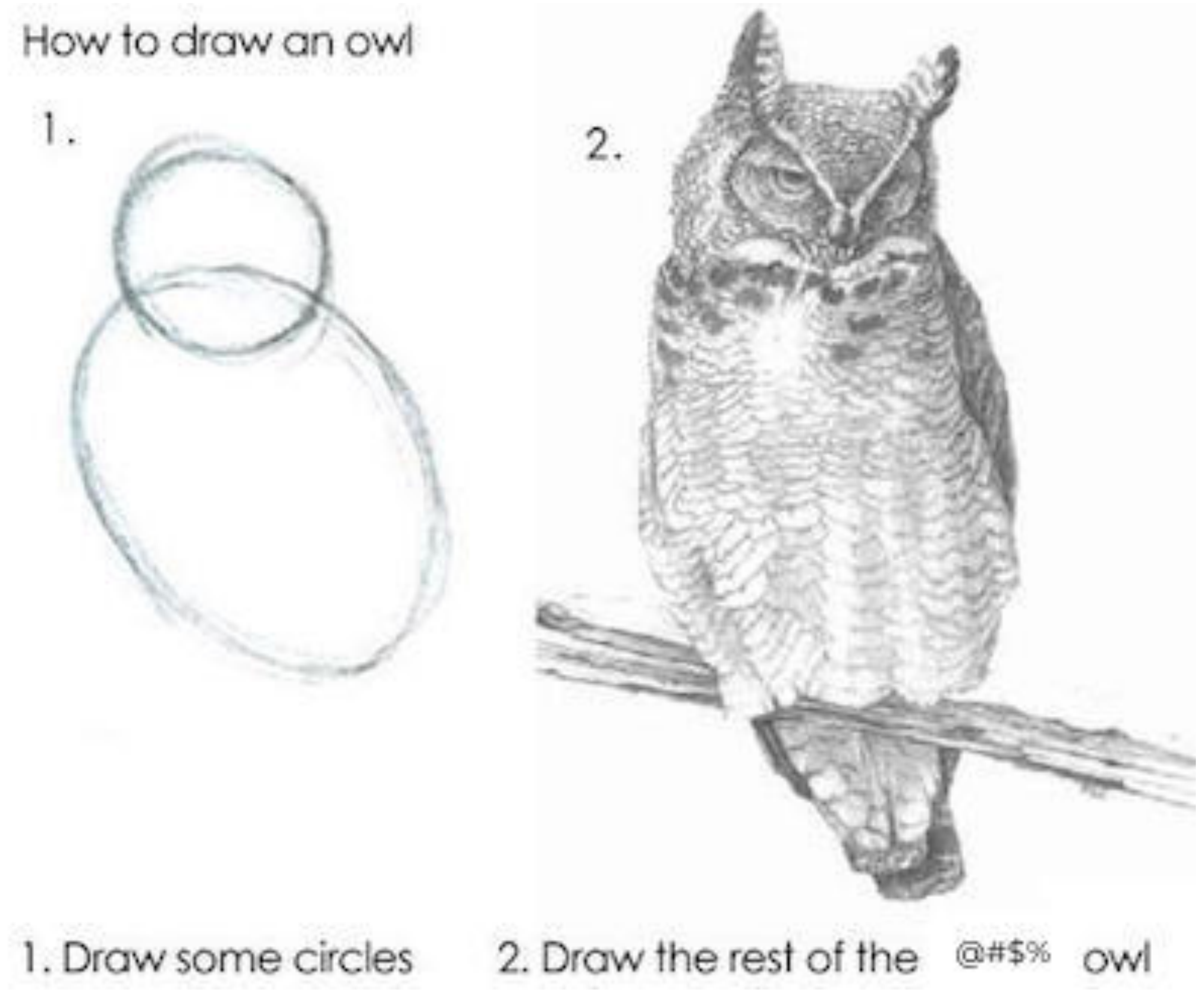
Package Development:

The Rest of the Owl



GitHub repo: pos.it/pkg-dev-conf25

Wifi: Posit Conf 2025 | conf2025

Discord: #workshop-pkg-dev

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

```r
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

```r
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()`

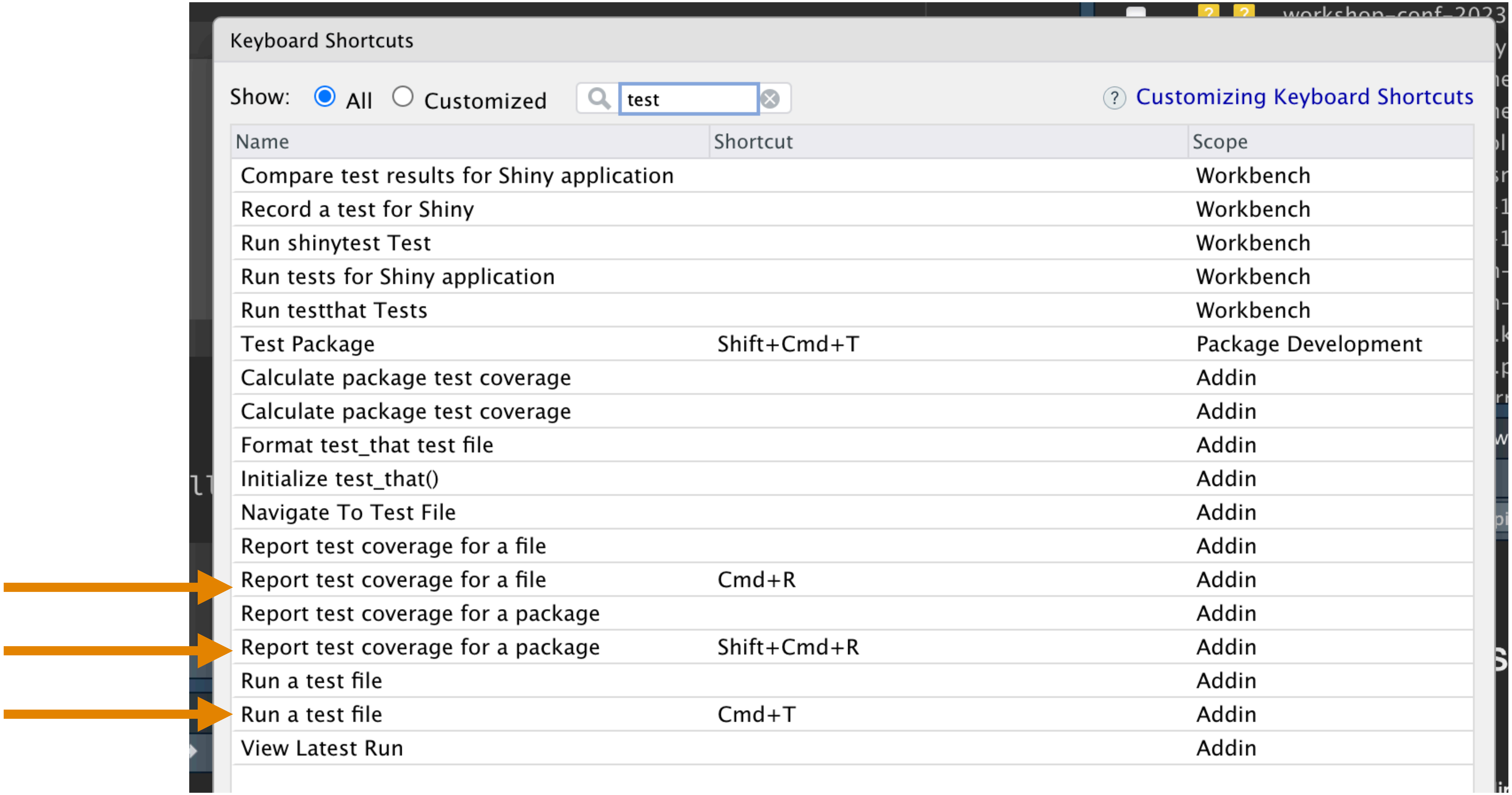
- testthat's workflow is designed around `load_all()`
- Makes entire package namespace available
- Attaches testthat
- Sources `tests/testthat/helper.R`

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)

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

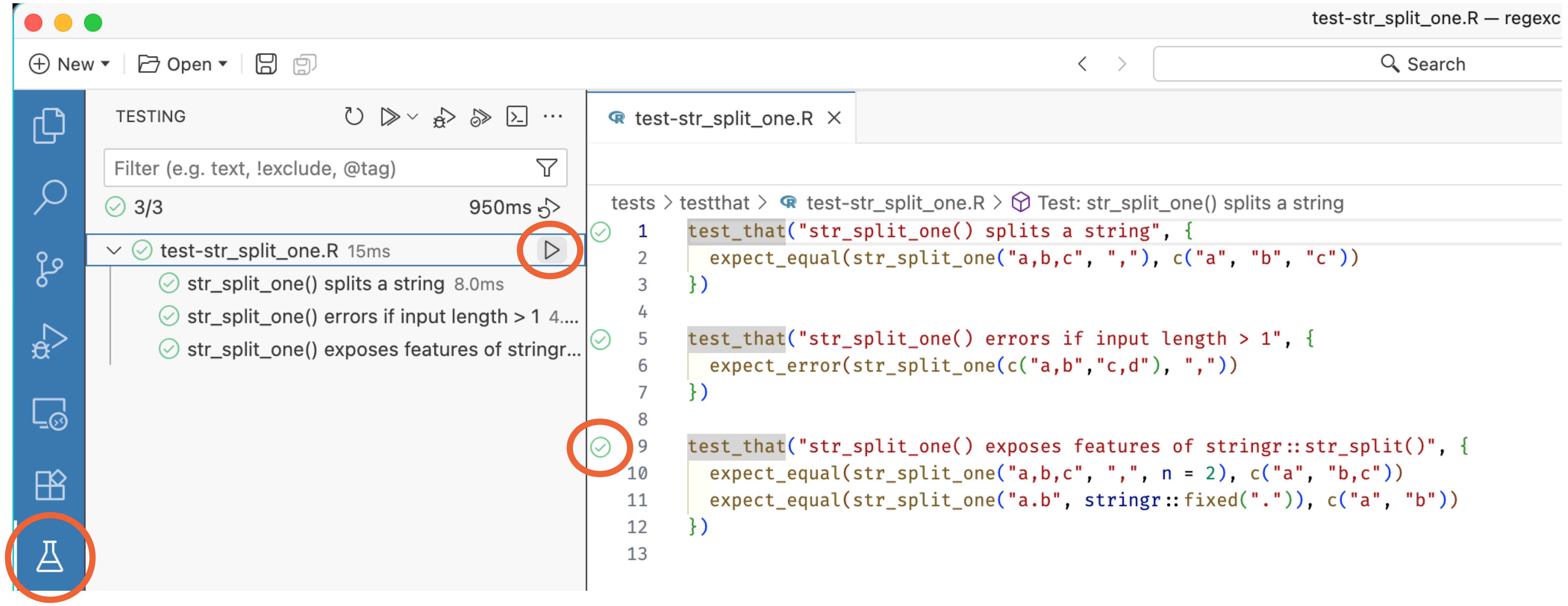
Workflow: mezzo-iteration, whole test file

```r
load_all()

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

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



Search: test

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


test-str_split_one.R — regexc
New
Open
Search
TESTING
Filter (e.g. text, !exclude, @tag)
3/3
950ms
test-str_split_one.R 15ms
str_split_one() splits a string 8.0ms
str_split_one() errors if input length > 1 4....
str_split_one() exposes features of stringr...
test-str_split_one.R
tests > testthat > test-str_split_one.R > Test: str_split_one() splits a string

```r
test_that("str_split_one() splits a string", {
  expect_equal(str_split_one("a,b,c", ","), c("a", "b", "c"))
})

test_that("str_split_one() errors if input length > 1", {
  expect_error(str_split_one(c("a,b","c,d"), ","))
})

test_that("str_split_one() exposes features of stringr::str_split()", {
  expect_equal(str_split_one("a,b,c", ",", n = 2), c("a", "b,c"))
  expect_equal(str_split_one("a.b", stringr::fixed(".")), c("a", "b"))
})

```

Positron's Test Explorer helps you run all tests, all test from a specific file, an individual test, all failing tests, etc.

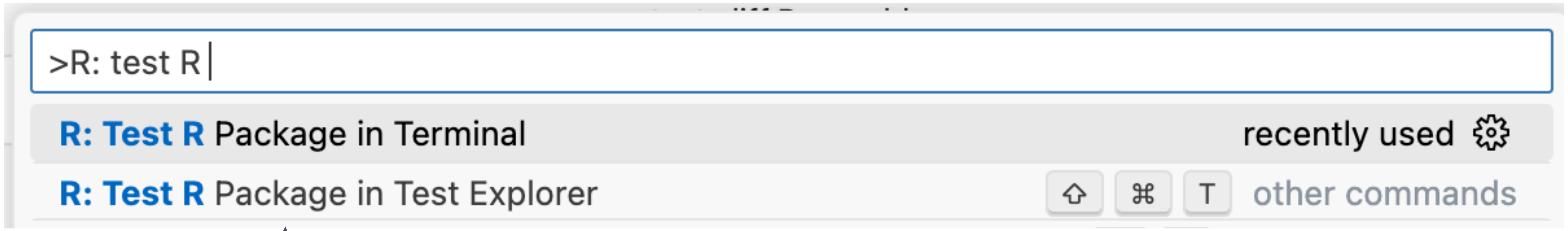
Probably the best / most promising tool for "mezzo" and "macro" iteration on a test suite.

Workflow: macro-iteration, all tests

```
test()
```

```
test_coverage()
```

```
check()
```

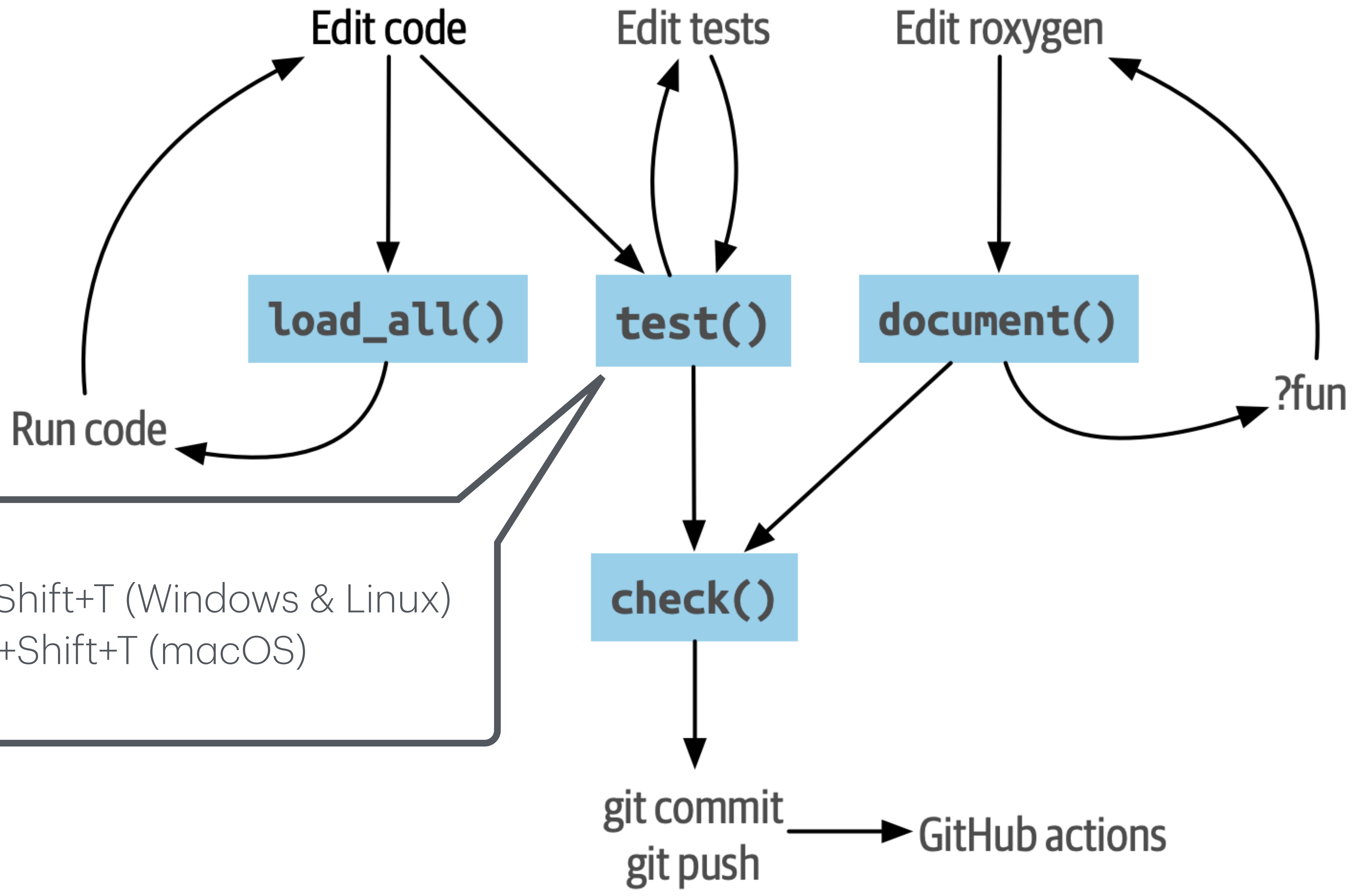

>R: test R

R: Test R Package in Terminal — recently used

R: Test R Package in Test Explorer — ⇧ ⌘ T other commands

Positron has two commands for running all tests.

It is also possible to configure keyboard shortcuts for `use_r()`, `use_test()`, `test_active_file()`, etc. See Emil Hvitfeldt's blog post on Positron keybindings for inspiration.



Ctrl+Shift+T (Windows & Linux)
Cmd+Shift+T (macOS)

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".

All tests should strive to be hermetic: **a test should contain all of the information necessary to set up, execute, and tear down its environment**. Tests should assume as little as possible about the outside environment ….

Software Engineering at Google, Chapter 11

In its purest form, automating testing consists of three activities: writing tests, running tests, and **reacting to test failures**....

Remember that tests are often revisited only when something breaks. When you are called to fix a broken test that you have never seen before, you will be thankful someone took the time to make it easy to understand. Code is read far more than it is written, so make sure you write the test you'd like to read!

Software Engineering at Google, Chapter 11

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

```r
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`

```r
library(testthat)
library(abcde)

test_check("abcde")
```

DO NOT MESS WITH THIS FILE.
JUST DON'T.

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

```r
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"

```
── Snapshot ──────────────────────────────────────────────────
ℹ Can't save or compare to reference when testing interactively.
```

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

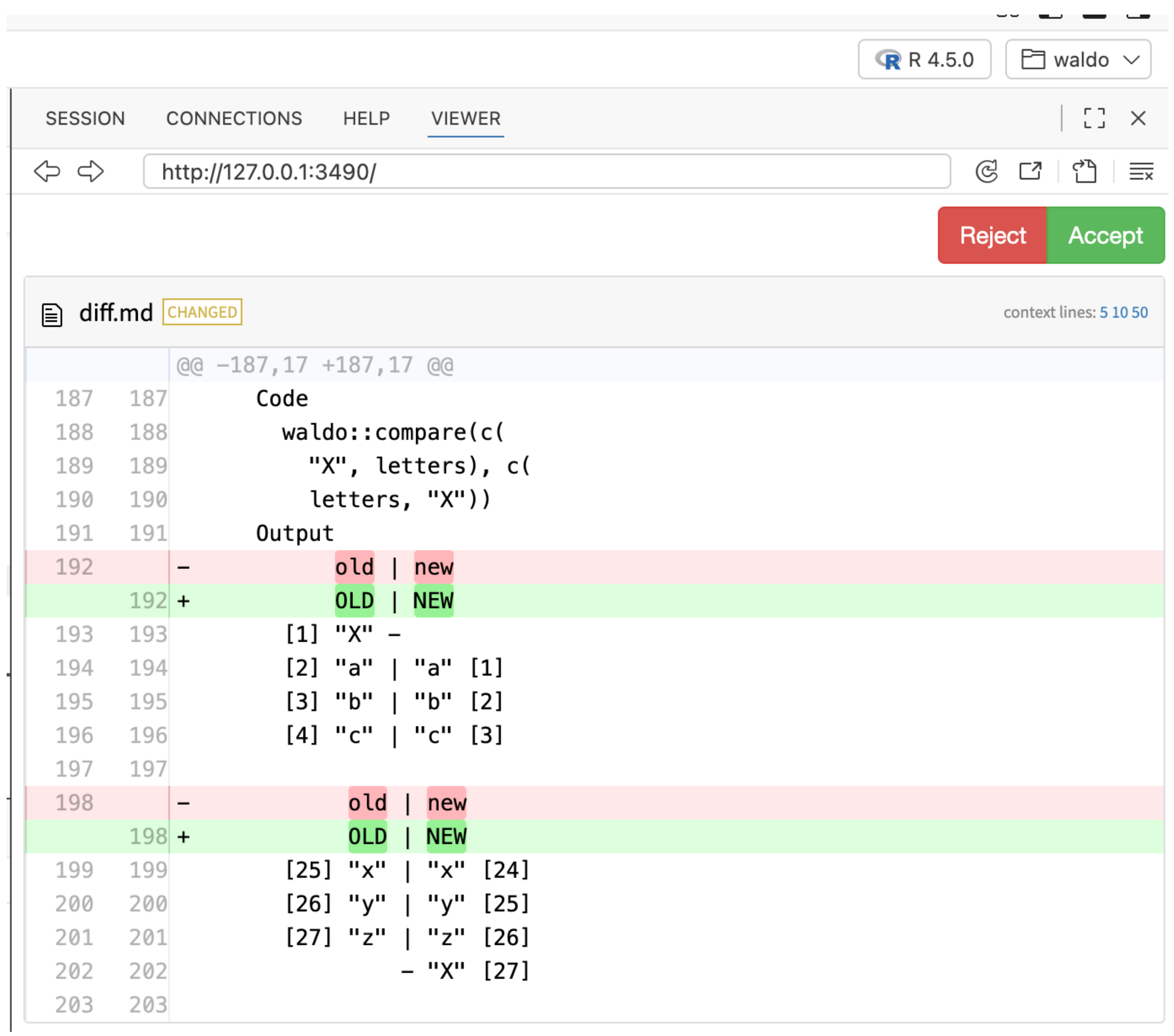
```
── Failure (test-diff.R:63:3): side-by-side diffs work──────────────────────────
Snapshot of code has changed:
old[3:15] vs new[3:15]
  "    \"X\", letters), c("
  "    letters, \"X\"))"
  "Output"
- "      old | new    "
+ "      OLD | NEW    "
  "  [1] \"X\" -        "
  "  [2] \"a\" | \"a\" [1]"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



```
  @@ -187,17 +187,17 @@
    Code
      waldo::compare(c(
        "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
      [25] "x" | "x" [24]
      [26] "y" | "y" [25]
      [27] "z" | "z" [26]
               - "X" [27]

```

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

Key arguments to `expect_snapshot()`

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

Safe filepaths inside your tests

```
test_that("Stata %td (date) and %tc (datetime) read into expected classes", {
  df ← read_stata(test_path("stata/types.dta"))

  expect_s3_class(df$vdat   "Date")
  expect_s3_class(df$vdat  ime, "POSIXct")
})
```

`test_path()` builds a filepath inside tests/testthat/ that works during interactive test tinkering (working directory = package root) and during automated test runs (working directory != package root).

Your turn

Ideas for test-related activities:

https://github.com/posit-conf-2025/pkg-dev/blob/main/testing-prompts.md

Feel free to share what you're doing, how it's going, etc. in the Discord channel.

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`
	`test_path()`

Avoid (at least minimize) test code outside of test_that()	`test helpers`
	`test fixtures`

Leave the world as you found it	`withr::local_*()`