OO programming

Jan 2019

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

sorry, I forgot this one

```
install.packages("sloop")
```

Adapted from *Tidy Tools* by Hadley Wickham

What is S3?

What does S3 do?

S3 powers context specific behaviour

```
x <- 1:5
y <- factor(letters[1:5])

summary(x)
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#       1       2       3       3       4       5

summary(y)
# a b c d e
# 1 1 1 1 1
```

A 6-number summary

A table of categories

summary() is an S3 generic

```
sloop::ftype(summary)
# [1] "S3"      "generic"

# summary() will look for methods based on an
# objects class
sloop::s3_class(y)
# [1] "factor"

sloop::s3_dispatch(summary(x))
# => summary.factor     => this method gets called
#  * summary.default    *   this method exists but wasn't called
```

Your Turn

```
mod <- lm(mpg ~ wt, data = mtcars)
summary(mod)
```

What is the **class** of `mod`?

Which **method** gets dispatched by `summary()`?

Can you find the code for the method?

```r
library(sloop)
mod <- lm(mpg ~ wt, data = mtcars)
summary(mod)

s3_class(mod)
# [1] "lm"

s3_dispatch(summary(mod))
# => summary.lm
#  * summary.default

summary.lm

# won't always work
# use `s3_get_method()` to find non-exported methods
s3_get_method(summary.lm)
```

Motivation

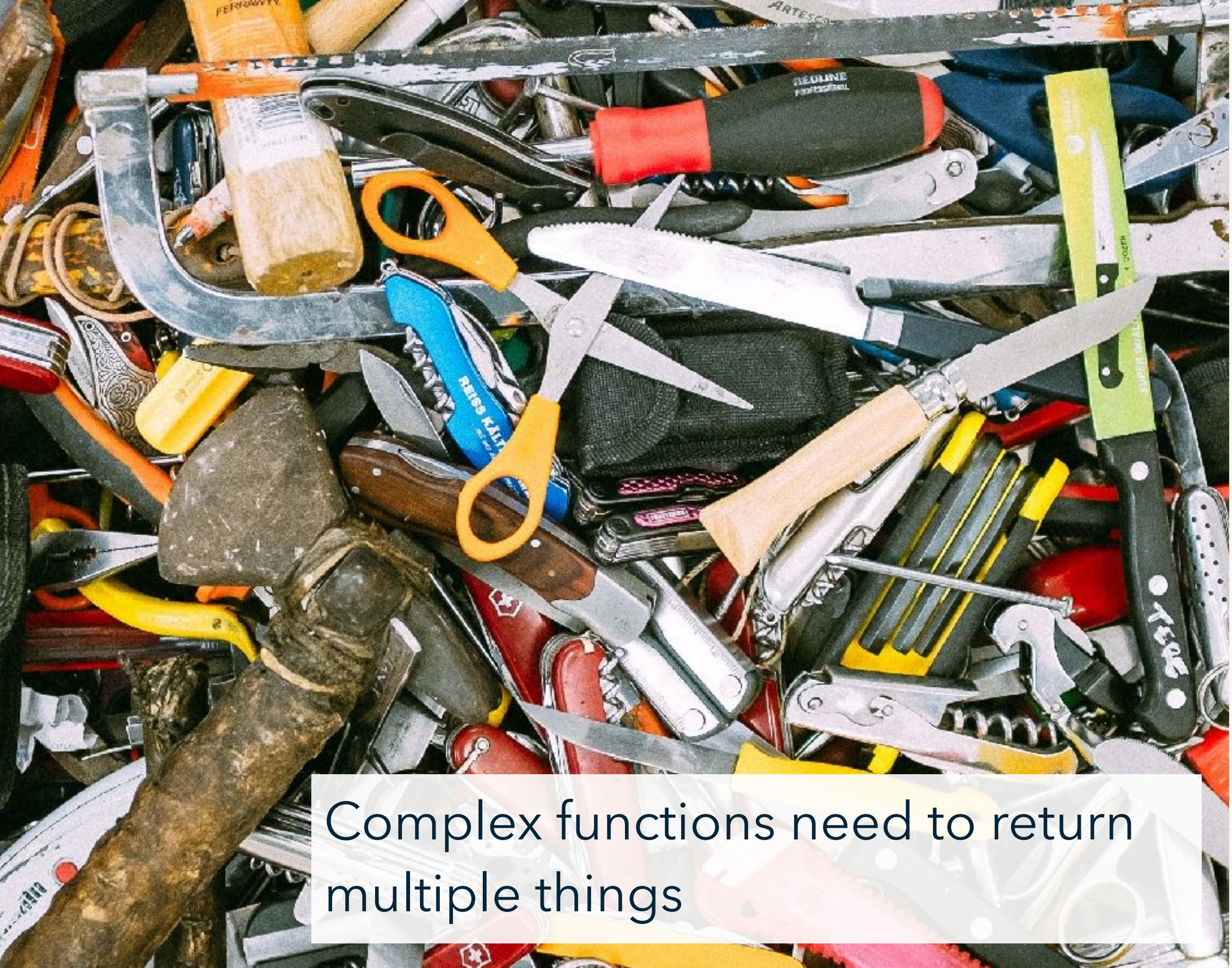
Why should you care about S3?

You’re already using S3 objects in your analyses

https://unsplash.com/photos/t5YUoHW6zRo

Important S3 objects in base R

```
data.frame()
factor()
Sys.Date()
Sys.time()
table()
```



Complex functions need to return multiple things

https://unsplash.com/photos/Kw_zQBAChws

This is obviously important for linear models

```
mod <- lm(mpg ~ wt, data = mtcars)
str(mod)
```

```
# But also their summaries
sum <- summary(mod)
str(sum)
```

Form follows function

https://unsplash.com/photos/gbP6hnvfYCM

One example is linear models

```
sum
#> Call:
#> lm(formula = mpg ~ wt, data = mtcars)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -4.5432 -2.3647 -0.1252  1.4096  6.8727
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
#> wt           -5.3445     0.5591  -9.559 1.29e-10 ***
#> ---
#> Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
#>
#> Residual standard error: 3.046 on 30 degrees of freedom
#> Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
#> F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

Another example is tibbles

Total size

Variable type

```
# A tibble: 53,940 x 10
   carat cut         color clarity depth table price     x     y     z
   <dbl> <ord>       <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1 0.230 Ideal       E     SI2      61.5  55.0   326  3.95  3.98  2.43
 2 0.210 Premium     E     SI1      59.8  61.0   326  3.89  3.84  2.31
 3 0.230 Good        E     VS1      56.9  65.0   327  4.05  4.07  2.31
 4 0.290 Premium     I     VS2      62.4  58.0   334  4.20  4.23  2.63
 5 0.310 Good        J     SI2      63.3  58.0   335  4.34  4.35  2.75
 6 0.240 "Very Good" J     VVS2     62.8  57.0   336  3.94  3.96  2.48
 7 0.240 "Very Good" I     VVS1     62.3  57.0   336  3.95  3.98  2.47
 8 0.260 "Very Good" H     SI1      61.9  55.0   337  4.07  4.11  2.53
 9 0.220 Fair        E     VS2      65.1  61.0   337  3.87  3.78  2.49
10 0.230 "Very Good" H     VS1      59.4  61.0   338  4.00  4.05  2.39
# ... with 53,930 more rows
```

Only shows first 10 rows

S3 makes packages extensible

New methods

Lets you extend other packages

New generics

Write packages in way that others can easily extend.

“Scalar” classes

a single complex object

Principle:

Provide consistent structure and print method for complex return values

Change project to:

[safely]

Challenge: how can the output of safely be improved?

```
library(purrr)
safe_log <- safely(log)

safe_log("a")
#> $result
#> NULL
#>
#> $error
#> <simpleError in log(...):
#>  non-numeric argument to
#>  mathematical function>
```

```
safe_log(10)
#> $result
#> [1] 2.302585
#>
#> $error
#> NULL
```

Creating a new S3 class

1. Figure out name `safely`
2. Define properties of the class
3. Write the constructor
4. Write methods

Your turn

What are the invariants of the results of safely?

```
safe_log <- purrr::safely(log)
# what do we know to be always true
# about the result of safe_log?
```

Invariants

Returns a list

- two components: result and error
- result should always come first
- one is always NULL

Now, write the constructor

```
new_safely <- function(result = NULL, error = NULL) {
  if (!is.null(result) && !is.null(error)) {
    stop(
      "One of `result` and `error` must be NULL",
      call. = FALSE
    )
  }

  structure(
    list(
      result = result,
      error = error
    ),
    class = "safely"
  )
}
```

Check inputs

Enforce structure and apply class

Definition of safely

```
safely <-  function(.f) {
  stopifnot(is.function(.f))

  function(...) {
    tryCatch({
      list(result = .f(...), error = NULL)
    }, error = function(e) {
      list(result = NULL, error = e)
    })
  }
}
```

Then use the constructor

```
safely <-  function(.f) {
  stopifnot(is.function(.f))

  function(...) {
    tryCatch({
      new_safely(result = .f(...))
    }, error = function(e) {
      new_safely(error = e)
    })
  }
}
```

Abbreviation	Test
`expect_null()`	Checks if a literal NULL

`expect_type()`
`expect_s3_class()`
`expect_s4_class()`
	Check that inherits from a given base type, S3 class, or S4 class.

`expect_true()` `expect_false()`	Catch all expectations for anything not otherwise covered

Your turn

Write tests to ensure that our new_safely() function returns the correct type of output regardless of whether or not an error occurs. (i.e. express the invariants as unit tests)

```r
# In tests/testthat/test-safely.R
context("test-safely.R")

test_that("can only supply error or result", {
  expect_error(new_safely(1, 2), "must be NULL")
})

test_that("it's ok for both to be null", {
  expect_error(new_safely(NULL, NULL), NA)
})
```

Expect no error

```r
test_that("result and error are captured", {
 s1 <- new_safely(result = 1)
 s2 <- new_safely(error = 1)

 expect_s3_class(s1, "safely")
 expect_equal(s1$result, 1)
 expect_equal(s1$error, NULL)

 expect_s3_class(s2, "safely")
 expect_equal(s2$result, NULL)
 expect_equal(s2$error, 1)
})
```

Now we can improve the output with a print method

```
safe_log(10)
#> <safely: ok>
#> [1] 2.302585
```

I think it's good practice to include type in <>

```
safe_log("a")
#> <safely: error>
#> Error: non-numeric argument to
#>  mathematical function
```

S3 methods all have the same basic structure

generic

Same arguments as generic

```
print.safely <- function(x, ...) {
}
```

class

Methods belong to **generics**, not classes

	Date	POSIXct	integer
print			
mean			
sum			

	Date	POSIXct	integer
print			
mean			
sum			

Your turn: fill in the blanks

```
# In R/safely.R
print.safely <- function(x, ...) {


}
```

```
# Useful helper found in utils.R
cat_line <- function(...) {
  cat(..., "\n", sep = "")
}
# See https://github.com/r-lib/cli for
# many more helpers.
```

Some test cases

```
f <- function() stop("message")
g <- function() 1
```

```
safe_f <- safely(f)
safe_g <- safely(g)

safe_f()
safe_g()
```

My print method

```
print.safely <- function(x, ...) {
  if (!is.null(x$error)) {
    cat_line("<safely: error>")
    cat_line("Error: ", x$error$message)
  } else {
    cat_line("<safely: ok>")
    print(x$result)
  }

  invisible(x)
}
```

Called primarily for side-effects

A little colour can be transformative

```
print.safely <- function(x, ...) {
  if (!is.null(x$error)) {
    cat_line("<safely: ", crayon::bold(crayon::red("error")), ">")
    cat_line(crayon::red("Error: "), x$error$message)
  } else {
    cat_line("<safely: ", crayon::green("ok"), ">")
    print(x$result)
  }

  invisible(x)
}
```

New generic

Change project to:

[bizarro]

Goal: create a bizarro function

```
bizarro("abc")
#> [1] "cba"
```

```
bizarro(1)
#> [1] -1
```

```
bizarro(c(TRUE, FALSE))
#> [1] FALSE TRUE
```

We could use if + else

```
str_reverse <- function(x) {
  purrr::map_chr(stringr::str_split(x, ""),
    ~ stringr::str_flatten(rev(.x))
  )
}

bizarro <- function(x) {
  if (is.character(x)) {
    str_reverse(x)
  } else if (is.numeric(x)) {
    -x
  } else if (is.logical(x)) {
    !x
  } else {
    stop(
      "Don't know how to make bizzaro <", class(x)[[1]], ">",
      call. = FALSE)
  }
}
```

But instead we'll create a new S3 generic

```
bizarro <- function(x) {
  UseMethod("bizarro")
}
```

Magically passes all arguments to correct method

`generic.class`

```
bizarro.character <- function(x) {
  str_reverse(x)
}

bizarro("abc")
#> [1] cba
```

Allows anyone to extend

Your turn

Implement:

1. a numeric method that multiplies by -1
2. a logical method which inverts TRUE/FALSE
3. a data frame method that bizarros the column names, as well as each column.

(i.e. get tests passing)

```r
bizarro.numeric <- function(x) {
  -x
}

bizarro.logical <- function(x) {
  !x
}
```

```r
bizarro.data.frame <- function(x) {
  names(x) <- bizarro(names(x))
  x[] <- purrr::map(x, bizarro)
  x
}
```

Useful technique Method for complex object: apply generic to components.

What happens when a method isn't available?

```
bizarro(factor(letters))
#> Error in UseMethod("bizarro") :
#>  no applicable method for 'bizarro'
#>  applied to an object of class "factor"

# How can we do better?
# We need to provide a catch-all default
method
```

```
bizarro.default <- function(x) {
  stop(
   "Don't know how to make bizzaro <",
   class(x)[[1]], ">",
   call. = FALSE
  )
}

bizarro(factor(letters))
#> Error: Don't know how to make
#> bizzaro <factor>
```

Your turn

What should bizzaro(factor("abc")) return?

Decide, encode your decisions in tests, then implement bizarro.factor().

One idea: reverse letters in factor levels

```
# In tests/testthat/test-bizarro.R
test_that("bizarro factors have levels reversed", {
  f1 <- factor(c("abc", "def", "abc"))
  f2 <- factor(c("cba", "fed", "cba"))

  expect_equal(bizarro(f1), f2)
  expect_equal(bizarro(f2), f1)
})
```

```r
# In R/bizarro.R
bizarro.factor <- function(x) {
  levels(x) <- bizarro(levels(x))
  x
}
```

Learning more

Advanced R (2nd ed) has four chapters

S3: https://adv-r.hadley.nz/s3.html
S4: https://adv-r.hadley.nz/s4.html
R6: https://adv-r.hadley.nz/r6.html
Trade-offs: https://adv-r.hadley.nz/oo-tradeoffs.html

Vector classes

Classes built on top of vector types

"vctrs will typically be used by other packages, making it easy for them to provide new classes of S3 vectors that are supported throughout the tidyverse (and beyond)."

--https://vctrs.r-lib.org/

Adapted from *Tidy Tools* by Hadley Wickham