Advanced testing

Drawing from newly updated
https://r-pkgs.org/tests.html

September 2023

1. Improving your test quality
2. Testing challenging functions

Improving test quality

1. Support interactive debugging
2. Leave the world the way you found it
3. Extract out duplicated code

Support interactive debugging

What makes these tests hard to run interactively?

```
dat ← data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))

skip_if(today_is_a_monday())

test_that("foofy() does this", {
  expect_equal(foofy(dat), … )
})

dat2 ← data.frame(x = c("x", "y", "z"), y = c(4, 5, 6))

skip_on_os("windows")

test_that("foofy2() does that", {
  expect_snapshot(foofy2(dat, dat2)
})
```

You have to carefully run code outside of each test

```
dat ← data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))

skip_if(today_is_a_monday())

test_that("foofy() does this", {
  expect_equal(foofy(dat), ... )
})

dat2 ← data.frame(x = c("x", "y", "z"), y = c(4, 5, 6))

skip_on_os("windows")

test_that("foofy2() does that", {
  expect_snapshot(foofy2(dat, dat2)
})
```

To avoid code outside of test_that():

- Move file-scope logic to either narrower scope (just this test) or a broader scope (all files).
- It's ok to copy and paste: test code doesn't have to be super DRY. Obvious >>> DRY

Deodorizing the previous example

```
test_that("foofy() does this", {
  skip_if(today_is_a_monday())

  dat <- data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))

  expect_equal(foofy(dat), ... )
})

test_that("foofy() does that", {
  skip_if(today_is_a_monday())
  skip_on_os("windows")

  dat <- data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))
  dat2 <- data.frame(x = c("x", "y", "z"), y = c(4, 5, 6))

  expect_snapshot(foofy(dat, dat2))
})
```

Deodorizing the previous example

```
test_that("foofy() does this", {
  skip_if(today_is_a_monday())

  dat ← data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))

  expect_equal(foofy(dat), ... )
})
```

Deodorizing the previous example

```
test_that("foofy() does that", {
  skip_if(today_is_a_monday())
  skip_on_os("windows")

  dat <- data.frame(x = c("a", "b", "c"), y = c(1, 2, 3))
  dat2 <- data.frame(x = c("x", "y", "z"), y = c(4, 5, 6))

  expect_snapshot(foofy(dat, dat2))
})
```

Leave the world the way you found it

Your tests should leave the world the way they found it

```
test_that("side-by-side diffs work", {
  options(width = 20)
  expect_snapshot(
    waldo::compare(c("X", letters), c(letters, "X"))
  )
})
```

Your tests should leave the world the way they found it

```
test_that("side-by-side diffs work", {
  withr::local_options(width = 20)
  expect_snapshot(
    waldo::compare(c("X", letters), c(letters, "X"))
  )
})
```

Instead of this	do this
`options()`	`withr::local_options()`
`Sys.setenv()`	`withr::local_envvar()`
`setwd()`	`withr::local_dir()`
`Sys.setlocale()`	`withr::local_locale()`
`tempfile()`	`withr::with_tempfile()`

This is particularly important when creating files

- You should only write files inside the session temp directory.
- Do not write into your package's tests/ directory.
- Do not write into the current working directory.
- Do not write into the user's home directory.

You should still clean up after yourself even in the temp dir

```
test_that("can read from file name with utf-8 path", {
  path <- withr::local_tempfile(
    pattern = "Universit\u00e0-",
    lines = c("#' @include foo.R", NULL)
  )
  expect_equal(find_includes(path), "foo.R")
})
```

What about code before you knew how to do this right?

```
test_that("side-by-side diffs work", {
  options(width = 20)
  expect_snapshot(
    waldo::compare(c("X", letters), c(letters, "X"))
  )
})

# It used to be diabolically hard to figure out
# where you accidentally did this
```

Can set up a state inspector

```
set_state_inspector(function() {
  list(
    options = options(),
    envvars = Sys.getenv()
  )
})

# Run before and after every test and warns if
# there's a difference. Put in setup.R
```

Others values are useful for R CMD check

```
set_state_inspector(function() {
  list(
    connections = nrow(showConnections()),
    temp = dir(tempdir()),
    home = dir("~")
  )
})
```

Extract out repeated code

Common source of duplicated code

- Repeated test data
- Commonly used skip functions
- Custom expectations
- Repeated local_*() functions

Most data functions are straightforward

```
my_data ← function() {
  data.frame(x = 1, y = 2, z = 3)
}
```

Although for bigger data you might use a fixture

```
my_data <- function() {
  readRDS(test_path("fixtures", "data.rds"))
}
```

Just make sure it’s reproducible!

```
.
├── ...
└── tests
    ├── testthat
    │   ├── fixtures
    │   │   ├── make-useful-things.R
    │   │   ├── useful_thing1.rds
    │   │   └── useful_thing2.rds
    │   ├── helper.R
    │   ├── setup.R
    │   └── (all the test files)
    └── testthat.R
```

Most skip functions are straightforward

```
my_skip <- function() {
  skip_on_ci()
  skip_on_windows()
  skip_if(as.POSIXlt(Sys.time())$wday %in% 6:7)
}
```

But local functions are more challenging

```
my_local <- function() {
  withr::local_options(rlang_interactive = FALSE)
  withr::local_envvar(MY_ENVVAR = "xyz")
}


local({
  my_local()
  getOption("rlang_interactive")
})
```

Need to capture execution environment

```
my_local <- function(env = parent.frame()) {
  withr::local_options(
    rlang_interactive = FALSE,
    .local_envir = env
  )
  withr::local_envvar(
    MY_ENVVAR = "xyz",
    .local_envir = env
  )
}

# The name varies a bit from function to function but always
# includes env or frame.
```

Where should these functions live?

```
.
├── ...
├── R/
│   ├── x.R                 load_all()
│   └── y.R                 load_all()
└── tests/
    ├── testthat.R          R CMD check
    └── testthat/
        ├── test-x.R        test()
        ├── test-y.R        test()
        ├── helper-1.R      load_all() + test()
        ├── helper-2.R      load_all() + test()
        └── setup.R         test()
```

Most of these are run even when testing a single file

```
.
├── ...
├── R/
│   ├── x.R
│   └── y.R
└── tests/
    ├── testthat.R
    └── testthat/
        ├── test-x.R        test_file()
        ├── test-y.R
        ├── helper-1.R      test_file()
        ├── helper-2.R      test_file()
        └── setup.R         test_file()
```

Helpers in the tidyverse

https://github.com/tidyverse/readxl/blob/3aa8c2ddf9f1d8921f2a8b42ae0bdfa69a22ed9b/tests/testthat/helper.R#L2

https://github.com/tidyverse/haven/blob/9b3b21b5e9b64867eb53818faa7e9a22480f347d/tests/testthat/helper-roundtrip.R#L4

https://github.com/tidyverse/dbplyr/blob/9ff37e05e83aebd9d196ed5c8d198c4681d80a18/tests/testthat/helper-src.R#L4

Always worth considering if would be better in R/

- Can use in vignettes
- Can test it
- Can document and export it
- Stricter R CMD check checks

Challenging functions

Challenging functions include:

- Dependencies on other state — withr::local_options()/withr::local_envvar() Earlier
- Errors & other user facing tests — snapshots Earlier
- Interactivity — mocking Next
- Output affected by RNG — withr::local_seed()
- HTTP responses — httr2 mocking + httptest2
- Graphical output — https://vdiffr.r-lib.org/

Mocking

How do you test code that requires user interaction?

```
httr2:::oauth_flow_auth_code_read("abcd")
#> Enter authorization code:
#> Enter state parameter:
```

Mocking functions can be simple

```
test_that("JSON-encoded authorisation codes can be input manually", {
  state <- base64_url_rand(32)
  input <- list(state = state, code = "abc123")
  encoded <- openssl::base64_encode(jsonlite::toJSON(input))
  local_mocked_bindings(
    read_line = function(prompt = "") encoded
  )
  expect_equal(oauth_flow_auth_code_read(state), "abc123")
  expect_error(oauth_flow_auth_code_read("invalid"), "state does not match")
})
```

Or more complex

```
test_that("bare authorisation codes can be input manually", {
  state <- base64_url_rand(32)
  sent_code <- FALSE
  local_mocked_bindings(
    read_line = function(prompt = "") {
      if (sent_code) {
        state
      } else {
        sent_code <<- TRUE
        "zyx987"
      }
    }
  )
  expect_equal(oauth_flow_auth_code_read(state), "zyx987")
  expect_error(oauth_flow_auth_code_read("invalid"), "state does not match")
})
```

How would you test this function?

```
library(rlang)
check_installed <- function(package) {
  if (is_installed(package)) {
    return(invisible())
  }

  stop("Please install '", package, "' before continuing")
}

# Very loosely inspired by rlang::check_installed()
```

Your turn

- Make a new package, e.g. usethis::create_package("~/desktop/practice")
- Add an rlang dependency & import is_installed.
- Create an R file and copy in the code from the last slide.
- Test it interactively
- Try to use local_mocked_bindings() to elicit both states in a test
- Use code coverage to check your work

```
test_that("errors if package not found", {
  local_mocked_bindings(is_installed = function(package) FALSE)
  expect_error(check_installed("foo"))
})

test_that("errors if package not found", {
  local_mocked_bindings(is_installed = function(package) TRUE)
  expect_invisible(check_installed("foo"))
})
```

Mocking challenges

Challenge: mocking a function called with ::

Solution: import it instead

Challenge: mocking a function from the base package

Solution: ensure you have basefunction <- NULL in your package

See ?local_mocked_bindings for more details

Case study: is_interactive()

```
is_interactive <- function() {
  opt <- options("rlang_interactive")
  if (!is.null(opt)) {
    return(opt)
  }

  is_knitting <- isTRUE(options("knitr.in.progress"))
  if (is_knitting) {
    return(TRUE)
  }

  is_testing <- identical(Sys.getenv("TESTTHAT"), "true")
  if (is_testing) {
    return(TRUE)
  }

  interactive()
}
```

This makes it much easier to affect behaviour in tests

```
withr::local_options(rlang_interactive = TRUE)

# vs
local_mocked_bindings(
  interactive = function() FALSE
)
# plus interactive <- NULL
```