

# Intro to iteration with Base R

---

Daniel Anderson

Week 2, Class 1

# Agenda

---

- For loops
- Apply family of loops
  - `lapply()`
  - `sapply()`
  - `vapply()`

Note – we won't get to `apply` or `tapply`, but the former in particular is probably worth investigating.

# Learning objectives

---

- Understand the basics of what it means to loop through a vector
- Begin to recognize use cases
- Be able to apply basic **for** loops and write their equivalents with **lapply**.

# Basic overview: **for** loops

---

```
39 for(i in 1:5){  
40   print(a[i])  
41 }
```

Index Sequence Loop

```
a <- letters[1:26]  
a
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"  
## [26] "z"
```

```
for(i in 1:5){  
  print(a[i])  
}
```

```
## [1] "a"  
## [1] "b"  
## [1] "c"  
## [1] "d"  
## [1] "e"
```

Note these are five different character scalars (atomic vectors of length one). It is NOT a single vector.

# Another basic example

---

Simulate tossing a coin, record results

- For a single toss

```
sample(c("Heads", "Tails"), 1)
```

```
## [1] "Tails"
```

- For multiple tosses, first allocate a vector with **length** equal to the number of iterations

```
result <- rep(NA, 10)  
result
```

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

- Next, run the trial  $n$  times, storing the result in your pre-allocated vector.

```
for(i in seq_along(result)) {  
  result[i] <- sample(c("Heads", "Tails"), 1)  
}  
result
```

```
## [1] "Tails" "Heads" "Tails" "Tails" "Heads" "Heads" "Heads" "Heads" "He
```

# Growing vectors

---

- **Always** pre-allocate a vector for storage before running a `for` loop.
- Contrary to some opinions you may see out there, `for` loops are not actually slower than `lapply`, etc., provided the `for` loop is written well
- This primarily means not growing a vector

# Example

---

100,000 coin flips by growing a vector

```
library(tictoc)

set.seed(1)
tic()
not_allocated <- sample(c("Heads", "Tails"), 1)
for(i in seq_len(1e5 - 1)) {
  not_allocated <- c(
    not_allocated,
    sample(c("Heads", "Tails"), 1)
  )
}
toc()
```

## 37.313 sec elapsed



same *exact* thing with pre-allocated vector

```
set.seed(1)
tic()
allocated <- rep(NA, 1e5)
for(i in seq_len(1e5)) {
  allocated[i] <- sample(c("Heads", "Tails"), 1)
}
toc()
```

```
## 0.622 sec elapsed
```

# Result

---

- The result is the same, regardless of the approach (notice I forced the random number generator to start at the same place in both samples)

```
identical(not_allocated, allocated)
```

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

- Speed is obviously not identical

# You try

---

Base R comes with `letters` and `LETTERS`

- Make an alphabet of upper/lower case. For example, create "Aa" with `paste0(LETTERS[1], letters[1])`
- Write a `for` loop for all letters

03:00

# Answer

---

```
alphabet <- rep(NA, length(letters))  
  
for(i in seq_along(alphabet)) {  
  alphabet[i] <- paste0(LETTERS[i], letters[i])  
}  
alphabet
```

```
## [1] "Aa" "Bb" "Cc" "Dd" "Ee" "Ff" "Gg" "Hh" "Ii" "Jj" "Kk" "Ll" "Mm" "Nn"  
## [21] "Uu" "Vv" "Ww" "Xx" "Yy" "Zz"
```

# Quick style note

---

- Why am I always using `seq_along`?
- When writing functions, it's safer to use `seq_*` because you can't always be guaranteed of the input

```
x <- data.frame()  
1:length(x)
```

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

```
seq_along(x)
```

```
## integer(0)
```

# Running the loop

---

```
for(i in 1:length(x)) {  
  print(letters[i])  
}
```

```
## [1] "a"  
## character(0)
```

```
for(i in seq_along(x)) {  
  print(letters[i])  
}
```

- The first may return unhelpful error messages or unexpected output, while the latter simply won't run, which is generally easier to diagnose.
- Even better, if you're using a loop in a function, you should probably have a condition that checks the input before running it

# Another example

---

- Say we wanted to simulate 100 cases from random normal data, where we varied the standard deviation in increments of 0.2, ranging from 1 to 5
- First, specify a vector standard deviations

```
increments <- seq(1, 5, by = 0.2)
```

- Next, allocate a vector. There are many ways I could store this result (data frame, matrix, list). I'll do it in a list.

```
simulated <- vector("list", length(increments))  
str(simulated)
```

```
## List of 21  
## $ : NULL  
## $ : NULL  
## $ : NULL  
## $ : NULL
```

# Write **for** loop

---

```
for(i in seq_along(simulated)) {  
  simulated[[i]] <- rnorm(100, 0, increments[i])  
  # note use of `[` above  
}  
str(simulated)
```

```
## List of 21  
## $ : num [1:100] -2.387 0.405 -1.599 -0.285 0.288 ...  
## $ : num [1:100] 0.298 0.433 -1.021 1.384 -0.323 ...  
## $ : num [1:100] 0.893 -1.799 -0.819 -1.11 -2.198 ...  
## $ : num [1:100] -0.332 1.067 -0.823 2.899 1.863 ...  
## $ : num [1:100] -2.568 -0.672 -0.244 -1.645 2.221 ...  
## $ : num [1:100] 2.4 -1.95 1.13 3.05 3.56 ...  
## $ : num [1:100] -2.978 0.798 2.212 2.15 -2.197 ...  
## $ : num [1:100] -0.211 -1.768 3.35 2.06 0.213 ...  
## $ : num [1:100] 0.718 -4.029 -1.093 0.417 -3.952 ...  
## $ : num [1:100] 0.632 3.084 -2.62 -1.282 -2.965 ...  
## $ : num [1:100] 2.1759 -0.4681 1.6349 -0.0809 -0.7611 ...  
## $ : num [1:100] 1.236 3.055 -2.575 -0.868 4.369 ...  
## $ : num [1:100] 0.7795 -1.0125 -6.465 0.0926 1.8629 ...  
## $ : num [1:100] 3.466 -1.245 0.496 3.67 -2.207 ...  
## $ : num [1:100] -2.712 -4.21 -3.686 -0.728 -0.142 ...  
## $ : num [1:100] 2.83 6.08 -3 4.29 4.18 ...  
## $ : num [1:100] 0.335 0.574 4.106 4.414 0.897 ...  
## $ : num [1:100] -2.123 3.165 1.104 -4.065 0.578 ...  
## $ : num [1:100] 2.448 -1.472 4.411 2.34 -0.346 ...
```



# List/data frame

---

- Remember, if all the vectors of our list are the same length, it can be transformed into a data frame.
- First, let's provide meaningful names

```
names(simulated) <- paste0("sd_", increments)
sim_d <- data.frame(simulated)
head(sim_d)
```

```
##           sd_1      sd_1.2      sd_1.4      sd_1.6      sd_1.8      sd_2
## 1 -2.3872613  0.2979273  0.8930471 -0.3319310 -2.5676229  2.3954045 -2.9
## 2  0.4051212  0.4329239 -1.7989656  1.0673114 -0.6722755 -1.9542566  0.7
## 3 -1.5992856 -1.0209222 -0.8192303 -0.8232251 -0.2435614  1.1309855  2.2
## 4 -0.2847246  1.3838266 -1.1097486  2.8991565 -1.6445420  3.0545722  2.1
## 5  0.2881735 -0.3233308 -2.1982580  1.8633398  2.2213515  3.5578607 -2.1
## 6  0.1175257 -0.4150724  0.6353818  0.7142314 -0.5219616 -0.8029945  0.1
##           sd_2.8      sd_3      sd_3.2      sd_3.4      sd_3.6      sd_3.8
## 1  0.6321448  2.17589282  1.2362364  0.77947966  3.4658329 -2.7119876  2
## 2  3.0843027 -0.46812673  3.0554725 -1.01245886 -1.2451870 -4.2104097  6
## 3 -2.6196216  1.63492841 -2.5751022 -6.46499466  0.4960868 -3.6861183 -3
## 4 -1.2824915 -0.08085208 -0.8678742  0.09259855  3.6701383 -0.7277885  4
## 5 -2.9646399 -0.76111234  4.3687915  1.86290325 -2.2067855 -0.1416683  4
## 6  4.5420524  3.53122922 -2.4194781 -1.14660593 -1.7557261 -2.2110984  4
```

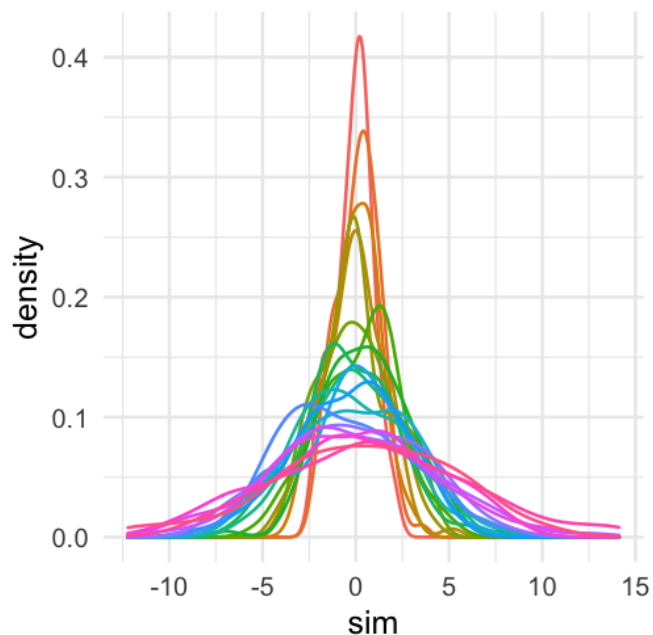
# tidyverse

---

- One of the *best* things about the tidyverse is that it often does the looping for you

```
library(tidyverse)
pd <- sim_d %>%
  pivot_longer(
    everything(),
    names_to = "sd",
    values_to = "sim",
    names_prefix = "sd_",
    names_ptypes = list(
      sd = factor()
    )
  )

ggplot(pd, aes(sim)) +
  geom_density(
    aes(color = sd)
  ) +
  guides(color = "none")
```





# Base R Method

---

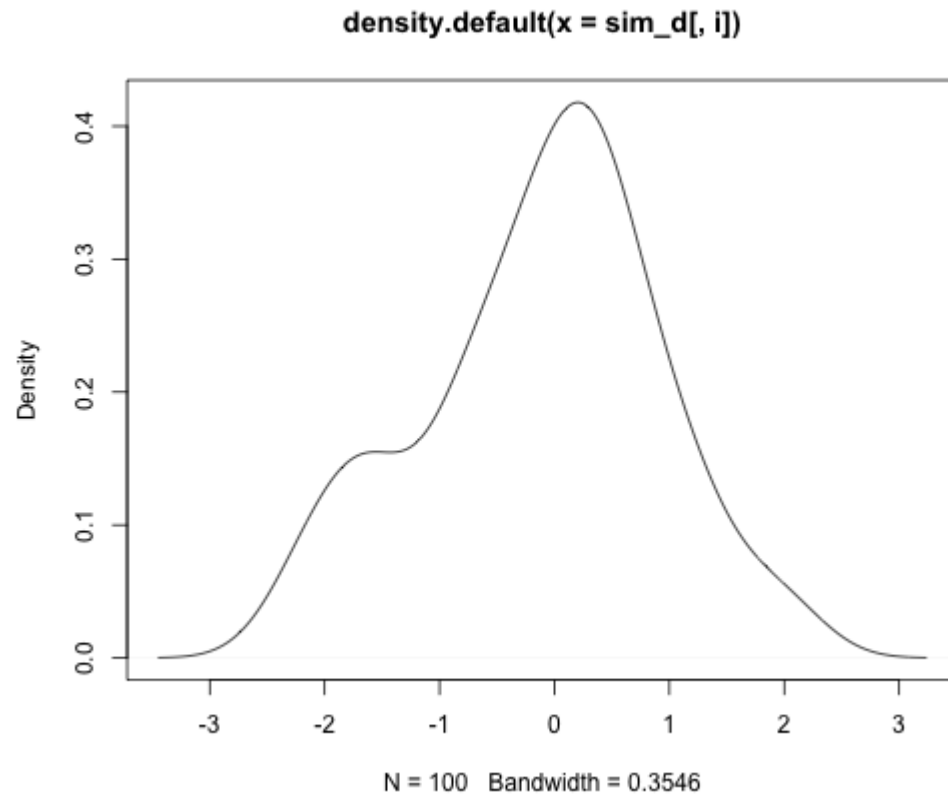
- Calculate all the densities

```
densities <- vector("list", length(sim_d))
for(i in seq_along(densities)) {
  densities[[i]] <- density(sim_d[,i])
}
str(densities)
```

```
## List of 21
## $ :List of 7
## ..$ x      : num [1:512] -3.45 -3.44 -3.42 -3.41 -3.4 ...
## ..$ y      : num [1:512] 0.000173 0.000195 0.000219 0.000245 0.00027
## ..$ bw     : num 0.355
## ..$ n      : int 100
## ..$ call   : language density.default(x = sim_d[, i])
## ..$ data.name: chr "sim_d[, i]"
## ..$ has.na  : logi FALSE
## ..- attr(*, "class")= chr "density"
## $ :List of 7
## ..$ x      : num [1:512] -3.39 -3.38 -3.36 -3.35 -3.33 ...
## ..$ y      : num [1:512] 0.000261 0.000296 0.000334 0.000377 0.00042
## ..$ bw     : num 0.405
## ..$ n      : int 100
## ..$ call   : language density.default(x = sim_d[, i])
```

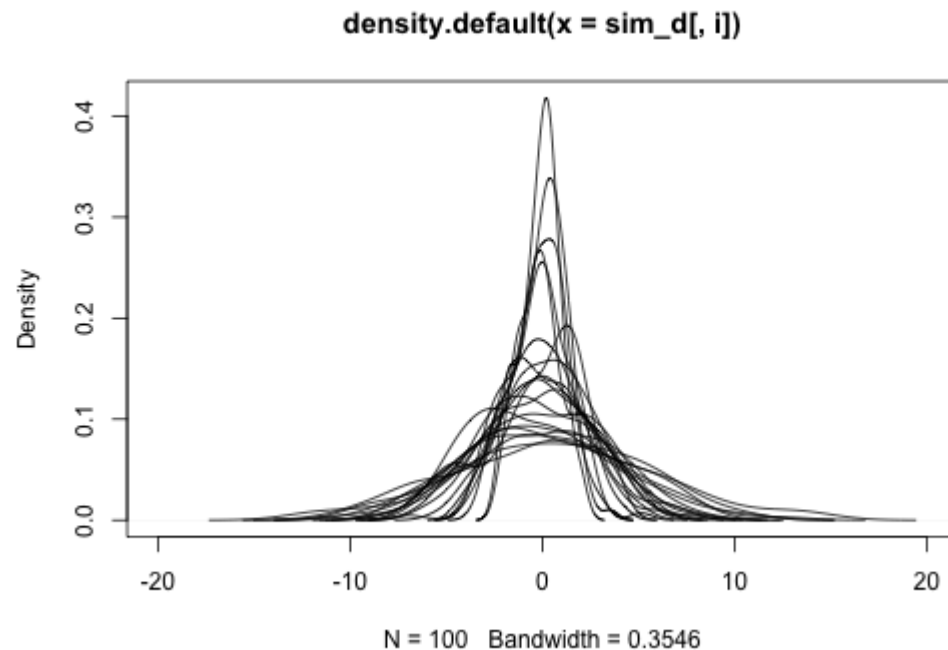
- Next, plot the first density

```
plot(densities[[1]])
```



- Finally, loop through all the other densities

```
plot(densities[[1]], xlim = c(-20, 20))  
  
for(i in seq(2, length(densities))) {  
  lines(x = densities[[i]]$x,  
        y = densities[[i]]$y)  
}
```



# Skipping iterations

---

- On the prior slide, I set the index to skip over the first by using `seq(2, length(densities))`
- Alternatively, the loop could have been written like this

```
plot(densities[[1]], xlim = c(-20, 20))

for(i in seq_along(densities)) {
  if(i == 1) next
  lines(x = densities[[i]]$x,
        y = densities[[i]]$y)
}
```

# Breaking loops

---

- Similarly, if a condition is met, you may want to break out of the loop

```
set.seed(1)

rand_unif <- vector("double", 10)

for(i in seq_along(rand_unif)) {
  rand_unif[i] <- runif(1, 0, 10)
  if(any(rand_unif > 5)) {
    break
  }
}

rand_unif
```

```
## [1] 2.655087 3.721239 5.728534 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```



\*apply

---

# lapply

---

- One of numerous *functionals* in R
- A functional "takes a function as an input and returns a vector as output" (adv-r, Chpt 9)
- `lapply` will **always** return a list

# Revisiting our simulation with $n = 10$

Our **for** loop version

```
increments <- seq(1, 5, by = 0.2)

simulated <- vector("list", length(increments))

for(i in seq_along(simulated)) {
  simulated[[i]] <- rnorm(10, 0, increments[i])
  # note use of `[[`
}

simulated
```

```
## [[1]]
## [1]  1.329799263  1.272429321  0.414641434 -1.539950042 -0.928567035 -0.
## [8]  2.404653389  0.763593461 -0.799009249
##
## [[2]]
## [1] -1.3771884 -0.3473539 -0.3590581 -0.4938130  0.3026681 -1.0703054
## [10]  0.4528748
##
## [[3]]
```

# The `lapply` version

---

```
sim_l <- lapply(seq(1, 5, by = 0.2), function(sd) {  
  rnorm(10, 0, sd)  
})  
sim_l
```

```
## [[1]]  
## [1] -1.06620017 -0.23845635 1.49522344 1.17215855 -1.45770721 0.0950  
## [9] 1.40856336 -0.54176036  
##  
## [[2]]  
## [1] 0.33439767 -0.23276729 1.89138982 -1.77065716 -0.17352985 -1.1438  
## [9] -1.81739641 -0.07404891  
##  
## [[3]]  
## [1] -0.2061791 2.1582303 -1.3745979 0.6952094 2.3757270 -0.3650308 -  
## [10] -1.4189555  
##  
## [[4]]  
## [1] 2.583603577 0.009027176 -4.647838497 -1.771463710 2.476107092 -1  
## [8] 0.068240400 -2.554748823 0.785547796  
##  
## [[5]]  
## [1] 0.7588861 3.3730270 1.8621258 0.1472586 -0.1485428 1.0909322 -  
## [10] 0.9907080  
##  
## [[6]]
```

# Some more examples

---

## Loop through a data frame

- Remember – a data frame is a list. We can loop through it easily

```
library(palmerpenguins)  
lapply(penguins, is.double)
```

```
## $species  
## [1] FALSE  
##  
## $island  
## [1] FALSE  
##  
## $bill_length_mm  
## [1] TRUE  
##  
## $bill_depth_mm  
## [1] TRUE  
##  
## $flipper_length_mm  
## [1] FALSE  
##
```

```
lapply(mtcars, mean)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
##
## $hp
## [1] 146.6875
##
## $drat
## [1] 3.596563
##
## $wt
## [1] 3.21725
##
## $qsec
## [1] 17.84875
##
## $vs
## [1] 0.4375
##
## $am
## [1] 0.40625
##
## $gear
```

# Add a condition

---

```
lapply(penguins, function(x) {  
  if(is.numeric(x)) {  
    mean(x, na.rm = TRUE)  
  }  
})
```

```
## $species  
## NULL  
##  
## $island  
## NULL  
##  
## $bill_length_mm  
## [1] 43.92193  
##  
## $bill_depth_mm  
## [1] 17.15117  
##  
## $flipper_length_mm  
## [1] 200.9152  
##  
## $body_mass_g  
## [1] 4201.754  
##  
## $sex  
## NULL
```

# Add a second condition

---

```
lapply(penguins, function(x) {  
  if(is.numeric(x)) {  
    return(mean(x, na.rm = TRUE))  
  }  
  else if(is.character(x) |  
          is.factor(x)) {  
    return(table(x))  
  }  
})
```

```
## $species  
## x  
##      Adelie Chinstrap      Gentoo  
##      152         68        124  
##  
## $island  
## x  
##      Biscoe      Dream Torgersen  
##      168        124         52  
##  
## $bill_length_mm  
## [1] 43.92193  
##  
## $bill_depth_mm  
## [1] 17.15117
```



# Passing arguments

---

```
head(airquality)
```

```
##      Ozone  Solar.R Wind Temp Month Day
## 1      41      190  7.4   67     5   1
## 2      36      118  8.0   72     5   2
## 3      12      149 12.6   74     5   3
## 4      18      313 11.5   62     5   4
## 5      NA       NA 14.3   56     5   5
## 6      28       NA 14.9   66     5   6
```

```
lapply(airquality, mean, na.rm = TRUE)
```

```
## $Ozone
## [1] 42.12931
##
## $Solar.R
## [1] 185.9315
##
## $Wind
## [1] 9.957516
##
## $Temp
## [1] 77.88235
##
## $Month
```

# Simulation again

---

```
lapply(seq(1, 5, 0.2), rnorm, n = 10, mean = 0)
```

```
## [[1]]
## [1] -0.02516264 -0.16367334  0.37005975 -0.38082454  0.65295237  2.0613
## [9] -0.72275312 -0.62916466
##
## [[2]]
## [1] -2.1794473 -0.3111469  0.4015587 -1.7126011  2.3263539 -0.9114363 -
## [10]  1.9155850
##
## [[3]]
## [1]  1.7884237  1.1045592  0.6460515 -0.6132968 -2.1109298 -3.1121246 -
## [10]  1.0198842
##
## [[4]]
## [1] -1.4154959 -2.4615063 -1.6710007 -2.7490179  1.2860121 -2.4028595 -
## [10]  3.0302573
##
## [[5]]
## [1] -3.1684074  1.6641842 -1.0017759 -0.3250514  2.6053925 -1.0928366
## [10]  2.5391869
##
## [[6]]
## [1] -0.4491476 -0.4249910  1.3927569  1.8303650 -1.8467486  2.2937465 -
## [10] -0.2909816
##
```

# Mimic `dplyr::group_by`

```
by_cyl <- split(mtcars, mtcars$cyl)
str(by_cyl)
```

```
## List of 3
## $ 4:'data.frame': 11 obs. of 11 variables:
##   ..$ mpg : num [1:11] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26 30.4 ...
##   ..$ cyl : num [1:11] 4 4 4 4 4 4 4 4 4 4 4 ...
##   ..$ disp: num [1:11] 108 146.7 140.8 78.7 75.7 ...
##   ..$ hp : num [1:11] 93 62 95 66 52 65 97 66 91 113 ...
##   ..$ drat: num [1:11] 3.85 3.69 3.92 4.08 4.93 4.22 3.7 4.08 4.43 3.77
##   ..$ wt : num [1:11] 2.32 3.19 3.15 2.2 1.61 ...
##   ..$ qsec: num [1:11] 18.6 20 22.9 19.5 18.5 ...
##   ..$ vs : num [1:11] 1 1 1 1 1 1 1 1 0 1 ...
##   ..$ am : num [1:11] 1 0 0 1 1 1 0 1 1 1 ...
##   ..$ gear: num [1:11] 4 4 4 4 4 4 3 4 5 5 ...
##   ..$ carb: num [1:11] 1 2 2 1 2 1 1 1 2 2 ...
## $ 6:'data.frame': 7 obs. of 11 variables:
##   ..$ mpg : num [1:7] 21 21 21.4 18.1 19.2 17.8 19.7
##   ..$ cyl : num [1:7] 6 6 6 6 6 6 6
##   ..$ disp: num [1:7] 160 160 258 225 168 ...
##   ..$ hp : num [1:7] 110 110 110 105 123 123 175
##   ..$ drat: num [1:7] 3.9 3.9 3.08 2.76 3.92 3.92 3.62
##   ..$ wt : num [1:7] 2.62 2.88 3.21 3.46 3.44 ...
##   ..$ qsec: num [1:7] 16.5 17 19.4 20.2 18.3 ...
##   ..$ vs : num [1:7] 0 0 1 1 1 1 0
##   ..$ am : num [1:7] 1 1 0 0 0 0 1
```

```
lapply(by_cyl, function(x) mean(x$mpg))
```

```
## $`4`  
## [1] 26.66364  
##  
## $`6`  
## [1] 19.74286  
##  
## $`8`  
## [1] 15.1
```

# Your turn

---

Try splitting the penguins dataset by species and calculating the average `bill_length_mm`

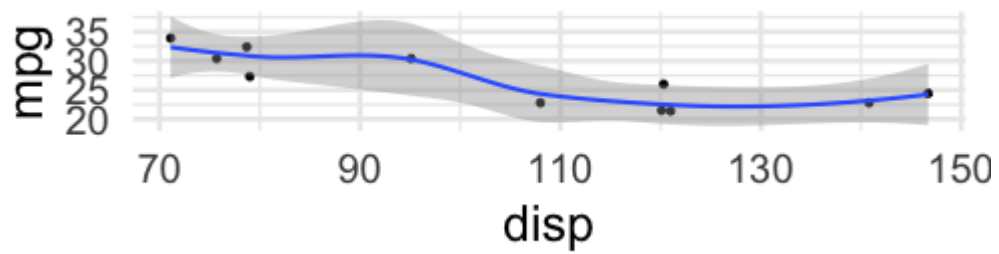
04:00

# Produce separate plots

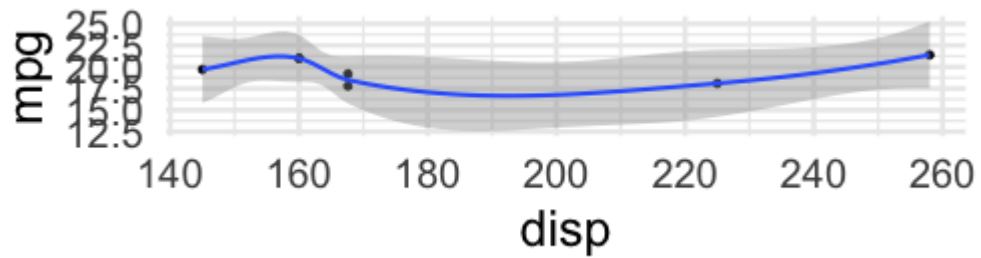
---

```
lapply(by_cyl, function(x) {  
  ggplot(x, aes(displacement, mpg)) +  
    geom_point() +  
    geom_smooth()  
})
```

```
## $`4`
```



```
##  
## $`6`
```



```
##  
## $`8`
```

# Your turn

---

Produce separate plots of the relation between  
`bill_length_mm` and `body_mass_g`

04:00



# Saving

---

- You can extend this example further by saving the plot outputs to an object, then looping through that object to save the plots to disk.
- Using functionals, this would require parallel iterations, which we'll cover later (need to loop through plots and a file name)
- Could extend it fairly easily with a **for** loop

# Saving w/ **for** loop

---

Save plots to an object (list)

```
plots <- lapply(by_cyl, function(x) {  
  ggplot(x, aes(displacement, mpg)) +  
    geom_point() +  
    geom_smooth()  
})
```

Specify file names/directory

```
#dir.create(here::here("plots"))  
filenames <- here::here("plots",  
                        paste0("cyl", names(by_cyl), ".png"))  
filenames
```

```
## [1] "/Users/daniel/Teaching/data_sci_specialization/2020-21/c3-fp-2021/p  
## [2] "/Users/daniel/Teaching/data_sci_specialization/2020-21/c3-fp-2021/p  
## [3] "/Users/daniel/Teaching/data_sci_specialization/2020-21/c3-fp-2021/p
```

# Saving

---

```
for(i in seq_along(plots)) {  
  ggsave(filenamees[i], # single bracket  
    plots[[i]], # double bracket  
    device = "png",  
    width = 6.5,  
    height = 8)  
}
```

You try!

---

04:00

# Variants of `lapply`

---

- `sapply`
  - Will try to **s**implify the output, if possible. Otherwise it will return a list.
  - Fine for interactive work, but I strongly recommend against it if writing a function (difficult to predict the output)
- `vapply`
  - Strict – you specify the output
  - Use if writing functions (or just always stick with `lapply`), or consider jumping to `{purrr}` (next week)

# Examples

---

## Our simulation

```
sim_s <- sapply(seq(1, 5, by = 0.2), function(x) {  
  rnorm(10, 0, x)  
})  
class(sim_s)
```

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

```
dim(sim_s)
```

```
## [1] 10 21
```

```
sim_s
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]  
## [1,] -2.939773695 -0.38696441 -2.0067288 -0.6857975  0.26119194 -2.6882  
## [2,]  0.002415809 -1.60656089 -1.4772590 -1.0498872 -4.39616038 -3.0463  
## [3,]  0.509665571  0.82578723 -1.0263566  1.5350309  1.04457363 -0.8439  
## [4,] -1.084720001  0.08553678  0.2952702  2.4896842  1.17909360  2.7218  
## [5,]  0.704832977  2.62770283 -1.3984890 -1.6652743 -0.54811591  3.5075  
## [6,]  0.330976350 -1.38924912  1.5089905  1.4889159 -1.27362282  3.1367
```

```
sapply(penguins, is.double)
```

```
##           species           island  bill_length_mm  bill_depth_mm
##           FALSE           FALSE           TRUE           TRUE
##  body_mass_g           sex           year
##           FALSE           FALSE           FALSE
```

- Now that it's a vector we can easily use it for subsetting

```
head(penguins)
```

```
## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_
##   <fct>   <fct>         <dbl>         <dbl>         <int>
## 1 Adelie  Torgersen         39.1           18.7           181
## 2 Adelie  Torgersen         39.5           17.400          186
## 3 Adelie  Torgersen        40.300           18           195
## 4 Adelie  Torgersen         NA             NA             NA
## 5 Adelie  Torgersen         36.7           19.3           193
## 6 Adelie  Torgersen        39.300           20.6           190
```

```
head( penguins[ ,sapply(penguins, is.double)] )
```

```
## # A tibble: 6 x 2
##   bill_length_mm bill_depth_mm
##         <dbl>         <dbl>
## 1         39.1           18.7
## 2         39.5           17.400
## 3        40.300           18
## 4          NA             NA
## 5         36.7           19.3
## 6        39.300           20.6
```



# Challenge

---

Can you make return the opposite? In other words – all those that are *not* double?

02:00

```
head( penguins[ ,!sapply(penguins, is.double)] )
```

```
## # A tibble: 6 x 6
##   species island flipper_length_mm body_mass_g sex    year
##   <fct>    <fct>          <int>         <int> <fct>  <int>
## 1 Adelie  Torgersen           181          3750 male    2007
## 2 Adelie  Torgersen           186          3800 female  2007
## 3 Adelie  Torgersen           195          3250 female  2007
## 4 Adelie  Torgersen            NA            NA <NA>    2007
## 5 Adelie  Torgersen           193          3450 female  2007
## 6 Adelie  Torgersen           190          3650 male    2007
```

# vapply

---

- As you can probably see, simplifying can be *really* helpful for interactive work.

BUT

- Not ideal for programmatic work – need to be able to reliably predict the output
- **vapply** solves this issue.

```
vapply(mtcars, mean, FUN.VALUE = double(1))
```

```
##           mpg           cyl           disp           hp           drat           wt
## 20.090625    6.187500 230.721875 146.687500    3.596563    3.217250    17.84
##           gear           carb
##    3.687500    2.812500
```

```
vapply(penguins, is.double, FUN.VALUE = character(1))
```

```
## Error in vapply(penguins, is.double, FUN.VALUE = character(1)): values m
## but FUN(X[[1]]) result is type 'logical'
```

```
vapply(penguins, is.double, FUN.VALUE = logical(1))
```

```
##           species           island  bill_length_mm  bill_depth_mm
##           FALSE           FALSE           TRUE           TRUE
## body_mass_g           sex           year
##           FALSE           FALSE           FALSE
```

# Coercion with `vapply`

---

- If it can coerce the vector without loss of information, it will

```
vapply(penguins, is.double, FUN.VALUE = double(1))
```

```
##           species            island  bill_length_mm  bill_depth_mm
##                0                0                1                1
##  body_mass_g          sex          year
##                0                0                0
```

# Count missing data

---

```
vapply(airquality, function(col) {  
  sum(is.na(col))  
},  
double(1)  
)
```

##	Ozone	Solar.R	Wind	Temp	Month	Day
##	37	7	0	0	0	0

# sapply alternative

---

For interactive work, the code on the previous slide is maybe too much. Could be reduced to

```
sapply(airquality, function(col) sum(is.na(col)))
```

```
##      Ozone Solar.R      Wind      Temp      Month      Day  
##       37        7         0         0         0         0
```

# Summary

---

- **for** loops are incredibly flexible and there's nothing inherently "wrong" about them
  - Do require more text, and often repetitive text, which can lead to errors/bugs
  - The flexibility can actually be more of a curse than a blessing



# Summary

---

- The **lapply** family of functions help put the focus on a given function, and what values are being looped through the function
  - **lapply** will always return a list
  - **sapply** will try to simplify, which is problematic for programming, but fine for interactive work
  - **vapply** is strict, and will only return the type specified

# Next time

---

Lab 1