI software to automatically rerun/retest code
- Identify bugs/conflicts as soon as they're created

Tools: ~~Travis CI~~, GitHub Actions, Bitbucket Pipelines, Gitlab CI/CD, Jenkins

Continuous integration

Make life easier for yourself:

[1] Project metadata in a (dummy) `DESCRIPTION` file

[2] R code in `R/` subfolder

[3] Tests in a `tests/testthat/` subfolder

[4] Run tests quickly: `test_local()` / `<Ctrl>+<Shift>+T`

[5] Use Git for version control & CI for automatic testing

Minimal working `DESCRIPTION` file:

```
Package: blah
Version: 0.1
```

Example repo: https://github.com/Selbosh/unittesting

Thanks!

Based on a talk by Lewis Rendell
at Warwick R User Group, 2017.

CENTRE FOR EPIDEMIOLOGY VERSUS ARTHRITIS

Next meeting

Friday 3 December

Advent of Code discussion

https://adventofcode.com/

Attempt days 1–2 and share your approach

See also https://selbydavid.com/2020/12/06/advent-2020/