## Testing

#### **Hadley Wickham**

Assistant Professor / Dobelman Family Junior Chair Department of Statistics / Rice University





- 1. Motivation
- 2. Overview
- 3. Expectations
- 4. Tests
- 5. Context
- 6. Running tests

# Motivation

# You already know how to debug

- traceback() tells you where the problem is
- browser() starts an interactive debugger where it's called
- options(error = recover) starts
   interactive debugger automatically on error
- options(warn = 2) turns warnings into errors so you can find them more easily

#### Automated tests

How do you keep bugs from coming back?

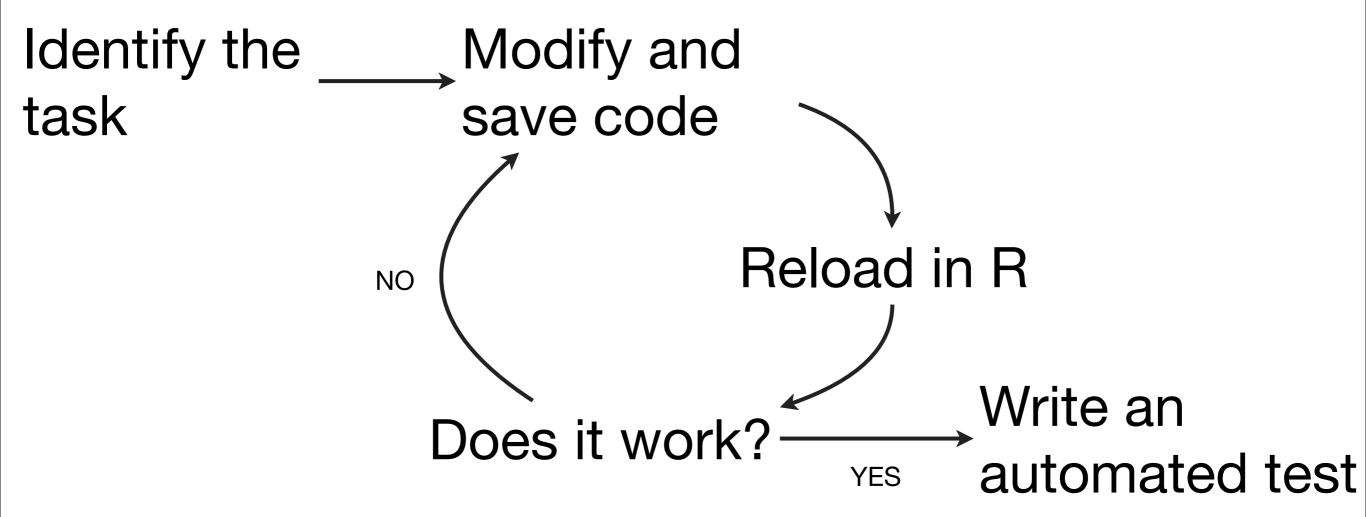
You can't manually check every function every time you make a change – it takes too long

Solution: automate your testing so you can quickly run tests after every change

#### **Confirmatory programming**

Write an Modify and save code automated test Reload in R NO Does it work? YES You're done

#### **Exploratory programming**



#### Other benefits

- Code that can be tested easily, often has a better, more modular, design
- When you stop working, leave a test failing. You'll know what to work on when you come back
- Make big changes without fear of accidentally breaking anything

## Testing packages

- RUnit
- svUnit
- testthat

### Why test that?

- Easy transition from informal to formal tests. Can be used in wide variety of situations
- Wide range of expectations/assertions
- Fun, colourful output that keeps you motivated

https://github.com/hadley/devtools/wiki/Testing

# 

## Key components

- Expectations: what do you expect a function to do?
- Tests: a group of expectations that tests a small piece
- Contexts: a group of tests that tests behaviour of a large piece of functionality (function, class, etc)

```
context("String length")
test_that("str_length is number of characters", {
 expect_that(str_length("a"), equals(1))
 expect_that(str_length("ab"), equals(2))
 expect_that(str_length("abc"), equals(3))
})
test_that("str_length of missing string is missing", {
 expect_that(str_length(NA), equals(NA_integer_))
 expect_that(str_length(c(NA, 1)), equals(c(NA, 1)))
 expect_that(str_length("NA"), equals(2))
})
test_that("str_length of factor is length of level", {
 expect_that(str_length(factor("a")), equals(1))
 expect_that(str_length(factor("ab")), equals(2))
 expect_that(str_length(factor("abc")), equals(3))
})
```

#### Test structure

- Informal: just use expectations casually
- Structured: organised tests in files and directories
- Package: in inst/tests with test/run-all.r

### Running tests

- testthat::test\_file(path)
   Run all tests in a single file.
- testthat::test\_dir(path)
   Run all files starting with test- in a directory
- testthat::test\_package(installed\_package)
   Run all tests in an installed package. Tests run in package namespace
- devtools::test(package)
   Run all tests in a development package.

### Example packages

- · testthat, stringr, plyr, lubridate
- MSnbase
- Reverse suggests from: http://cran.rproject.org/web/packages/testthat/ index.html

#### Your turn

Download the source for these packages.

Where are the tests? How are they structured?

# Expectations

Expectation	Test	Abbreviation
equals	all.equals	expect_equal
is_identical_to	identical	expect_identical
is_equivalent_to	all.equals, check.attributes = FALSE	expect_equivalent
is_a	inherits	expect_is
is_true / is_false	identical	expect_true / expect_false

#### Your turn

Write down your expectations of output from nchar(). Try using character vectors, factors, missing values, escaped strings.

```
expect_that(nchar("a"), equals(1))
expect_equal(nchar("a"), 1)
expect_equal(nchar("abc"), 3)
expect_equal(nchar("\"\"\""), 3)
# But remember we want to avoid repetition!
check_length <- function(string, n) {</pre>
  expect_equal(nchar(string), n)
check_length("a", 1)
check_length("abc", 3)
check_length("\"\"\"", 3)
```

```
# However errors aren't very informative
check_length(NA, NA)
# But all expect functions take extra arguments for
# just this reason (or you could use substitute +
# eval)
check_length <- function(string, n) {</pre>
  expect_equal(nchar(string), n,
    label = paste("nchar(", string, ")", sep = ""),
    expected.label = n)
check_length(NA, NA)
check_length(factor("a"), 1)
check_length(factor("abc"), 3)
```

Expectation	Test	Abbreviation
matches	grepl + any	expect_matches
prints_text	matches applied to output	expect_output
shows_message	matches applied to messages	expect_message
gives_warning	matches applied to warnings	expect_warning
throws_error	matches on errors	expect_error

# 

```
# Tests collect related expectations and name them
test_that("str_length of missing string is missing", {
  expect_that(str_length(NA), equals(NA_integer_))
  expect_that(str_length(c(NA, 1)), equals(c(NA, 1)))
  expect_that(str_length("NA"), equals(2))
})
test_that("nchar of missing string is missing", {
  expect_that(nchar(NA), equals(NA_integer_))
  expect_that(nchar(c(NA, 1)), equals(c(NA, 1)))
  expect_that(nchar("NA"), equals(2))
})
```

```
# But real advantage is when you put them in
# a file, and run all at once.
library(testthat)
test_file("7-nchar-test.r")
# Green . = passing test
# Red number = failing test (or error)
# Numbers index list of all failed expectation
# giving message and test name.
```

#### Your turn

Create a file that tests tapply. Test:

What happens when index and group vectors are different lengths (or require recycling).

What happens with length 1 or length 0 inputs?

If group = seq\_along(index), does output =
input?

```
# As far as I can tell, there's only one test for
# tapply in the R source:
tapply(character(0), factor(letters)[FALSE], length)
# And in my opinion this output is incorrect. It
# should be a vector of 0s.
pieces <- split(character(), factor(letters)[FALSE])</pre>
vapply(pieces, length, integer(1))
# Not quite as worrying as it might seem, because
# R-core tends to rely on output tests...
```

### Output tests

For each .R in tests/, a .Rout file is created. If a .Rout.save file exists, then the two are compared, and any differences cause an error

Useful for testing large scale behaviour. But very sensitive to output format.

# Contexts

```
context("My context")

# Used to label a file

# Up to you how to arrange tests in files

# But keep related tests together.
```

# Ruming tests

### Running tests

- testthat::test\_file(path)
   Run all tests in a single file.
- testthat::test\_dir(path)
   Sources helper-\*.R then runs all tests in test-\*.R
- testthat::test\_package(installed\_package)
   Run all tests in an installed package. Tests run in package namespace
- devtools::test(package)
   Run all tests in a development package.

```
# Casually, during development
load_all("stringr")
test("stringr")

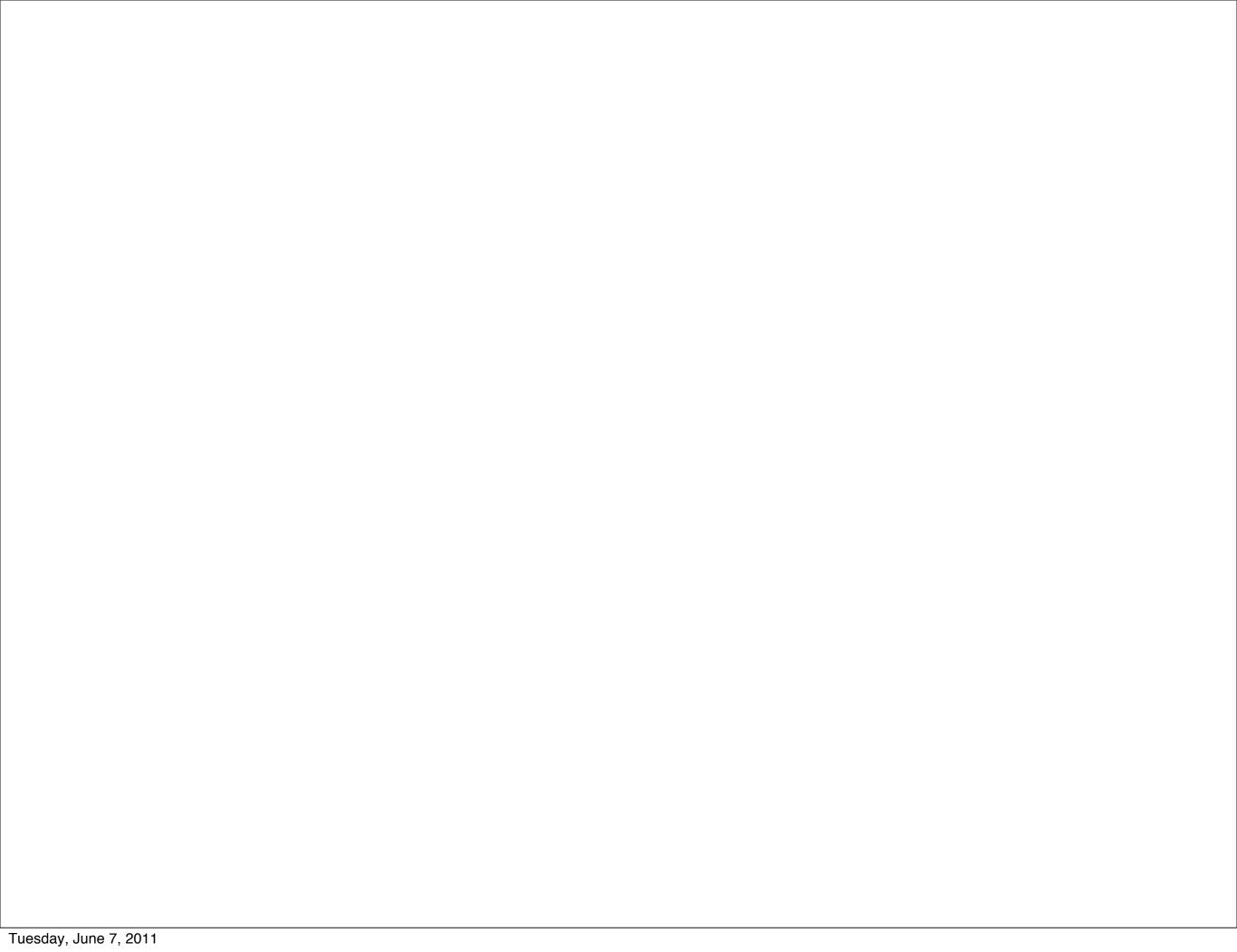
# More formally
install("stringr")
reload("stringr")
test_package("stringr")
```

### Package tests

- Store all tests in inst/tests so they are installed with the package. Then users can run to check their installation/OS is ok
- Include the following code in tests/ test-all.R (note capital R). This ensures R CMD check will not pass unless all tests pass

```
library(testthat)
# This loads the version being tested
library(stringr)

test_package("stringr")
```



This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.