10_Function_Factories

2023-01-31

10 Function factories

10.1 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()
```

```
power1 <- function(exp) {
  function(x) {
    x ^ exp
  }
}

square <- power1(2)
cube <- power1(3)
```

```
library(rlang)
```

```
##
## Attaching package: 'rlang'

## The following objects are masked from 'package:purrr':
##
##     %@%, flatten, flatten_chr, flatten_dbl, flatten_int, flatten_lgl,
##     flatten_raw, invoke, splice
```

```
library(ggplot2)
library(scales)
```

```
##
## Attaching package: 'scales'
```

```
## The following object is masked from 'package:purrr':
##
##     discard

## The following object is masked from 'package:readr':
##
##     col_factor
```

10.2 Factory fundamentals

```
square
```

```
## function(x) {
##     x ^ exp
##   }
## <environment: 0x000001f7088bc7b8>
```

```
cube
```

```
## function(x) {
##     x ^ exp
##   }
## <bytecode: 0x000001f708a34718>
## <environment: 0x000001f708913a30>
```

```
env_print(square)
```

```
## <environment: 0x000001f7088bc7b8>
## Parent: <environment: global>
## Bindings:
## * exp: <lazy>
```

```
env_print(cube)
```

```
## <environment: 0x000001f708913a30>
## Parent: <environment: global>
## Bindings:
## * exp: <lazy>
```

```
fn_env(square)$exp
```

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

```
fn_env(cube)$exp
```

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

10.2.2 Diagram conventions

```r
square(10)
```

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

10.2.3 Forcing evaluation

```r
x <- 2
square <- power1(x)
x <- 3
```

```r
square(2)
```

```
## [1] 8
```

```r
power2 <- function(exp) {
  force(exp)
  function(x) {
    x ^ exp
  }
}

x <- 2
square <- power2(x)
x <- 3
square(2)
```

```
## [1] 4
```

10.2.4 Stateful functions

```r
new_counter <- function() {
  i <- 0

  function() {
    i <<- i + 1
    i
  }
}

counter_one <- new_counter()
counter_two <- new_counter()
```

```r
counter_one()
```

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

```r
counter_one()
```

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

```r
counter_two()
```

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

```r
counter_one()
```

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

```r
counter_two()
```

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

10.2.5 Garbage collection

```r
f1 <- function(n) {
  x <- runif(n)
  m <- mean(x)
  function() m
}

g1 <- f1(1e6)
lobstr::obj_size(g1)
```

```
## 8.01 MB
```

```r
#> 8,013,104 B

f2 <- function(n) {
  x <- runif(n)
  m <- mean(x)
  rm(x)
  function() m
}

g2 <- f2(1e6)
lobstr::obj_size(g2)
```

```
## 13.12 kB
```

10.2.6 Exercises

1. The definition of `force()` is simple:

```
force
```

```
## function (x)
## x
## <bytecode: 0x000001f77fa74b60>
## <environment: namespace:base>
```

Why is it better to force(x) instead of just x?

Using `force(x)` is better than `x` because it makes the evaluation of x occur immediately when the function is created rather than lazily executing the first time the function is called. If the value of `x` changes before the function is called, the value of `x` when the function was created will not be used and the new value of `x` will be used

```
timeser <- function(multiplier){
  function(x) x * multiplier
}
x <- 2
doubler <- timeser(x)
# Function called before x changed
doubler(2)
```

```
## [1] 4
```

```
x <- 3
doubler(2)
```

```
## [1] 4
```

```
x <- 2
doubler <- timeser(x)
# Function not called before x changed
x <- 3
doubler(2)
```

```
## [1] 6
```

```
# Forced operation
timeser2 <- function(multiplier){
  force(multiplier)
  function(x) x * multiplier
}
x <- 2
doubler <- timeser2(x)
# Function not called before x changed
x <- 3
doubler(2)
```

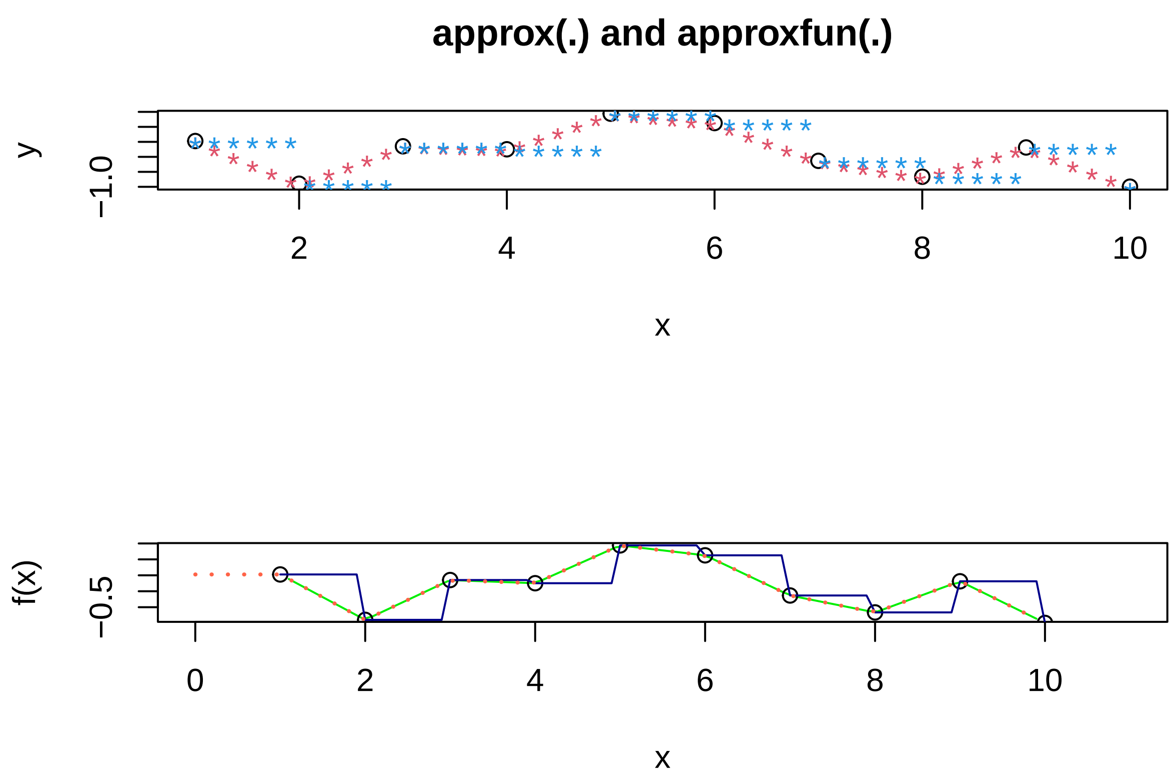
```
## [1] 4
```

2. Base R contains two function factories, `approxfun()` and `ecdf()`. Read their documentation and experiment to figure out what the functions do and what they return.

```r
x <- 1:10
y <- rnorm(10)
par(mfrow = c(2,1))
plot(x, y, main = "approx(.) and approxfun(.)")
points(approx(x, y), col = 2, pch = "*")
points(approx(x, y, method = "constant"), col = 4, pch = "*")

f <- approxfun(x, y)
curve(f(x), 0, 11, col = "green2")
points(x, y)
is.function(fc <- approxfun(x, y, method = "const")) # TRUE
```

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

```r
curve(fc(x), 0, 10, col = "darkblue", add = TRUE)
## different extrapolation on left and right side :
plot(approxfun(x, y, rule = 2:1), 0, 11,
     col = "tomato", add = TRUE, lty = 3, lwd = 2)
```




```r
x <- rnorm(12)
Fn <- ecdf(x)
Fn     # a *function*
```

```
## Empirical CDF
```

```
## Call: ecdf(x)
##  x[1:12] = -1.1656, -0.65655, -0.65042,  ..., 1.1752, 1.2699
```

```
Fn(x)  # returns the percentiles for x
```

```
##  [1] 0.50000000 0.08333333 0.75000000 0.41666667 1.00000000 0.83333333
##  [7] 0.58333333 0.25000000 0.16666667 0.91666667 0.66666667 0.33333333
```

```
tt <- seq(-2, 2, by = 0.1)
12 * Fn(tt) # Fn is a 'simple' function {with values k/12}
```

```
##  [1]  0  0  0  0  0  0  0  0  0  1  1  1  1  1  3  3  3  3  3  6  7  7  7  8  8
## [26]  9  9  9  9 10 10 10 11 12 12 12 12 12 12 12 12
```

```
summary(Fn)
```

```
## Empirical CDF:     12 unique values with summary
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.16556 -0.30789 -0.07721  0.08537  0.53374  1.26989
```

```
##--> see below for graphics
knots(Fn)  # the unique data values {12 of them if there were no ties}
```

```
##  [1] -1.16555703 -0.65654755 -0.65042344 -0.19371486 -0.13732884 -0.13231493
##  [7] -0.02210469  0.24075921  0.41917073  0.87744878  1.17516905  1.26988685
```

```
y <- round(rnorm(12), 1); y[3] <- y[1]
Fn12 <- ecdf(y)
Fn12
```

```
## Empirical CDF
## Call: ecdf(y)
##  x[1:10] =   -0.8,   -0.3,   -0.1,  ...,    1.1,    1.3
```

```
knots(Fn12) # unique values (always less than 12!)
```

```
##  [1] -0.8 -0.3 -0.1  0.1  0.3  0.6  0.7  1.0  1.1  1.3
```

```
summary(Fn12)
```

```
## Empirical CDF:     10 unique values with summary
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -0.800  -0.050   0.450   0.390   0.925   1.300
```

```
summary.stepfun(Fn12)
```

```
## Step function with continuity 'f'= 0 ,  10 knots with summary
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -0.800  -0.050   0.450   0.390   0.925   1.300
##
## and  11 plateau levels (y) with summary
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.2083  0.5000  0.4924  0.7500  1.0000
```

3. Create a function `pick()` that takes an index, `i`, as an argument and returns a function with an argument `x` that subsets `x` with `i`.

```
pick <- function(index){
  function(x) x[index]
}

a <- 1:10
b <- LETTERS[1:10]
pick_2s <- pick(c(2,4,8))
pick_2s(a)
```

```
## [1] 2 4 8
```

```
pick_2s(b)
```

```
## [1] "B" "D" "H"
```

4. Create a function that creates functions that compute the i^th central moment of a numeric vector. You can test it by running the following code:

```
moment <- function(index){
  function(x){
    sum((x - mean(x)) ^ index) / length(x)
  }
}
m1 <- moment(1)
m2 <- moment(2)

x <- runif(100)
stopifnot(all.equal(m1(x), 0))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

5. What happens if you don't use a closure? Make predictions, then verify with the code below.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}

new_counter2()
```

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

```r
new_counter2()
```

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

```r
i
```

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

```r
i <- 9
new_counter2()
```

```
## [1] 10
```

```r
i
```

```
## [1] 10
```

Uses the parent environment and changes when global `i` changes in this case. Function factory can help prevent this

```r
i <- 0
new_counter2 <- function() {
  i <- 0
  function(){
    i <<- i + 1
    i
  }
}
i
```

```
## [1] 0
```

```r
new_counter3 <- new_counter2()
i
```

```
## [1] 0
```

```r
new_counter3()
```

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

```r
new_counter3()
```

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

```r
i <- 0
new_counter3()
```

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

6. What happens if you use `<-` instead of `<<-`? Make predictions, then verify with the code below.

```r
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}

i # 0
```

```
## [1] 0
```

```r
new_counter4 <- new_counter3()
i # 0
```

```
## [1] 0
```

```r
new_counter4() # 1
```

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

```r
i # 0
```

```
## [1] 0
```

```r
new_counter4() # 1
```

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

```r
i # 0
```

```
## [1] 0
```

It doesn't update `i` in the global environment so each run `i` doesn't change.

10.3 Graphical factories

10.3.1 Labelling

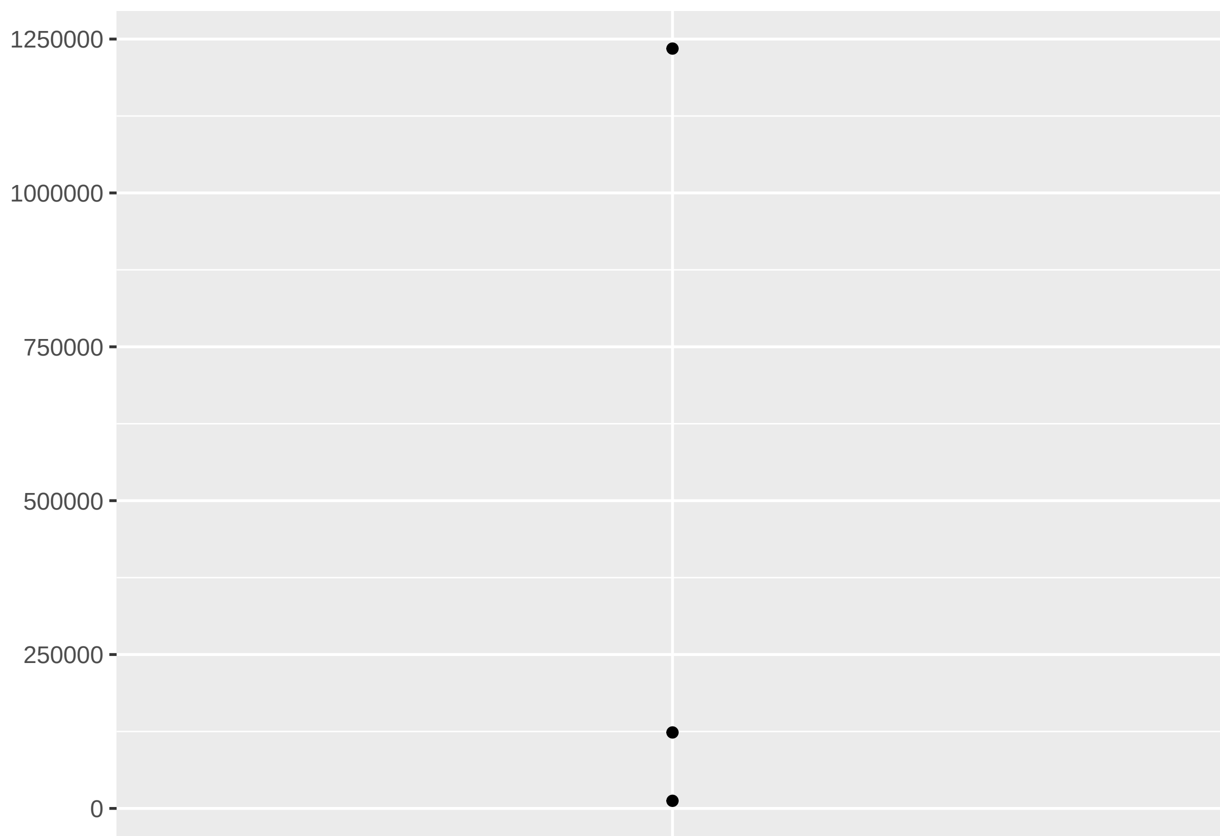
```r
y <- c(12345, 123456, 1234567)
comma_format()(y)
```

```
## [1] "12,345"    "123,456"   "1,234,567"
```

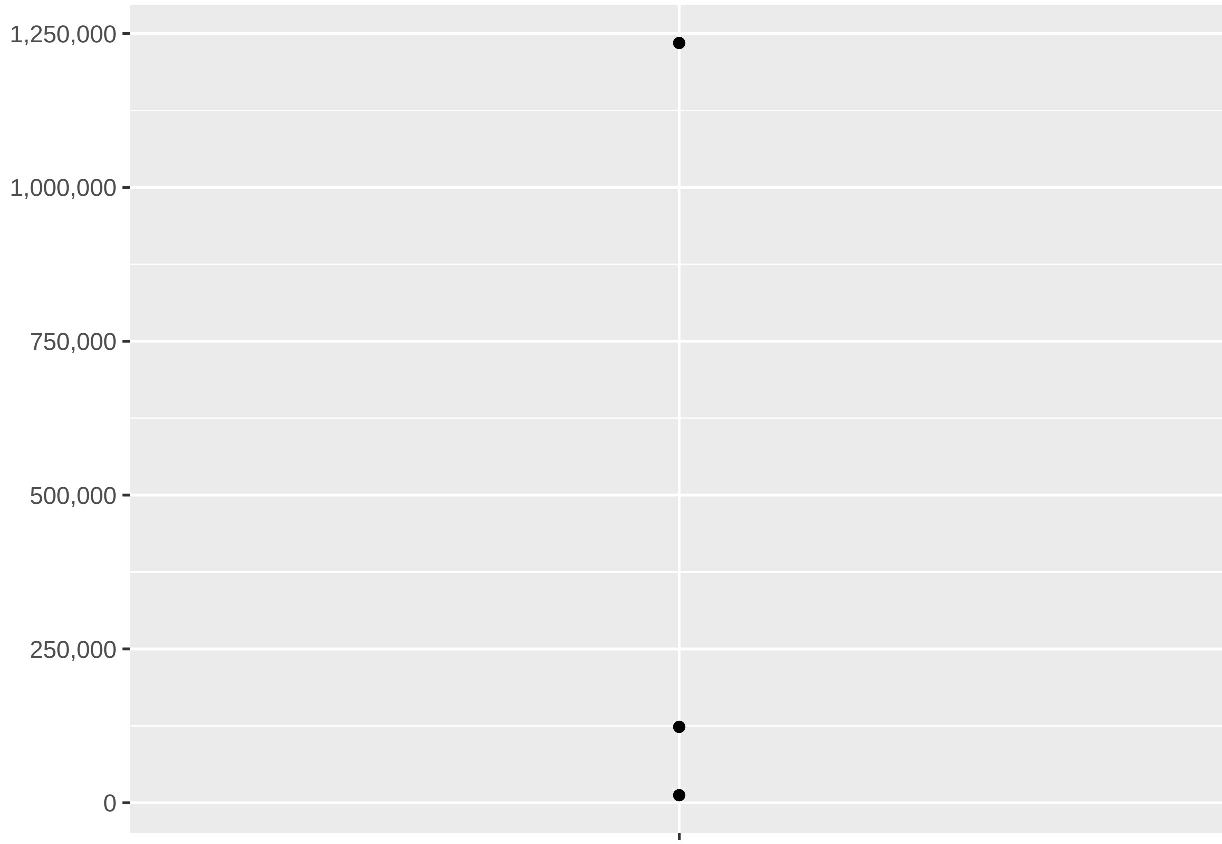
```r
number_format(scale = 1e-3, suffix = " K")(y)
```

```
## [1] "12 K"    "123 K"   "1 235 K"
```

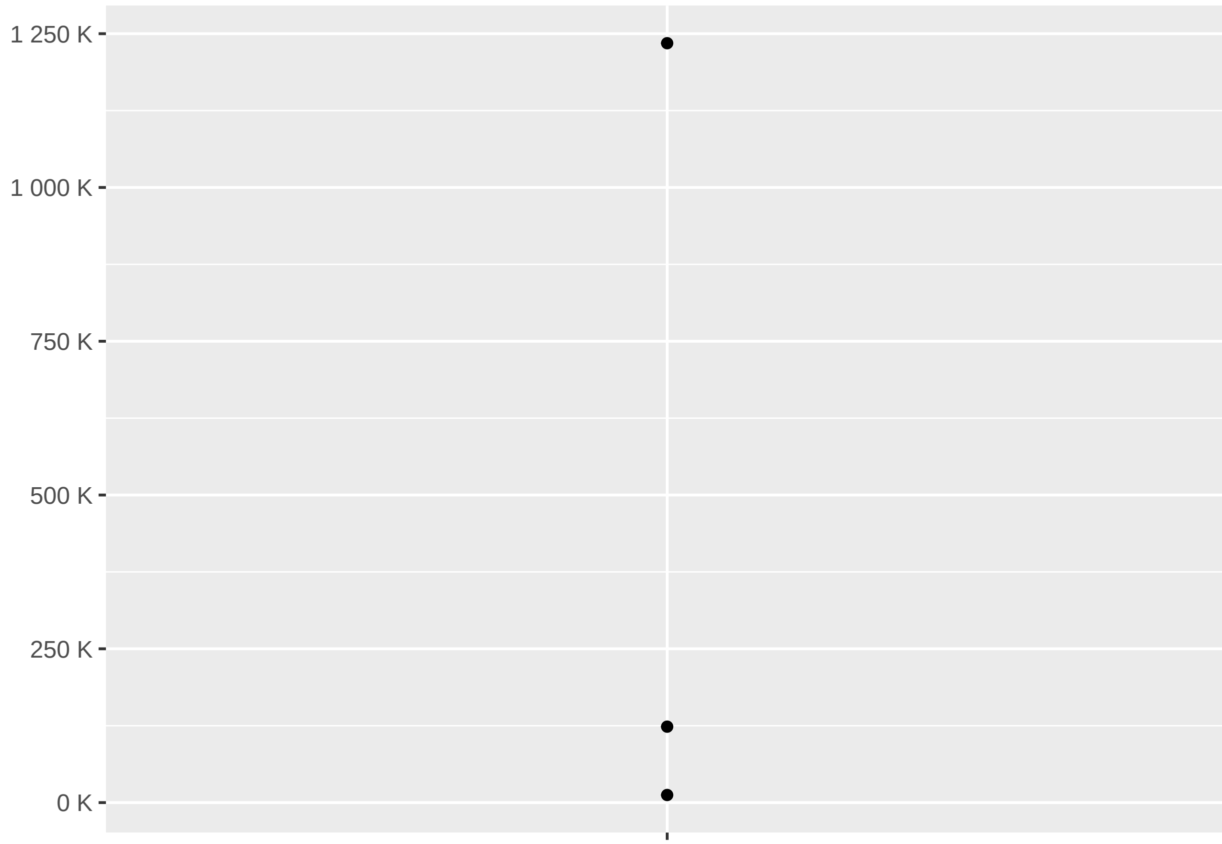
```r
df <- data.frame(x = 1, y = y)
core <- ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_continuous(breaks = 1, labels = NULL) +
  labs(x = NULL, y = NULL)

core
```



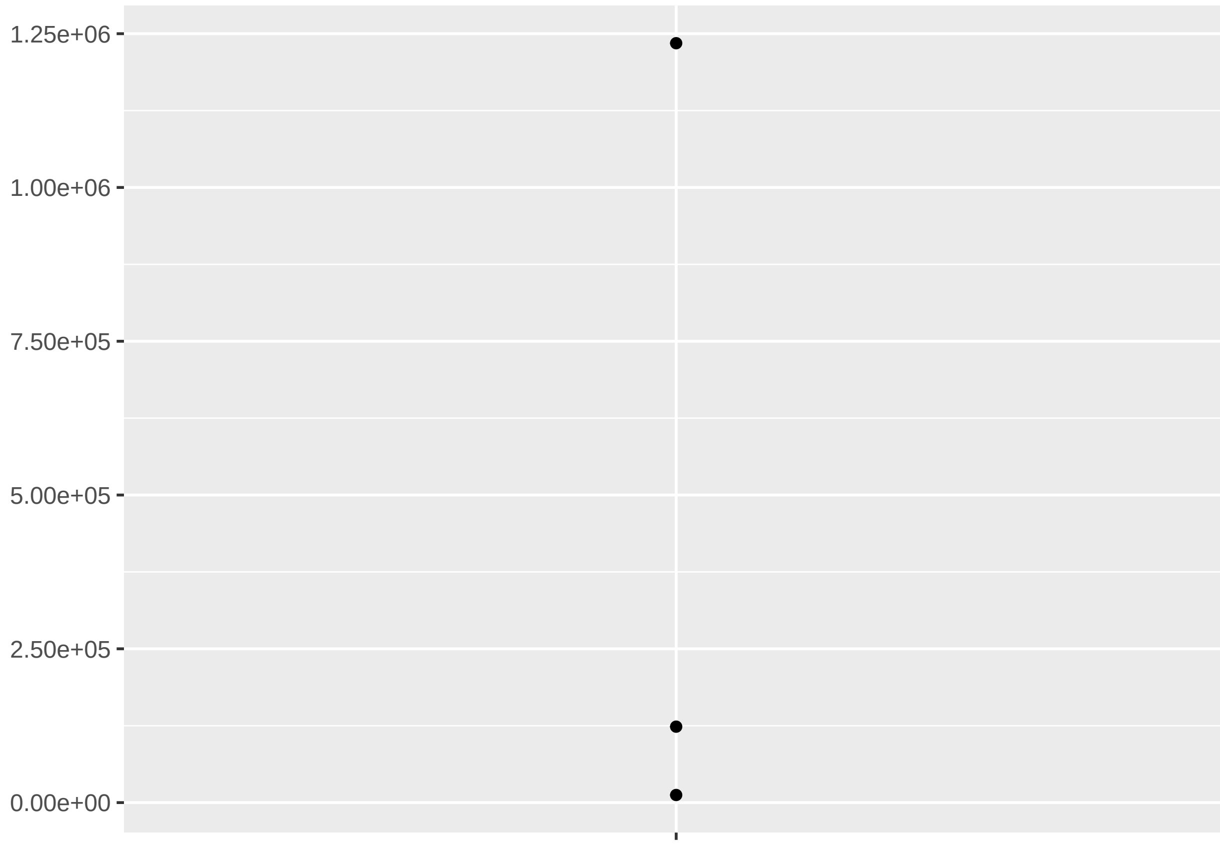

```r
core + scale_y_continuous(
  labels = comma_format()
)
```




```
core + scale_y_continuous(
  labels = number_format(scale = 1e-3, suffix = " K")
)
```

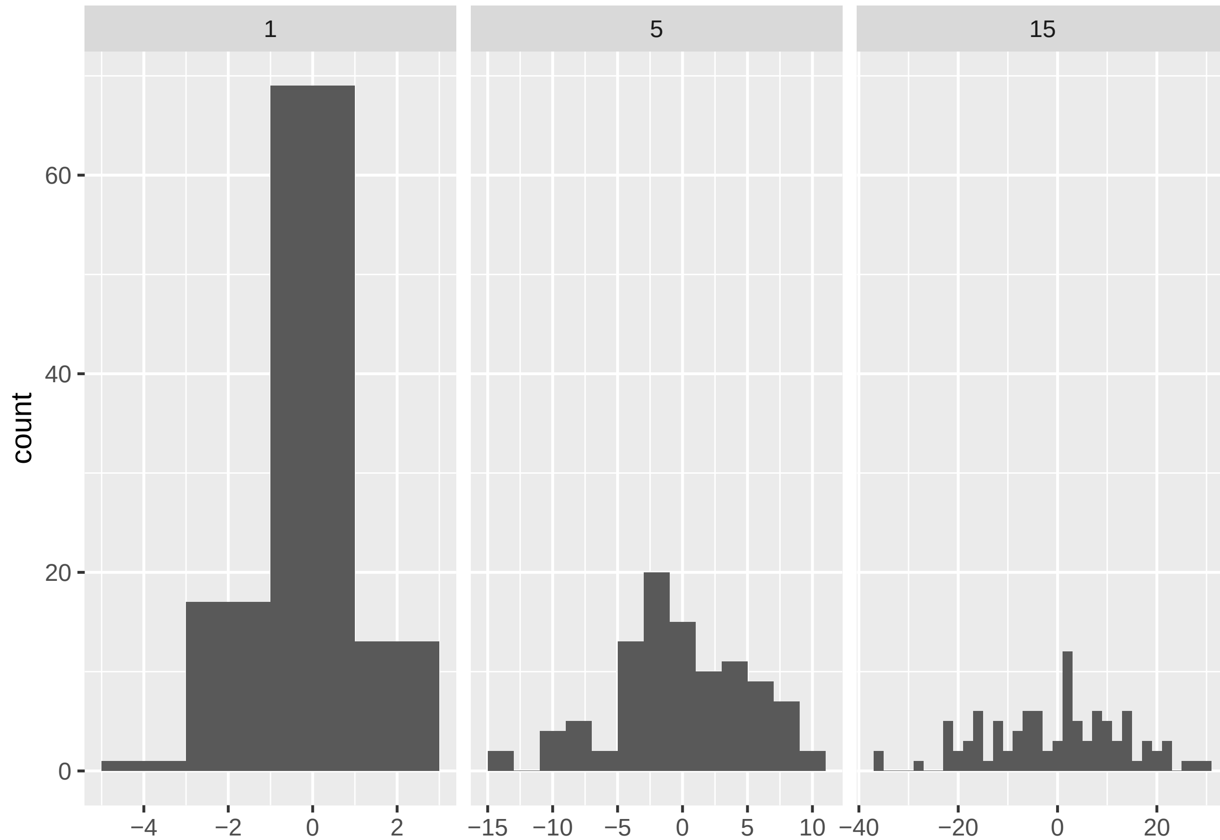



```
core + scale_y_continuous(
  labels = scientific_format()
)
```

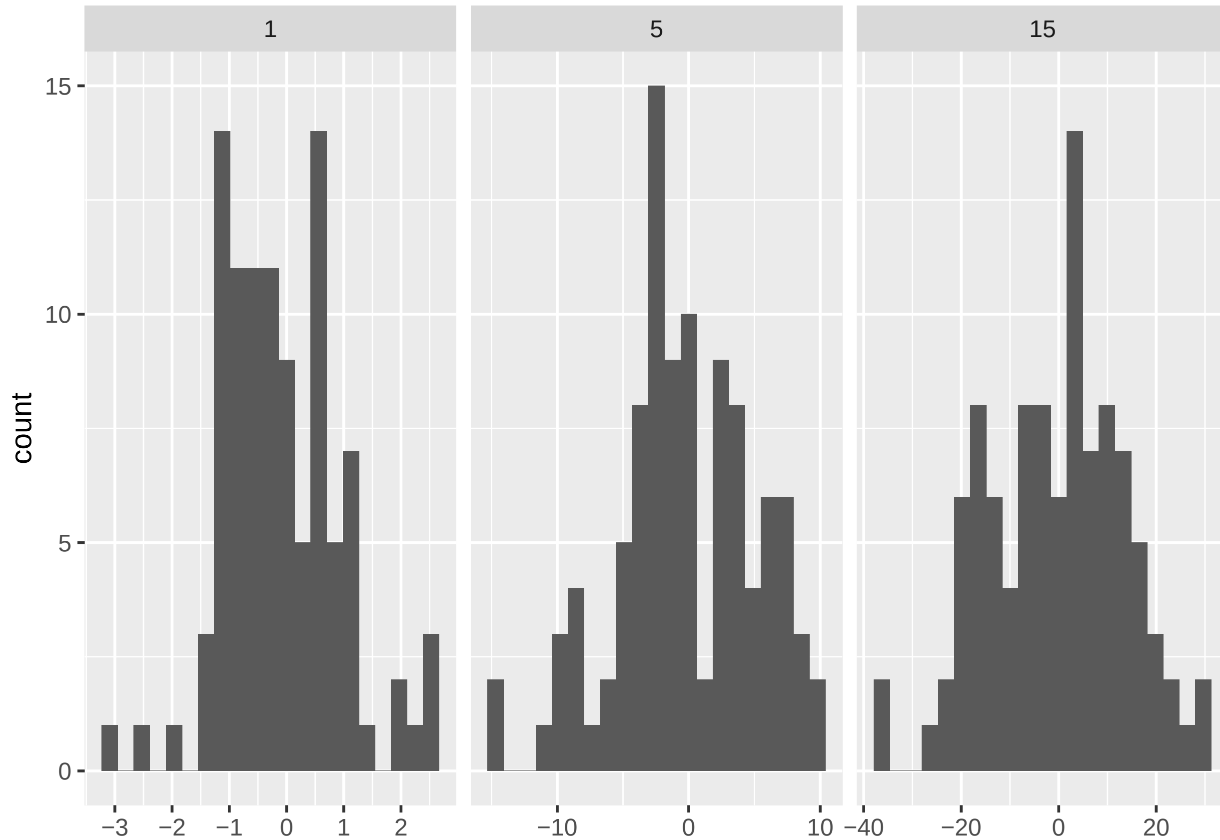


10.3.2 Histogram bins

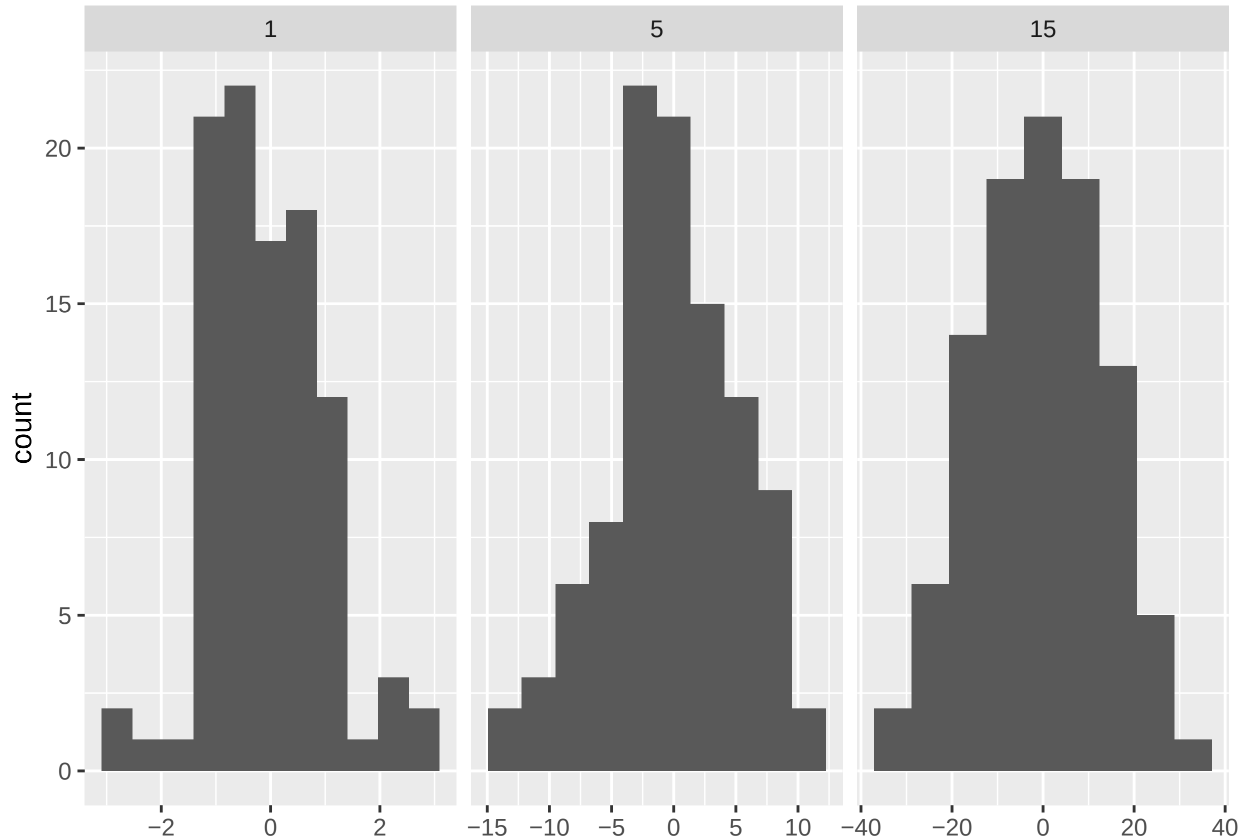
```
# construct some sample data with very different numbers in each cell
sd <- c(1, 5, 15)
n <- 100

df <- data.frame(x = rnorm(3 * n, sd = sd), sd = rep(sd, n))

ggplot(df, aes(x)) +
  geom_histogram(binwidth = 2) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```




```
binwidth_bins <- function(n) {
  force(n)

  function(x) {
    (max(x) - min(x)) / n
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = binwidth_bins(20)) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```




```r
base_bins <- function(type) {
  fun <- switch(type,
    Sturges = nclass.Sturges,
    scott = nclass.scott,
    FD = nclass.FD,
    stop("Unknown type", call. = FALSE)
  )

  function(x) {
    (max(x) - min(x)) / fun(x)
  }
}

ggplot(df, aes(x)) +
  geom_histogram(binwidth = base_bins("FD")) +
  facet_wrap(~ sd, scales = "free_x") +
  labs(x = NULL)
```



10.3.3 ggsave()

```
plot_dev <- function(ext, dpi = 96) {
  force(dpi)

  switch(ext,
    eps =   ,
    ps  =  function(path, ...) {
      grDevices::postscript(
        file = filename, ..., onefile = FALSE,
        horizontal = FALSE, paper = "special"
      )
    },
    pdf = function(filename, ...) grDevices::pdf(file = filename, ...),
    svg = function(filename, ...) svglite::svglite(file = filename, ...),
    emf = ,
    wmf = function(...) grDevices::win.metafile(...),
    png = function(...) grDevices::png(..., res = dpi, units = "in"),
    jpg = ,
    jpeg = function(...) grDevices::jpeg(..., res = dpi, units = "in"),
    bmp = function(...) grDevices::bmp(..., res = dpi, units = "in"),
    tiff = function(...) grDevices::tiff(..., res = dpi, units = "in"),
    stop("Unknown graphics extension: ", ext, call. = FALSE)
  )
```

```
}

plot_dev("pdf")
```

```
## function(filename, ...) grDevices::pdf(file = filename, ...)
## <bytecode: 0x000001f709a18bd0>
## <environment: 0x000001f709699040>
```

```
plot_dev("png")
```

```
## function(...) grDevices::png(..., res = dpi, units = "in")
## <bytecode: 0x000001f709c978b8>
## <environment: 0x000001f70a1d5db8>
```

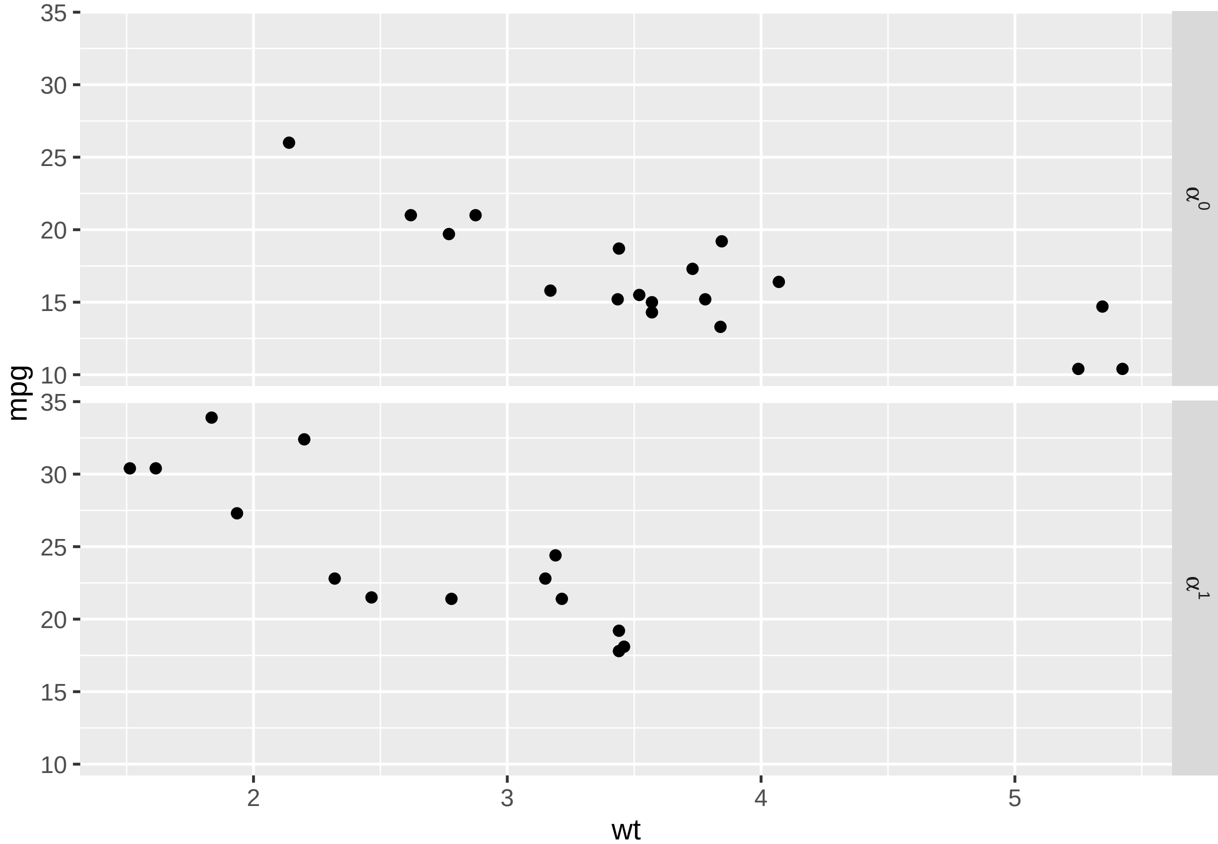
10.3.4 Exercises

1. Compare and contrast `ggplot2::label_bquote()` with `scales::number_format()`

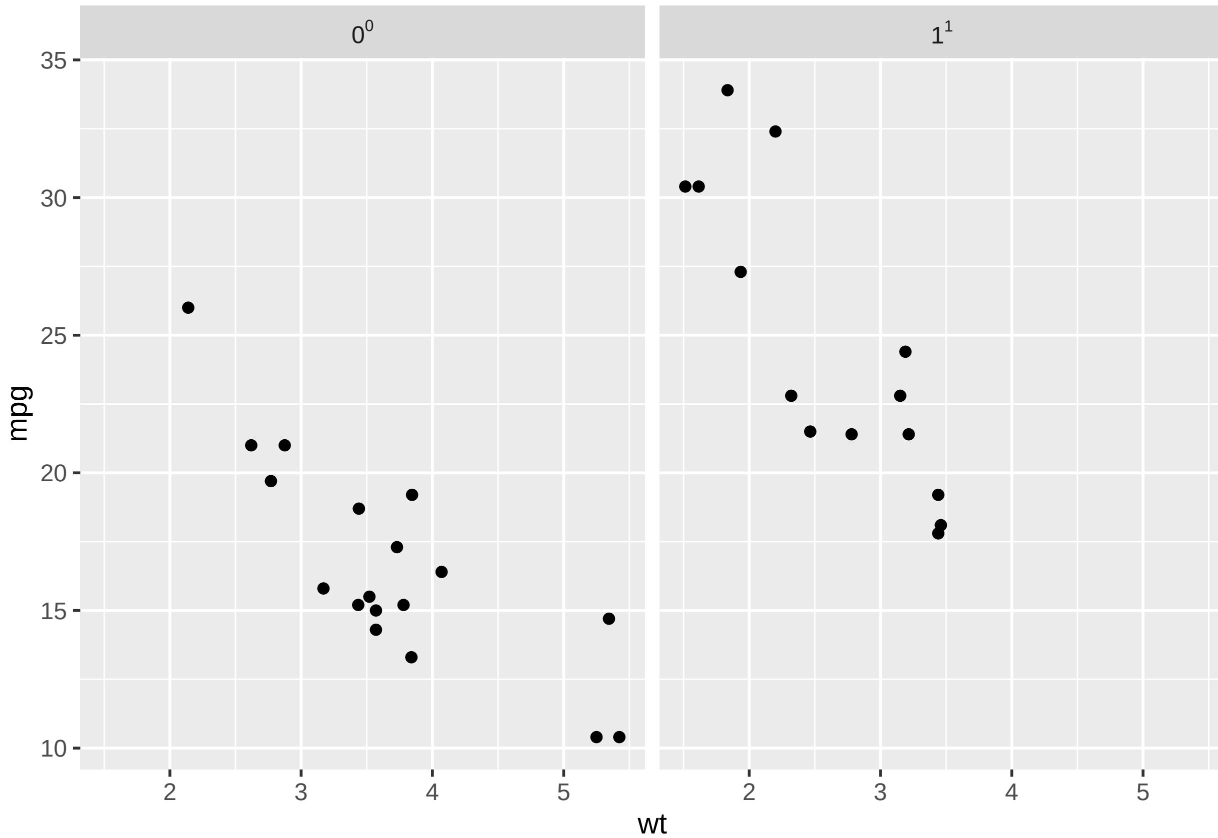
```
y <- c(12345, 123456, 1234567)

scales::number_format(scale = 1e-3, suffix = " K")(y)
```

```
## [1] "12 K"    "123 K"   "1 235 K"
```

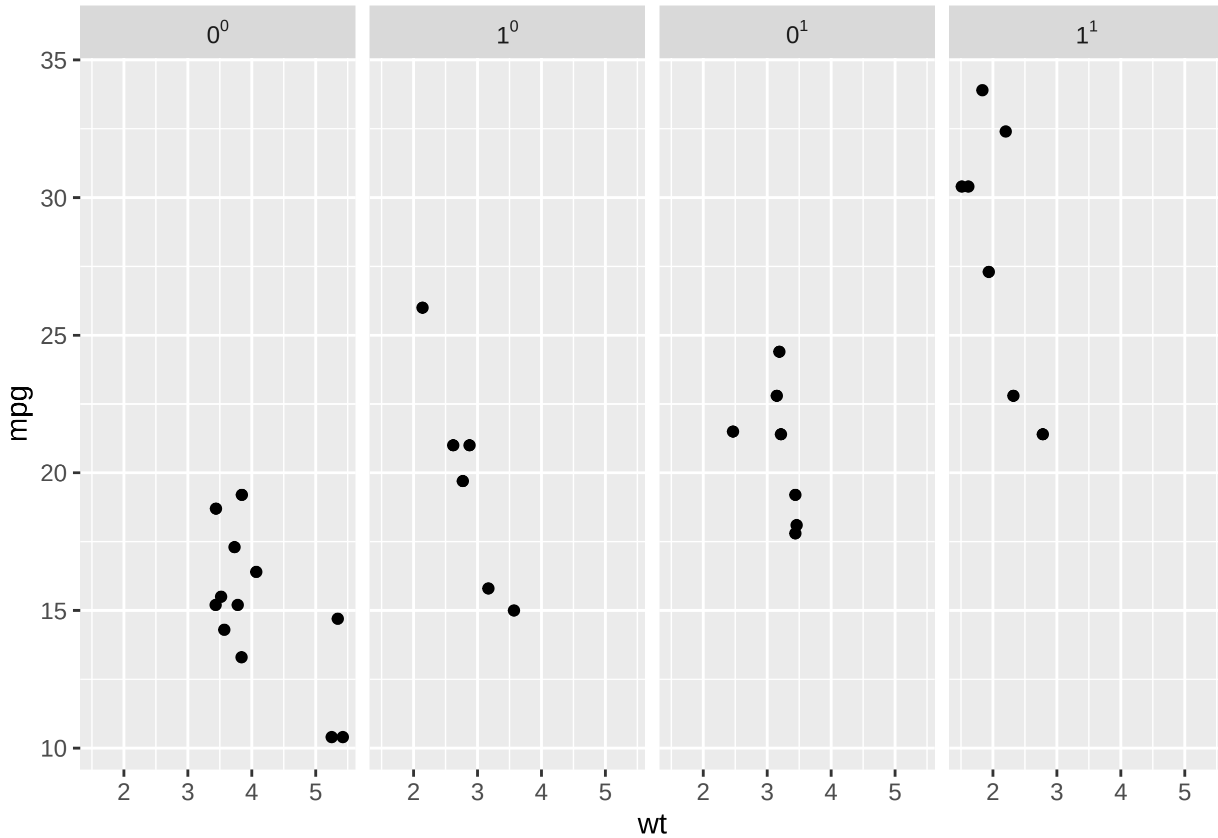
```
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
p + facet_grid(vs ~ ., labeller = label_bquote(alpha ^ .(vs)))
```




```
p + facet_grid(. ~ vs, labeller = label_bquote(cols = .(vs) ^ .(vs)))
```




```
p + facet_grid(. ~ vs + am, labeller = label_bquote(cols = .(am) ^ .(vs)))
```



One lets you use math notations in your labeller facet names, while the other let's you transform your number values

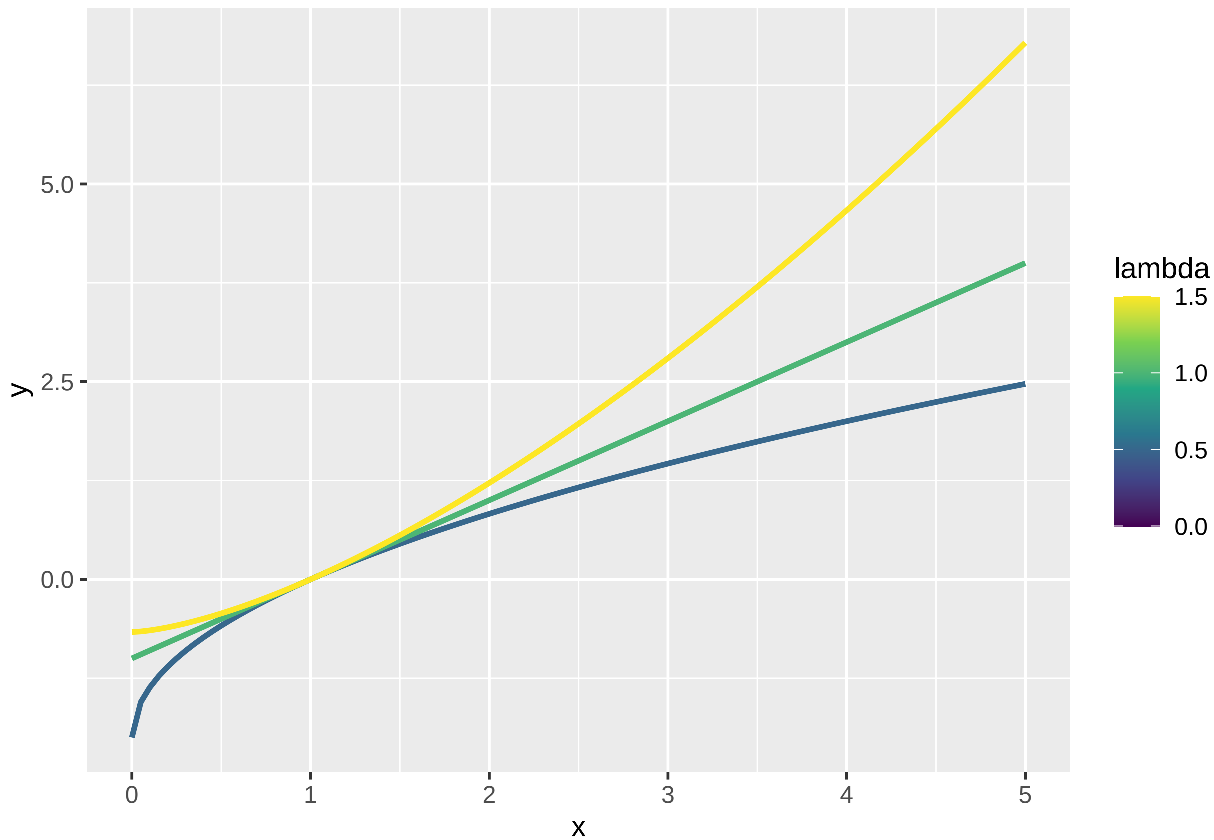
10.4 Statistical factories

10.4.1 Box-Cox transformation

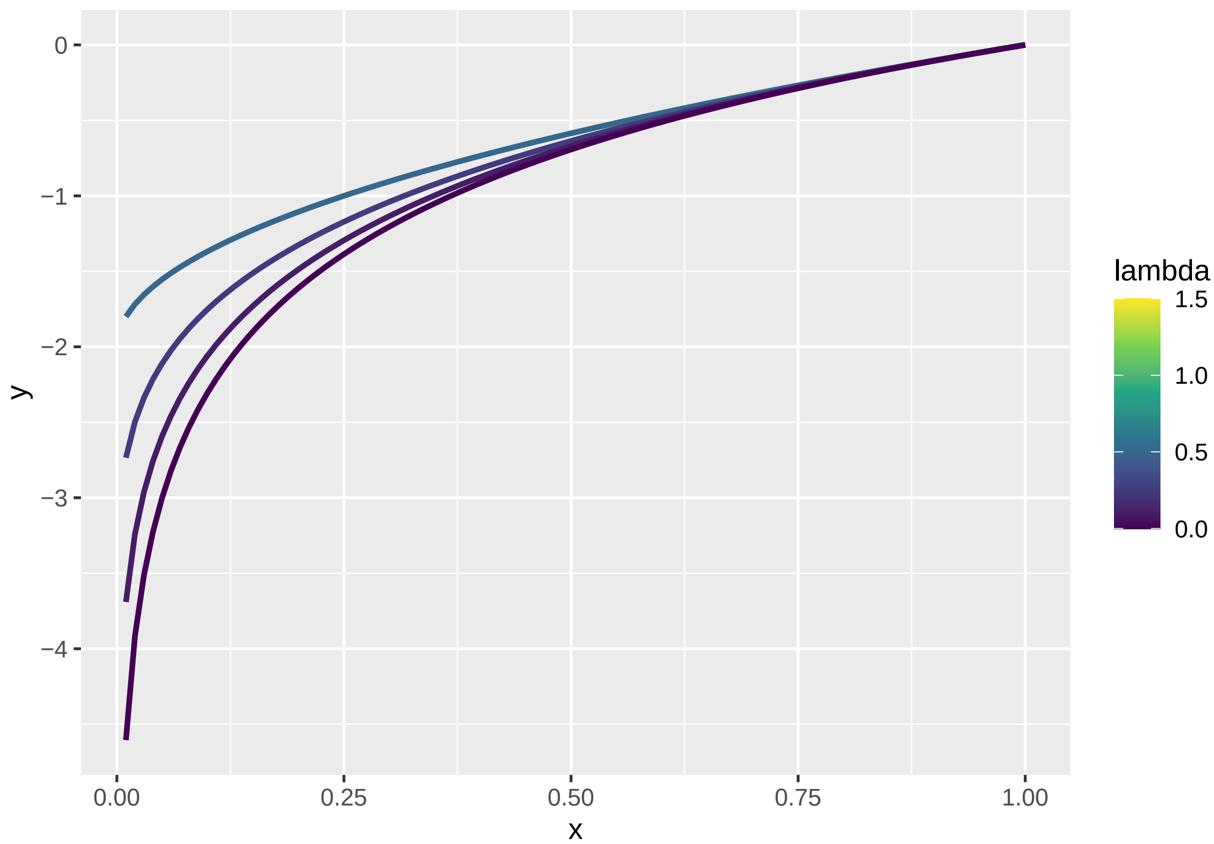
```
boxcox1 <- function(x, lambda) {
  stopifnot(length(lambda) == 1)

  if (lambda == 0) {
    log(x)
  } else {
    (x ^ lambda - 1) / lambda
  }
}
```

```
boxcox2 <- function(lambda) {
  if (lambda == 0) {
    function(x) log(x)
  } else {
    function(x) (x ^ lambda - 1) / lambda
  }
}
```

```
stat_boxcox <- function(lambda) {
  stat_function(aes(colour = lambda), fun = boxcox2(lambda), linewidth = 1)
}

ggplot(data.frame(x = c(0, 5)), aes(x)) +
  lapply(c(0.5, 1, 1.5), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))
```




```
ggplot(data.frame(x = c(0.01, 1)), aes(x)) +
  lapply(c(0.5, 0.25, 0.1, 0), stat_boxcox) +
  scale_colour_viridis_c(limits = c(0, 1.5))
```



10.4.2 Bootstrap generators

```
boot_permute <- function(df, var) {
  n <- nrow(df)
  force(var)

  function() {
    col <- df[[var]]
    col[sample(n, replace = TRUE)]
  }
}

boot_mtcars1 <- boot_permute(mtcars, "mpg")
head(boot_mtcars1())
```

```
## [1] 14.7 30.4 21.0 15.0 30.4 33.9
```

```
head(boot_mtcars1())
```

```
## [1] 15.2 22.8 15.8 17.8 10.4 18.7
```

```r
boot_model <- function(df, formula) {
  mod <- lm(formula, data = df)
  fitted <- unname(fitted(mod))
  resid <- unname(resid(mod))
  rm(mod)

  function() {
    fitted + sample(resid)
  }
}

boot_mtcars2 <- boot_model(mtcars, mpg ~ wt)
head(boot_mtcars2())
```

```
## [1] 20.07725 22.21963 31.30793 22.45261 18.70000 19.14968
```

```r
head(boot_mtcars2())
```

```
## [1] 20.31002 18.45742 20.98059 21.30371 17.87265 16.70730
```

10.4.3 Maximum likelihood estimation

```r
lprob_poisson <- function(lambda, x) {
  n <- length(x)
  (log(lambda) * sum(x)) - (n * lambda) - sum(lfactorial(x))
}
```

```r
x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
```

```r
lprob_poisson(10, x1)
```

```
## [1] -183.6405
```

```r
#> [1] -184
lprob_poisson(20, x1)
```

```
## [1] -61.14028
```

```r
#> [1] -61.1
lprob_poisson(30, x1)
```

```
## [1] -30.98598
```

```r
#> [1] -31
```

```r
ll_poisson1 <- function(x) {
  n <- length(x)

  function(lambda) {
    log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
  }
}
```

```r
ll_poisson2 <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  c <- sum(lfactorial(x))

  function(lambda) {
    log(lambda) * sum_x - n * lambda - c
  }
}
```

```r
ll1 <- ll_poisson2(x1)

ll1(10)
```

```
## [1] -183.6405
```

```r
ll1(20)
```

```
## [1] -61.14028
```

```r
ll1(30)
```

```
## [1] -30.98598
```

```r
optimise(ll1, c(0, 100), maximum = TRUE)
```

```
## $maximum
## [1] 32.09999
##
## $objective
## [1] -30.26755
```

```r
optimise(lprob_poisson, c(0, 100), x = x1, maximum = TRUE)
```

```
## $maximum
## [1] 32.09999
##
## $objective
## [1] -30.26755
```

10.4.4 Exercises

1. In `boot_model()`, why don't I need to force the evaluation of `df` or `model`?

```
boot_model <- function(df, formula) {
  mod <- lm(formula, data = df)
  fitted <- unname(fitted(mod))
  resid <- unname(resid(mod))
  rm(mod)

  function() {
    fitted + sample(resid)
  }
}
```

The call to `fitted` and `resid` implicitly forces the evaluation of `df` and `mod` when the factory is called and luckily these values only need to be calculated once.

2. Why might you formulate the Box-Cox transformation like this?

```
boxcox3 <- function(x) {
  function(lambda) {
    if (lambda == 0) {
      log(x)
    } else {
      (x ^ lambda - 1) / lambda
    }
  }
}
```

In this case `x` is fixed when first created, but can change later on by changing `x`. You can then manipulate `lambda` and also change `x` is need be. ie change your dataset but keep the overall function the same.

3. Why don't you need to worry that `boot_permute()` stores a copy of the data inside the function that it generates?

```
boot_permute <- function(df, var) {
  n <- nrow(df)
  force(var)

  function() {
    col <- df[[var]]
    col[sample(n, replace = TRUE)]
  }
}

boot_mtcars1 <- boot_permute(mtcars, "mpg")
head(boot_mtcars1())
```

```
## [1] 15.8 19.7 21.4 19.7 15.2 18.7
```

It doesn't actually store a copy. It just describes an object already in memory. A sampling of `mtcar` where no new values are created.

4. How much time does `ll_poisson2()` save compared to `ll_poisson1()`? Use `bench::mark()` to see how much faster the optimisation occurs. How does changing the length of `x` change the results?

```
ll_poisson1 <- function(x) {
  n <- length(x)

  function(lambda) {
    log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
  }
}
```

```
ll_poisson1 <- function(x) {
  n <- length(x)

  function(lambda) {
    log(lambda) * sum(x) - n * lambda - sum(lfactorial(x))
  }
}
ll_poisson2 <- function(x) {
  n <- length(x)
  sum_x <- sum(x)
  c <- sum(lfactorial(x))

  function(lambda) {
    log(lambda) * sum_x - n * lambda - c
  }
}
x1 <- c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38)
x2 <- sample(1:1000, 100)
x3 <- sample(1:1e6, 10000)

things <- expand_grid(functional = c("ll_poisson1", "ll_poisson2"),
                      x_values = c("x1","x2","x3"))

things <- pmap(things,
     function(functional, x_values) {
       fun <- get(functional)
       x <- get(x_values)
       fun(x)
     })
names(things) <- c("ll11", "ll21",
                   "ll31", "ll12",
                   "ll22", "ll32")
map_df(things, ~bench::mark(optimise(.x, c(0,100), maximum = TRUE))) %>%
  mutate(input = names(things)) %>%
  select(input, everything())
```

```
## # A tibble: 6 x 7
##   input expression                                   min  median itr/s~1 mem_a~2
##   <chr> <bch:expr>                              <bch:tm> <bch:t>   <dbl> <bch:b>
## 1 ll11  optimise(.x, c(0, 100), maximum = TRUE)   35.6us  50.5us 16472.       0B
## 2 ll21  optimise(.x, c(0, 100), maximum = TRUE)  228.8us 254.8us  3429.  25.67KB
## 3 ll31  optimise(.x, c(0, 100), maximum = TRUE)   19.1ms  20.6ms    49.1  2.37MB
```

```
## 4 l112  optimise(.x, c(0, 100), maximum = TRUE)   19.7us  22.5us 37707.       0B
## 5 l122  optimise(.x, c(0, 100), maximum = TRUE)   25.2us    29us 29667.       0B
## 6 l132  optimise(.x, c(0, 100), maximum = TRUE)   25.2us    29us 28804.       0B
## # ... with 1 more variable: `gc/sec` <dbl>, and abbreviated variable names
## #   1: `itr/sec`, 2: mem_alloc
```

10.5 Function factories + functionals

```
names <- list(
  square = 2,
  cube = 3,
  root = 1/2,
  cuberoot = 1/3,
  reciprocal = -1
)
funs <- purrr::map(names, power1)

funs$root(64)
```

```
## [1] 8
```

```
funs$root
```

```
## function(x) {
##     x ^ exp
##   }
## <bytecode: 0x000001f708a34718>
## <environment: 0x000001f708bb3b00>
```

```
with(funs, root(100))
```

```
## [1] 10
```

```
attach(funs)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     cube, square
```

```
root(100)
```

```
## [1] 10
```

```
detach(funs)
```

```
rlang::env_bind(globalenv(), !!!funs)
root(100)
```

```
## [1] 10
```

```r
rlang::env_unbind(globalenv(), names(funs))
```

10.5.1 Exercises

1. Which of the following commands is equivalent to `with(x, f(z))`?

E. It depends

2. Compare and contrast the effects of `env_bind()` vs. `attach()` for the following code.

```r
funs <- list(
  mean = function(x) mean(x, na.rm = TRUE),
  sum = function(x) sum(x, na.rm = TRUE)
)

attach(funs)
```

```
## The following objects are masked from package:base:
##
##     mean, sum
```

```r
#> The following objects are masked from package:base:
#>
#>     mean, sum
mean <- function(x) stop("Hi!")
detach(funs)

mean(1:5)
```

```
## Error in mean(1:5): Hi!
```

```r
sum(1:5)
```

```
## [1] 15
```

```r
rlang::env_bind(globalenv(), !!!funs)
mean(1:5)
```

```
## Error in mean(x, na.rm = TRUE): unused argument (na.rm = TRUE)
```

```r
sum(1:5)
```

```
## Error in sum(x, na.rm = TRUE): unused argument (na.rm = TRUE)
```

```r
mean <- function(x) stop("Hi!")
mean(1:5)
```

```
## Error in mean(1:5): Hi!
```

```r
rlang::env_unbind(globalenv(), names(funs))
mean(1:5)
```

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

```r
sum(1:5)
```

```
## [1] 15
```

with `attach` the base `mean` and `sum` functions are masked because you are adding `funs` to the search path. Detaching `funs` removes the masking and `funs` from the search path. However if you change `mean` you change the base `mean` function as well. With `env_bind` you can change the functions in the global environment, but then when you unbind them they revert back to their original format.