Upgrading to testthat edition 3

Photo by David Pisnoy

2022/02/22

devtools, testthat, usethis

Hannah Frick

As the collection of packages in the tidyverse grows, maintenance becomes increasingly important, and Hadley made this the topic of his keynote at rstudio::global 2021.

In this blog post, I discuss my process for a recent maintenance task, upgrading package tests to use the third edition of testthat.

testthat 3e

The testthat package introduced the idea of an “edition” in version 3.0.0:

Contents

> *backward incompatible changes.*

If you haven't heard of testthat 3e yet, the testthat article introducing the 3rd edition is a great place to start. It outlines all the changes this edition brings.

While you can continue to use testthat's previous behaviour, it's a good idea to upgrade so that you can make use of handy new features. As some of the changes may break your tests, you might have been putting that off, though. You would not be alone in that! Several tidymodels packages still have to make the jump, but I recently upgraded dials and censored to testthat edition 3. Here is what I did and learned along the way.

Workflow to upgrade

The testthat article tells you how you can opt in to the new edition, and about major changes: deprecations, how messages and warnings are handled, and how comparisons of objects are made.

The main guidance for a workflow is:

1. Activate edition 3.
2. Remove or replace deprecated functions.
3. If your output got noisy, quiet things down as needed.
4. Think about what it means if things are not "all equal" anymore.

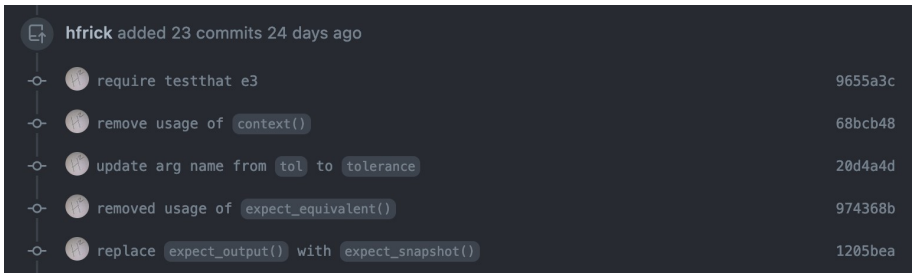
Activation 🚀

To activate you need to do two things in the DESCRIPTION -

- Increase the testthat version to `>= 3.0.0`.
- Set the `Config/testthat/edition` field to `3`.

Moving on from deprecations ✨

The article on testthat 3e contains a list of deprecated functions together with their successors. You can work your way through it, searching for the deprecated function and then replacing it with the most suitable alternative. The first one in that list is `context()` as testthat will use the file name instead, ensuring that context and file name are in sync. As such, `context()` does not have a replacement. My first commit after activating the third edition was to remove all calls to `context()`, followed by replacing other deprecated functions and arguments.



Warnings and messages 😲

testthat edition 3 handles warnings and messages differently than edition 2: `expect_warning()` captures at most one warning, so if your code generates multiple warnings, they will bubble up now. Messages were previously silently ignored, now they also bubble up. That means the output may be a lot noisier after switching to edition 3. If the warnings or messages are important, you should explicitly capture them. Otherwise you can suppress them to clean up the output and make it easier to focus on what's important. Again, the

Comparing things 🍎 🍊

The last big change from edition 2 to edition 3 that I want to mention is what is happening under the hood of `expect_equal()` and `expect_identical()`. Edition 3 uses `waldo::compare()` while edition 2 uses `all.equal()`. For the most part, that meant changing the argument name from `tol` to `tolerance`, like in my third commit above.

I did, however, run into a situation where a test newly failed. Those are the situations where general advice is hard because it depends so much on the context. In my case, I made use of the `ignore_function_env` and `ignore_formula_env` arguments to `waldo::compare()` to exclude those environments from the comparison. Those are probably useful to know about if you are upgrading a modelling package, but not particularly important otherwise. For dials and censored, that solved most of the cases. In one instance, I ended up tweaking the reference value based on theoretical considerations of the model I was dealing with rather than increasing the tolerance.

Those instances may be the most work when upgrading to edition 3, but I did not encounter many of them – and, when I did, it was valuable to know about the differences (well, those which I didn't choose to ignore).

More testing made easier

While I was going over all the test files, I also decided to cover

Nested expectations

When Davis Vaughan moved other tidymodels packages to testthat 3e, I saw him disentangle nested expectations. For example, patterns like

```
expect_warning(expect_equal(one_call,
another_call))
```

or

```
expect_equal(expect_warning(one_call),
expect_warning(another_call))
```

can be re-written as

```
expect_snapshot({
    object_from_one_call <- one_call()
    object_from_another_call <- another_call()
})
expect_equal(object_from_one_call,
object_from_another_call)
```

This separates an expectation about the warnings from the expectation about the value, making it easier to see which part(s) fail. Snapshots can also be particularly helpful in situations where you are trying to test for a combination of warnings, messages, and/or errors because they cover them all.

Self-contained tests

I wanted to make the tests more self-contained so that a test could run with a single call to `test_that()`. Specifically, I didn't want to have to scroll back up to the top of the file to

You can avoid the former by prefixing functions with the package they belong to, i.e. using `dplyr::mutate()` instead of `library(dplyr)` at the top of the file and later `mutate()` inside of the expression for `test_that()`.

If creating a helper object is short, I might move the code inside of `test_that()`. If you create the same helper objects multiple times and don't want to see the code repeatedly, you can move it into a helper function. Files inside the `testthat` folder of your source code with file names starting with `helper` are executed before tests are run. You could put your helper code there but it is recommended to put the helper code in your `R/` folder, for example as `test-helpers.R`.

An example helper function is called `make_test_model()`, which returns a list with training and testing data as well as the fitted model. A test on the prediction method could then look like this:

```
test_that("prediction returns the correct number
of records", {
    helper_objects <- make_test_model()
    pred <- predict(helper_objects$model,
helper_objects$test_data)
    expect_equal(nrow(pred),
nrow(helper_objects$test_data))
})
```

Any other data objects needed for testing I moved into `tests/testthat/data/`.

Corresponding files in `R/` and `tests/testthat/`

`test-monstera.R`.

This gives you access to some convenient features of usethis and devtools:

- When you have the R file open, it's easy to open the corresponding test file with `usethis::use_test()` - and vice versa with `usethis::use_r()`. No clicking around needed!
- When you have either file open, you can run the tests with `devtools:::test_active_file()`, and see the test coverage report with `test_coverage_active_file()` (which also shows you which lines are actually being tested). Both also have an RStudio addin, which means you can add keyboard shortcuts for them!

And, with that, dials and censored were ready for more snapshot tests in the future!

For more guidance on implementing tidy standards, check out `usethis::use_tidy_upkeep_issue()`. It creates a GitHub issue with a handy checklist. You will be seeing those popping up in our repositories soon when we do some spring