13_S3

2023-03-06

13 S3

13.1 Introduction

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.4.0      v purrr   1.0.1
## v tibble  3.1.8      v dplyr   1.0.10
## v tidyr   1.2.1      v stringr 1.5.0
## v readr   2.1.3      v forcats 0.5.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(sloop)
```

13.2 Basics

```
f <- factor(c("a", "b", "c"))

typeof(f)
```

```
## [1] "integer"
```

```
attributes(f)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

```
unclass(f)
```

```
## [1] 1 2 3
## attr(,"levels")
## [1] "a" "b" "c"
```

```
ftype(print)
```

```
## [1] "S3"      "generic"
```

```
ftype(str)
```

```
## [1] "S3"      "generic"
```

```
ftype(unclass)
```

```
## [1] "primitive"
```

```
print(f)
```

```
## [1] a b c
## Levels: a b c
```

```
print(unclass(f))
```

```
## [1] 1 2 3
## attr(,"levels")
## [1] "a" "b" "c"
```

```
time <- strptime(c("2017-01-01", "2020-05-04 03:21"), "%Y-%m-%d")
str(time)
```

```
##  POSIXlt[1:2], format: "2017-01-01" "2020-05-04"
```

```
str(unclass(time))
```

```
## List of 11
##  $ sec   : num [1:2] 0 0
##  $ min   : int [1:2] 0 0
##  $ hour  : int [1:2] 0 0
##  $ mday  : int [1:2] 1 4
##  $ mon   : int [1:2] 0 4
##  $ year  : int [1:2] 117 120
##  $ wday  : int [1:2] 0 1
##  $ yday  : int [1:2] 0 124
##  $ isdst : int [1:2] 0 1
##  $ zone  : chr [1:2] "PST" "PDT"
##  $ gmtoff: int [1:2] NA NA
```

```
s3_dispatch(print(f))
```

```
## => print.factor
##  * print.default
```

```
ftype(t.test)
```

```
## [1] "S3"      "generic"
```

```
ftype(t.data.frame)
```

```
## [1] "S3"     "method"
```

```
weighted.mean.Date
```

```
## Error in eval(expr, envir, enclos): object 'weighted.mean.Date' not found
```

```
s3_get_method(weighted.mean.Date)
```

```
## function (x, w, ...)
## .Date(weighted.mean(unclass(x), w, ...))
## <bytecode: 0x000001c2608e8b60>
## <environment: namespace:stats>
```

When using `s3_dispatch()`

`=>` method exists and is found by `UseMethod()`.

`->` method exists and is used by `NextMethod()`.

`*` method exists but is not used.

Nothing (and greyed out in console): method does not exist.

13.2.1 Exercises

1. Describe the difference between `t.test()` and `t.data.frame()`. When is each function called?

```
ftype(t.test)
```

```
## [1] "S3"      "generic"
```

```
s3_dispatch(t.test(1:10, y = c(7:20)))
```

```
##    t.test.integer
##    t.test.numeric
## => t.test.default
```

```
ftype(t.data.frame)
```

```
## [1] "S3"     "method"
```

```
s3_dispatch(t(data.frame(a=1:5, b = 6:10)))
```

```
## => t.data.frame
## -> t.default
```

```
s3_dispatch(t.data.frame(data.frame(a=1:5, b = 6:10)))
```

```
##    t.data.frame.data.frame
##    t.data.frame.default
```

t.test is a generic while t.data.frame is a method. t.test gets called first since it is a generic and then it finds the right implementation for the job. t.data.frame is called once the generic determines it is the correct method by performing method dispatch.

2. Make a list of commonly used base R functions that contain `.` in their name but are not S3 methods.

```
ftype(read.csv)
```

```
## [1] "function"
```

```
ftype(as.character)
```

```
## [1] "primitive" "generic"
```

```
ftype(all.equal)
```

```
## [1] "S3"      "generic"
```

```
ftype(file.copy)
```

```
## [1] "internal"
```

```
ftype(format.info)
```

```
## [1] "internal"
```

```
ftype(is.na)
```

```
## [1] "primitive" "generic"
```

```
ftype(Sys.info)
```

```
## [1] "internal"
```

3. What does the `as.data.frame.data.frame()` method do? Why is it confusing? How could you avoid this confusion in your own code?

```
ftype(as.data.frame.data.frame)
```

```
## [1] "S3"     "method"
```

```
s3_dispatch(as.data.frame.data.frame(mtcars))
```

```
##    as.data.frame.data.frame.data.frame
##    as.data.frame.data.frame.default
```

```
as.data.frame.data.frame(mtcars)
```

```
##                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant            18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360         14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D          24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230           22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280           19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C          17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE         16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL         17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC        15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4  8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial  14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128           32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic        30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla     33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona      21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin        15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28         13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird   19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9          27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2      26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa       30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L     15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino       19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora      15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E         21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
s3_dispatch(as.data.frame.data.frame(matrix(1:25, nrow = 5)))
```

```
##    as.data.frame.data.frame.matrix
##    as.data.frame.data.frame.integer
##    as.data.frame.data.frame.numeric
##    as.data.frame.data.frame.default
```

```
as.data.frame.data.frame(matrix(1:25, nrow = 5))
```

```
## Error in if (i > 1L) class(x) <- cl[-(1L:(i - 1L))]: missing value where TRUE/FALSE needed
```

```
s3_dispatch(as.data.frame(matrix(1:25, nrow = 5)))
```

```
## => as.data.frame.matrix
##  * as.data.frame.integer
##  * as.data.frame.numeric
##  * as.data.frame.default
```

```
as.data.frame(matrix(1:25, nrow = 5))
```

```
##   V1 V2 V3 V4 V5
## 1  1  6 11 16 21
## 2  2  7 12 17 22
## 3  3  8 13 18 23
## 4  4  9 14 19 24
## 5  5 10 15 20 25
```

It checks if the object is a `data.frame` and coerces it if possible and then attempts to cast it as a `data.frame` Easier to just use the generic instead of the specific method, let method dispatch do the work for you

4. Describe the difference in behaviour in these two calls.

```
set.seed(1014)
some_days <- as.Date("2017-01-31") + sample(10, 5)

some_days
```

```
## [1] "2017-02-07" "2017-02-05" "2017-02-06" "2017-02-10" "2017-02-04"
```

```
s3_dispatch(mean(some_days))
```

```
## => mean.Date
##  * mean.default
```

```
class(some_days)
```

```
## [1] "Date"
```

```
mean(some_days)
```

```
## [1] "2017-02-06"
```

```
s3_dispatch(mean(unclass(some_days)))
```

```
##    mean.double
##    mean.numeric
## => mean.default
```

```
class(unclass(some_days))
```

```
## [1] "numeric"
```

```
str(unclass(some_days))
```

```
##  num [1:5] 17204 17202 17203 17207 17201
```

```
mean(unclass(some_days))
```

```
## [1] 17203.4
```

The first one calculates mean using the `mean.Date` method since it sees the class is "Date". In the second class is stripped so it becomes a numeric and this causes the `mean.default` method to be used.

5. What class of object does the following code return? What base type is it built on? What attributes does it use?

```
x <- ecdf(rpois(100, 10))
x
```

```
## Empirical CDF
## Call: ecdf(rpois(100, 10))
##  x[1:18] =      2,      3,      4,  ...,     18,     19
```

```
class(x)
```

```
## [1] "ecdf"     "stepfun"  "function"
```

```
str(x)
```

```
## function (v)
##  - attr(*, "class")= chr [1:3] "ecdf" "stepfun" "function"
##  - attr(*, "call")= language ecdf(rpois(100, 10))
```

```
typeof(unclass(x))
```

```
## [1] "closure"
```

`x` is class ecdf, with two more classes of stepfun and function. it's base class is a closure (function). The attribute it uses is the expression used when it was created `rpois(100,10)`.

6. What class of object does the following code return? What base type is it built on? What attributes does it use?

```r
x <- table(rpois(100, 5))
x
```

```
##
##  1  2  3  4  5  6  7  8  9 10
##  7  5 18 14 15 15 14  4  5  3
```

```r
class(x)
```

```
## [1] "table"
```

```r
typeof(x)
```

```
## [1] "integer"
```

```r
str(x)
```

```
##  'table' int [1:10(1d)] 7 5 18 14 15 15 14 4 5 3
##  - attr(*, "dimnames")=List of 1
##   ..$ : chr [1:10] "1" "2" "3" "4" ...
```

```r
attributes(x)
```

```
## $dim
## [1] 10
##
## $dimnames
## $dimnames[[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
##
##
## $class
## [1] "table"
```

```r
class(unclass(x))
```

```
## [1] "array"
```

Class is `table`. It is built on top of the base type `integer`. It uses the attribute `dimnames`

13.3 Classes

```r
# Create and assign class in one step
x <- structure(list(), class = "my_class")

# Create, then set class
x <- list()
class(x) <- "my_class"
```

```
class(x)
```

```
## [1] "my_class"
```

```
inherits(x, "my_class")
```

```
## [1] TRUE
```

```
inherits(x, "your_class")
```

```
## [1] FALSE
```

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
```

```
## [1] "lm"
```

```
print(mod)
```

```
##
## Call:
## lm(formula = log(mpg) ~ log(disp), data = mtcars)
##
## Coefficients:
## (Intercept)    log(disp)
##      5.3810      -0.4586
```

```
# Turn it into a date (?!)
class(mod) <- "Date"

# Unsurprisingly this doesn't work very well
print(mod)
```

```
## Error in as.POSIXlt.Date(x): 'list' object cannot be coerced to type 'integer'
```

13.3.1 Constructors

```
new_Date <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "Date")
}

new_Date(c(-1, 0, 1))
```

```
## [1] "1969-12-31" "1970-01-01" "1970-01-02"
```

```
new_difftime <- function(x = double(), units = "secs") {
  stopifnot(is.double(x))
  units <- match.arg(units, c("secs", "mins", "hours", "days", "weeks"))

  structure(x,
    class = "difftime",
    units = units
  )
}

new_difftime(c(1, 10, 3600), "secs")
```

```
## Time differences in secs
## [1]    1   10 3600
```

```
new_difftime(52, "weeks")
```

```
## Time difference of 52 weeks
```

13.3.2 Validators

```
new_factor <- function(x = integer(), levels = character()) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  structure(
    x,
    levels = levels,
    class = "factor"
  )
}

new_factor(1:5, "a")
```

```
## Error in as.character.factor(x): malformed factor
```

```
new_factor(0:1, "a")
```

```
## Error in as.character.factor(x): malformed factor
```

```
validate_factor <- function(x) {
  values <- unclass(x)
  levels <- attr(x, "levels")

  if (!all(!is.na(values) & values > 0)) {
    stop(
      "All `x` values must be non-missing and greater than zero",
      call. = FALSE
    )
```

```
  }

  if (length(levels) < max(values)) {
    stop(
      "There must be at least as many `levels` as possible values in `x`",
      call. = FALSE
    )
  }

  x
}

validate_factor(new_factor(1:5, "a"))
```

```
## Error: There must be at least as many `levels` as possible values in `x`
```

```
validate_factor(new_factor(0:1, "a"))
```

```
## Error: All `x` values must be non-missing and greater than zero
```

13.3.3 Helpers

```
new_difftime(1:10)
```

```
## Error in new_difftime(1:10): is.double(x) is not TRUE
```

```
difftime <- function(x = double(), units = "secs") {
  x <- as.double(x)
  new_difftime(x, units = units)
}

difftime(1:10)
```

```
## Time differences in secs
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)
  validate_factor(new_factor(ind, levels))
}

factor(c("a", "a", "b"))
```

```
## [1] a a b
## Levels: a b
```

```
POSIXct <- function(year = integer(),
                    month = integer(),
                    day = integer(),
                    hour = 0L,
                    minute = 0L,
                    sec = 0,
                    tzone = "") {
  ISOdatetime(year, month, day, hour, minute, sec, tz = tzone)
}

POSIXct(2020, 1, 1, tzone = "America/New_York")
```

```
## [1] "2020-01-01 EST"
```

13.3.4 Exercises

1. Write a constructor for `data.frame` objects. What base type is a data frame built on? What attributes does it use? What are the restrictions placed on the individual elements? What about the names?

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
class(mtcars)
```

```
## [1] "data.frame"
```

```
unclass(mtcars)
```

```
## $mpg
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
##
## $cyl
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
##
## $disp
##  [1] 160.0 160.0 108.0 258.0 360.0 225.0 360.0 146.7 140.8 167.6 167.6 275.8
```

```
## [13] 275.8 275.8 472.0 460.0 440.0  78.7  75.7  71.1 120.1 318.0 304.0 350.0
## [25] 400.0  79.0 120.3  95.1 351.0 145.0 301.0 121.0
##
## $hp
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
##
## $drat
##  [1] 3.90 3.90 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 3.92 3.07 3.07 3.07 2.93
## [16] 3.00 3.23 4.08 4.93 4.22 3.70 2.76 3.15 3.73 3.08 4.08 4.43 3.77 4.22 3.62
## [31] 3.54 4.11
##
## $wt
##  [1] 2.620 2.875 2.320 3.215 3.440 3.460 3.570 3.190 3.150 3.440 3.440 4.070
## [13] 3.730 3.780 5.250 5.424 5.345 2.200 1.615 1.835 2.465 3.520 3.435 3.840
## [25] 3.845 1.935 2.140 1.513 3.170 2.770 3.570 2.780
##
## $qsec
##  [1] 16.46 17.02 18.61 19.44 17.02 20.22 15.84 20.00 22.90 18.30 18.90 17.40
## [13] 17.60 18.00 17.98 17.82 17.42 19.47 18.52 19.90 20.01 16.87 17.30 15.41
## [25] 17.05 18.90 16.70 16.90 14.50 15.50 14.60 18.60
##
## $vs
##  [1] 0 0 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 0 0 0 1
##
## $am
##  [1] 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1
##
## $gear
##  [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4
##
## $carb
##  [1] 4 4 1 1 2 1 4 2 2 4 4 3 3 3 4 4 4 1 2 1 1 2 2 4 2 1 2 2 4 6 8 2
##
## attr(,"row.names")
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```
str(unclass(mtcars))
```

```
## List of 11
##  $ mpg : num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num [1:32] 6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num [1:32] 160 160 108 258 360 ...
##  $ hp  : num [1:32] 110 110 93 110 175 105 245 62 95 123 ...
```

```
##  $ drat: num [1:32] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num [1:32] 16.5 17 18.6 19.4 17 ...
##  $ vs  : num [1:32] 0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num [1:32] 1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num [1:32] 4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num [1:32] 4 4 1 1 2 1 4 2 2 4 ...
##  - attr(*, "row.names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
```

```
new_data.frame <- function(x, row.names = NULL){ # x is the input list
  stopifnot(is.list(x)) # Require list
  stopifnot(length(unique(lengths(x))) == 1) # Require same length vectors
  n <- unique(lengths(x))
  if(is.null(row.names)){
    row.names <- as.character(1:n)
  } else {
    stopifnot(is.character(row.names), length(row.names) == n)
  }
  structure(
    x,
    class = "data.frame",
    row.names = row.names
  )
}

dat_list <- list(a = 1:5, b = 6:10)
new_data.frame(dat_list)
```

```
##   a  b
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

```
new_data.frame(dat_list, row.names = LETTERS[1:5])
```

```
##   a  b
## A 1  6
## B 2  7
## C 3  8
## D 4  9
## E 5 10
```

```
dat_list2 <- list(a = 1:5, b = 6:11)
new_data.frame(dat_list2)
```

```
## Error in new_data.frame(dat_list2): length(unique(lengths(x))) == 1 is not TRUE
```

data.frames are built on top of lists. The list contains named vectors and a row.names attribute which is optional. Each name in row.names has to be unique and each element of the list must be the same length

2. Enhance my `factor()` helper to have better behaviour when one or more values is not found in levels. What does `base::factor()` do in this situation?

```
factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)
  if(any(is.na(ind))){
    cat("removing", x[is.na(ind)], "since they are not present in levels\n")
    x <- x[!is.na(ind)]
    ind <- match(x, levels)
  }
  validate_factor(new_factor(ind, levels))
}

x <- c("a", "a", "b")
factor(x)
```

```
## [1] a a b
## Levels: a b
```

```
factor(x = x,
       levels = c("a", "c"))
```

```
## removing b since they are not present in levels

## [1] a a
## Levels: a c
```

```
base::factor(x = x,
       levels = c("a", "c"))
```

```
## [1] a    a    <NA>
## Levels: a c
```

In this case `base::factor` will convert values not present in level to `NA`

3. Carefully read the source code of `factor()`. What does it do that my constructor does not?

If any of the indexes are not an integer it stops the execution. This means an `NA`s from `ind` in the helper function will cause the function to stop. In the base function if levels are not provided, it generates levels from input. The base function handles cases of input where the input has levels not listed in the levels. It creates the vector, converts non-level values to `NA` and then adds the levels attribute.

4. Factors have an optional "contrasts" attribute. Read the help for `C()`, and briefly describe the purpose of the attribute. What type should it have? Rewrite the `new_factor()` constructor to include this attribute.

```
new_factor <- function(x = integer(), levels = character(), contrasts = NULL) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))
  if(!is.null(contrasts)){
```

```
    stopifnot(is.matrix(contrasts) && is.numeric(contrasts))
  }

  structure(
    x,
    levels = levels,
    class = "factor",
    contrasts = contrasts
  )
}

model3 <- glm(cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
     C(alcgp, , 1), data = esoph, family = binomial())
summary(model3)
```

```
##
## Call:
## glm(formula = cbind(ncases, ncontrols) ~ agegp + C(tobgp, , 1) +
##     C(alcgp, , 1), family = binomial(), data = esoph)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.3018  -0.7234  -0.2306   0.5737   2.4290
##
## Coefficients:
##                 Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -1.15264    0.20326  -5.671 1.42e-08 ***
## agegp.L          3.81892    0.67862   5.627 1.83e-08 ***
## agegp.Q         -1.49473    0.60671  -2.464   0.0138 *
## agegp.C          0.07923    0.46318   0.171   0.8642
## agegp^4          0.12136    0.32203   0.377   0.7063
## agegp^5         -0.24856    0.21153  -1.175   0.2400
## C(tobgp, , 1).L  0.98287    0.21519   4.568 4.93e-06 ***
## C(alcgp, , 1).L  2.38736    0.23462  10.175  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 367.953  on 87  degrees of freedom
## Residual deviance:  91.121  on 80  degrees of freedom
## AIC: 222.18
##
## Number of Fisher Scoring iterations: 6
```

C needs a mumeric matrix or a suitable function. When modeling it allows us to preset the contrasts we want to use.

5. Read the documentation for `utils::as.roman()`. How would you write a constructor for this class? Does it need a validator? What might a helper do?

It converts integer numbers into roman numerals. The new object is has the class "roman" Number range is 1 to 3899. It uses a basic dictionary called `.romans`. Constructor makes sure we have in integer input. Validator can check to make sure it's in range. Helper could round numbers to the nearest integer

```
.romans
```

```
##    M   CM    D   CD    C   XC    L   XL    X   IX    V   IV    I
## 1000  900  500  400  100   90   50   40   10    9    5    4    1
```

```
a <- as.roman(13)
a
```

```
## [1] XIII
```

```
typeof(a)
```

```
## [1] "integer"
```

```
str(a)
```

```
##  'roman' int XIII
```

```
attributes(a)
```

```
## $class
## [1] "roman"
```

13.4 Generics and methods

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x000001c259e60268>
## <environment: namespace:base>
```

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

13.4.1 Method dispatch

```
x <- Sys.Date()
s3_dispatch(print(x))
```

```
## => print.Date
##  * print.default
```

```
paste0("generic", ".", c(class(x), "default"))
```

```
## [1] "generic.Date"    "generic.default"
```

```
x <- matrix(1:10, nrow = 2)
s3_dispatch(mean(x))
```

```
##    mean.matrix
##    mean.integer
##    mean.numeric
## => mean.default
```

```
paste0("generic", ".", c(class(x), "default"))
```

```
## [1] "generic.matrix"  "generic.array"   "generic.default"
```

```
s3_dispatch(sum(Sys.time()))
```

```
##    sum.POSIXct
##    sum.POSIXt
##    sum.default
## => Summary.POSIXct
##    Summary.POSIXt
##    Summary.default
## -> sum (internal)
```

13.4.2 Finding methods

```
s3_methods_generic("mean")
```

```
## # A tibble: 7 x 4
##   generic class      visible source
##   <chr>   <chr>      <lgl>   <chr>
## 1 mean    Date       TRUE    base
## 2 mean    default    TRUE    base
## 3 mean    difftime   TRUE    base
## 4 mean    POSIXct    TRUE    base
## 5 mean    POSIXlt    TRUE    base
## 6 mean    quosure    FALSE   registered S3method
## 7 mean    vctrs_vctr FALSE   registered S3method
```

```
s3_methods_class("ordered")
```

```
## # A tibble: 6 x 4
##   generic       class   visible source
##   <chr>         <chr>   <lgl>   <chr>
## 1 as.data.frame ordered TRUE    base
## 2 Ops           ordered TRUE    base
## 3 relevel       ordered FALSE   registered S3method
## 4 scale_type    ordered FALSE   registered S3method
## 5 Summary       ordered TRUE    base
## 6 type_sum      ordered FALSE   registered S3method
```

13.4.3 Creating methods

13.4.4 Exercises

1. Read the source code for `t()` and `t.test()` and confirm that `t.test()` is an S3 generic and not an S3 method. What happens if you create an object with class `test` and call `t()` with it? Why?

```
t
```

```
## function (x)
## UseMethod("t")
## <bytecode: 0x000001c26045ec08>
## <environment: namespace:base>
```

```
s3_methods_generic("t")
```

```
## # A tibble: 6 x 4
##   generic class      visible source
##   <chr>   <chr>      <lgl>   <chr>
## 1 t       data.frame TRUE    base
## 2 t       default    TRUE    base
## 3 t       gtable     FALSE   registered S3method
## 4 t       ts         FALSE   registered S3method
## 5 t       vctrs_sclr FALSE   registered S3method
## 6 t       vctrs_vctr FALSE   registered S3method
```

```
ftype(t)
```

```
## [1] "S3"      "generic"
```

```
t.test
```

```
## function (x, ...)
## UseMethod("t.test")
## <bytecode: 0x000001c2593de4a0>
## <environment: namespace:stats>
```

```
s3_methods_generic("t.test")
```

```
## # A tibble: 2 x 4
##   generic class   visible source
##   <chr>   <chr>   <lgl>   <chr>
## 1 t.test  default FALSE   registered S3method
## 2 t.test  formula FALSE   registered S3method
```

```
ftype(t.test)
```

```
## [1] "S3"      "generic"
```

```
x <- matrix(1:25, ncol = 5)
x <- structure(x,
    class = "test"
  )

x
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    6   11   16   21
## [2,]    2    7   12   17   22
## [3,]    3    8   13   18   23
## [4,]    4    9   14   19   24
## [5,]    5   10   15   20   25
## attr(,"class")
## [1] "test"
```

```
class(x)
```

```
## [1] "test"
```

```
t(x)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
## attr(,"class")
## [1] "test"
```

```
x <- matrix(1:25, ncol = 5)
t(x)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
```

Still worked, used default method

2. What generics does the `table` class have methods for?

```
s3_methods_class("table")
```

```
## # A tibble: 11 x 4
##    generic       class visible source
##    <chr>         <chr> <lgl>   <chr>
```

```
##  1 [             table TRUE    base
##  2 aperm         table TRUE    base
##  3 as.data.frame table TRUE    base
##  4 as_tibble     table FALSE   registered S3method
##  5 Axis          table FALSE   registered S3method
##  6 lines         table FALSE   registered S3method
##  7 plot          table FALSE   registered S3method
##  8 points        table FALSE   registered S3method
##  9 print         table TRUE    base
## 10 summary       table TRUE    base
## 11 tail          table FALSE   registered S3method
```

3. What generics does the `ecdf` class have methods for?

```
s3_methods_class("ecdf")
```

```
## # A tibble: 4 x 4
##   generic  class visible source
##   <chr>    <chr> <lgl>   <chr>
## 1 plot     ecdf  TRUE    stats
## 2 print    ecdf  FALSE   registered S3method
## 3 quantile ecdf  FALSE   registered S3method
## 4 summary  ecdf  FALSE   registered S3method
```

4. Which base generic has the greatest number of defined methods?

```
possibles <- ls("package:base")
possibles_functions <- mget(possibles, envir = baseenv())
possibles_functions <- unlist(map(possibles_functions, is_function))
possibles <- possibles[possibles_functions]
possibles_generics <- unlist(map(possibles, is_s3_generic))
possibles <- possibles[possibles_generics]
names(possibles) <- possibles
for(i in 1:length(possibles)){
  possibles[i] <- nrow(s3_methods_generic(possibles[i]))
}

possibles %>%
  as.numeric() %>%
  set_names(., names(possibles)) %>%
  sort(decreasing = T) %>%
  head()
```

```
##         print        format             [  as.character       summary
##           345           147            68            50            45
## as.data.frame
##            37
```

5. Carefully read the documentation for `UseMethod()` and explain why the following code returns the results that it does. What two usual rules of function evaluation does `UseMethod()` violate?

```
g <- function(x) {
  x <- 10
  y <- 10
  UseMethod("g")
}
g.default <- function(x) c(x = x, y = y)

x <- 1
y <- 1
g(x)
```

```
##  x  y
##  1 10
```

```
g.default(x)
```

```
## x y
## 1 1
```

```
g <- function(x) {
  x <- 10
  y <- 10
  UseMethod("g")
}

g.factor <- function(x) cat("FACTOR\n")
g.integer <- function(x) cat("INTEGER\n")
g.default <- function(x) cat("NOT THOSE\n")
x <- factor("A")
x2 <- 1L

g(x)
```

```
## FACTOR
```

```
g(x2)
```

```
## INTEGER
```

```
g(list())
```

```
## NOT THOSE
```

```
g(c())
```

```
## NOT THOSE
```

The function uses the value of x we provide and not the value set in the generic. Method dispatch takes place based on the class(es) of the first argument to the generic function or of the object supplied as an argument to UseMethod or NextMethod.

6. What are the arguments to [? Why is this a hard question to answer?

```
ftype(`[`)
```

```
## [1] "primitive" "generic"
```

```
formals(`[`)
```

```
## NULL
```

```
s3_methods_generic("[")
```

```
## # A tibble: 68 x 4
##    generic class                visible source
##    <chr>   <chr>                <lgl>   <chr>
##  1 [       acf                  FALSE   registered S3method
##  2 [       arrow                FALSE   registered S3method
##  3 [       AsIs                 TRUE    base
##  4 [       bibentry             FALSE   registered S3method
##  5 [       cell_addr            FALSE   registered S3method
##  6 [       check_details_changes FALSE   registered S3method
##  7 [       cli_doc              FALSE   registered S3method
##  8 [       data.frame           TRUE    base
##  9 [       Date                 TRUE    base
## 10 [       difftime             TRUE    base
## # ... with 58 more rows
```

```
names(formals(`[.data.frame`))
```

```
## [1] "x"    "i"    "j"    "drop"
```

```
names(formals(`[.Date`))
```

```
## [1] "x"    "..."  "drop"
```

There's a lot of methods and each one has different arguments

13.5 Object styles

```
x <- as.POSIXlt(ISOdatetime(2020, 1, 1, 0, 0, 1:3))
x
```

```
## [1] "2020-01-01 00:00:01 PST" "2020-01-01 00:00:02 PST"
## [3] "2020-01-01 00:00:03 PST"
```

```
length(x)
```

```
## [1] 3
```

```
unclass(x)
```

```
## $sec
## [1] 1 2 3
##
## $min
## [1] 0 0 0
##
## $hour
## [1] 0 0 0
##
## $mday
## [1] 1 1 1
##
## $mon
## [1] 0 0 0
##
## $year
## [1] 120 120 120
##
## $wday
## [1] 3 3 3
##
## $yday
## [1] 0 0 0
##
## $isdst
## [1] 0 0 0
##
## $zone
## [1] "PST" "PST" "PST"
##
## $gmtoff
## [1] -28800 -28800 -28800
##
## attr(,"tzone")
## [1] ""    "PST" "PDT"
```

```
length(unclass(x))
```

```
## [1] 11
```

```
x[[1]]
```

```
## [1] "2020-01-01 00:00:01 PST"
```

```
unclass(x)[[1]]
```

```
## [1] 1 2 3
```

```
x <- data.frame(x = 1:100, y = 1:100)
length(x)
```

```
## [1] 2
```

```
nrow(x)
```

```
## [1] 100
```

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

```
## [1] 12
```

13.5.1 Exercises

1. Categorise the objects returned by `lm()`, `factor()`, `table()`, `as.Date()`, `as.POSIXct()` `ecdf()`, `ordered()`, `I()` into the styles described above.

```
# lm() is a scalar style object
mod <- lm(mpg ~ wt, data = mtcars)
length(mod)
```

```
## [1] 12
```

```
# factor() is a vector style object
x <- factor(c("a","b","c"))
length(x)
```

```
## [1] 3
```

```
# table() is a vector style object
x <- table(rpois(100, 5))
length(x)
```

```
## [1] 13
```

```
# as.Date() is a vector style object
x <- as.Date(c(32768, 1444), origin = "1900-01-01")
length(x)
```

```
## [1] 2
```

```
unclass(x)
```

```
## [1]   7201 -24123
```

```
# as.POSIXct() is a vector style object
x <- as.POSIXct(Sys.time(), origin = "1960-01-01")
length(x)
```

```
## [1] 1
```

```
unclass(x)
```

```
## [1] 1678911489
```

```
# ecdf() is a scalar style object
x <- ecdf(rpois(100, 10))
length(x)
```

```
## [1] 1
```

```
unclass(x)
```

```
## function (v)
## .approxfun(x, y, v, method, yleft, yright, f, na.rm)
## <bytecode: 0x000001c25ed0d5c8>
## <environment: 0x000001c25b0e6850>
## attr(,"call")
## ecdf(rpois(100, 10))
```

```
# ordered() is a vector style object
x <- ordered(c(1,5,2,5,3))
length(x)
```

```
## [1] 5
```

```
# I() depends
x <- I(c(1:5))
length(x)
```

```
## [1] 5
```

```
dim(x)
```

```
## NULL
```

```
x <- I(iris)
length(x)
```

```
## [1] 5
```

```
dim(x)
```

```
## [1] 150   5
```

2. What would a constructor function for lm objects, new_lm(), look like? Use ?lm and experimentation to figure out the required fields and their types.

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

```
## List of 12
##  $ coefficients : Named num [1:2] 37.29 -5.34
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "wt"
##  $ residuals    : Named num [1:32] -2.28 -0.92 -2.09 1.3 -0.2 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##  $ effects      : Named num [1:32] -113.65 -29.116 -1.661 1.631 0.111 ...
##   ..- attr(*, "names")= chr [1:32] "(Intercept)" "wt" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:32] 23.3 21.9 24.9 20.1 18.9 ...
##   ..- attr(*, "names")= chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "wt"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.18 1.05
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 30
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = mpg ~ wt, data = mtcars)
##  $ terms        :Classes 'terms', 'formula'  language mpg ~ wt
##   .. ..- attr(*, "variables")= language list(mpg, wt)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "mpg" "wt"
##   .. .. .. ..$ : chr "wt"
##   .. ..- attr(*, "term.labels")= chr "wt"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

```
##   .. ..- attr(*, "predvars")= language list(mpg, wt)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "mpg" "wt"
##  $ model        :'data.frame':   32 obs. of  2 variables:
##   ..$ mpg: num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##   ..$ wt : num [1:32] 2.62 2.88 2.32 3.21 3.44 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language mpg ~ wt
##   .. .. ..- attr(*, "variables")= language list(mpg, wt)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "mpg" "wt"
##   .. .. .. .. ..$ : chr "wt"
##   .. .. ..- attr(*, "term.labels")= chr "wt"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(mpg, wt)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "mpg" "wt"
##  - attr(*, "class")= chr "lm"
```

```
attributes(mod)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

```
map_chr(mod, typeof)
```

```
##  coefficients     residuals       effects          rank fitted.values
##      "double"      "double"      "double"     "integer"      "double"
##        assign            qr   df.residual       xlevels          call
##     "integer"        "list"     "integer"        "list"    "language"
##         terms         model
##    "language"        "list"
```

You would need to ensure each piece of lm object is included and is the right type as listed above.