Errors

Jan 2019

Charlotte Wickham
@cvwickham
cwickham@gmail.com
cwick.co.nz

Signaling errors as a function **author**

Change project to:

[hadcol-test]

Inside downloaded materials

Motivation: protect against bad inputs

```r
# Or Crtl/Cmd + Shift + L
devtools::load_all()

df <- data.frame(x = 1, y = 2)

add_col(df, name = "z", value = 3, where = 0)
#  Error in `[.default`(x, lhs) :
#  only 0's may be mixed with negative subscripts
```

Finding where errors occur

```
> add_col(df, name = "z", value = 3, where = 0)

Error in `[.default`(x, lhs) :
  only 0's may be mixed with negative
subscripts

  6. NextMethod("[")
  5. `[.data.frame`(x, lhs)
  4. x[lhs]
  3. cbind(x[lhs], y, x[-lhs]) at insert_into.R#8
  2. insert_into(x, df, where = where) at add_col.R#7
  1. add_col(df, name = "z", value = 3, where = 0)
```

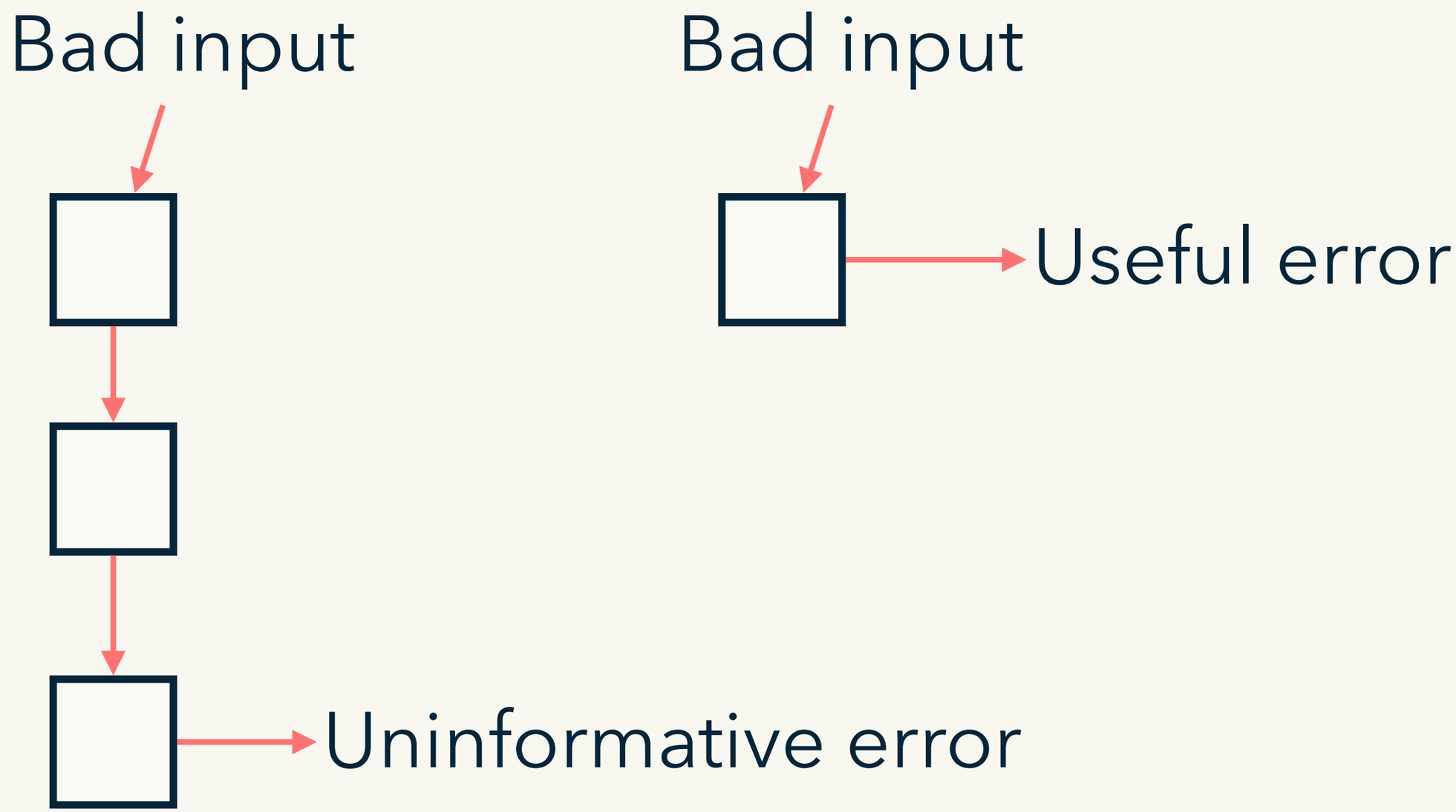
Show Traceback

Rerun with Debug

```
# Not in RStudio
traceback()
```

Fail fast

For robust code, fail early



Check inputs in insert_into()

```
df1 <- data.frame(a = 3, b = 4, c = 5)
df2 <- data.frame(X = 1, Y = 2)

# We need these to return errors
insert_into(df1, df2, where = 0)
insert_into(df1, df2, where = NA)
insert_into(df1, df2, where = 1:10)
insert_into(df1, df2, where = "a")
```

We could add to insert_into directly

```
insert_into <- function(x, y, where = 1) {
  if (!is.numeric(where) || length(where) != 1) {
    stop("`where` is not a number", call. = FALSE)
  } else if (where == 0 || is.na(where)) {
    stop("`where` must not be 0 or NA", call. = FALSE)
  } else if (where == 1) {
    cbind(y, x)
  } else if (where > ncol(x)) {
    cbind(x, y)
  } else {
    lhs <- 1:(where - 1)
    cbind(x[lhs], y, x[-lhs])
  }
}
```

But this confuses the intent of `insert_into()`

Better to have a function responsible for this

```
insert_into <- function(x, y, where = 1) {
  where <- check_where(where)
  if (where == 1) {
    cbind(y, x)
  } else if (where > ncol(x)) {
    cbind(x, y)
  } else {
    lhs <- 1:(where - 1)
    cbind(x[lhs], y, x[-lhs])
  }
}
```

Add protection against bad inputs

Test driven development

1. Decide what should happen with bad inputs
2. Write tests for `check_where()` that reflect #1
3. Write `check_where()`
4. Update `insert_into()` to use `check_where()`

Error message structure

1. **Problem statement**
   (use must or can't)
2. **Error location**
   (where possible)
3. **Hint**
   (if common)

http://style.tidyverse.org/error-messages.html

Your turn

```
# Write down the error message that you think
# each of these lines should generate

check_where(where = 0)
check_where(where = NA)
check_where(where = 1:10)
check_where(where = "a")
```

My results

```
check_where(0)
#> Error: `where` must not be
zero or missing.
check_where(NA)
#> Error: `where` must not be
zero or missing.
check_where(1:10)
#> Error: `where` must be a
length one numeric vector.
check_where("a")
#> Error: `where` must be a
length one numeric vector.
```

Style

- Surround variable names in `...`, and strings in '...'
- Sentence case

Use expect_error() to test for errors

```
# Test will pass if error occurs
expect_error(
  check_where("a")
)

# Test will pass if error message matches
expect_error(
  check_where("a"),
  "not a number"
)
```

A regular expression

Your turn

Write tests to ensure that `check_where()` only allows valid inputs.

(Where should the tests live? How many tests do you need? How many expectations?)

My tests

```
# I think should live in tests/testthat/test-insert_into.R

test_that("where must be valid value", {
  expect_error(check_where("a"), "length one numeric vector")
  expect_error(check_where(1:10), "length one numeric vector")

  expect_error(check_where(0), "not be zero or missing")
  expect_error(check_where(NA_real_), "not be zero or missing")
})
```

Signal an error with stop()

```
f <- function(){
  stop("This is an error message.",
    call. = FALSE)
}

f()
# Error : This is an error message.
```

Don't include the call in error message

Check inputs by combining with if()

```r
# A general pattern
f <- function(x){
  if (!is.numeric(x)) {
    stop("`x` must be numeric",
         call. = FALSE)
  }
  x
}
f("a")
```

Your turn

Write `check_where()`. It should throw an error if the input is incorrect. I suggest you put in the same file as `insert_into()`.

```
check_where(0)
check_where(NA)
check_where(1:10)
check_where("a")
```

Hint to get started on next slide

Hint: getting started

```r
# Start with a skeleton in R/insert_into.R
check_where <- function(where){

}

# Make sure you've put the tests in
# tests/testthat/test-insert_into.R
# Check you get four failures with
devtools::test()

# Edit check_where() until it passes tests
```

My answer

```
check_where <- function(x) {
  if (length(x) != 1 || !is.numeric(x)) {
    stop("`where` must be a length one numeric vector.",
      call. = FALSE)
  }
  x <- as.integer(x)

  if (x == 0 || is.na(x)) {
    stop("`where` must not be zero or missing",
      call. = FALSE)
  } else {
    x
  }
}
```

Other conditions

Errors `stop()`
No way for function to continue, execution must stop.

Warnings `warning()`
Signal that something has gone wrong, but the code has been able to recover and continue. Use sparingly, would an error be safer?

Messages `message()`
Informational only.

use cat() when primary purpose is output

https://adv-r.hadley.nz/conditions.html#signalling-conditions

Handling errors

as a function **user**

Iteration: what happens if there is an error?

```
library(purrr)

input <- list(1:10, sqrt(4), 5, "n")

map(input, log)
# Error in .Primitive("log")(x, base) :
#   non-numeric argument to mathematical
#   function
```

No results.
No idea which element was the problem.

Principle:

Turn side-effects into data

What does safely() do?

```r
input <- list(1:10, sqrt(4), 5, "n")

# This will never fail
map(input, safely(log))

# What does it return when the function succeeds?
# What does it return when the function fails?
```

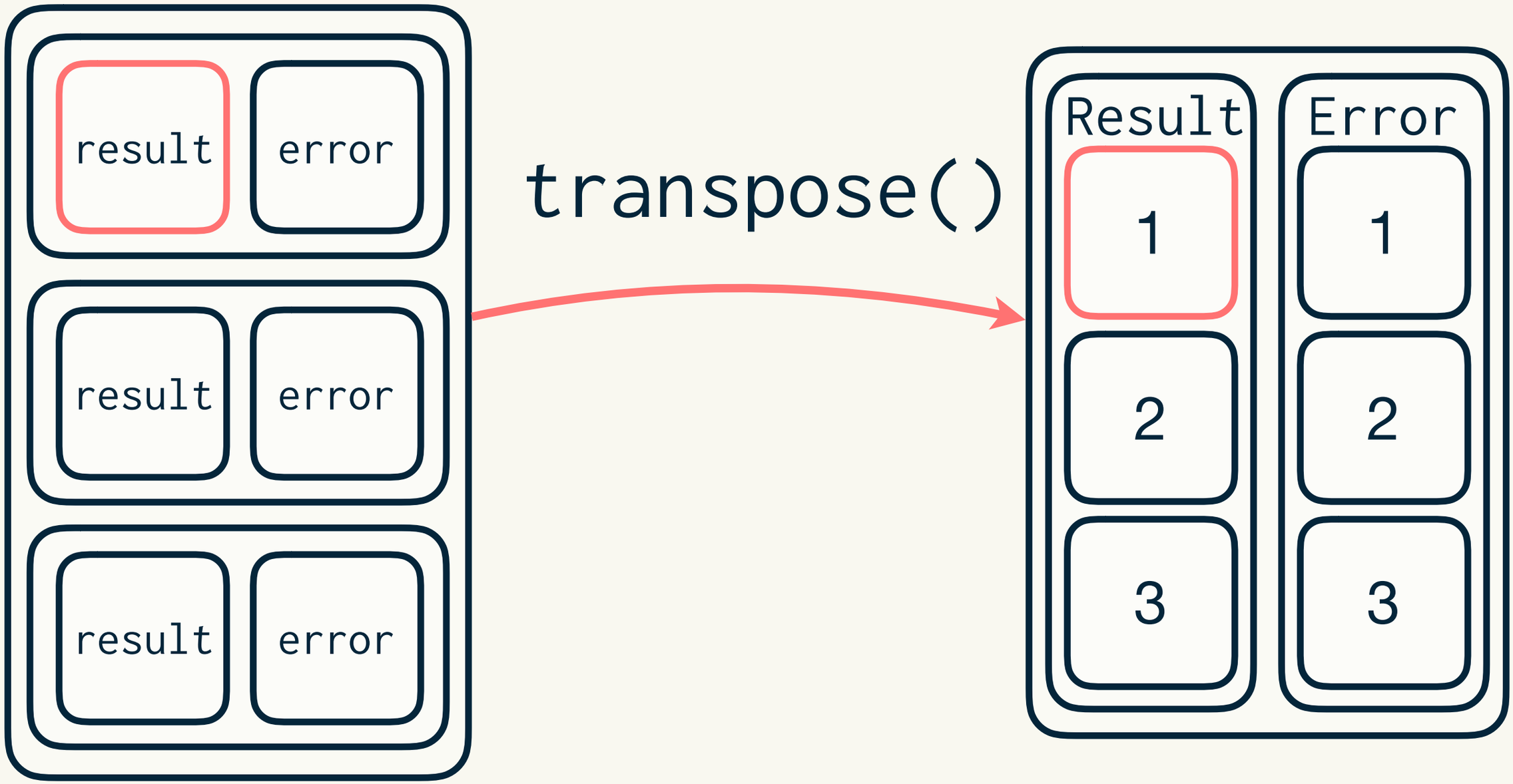
A more useful example

```
urls <- c(
  "https://google.com",
  "https://en.wikipedia.org",
  "asdfasdasdkfjlda"
)

# Fails
contents <- urls %>%
  map(readLines, warn = FALSE)

# Always succeeds
contents <- urls %>%
  map(safely(readLines), warn = FALSE)
str(contents)
```

But map() + safely() gives awkward output



`x[[1]][["result"]]` → `x[["result"]][[1]]`

Your turn

Apply `transpose()` to `contents` from "A more useful example" then:

1. Make logical vector that is `TRUE` if download succeeded. (`map_lgl()`)
2. List failed urls
3. Extract successfully retrieved text

Common pattern with safely()

```
contents <- urls %>%
  map(safely(readLines)) %>%
  transpose()

ok <- map_lgl(contents$error, is.null)
# This is suboptimal:
ok <- !map_lgl(contents$result, is.null)

urls[!ok]
contents$result[ok]
```

Functional operators

one or more function(s) as input, a function as output

`safely()`
`possibly()`
`quietly()`
	turn side effects into data

`partial()`	set some arguments
`lift()`	change how arguments are provided
`memoise::memoise()`	add a memory

Think **adverbs:** alter the behaviour of a function

Handling errors inside your own functions

```
try()
tryCatch()
```

To capture and handle errors in custom ways.

See: https://adv-r.hadley.nz/conditions.html#handling-conditions

Adapted from *Tidy Tools* by Hadley Wickham