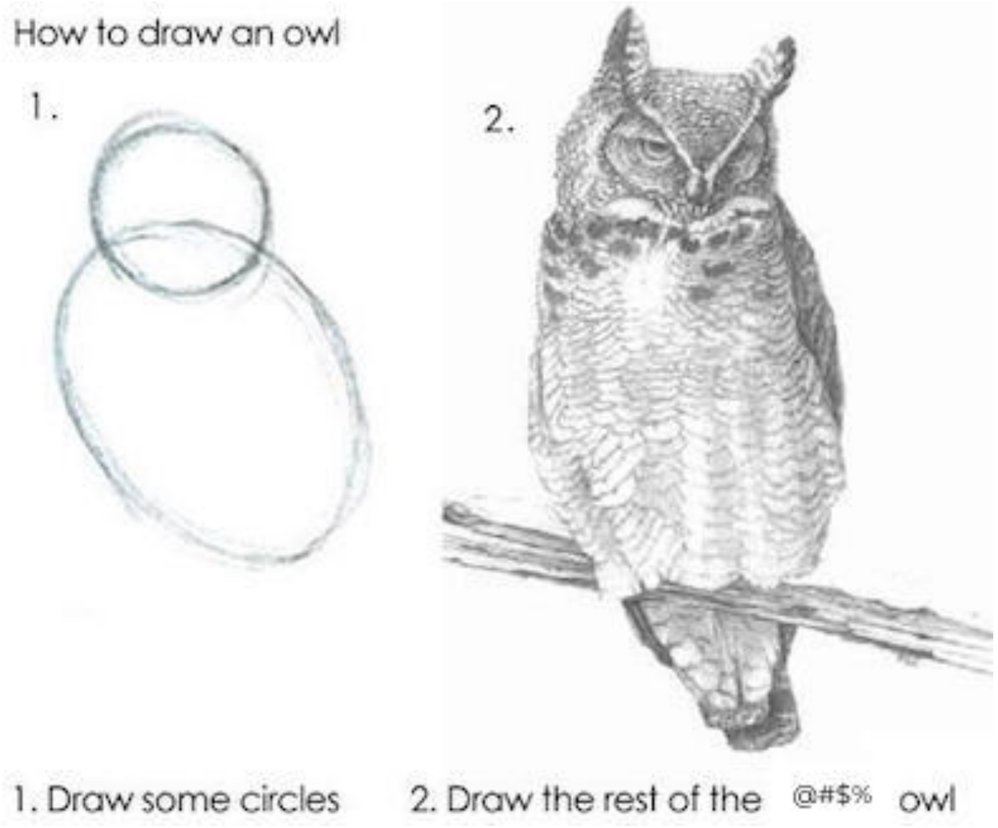
Package Development:

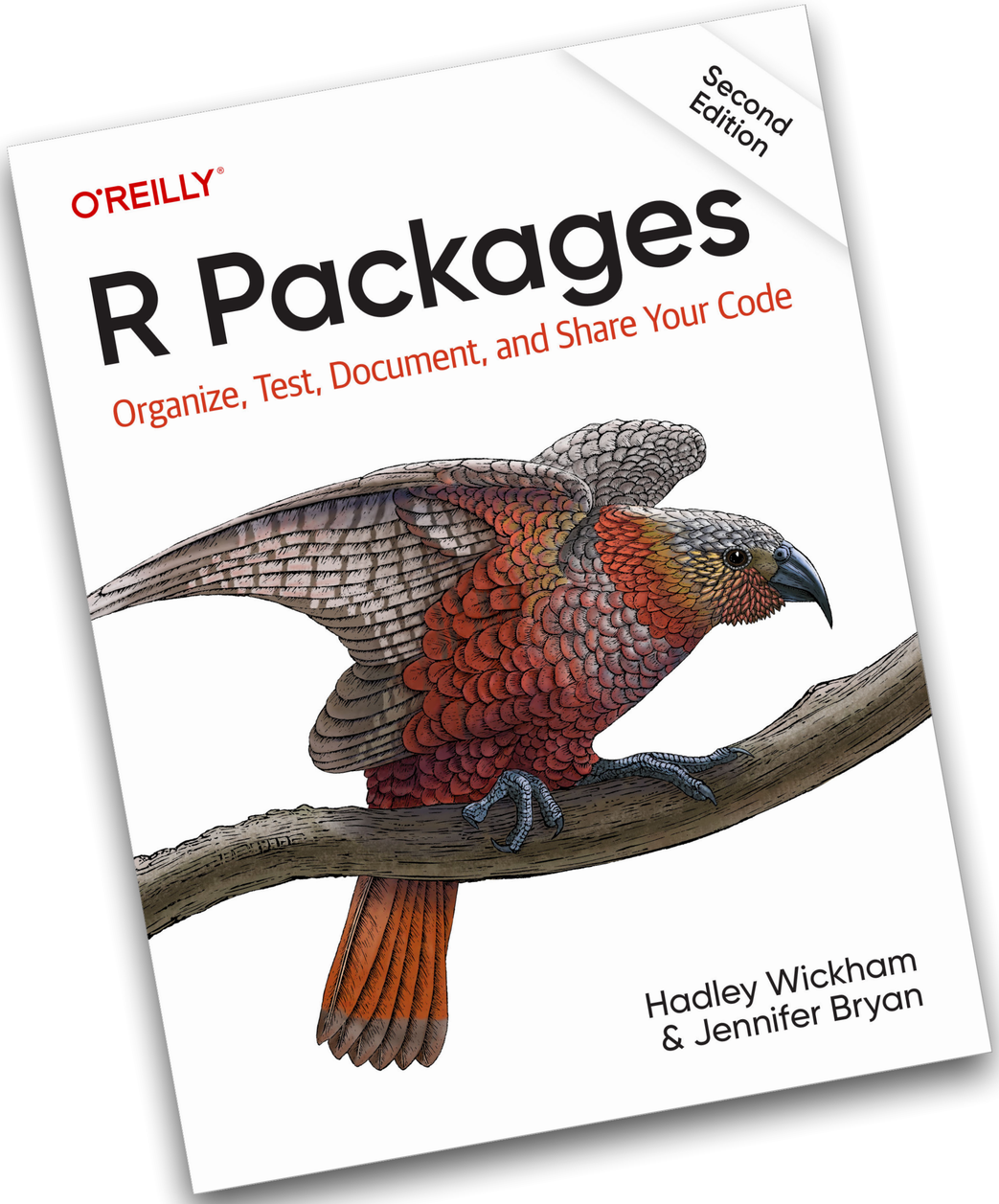
The Rest of the Owl

GitHub repo: pos.it/pkg_dev_24

Wifi: Posit Conf 2024 | conf2024

Discord: #workshop-pkg-dev





https://r-pkgs.org/


test
that

testthat setup: once per package*

```
# in a brand new package
use_testthat()

# switching to testthat 3e in an existing package
use_testthat(3)
```

* OK, technically once per package, per testthat edition

testthat 3e

- Snapshot tests
- Lots of deprecations, relative to legacy testthat
- `expect_equal()` and friends use waldo package
- Parallel testing
- More details in this article:
  - https://testthat.r-lib.org/articles/third-edition.html
- And in this blog post:
  - https://www.tidyverse.org/blog/2022/02/upkeep-testthat-3/

Maybe the most exciting thing?
Snapshots are pretty great, too.

waldo is great at reporting a difference

```
x1 ← x2 ← list(list(a = 1, b = 2, c = list(4, 5, list(6, 7))))
x2[[1]]$c[[3]][[2]] ← 10

waldo::compare(x1, x2)
#> `old[[1]]$c[[3]][[2]]`:  7
#> `new[[1]]$c[[3]][[2]]`: 10
```

testthat 2e uses `all.equal()`

```
library(testthat)
local_edition(2)

expect_equal(x1, x2)

#> Error:
#> ! `x1` not equal to `x2`.
#> Component 1: Component 3: Component 3:
#>   Component 2: Mean relative difference: 0.4285714
```

testthat 3e uses `waldo::compare()`

```
local_edition(3)
expect_equal(x1, x2)

#> Error:
#> ! `x1` (`actual`) not equal to `x2` (`expected`).
#>
#>   `actual[[1]]$c[[3]][[2]]`:  7
#> `expected[[1]]$c[[3]][[2]]`: 10
```

Create or navigate to a test file

```
use_test("whatever")

# in RStudio or Positron, with a R/*.R file open,
# target test file can be inferred
use_test()

# use_test() is half of a matched pair:
use_r()
```

`use_test()` + `use_r()` vibe with file pairs

`R/a.R`	`tests/testthat/test-a.R`
`R/b.R`	`tests/testthat/test-b.R`
`R/c.R`	`tests/testthat/test-c.R`
`R/data.R`	

`load_all()`

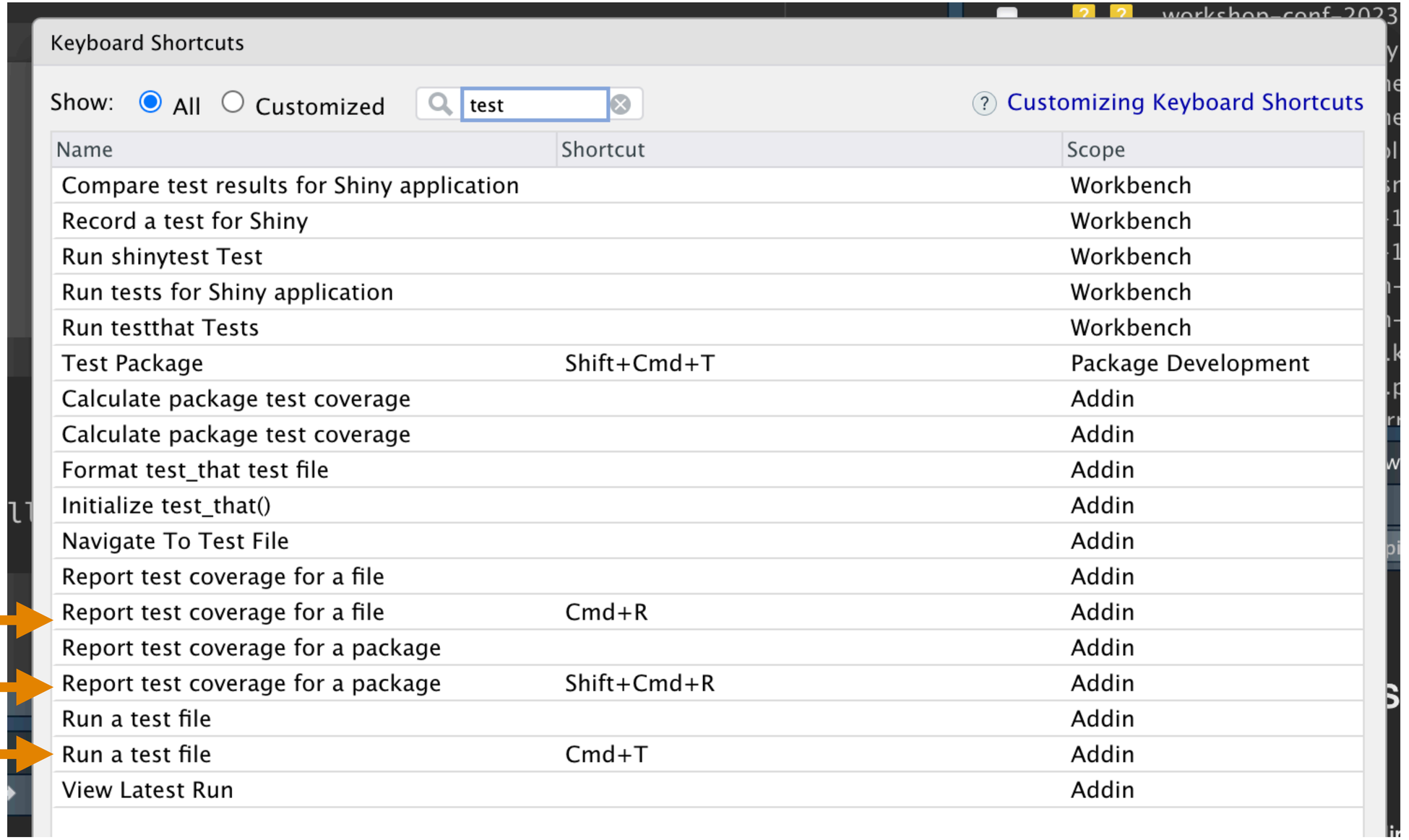
- testthat's workflow is designed around `load_all()`
- Makes entire package namespace available
- Attaches testthat
- Sources `tests/testthat/helper.R`

Workflow: micro-iteration, interactive experimentation

```
# tweak the foofy() function and re-load it
# also attach testthat and source test helpers
load_all()

# interactively explore and refine expectations
# and tests
expect_equal(foofy( ... ), EXPECTED_FOOFY_OUTPUT)

testthat("foofy does good things", { ... })
```

Workflow: mezzo-iteration, whole test file

```
test_file("tests/testthat/test-foofy.R")


# in RStudio, with test or R file focused
test_active_file()
test_coverage_active_file()
# consider binding these to Cmd + T, Cmd + R
```

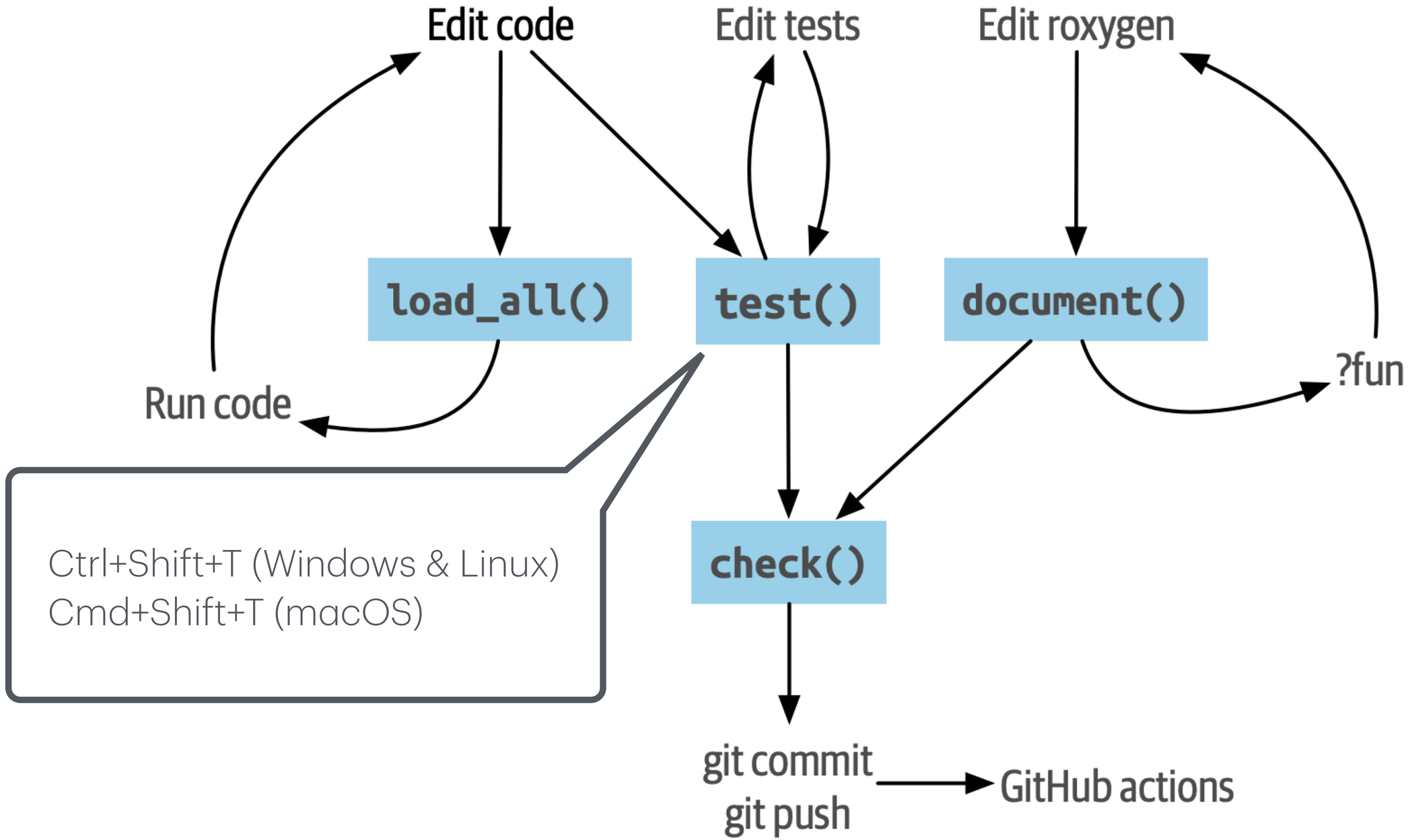

Keyboard Shortcuts
Show: All Customized test
Customizing Keyboard Shortcuts
Name Shortcut Scope
Compare test results for Shiny application Workbench
Record a test for Shiny Workbench
Run shinytest Test Workbench
Run tests for Shiny application Workbench
Run testthat Tests Workbench
Test Package Shift+Cmd+T Package Development
Calculate package test coverage Addin
Calculate package test coverage Addin
Format test_that test file Addin
Initialize test_that() Addin
Navigate To Test File Addin
Report test coverage for a file Addin
Report test coverage for a file Cmd+R Addin
Report test coverage for a package Addin
Report test coverage for a package Shift+Cmd+R Addin
Run a test file Addin
Run a test file Cmd+T Addin
View Latest Run Addin

Workflow: macro-iteration, all files

```
test()
```

```
test_coverage()
```

```
check()
```


Edit code
Edit tests
Edit roxygen
load_all()
test()
document()
Run code
?fun
Ctrl+Shift+T (Windows & Linux)
Cmd+Shift+T (macOS)
check()
git commit
git push
GitHub actions

Test suite design principles

- A test should be self-sufficient and self-contained.
- The interactive workflow is important.
- Obvious >>> DRY
- Don't let a nonstandard workflow "leak".

Test smell: top-level code that's outside `test_that()`

```
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)
})
```

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)
})
```

Move file-scope logic to a narrower scope (as done here) or a broader scope (coming soon).

It's OK to repeat yourself!

Leave the world the way you found it

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

withr's `local_*()` functions are super useful for making changes that are scoped to a single `test_that()`.

Functions to avoid in your tests

```
library(somedependency)
```

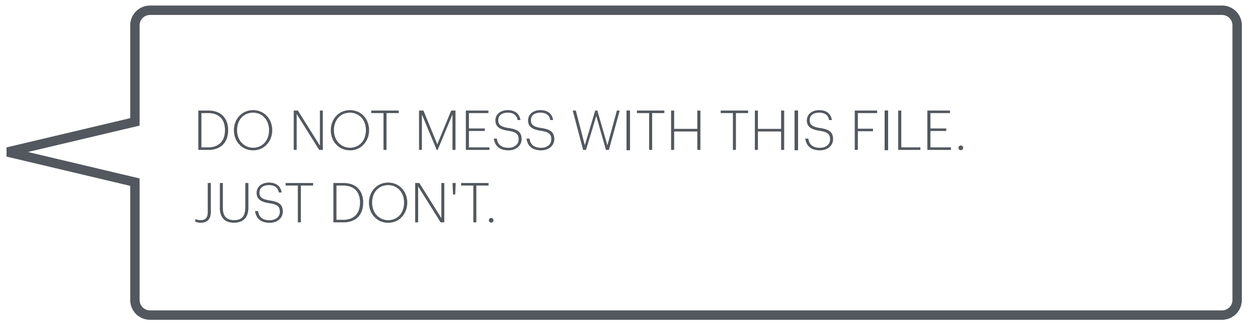
Please no! Access functions from your dependencies, in your tests, exactly as you do below R/.

```
source("stuff-thats-handy-for-your-tests.R")
```

Please no! Unexported helper functions and test helper files are a better mechanism for this.

Files relevant to testing: `tests/testthat.R`

```
library(testthat)
library(abcde)

test_check("abcde")
```



Files relevant to testing: `R/*.R`

```
.
├── ...
└── R
    ├── ...
    ├── test-helpers.R
    ├── test-utils.R
    ├── utils-testing.R
    └── ...
```

Test helpers can be internal (i.e. unexported) functions in your package.

Good example of relocating file-scope logic to a broader scope.

Files relevant to testing: `tests/testthat/helper.R`

```
.
├── ...
└── tests
    ├── testthat
    │   ├── helper.R
    │   ├── helper-blah.R
    │   ├── helper-foo.R
    │   ├── test-foofy.R
    │   └── (more test files)
    └── testthat.R
```

Test helper files are executed by `load_all()` and at the start of automated testing.

Another good example of relocating file-scope logic to a broader scope.

Files relevant to testing: everything else

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

Sometimes test fixtures are useful.
Keep the code to (re-)create them!

Setup files are good for certain types of setup + teardown.

I shall not today attempt further to define "hard-core pornography", and perhaps I could never succeed in intelligibly doing so.

But I know it when I see it, and the motion picture involved in this case is not that.

US Supreme Court Justice Potter Stewart

I shall not today attempt further to define this test's expected result, and perhaps I could never succeed in intelligibly doing so.

But I know it when I see it, and the actual result we're getting today is not that.

Your failing snapshot test

Big idea of snapshot tests

- Expected result is captured once and stored as a file.
- Future test runs compare current result to the snapshot file.
- Especially suitable for, e.g., testing messages, print methods, and errors.

Example: how waldo reports differences

```
withr::with_options(
  list(width = 20),
  waldo::compare(c("X", letters), c(letters, "X"))
)
#>     old | new
#> [1] "X" -
#> [2] "a" | "a" [1]
#> [3] "b" | "b" [2]
#> [4] "c" | "c" [3]
#>
#>      old | new
#> [25] "x" | "x" [24]
#> [26] "y" | "y" [25]
#> [27] "z" | "z" [26]
#>          - "X" [27]
```

Snapshot test of the example

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

New snapshot file! Warning is normal

```
── Warning (test-diff.R:63:3): side-by-side diffs work ─────────────────────
Adding new snapshot:
Code
  waldo::compare(c(
    "X", letters), c(
    letters, "X"))
Output
      old | new
  [1] "X" -
  [2] "a" | "a" [1]
  [3] "b" | "b" [2]
  [4] "c" | "c" [3]

       old | new
  [25] "x" | "x" [24]
  [26] "y" | "y" [25]
  [27] "z" | "z" [26]
           - "X" [27]
```

One-off execution of a snapshot test doesn't "work"

```
Can't compare snapshot to reference when testing interactively
ℹ Run `devtools::test()` or `testthat::test_file()` to see changes
```

It is harmless to execute snapshot tests interactively.

But it's a no-op.

No snapshot recording or comparison happens.

Snapshot tests only "work" in automated test runs

```
test_active_file()

test()

check()
```

Snapshot test recording and comparison only happens when the tests are being run at arms-length, via an automated process.

Specifically: running an entire test file or running the entire test suite.

When snapshot tests fail

```
── Failure (test-diff.R:63:3): side-by-side diffs work─────────────────────────
Snapshot of code has changed:
old[3:15] vs new[3:15]
  "    \"X\", letters), c("
  "    letters, \"X\"))"
  "Output"
- "      old | new    "
+ "      OLD | NEW    "
  "  [1] \"X\" -        "
  "  [2] \"a\" | \"a\" [1]"
  "  [3] \"b\" | \"b\" [2]"
  "  [4] \"c\" | \"c\" [3]"
  "  "
- "       old | new     "
+ "       OLD | NEW     "
and 3 more ...

* Run `snapshot_accept('diff')` to accept the change
* Run `snapshot_review('diff')` to interactively review the change
```

Notice that this print method switched from "old" and "new" to "OLD" and "NEW".

Reacting to snapshot test failure (change, really)

```
* Run `snapshot_accept('diff')` to accept the change
* Run `snapshot_review('diff')` to interactively review the change
```

snapshot_review() launches a nifty Shiny app when run inside RStudio.

Key arguments to `expect_snapshot()`

1. `cran = TRUE/FALSE`
2. `error = TRUE/FALSE`
3. `transform`
4. `variant`

Big Idea	Featured implementation
Make it easy to see what's changed	testthat 3e: waldo, snapshots

Run tests often, at appropriate scale	`load_all()` + interactive tinkering
	`test_active_file()`
	`test()`, `check()`

Use an interactive workflow that doesn't undermine automated testing	`load_all()` `test helpers`
Avoid (at least minimize) test code outside of test_that()	`test helpers` `test fixtures`
Leave the world as you found it	`withr::local_*()`