# Unit testing and continuous integration

## How to find bugs as soon as you create them

Lewis Rendell

Warwick R User Group, 11th May 2017

# Testing

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

# Unit testing

**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.

An alternative (not covered in this talk) is **RUnit**, by Matthias Burger, Klaus Jünemann and Thomas König.

# Expectation functions in **testthat**

These functions form the core of the testing framework. They all have the prefix `expect_`.

Several of these functions take an object to be tested as the *first argument*, and check some aspect of it against a reference value, given as the *second argument*.

If the two values **do not match**, an error is produced.

(If they do match, the function invisibly returns the object being tested.)

# Expectation functions in **testthat**

For example, the function `expect_equal` checks that the numerical value of the object is equal to a reference value (up to some tolerance, which may be given as a third argument).

```
a <- 3 + 2

expect_equal(a, 5)  # Runs without error

expect_equal(a, 6)  # Produces an error:
# Error: `a` not equal to 6.
# 1/1 mismatches
# [1] 5 - 6 == -1
```

# Expectation functions in **testthat**

Other such functions include:

- `expect_is`, which checks the class of an object.

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

- `expect_length`, which checks whether the object has a given length.

```
vec <- c(1, 7, 6, 4, 9)
expect_length(vec, 5)  # Runs without error
expect_length(vec, 8)  # Produces an error
```

# Expectation functions in **testthat**

Some functions only require one argument – the object to be tested.

For example, `expect_true` checks that the object evaluates to `TRUE`; similarly, `expect_false` checks for a `FALSE` evaluation.

```
root <- sqrt(64)

expect_true(root >= 0)  # Runs without error

expect_true(root == 1)  # Produces an error:
# Error: root == 1 isn't true.
```

Other such functions include `expect_error`, which checks that the object produces an error when it is evaluated. Analogously, there are `expect_warning`, `expect_message`, etc.

# Testing random functions

To test a function with random output, set the seed for the random number generation first using `set.seed`.

```
# Define a function that samples n numbers,
# with replacement, from (1, 2, ..., 10).
sampler <- function(n) {
  sample(1:10, n, replace = TRUE)
}

set.seed(123)       # Set random seed
foo <- sampler(3)
foo
# [1] 3 8 5

expect_equal(foo, c(3, 8, 5))  # Runs without error
```

# Useful functions when writing tests

`dput` (a base R function) takes any object and prints out the code required to recreate it.

```
set.seed(246)
dput(runif(4))
# c(0.703108243644238, 0.205851259641349,
# 0.599153293762356, 0.288029342656955)
```

This can be useful for specifying the reference value against which an object should be tested.

# Useful functions when writing tests

`capture.output` (from the package **utils**) takes any expression, and returns a character vector containing the text that would be printed to the console if it were run.

```
# Define a function that prints "Hello, world!".
hello <- function() {
  cat("Hello, world!")
}

hello()
# Hello, world!
capture.output(hello())
# [1] "Hello, world!"
```

# Adding tests to a package

When creating an R package, unit tests can be included. These will be run automatically whenever the package is built and checked (using `R CMD check`, for example).

In order to do so, a framework for the tests must be created. The correct framework can be created using the `use_testthat` function from the **devtools** package. This creates a directory in the package called 'tests', which contains:

- An R script called 'testthat.R', which contains the code for running the tests;
- A directory called 'testthat'. All tests must be contained in R scripts in this directory, with file names beginning 'test'.

# Writing tests

Within **testthat**, a *test* is a series of expectation functions, which collectively test one small unit of functionality.

A test is created using the `test_that` function. The first argument is the name of the test (as a character string), while the second argument contains the test code, including all of the expectation statements, between curly brackets.

When this function is run, the test code will be evaluated in its own environment. An error will be produced if any of the test code produces an error.

# Writing test scripts

Tests that check related functionality should all be put in the same R script. The `context` function should be used in the first line of the file, to provide a description for the group of tests that follows.

Each script should be saved in the 'testthat' directory with a file name beginning 'test'.

# An example test script

```
context("Function 'sampler' works correctly")

test_that("Function 'sampler' correctly samples
          with replacement from 1:10", {
  set.seed(123)
  expect_equal(sampler(3), c(3, 8, 5))

  set.seed(999)
  expect_equal(sampler(4), c(4, 6, 1, 9))
})

test_that("Function 'sampler' gives error if
          argument is not a positive number", {
  expect_error(sampler(-1))
  expect_error(sampler(FALSE))
})
```

# Running the tests

To run the tests, the `test` function from the **devtools** package may be used.

When checking a built package using `R CMD check`, any tests included in the package will be run automatically, alongside all other usual package checks. If any tests fail, an error message will be produced.

# Continuous integration

Version control systems are applications that track and manage changes to source code. This presentation shall consider only Git, and its web-based hosting service GitHub.

In software engineering, **continuous integration** (**CI**) is the practice of regularly merging all changes to source code to a central repository, building and testing the source code after every change. This allows any bugs that are introduced to be identified quickly.

# Continuous integration and R packages

Why use continuous integration software when developing an R package?

Every time a change is committed to the version control repository, this triggers the CI software to rebuild and check the package. This will include the running of any unit tests.

As such, all unit tests are run *automatically* every time any change is made to the source code – so any bugs can be identified as soon as they are created.

# Continuous integration with GitHub

**Travis CI** (travis-ci.org) is a continuous integration service for use with GitHub code repositories. For open-source projects it is free to use.

Alternative CI software exists for use with GitHub, and with alternative version control systems and hosting services (for example, Jenkins).

# How to use Travis CI

- Ensure your package is on GitHub, in the correct format so that it can be checked.
- Create an account at travis-ci.org.
- Activate the GitHub repository containing your package.
- Add a .travis.yml file to the root directory of your package – the `use_travis` function in the **devtools** package will do this.
- Commit this new file to your Git repository.

From now on, whenever any commits are pushed to your online GitHub repository, Travis will automatically build and check your package. Travis will notify you if the check results in any errors, including those generated by failed unit tests.

# Summary

- Testing is often done manually, in an *ad hoc* way; unit testing formalises this, allowing testing to be carried out in an automated way.
- Continuous integration software causes such tests to be carried out every time changes to source code are committed, allowing bugs to be identified as soon as they are created.
- This can make the process of writing code much easier, and gives assurances to the end user that everything works as the author intended.