Errors & snapshot tests

Back at 11am

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

September 2023

1. testthat 3e
2. Workflow advice
3. Snapshot tests
4. Errors
5. Type checking
6. Advanced topics

testthat 3e

testthat 3e

Activate (once per package) with `usethis::use_testthat(3)`

- Snapshot tests
- Lots of deprecations
- expect_equal() and friends use waldo package
- Parallel testing

IMO this alone is worth the cost of switching

More details at https://testthat.r-lib.org/articles/third-edition.html. (Likely to one day see a tidyverse edition)

Goal of waldo is to make it as easy as possible to spot the 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()

```
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
```

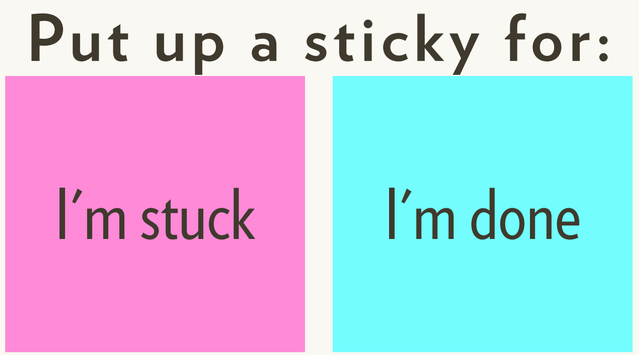
Your turn

Get a local copy of the stringb package with create_from_github("hadley/stringb")

Convert to testthat 3e (this will be easy!)

Enable parallel testing (https://testthat.r-lib.org/articles/parallel.html).

Verify that R CMD check passes.



Workflow

usethis provides helpers for creating & opening test files

```
usethis::use_test("whatever")


# in RStudio, with a R/.R file focused,
# target test file can be inferred
usethis::use_test()


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

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	

See code in ?use_r example to determine if this true for your package

Workflow: micro-iteration, interactive experimentation

```
# tweak the foofy() function and re-load it
devtools::load_all()

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

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

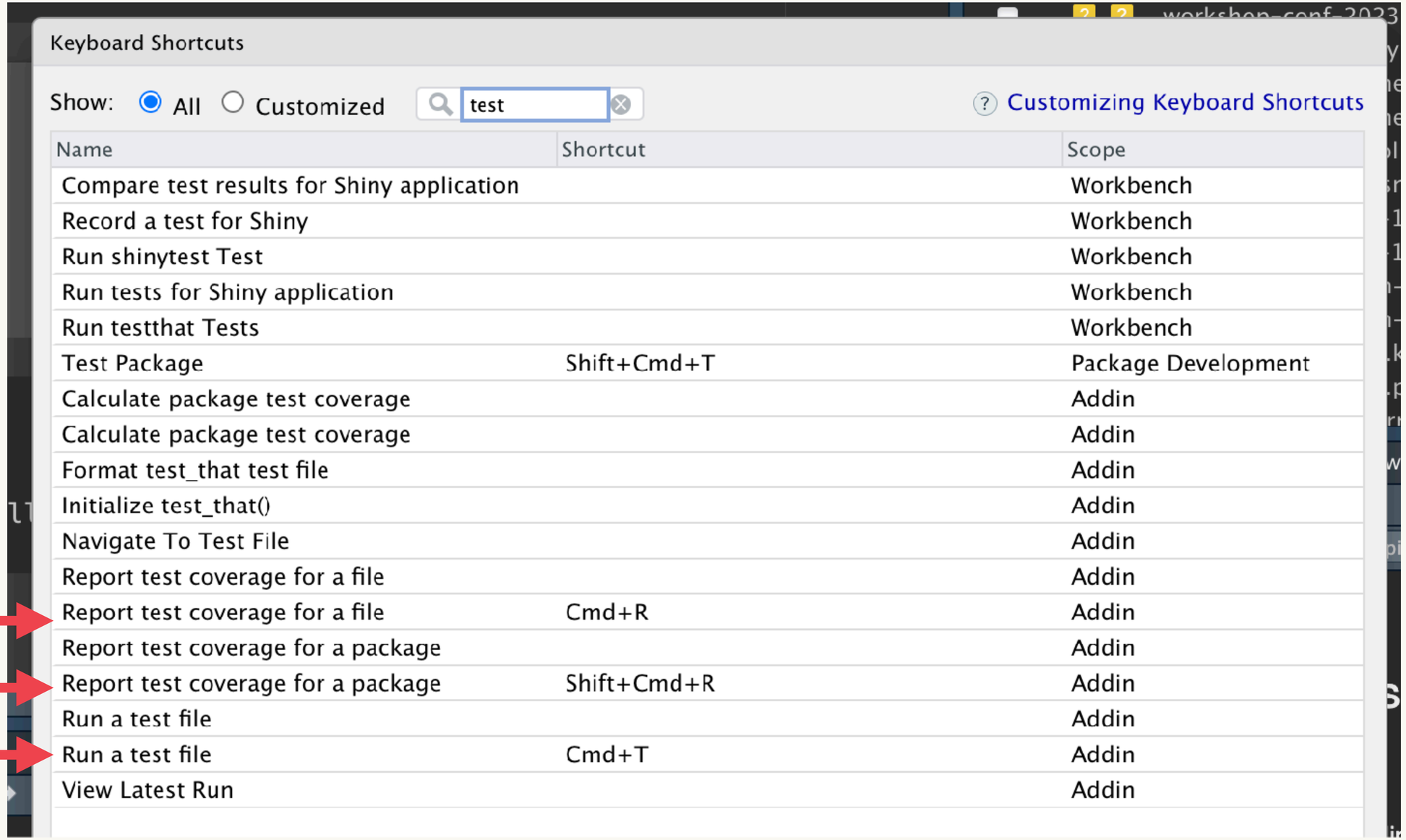
Workflow: mezzo-iteration, whole test file

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

# in RStudio, with test or R file focused
devtools::test_active_file()
devtools::test_coverage_active_file()

# we suggest binding these to Cmd + T, Cmd + R
```

Workflow: macro-iteration, all files

```
devtools::test()
devtools::test_coverage()

devtools::check()
```

Your turn


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

Your turn

Find which code isn't covered by tests
(you don't need to do anything about it yet).

```
> devtools:::test_coverage()
ℹ Computing test coverage for stringb
Error in idesc_create_package(self, private, package) :
  Cannot find DESCRIPTION for installed package stringb
```

Snapshot tests

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

The big idea of snapshot testing:

Capture the result of the first run, and store it in a (human readable) file.

Future test runs compare actual result to that file snapshot.

If you deliberate change the result, you have to manually approve it.

Especially suitable for testing human facing output, e.g., testing messages and errors.

For example, how would you test waldo?

```
withr::with_options(
  list(width = 10),
  waldo::compare(c("X", letters), c(letters, "X"))
)

withr::with_options(
  list(width = 20),
  waldo::compare(c("X", letters), c(letters, "X"))
)

withr::with_options(
  list(width = 60),
  waldo::compare(c("X", letters), c(letters, "X"))
)
```

One approach is to use snapshot tests

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

Interactive execution of snapshot tests 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 a non-interactive context

Snapshot test recording and comparison only happen when the tests are being run at arms-length, via some sort of automated process.

For example: running an entire test file, running the entire test suite.

New snapshot! 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]
```

When snapshot tests fail

```
── Failure (test-diff.R:63:3): side-by-side diffs work─────────────────────────

Snapshot of code has changed:
    old                    | new
[1] Code                   | Code                    [1]
[2]   waldo::compare(c(    |   waldo::compare(c(    [2]
[3]     "X", letters), c(  |     "X", letters), c(  [3]
[4]     letters, "X"))     -     letters, "Z"))     [4]
[5] Output                 | Output                  [5]
[6]       old | new        |       old | new         [6]
[7]   [1] "X" -            |   [1] "X" -             [7]

     old                     | new
[13]   [25] "x" | "x" [24] |   [25] "x" | "x" [24] [13]
[14]   [26] "y" | "y" [25] |   [26] "y" | "y" [25] [14]
[15]   [27] "z" | "z" [26] |   [27] "z" | "z" [26] [15]
[16]            - "X" [27] -            - "Z" [27] [16]
```

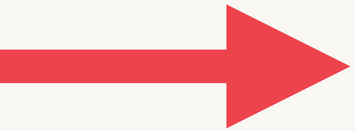
You have to determine if a change is a failure

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

(And both should be clickable in recent RStudio)

(If you need to undo an accidental acceptance, you'll need to use your git skills)

Other important arguments to `expect_snapshot()`



1. `error = TRUE`
2. `cran = TRUE`
3. `transform = \(x) ...`
4. `variant = \(x) ...`

Your turn

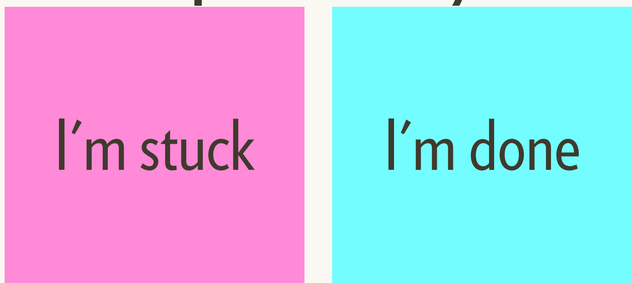
Convert the existing expect_error() to expect_snapshot(). (What argument do you need to set?)

Use expect_snapshot() to get to 100% test coverage for stringb. (Where should the new test live?)

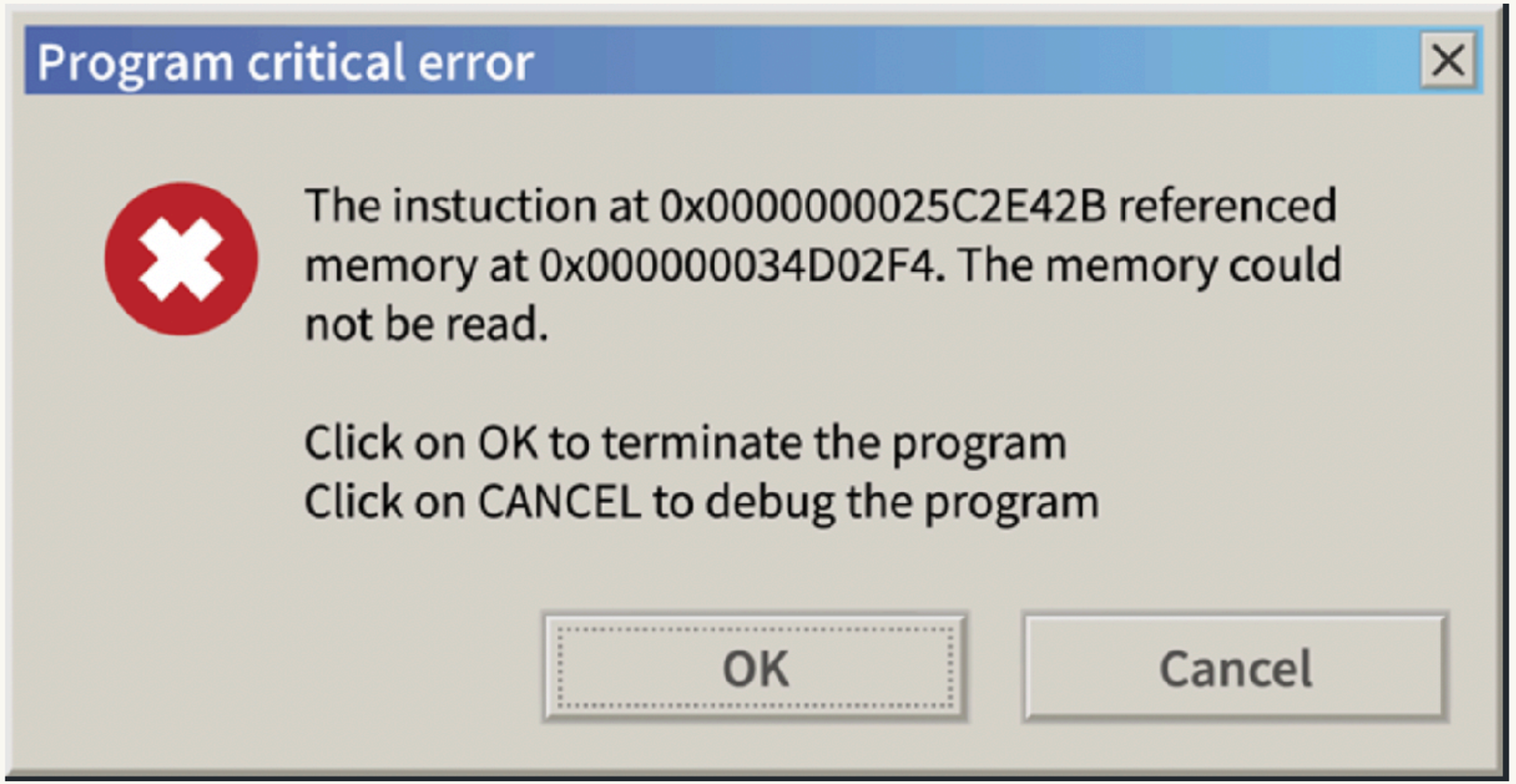
Our style guide now recommends that all error messages end in a full stop. Add that full stop to the messages, verify that the tests now fail, and then accept the change so that they pass again.

Verify that R CMD check passes.

Put up a sticky for:



Custom errors


Program critical error
The instuction at 0x0000000025C2E42B referenced memory at 0x000000034D02F4. The memory could not be read.
Click on OK to terminate the program
Click on CANCEL to debug the program
OK
Cancel

Our error principles

1. A **consistent** structure makes it easier for users to scan for key details.
2. Try to **communicate** exactly what went wrong. If you know what right looks like, show that too.
3. Include **context**, like the function call and argument name.
4. Strive to be **concise** so you don't overwhelm the reader, but provide links to more details.

Generating errors

```
# Base R
stop()

# rlang
abort()

# cli
cli::cli_abort()
```

We consider rlang and cli to be “free” dependencies for most packages

cli_abort() makes it easy to mix text and values

```
# Glue interpolation
x ← 10
cli::cli_abort("x ({x}) must be less than 10.")

# With styling
path ← "foo.txt"
cli::cli_abort("{.arg path} ({.path {path}}) doesn't exist.")
cli::cli_abort(
  "{.arg x} must be a string, not {.obj_type_friendly {x}}."
)
# https://cli.r-lib.org/reference/inline-markup.html
```

Pluralisation is a breeze

```
n_files ← 1
cli::cli_abort("Can't supply {n_files} file{?s}.")

n_files ← 2
cli::cli_abort("Can't supply {n_files} file{?s}.")
```

It’s easy to add links

```
cli::cli_abort("See {.url https://cli.r-lib.org} for details.")
cli::cli_abort("See {.fun stats::lm} to learn more.")
cli::cli_abort("See the tibble options at {.help tibble::tibble_options}.")

# Including links that run code
cli::cli_abort("Run {.run testthat::snapshot_review()} to review.")

# More at https://cli.r-lib.org/reference/links.html#hyperlink-support
```

Bulleted lists allow you to present multiple details

```
cli::cli_abort(c(
  "Unexpected content type {.str {content_type}}.",
  "*" = paste0(
    "Expecting {.str {type}}",
     if (!is.null(suffix)) " or suffix {.str {suffix}}",
     "."
  ),
  i = "Override check with {.code check_type = FALSE}."
))


# https://cli.r-lib.org/reference/cli_bullets.html#details
```

Your turn

use_package("cli")

Convert all existing uses of stop() to cli::cli_abort().

Use inline markup (https://cli.r-lib.org/reference/inline-markup.html) where appropriate.

Test your work. Do you need new tests or are you existing tests sufficient?

Ensure R CMD check passes.

Some style notes

```
# We use "must" when you know what is a valid input
dplyr::lag(1:5, "x")
#> Error in `dplyr::lag()`:
#> ! `n` must be a whole number, not the string "x".

# We use "Can't" when you can't state exactly what is expected
dplyr::select(mtcars, xyz)
#> Error in `dplyr::select()`:
#> ! Can't subset columns that don't exist.
#> ✖ Column `xyz` doesn't exist.

# More at <https://style.tidyverse.org/error-messages.html>
```

Error helpers

cli::cli_abort() automatically includes the function name

```
my_function ← function() {
  cli::cli_abort("An error")
}

my_function()
#> Error in `my_function()`:
#> ! An error
```

But what if you write a helper?

```
my_error_helper ← function() {
  cli::cli_abort("An error")
}
my_function ← function() {
  my_error_helper()
}
my_function()
Error in `my_error_helper()`:
! An error
```

Not useful to mention a function that users can't see

```
str_sub("x", 1:2)
#> Error in `recycle()`:
#> ! Can't recycle `arg` to length 1.
```

You need to capture the caller environment and pass it along

```
my_error_helper ← function(call = parent.frame()) {
  cli::cli_abort("An error", call = call)
}
my_function ← function() {
  my_error_helper()
}
my_function()
Error in `my_function()`:
! An error
```

Your turn

use_package("rlang", min_version = "1.0.0") &

update snapshots

Add the call argument to recycle() and check_pattern().

How do the snapshots change?

Verify that R CMD check passes

```
my_error_helper ← function(call = parent.frame()) {
  cli::cli_abort("An error", call = call)
}
```

Type checking

Bad types often give bad errors

```
str_sub("hello", "a")
#> [1] NA
#> Warning message:
#> In substr(string, start, end) :
#> NAs introduced by coercion
```

Clear failures are much easier to debug

```
str_sub("hello", "a")
#> Error in `str_sub()`:
#> ! `start` must be a whole number,
#>   not the string "a".
```

Begin by importing our type checker helpers

```
use_standalone("r-lib/rlang", "types-check")
# creates import-standalone-obj-type.R and
# import-standardalone-type-check.R.
# Assumes that you've already imported rlang with
#' @import rlang


# this somewhat of a stopgap until we figure
# exactly what the interface should be and
# where this code should live
```

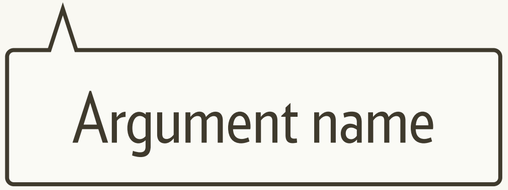
Checks are divided in to scalar and vector

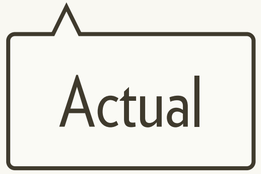
```
check_bool()
check_string()
check_number_decimal()
check_number_whole()


check_character()
check_logical()
check_data_frame()
```

Usage

```
my_function ← function(x, y, z) {
  check_bool(x)
  check_string(y)
  check_number_whole(z)
}
```

```
my_function(123)
#> Error in `my_function()`:
#> ! `x` must be `TRUE` or `FALSE`, not the number 123.
```





Arguments control allowed inputs

```
# make the argument "optional"
check_bool(x, allow_null = TRUE)

# NAs are not allowed by default
check_bool(x, allow_na = TRUE)

# empty strings are allowed by default
check_string(x, allow_empty = TRUE)

# number have optional ranges
check_number_whole(x, min = 0)
```

Your turn

```
use_standalone("r-lib/rlang", "types-check")

use_package_doc()

#' @import rlang
```

Add type checking functions to str_sub(). Test them with a snapshot test.

Add type checking functions to the rest of stringb.

Advanced topics

Chained errors

```
mtcars |>
  group_by(cyl) |>
  mutate(new = 1 + "")
#> Error in `mutate()`:
#> ! Problem while computing `new = 1 + ""`.
#> ℹ The error occurred in group 1: cyl = 4.
#> Caused by error in `1 + ""`:
#> ! non-numeric argument to binary operator
```

```
mtcars ▷
  ggplot() +
  geom_point(aes(1 + ""))
#> Error in `geom_point(aes(1 + ""))`:
#> ! Problem while computing aesthetics.
#> ℹ Error occurred in the 1st layer.
#> Caused by error in `1 + ""`:
#> ! non-numeric argument to binary operator
```

```
withCallingHandlers(
  code,
  error = function(err) {
    cli::cli_abort( ... , parent = err)
  }
)

# More at https://rlang.r-lib.org/reference/topic-error-chaining.html
```

Custom classes

A helper extracts out repeated error code:

```
resp_abort ← function(resp, info = NULL, call = caller_env()) {
  status ← resp_status(resp)
  desc ← resp_status_desc(resp)
  message ← glue("HTTP {status} {desc}.")

  abort(c(message, resp_auth_message(resp), info), call = call)
}
```

In some cases, you might make a richer error object

```
resp_abort ← function(resp, info = NULL, call = caller_env()) {
  status ← resp_status(resp)
  desc ← resp_status_desc(resp)
  message ← glue("HTTP {status} {desc}.")

  abort(
    c(message, resp_auth_message(resp), info),
    status = status,
    resp = resp,
    class = c(glue("httr2_http_{status}"), "httr2_http"),
    call = call
  )
}
```

Can test for specific class with expect_error()

```
test_that("can send username/password", {
  user ← "u"
  password ← "p"
  req1 ← request_test("/basic-auth/:user/:password")
  req2 ← req1 %>% req_auth_basic(user, password)

  expect_error(req_perform(req1), class = "httr2_http_401")
  expect_error(req_perform(req2), NA)
})
```

And gives user fine-grained control

```
tryCatch(
  req %>% req_perform() %>% resp_body_json(),
  httr2_http_404 = function(cnd) list()
)

# Learn more at
# <https://adv-r.hadley.nz/conditions.html>
```