Intro to iteration with Base R

Daniel Anderson
Week 2, Class 1

UNIVERSITY OF OREGON

Agenda

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

Agenda

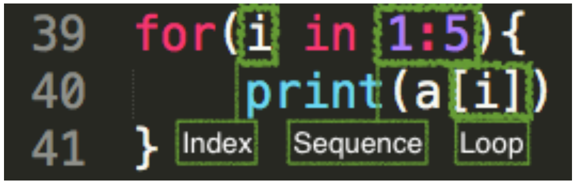
- For loops
- Apply family of loops
  - `lapply()`
  - `sapply()`
  - `vapply()`

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

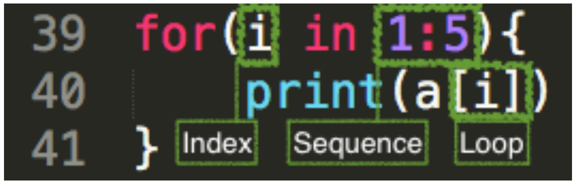
Learning objectives

- 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



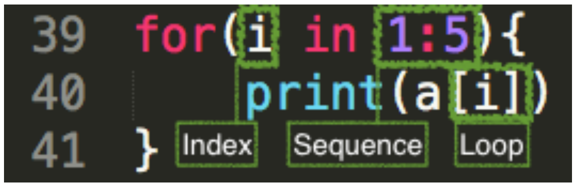
Basic overview: for loops




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

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Basic overview: for loops



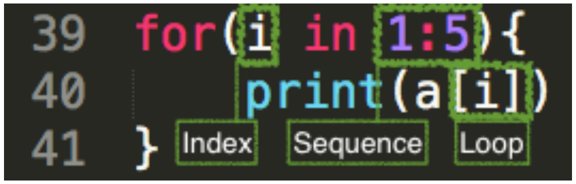

```
a <- letters[1:26]
a
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

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

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

Basic overview: `for` loops




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

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```r
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

Another basic example

Simulate tossing a coin, record results

- For a single toss

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

```
## [1] "Heads"
```

Another basic example

Simulate tossing a coin, record results

- For a single toss

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

```
## [1] "Heads"
```

- 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" "Tails" "Heads" "Tails" "Tails" "Tails" "Tails" "Tails"
##  [9] "Tails" "Heads"
```

Growing vectors

- **Always** pre-allocate a vector for storage before running a `for` loop.

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

Example

```r
library(tictoc)

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

```
## 37.716 sec elapsed
```

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

```
## 0.434 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"
  `paste0(LETTERS[1], letters[1])`

- Write a `for` loop for all letters

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"
## [15] "Oo" "Pp" "Qq" "Rr" "Ss" "Tt" "Uu" "Vv" "Ww" "Xx" "Yy" "Zz"
```

Quick style note

- Why am I always using `seq_along`?

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.

Running the loop

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

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

```r
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

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)
```

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
##  $ : NULL
##  $ : NULL
##  $ : NULL
```

Write for loop

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

```
## List of 21
##  $ : num [1:100] 0.526 -0.488 1.138 1.215 -0.425 ...
##  $ : num [1:100] -2.387 2.88 -1.083 -0.792 -1.722 ...
##  $ : num [1:100] 1.096 0.488 1.77 -2.129 -2.504 ...
##  $ : num [1:100] -0.873 0.464 -0.911 1.469 -0.207 ...
##  $ : num [1:100] 1.087 -0.363 -0.041 0.313 -4.339 ...
##  $ : num [1:100] -0.086 3.77 1.365 -0.301 0.181 ...
##  $ : num [1:100] 2.106 -2.533 1.264 -0.685 1.647 ...
##  $ : num [1:100] -2.66 2.81 -1.45 -1.71 -2.43 ...
##  $ : num [1:100] -4.808 4.387 2.268 0.712 0.458 ...
##  $ : num [1:100] -2.2601 0.6798 0.2835 0.0796 0.8568 ...
##  $ : num [1:100] 1.142 0.198 -4.268 -4.195 3.53 ...
##  $ : num [1:100] 1.36 -2.48 -1.44 3.52 -6.04 ...
##  $ : num [1:100] 0.9843 0.0286 6.8353 0.306 -11.3752 ...
##  $ : num [1:100] -1.31 0.731 -2.888 -5.283 4.79 ...
##  $ : num [1:100] 1.787 -5.524 0.728 -1.48 0.775 ...
##  $ : num [1:100] -0.468 -7.216 -4.278 0.522 -0.784 ...
##  $ : num [1:100] -10.036 -0.57 4.762 -0.229 1.634 ...
```

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)
```

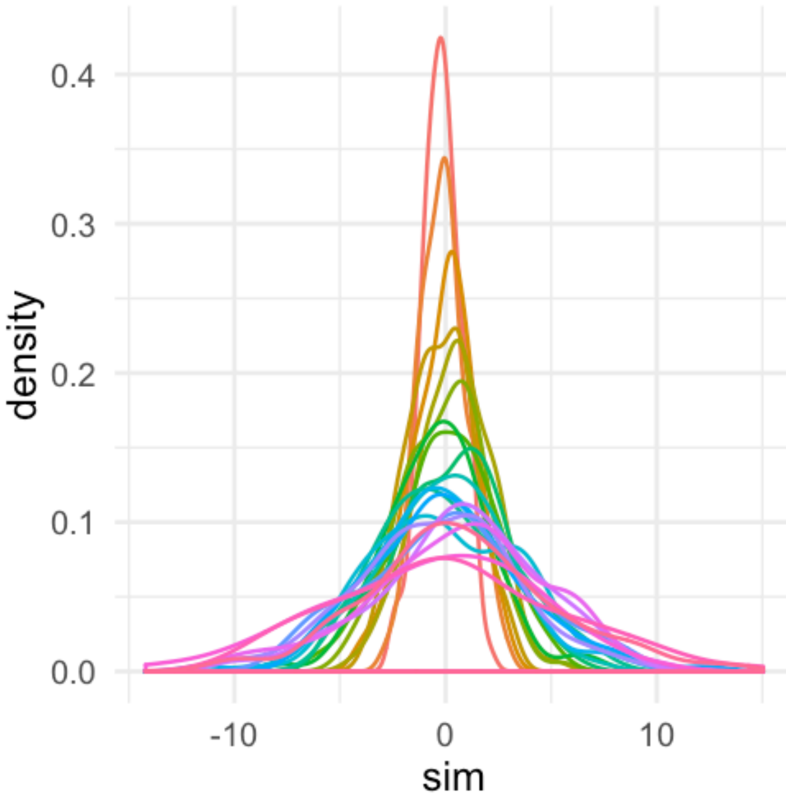
```
##         sd_1     sd_1.2     sd_1.4     sd_1.6      sd_1.8        sd_2
## 1  0.5258908 -2.3873130  1.0961069 -0.8730421  1.08667357 -0.08599321
## 2 -0.4875444  2.8798469  0.4882225  0.4641221 -0.36318687  3.77025309
## 3  1.1382508 -1.0825756  1.7704895 -0.9106457 -0.04103216  1.36549458
## 4  1.2151344 -0.7919963 -2.1289891  1.4693974  0.31305978 -0.30101182
## 5 -0.4248307 -1.7219648 -2.5036642 -0.2066456 -4.33921424  0.18097826
## 6 -1.4508403 -1.3647772  2.4902680 -3.0258191  3.84883329 -4.89857070
##       sd_2.2     sd_2.4     sd_2.6      sd_2.8       sd_3     sd_3.2
## 1  2.1059037 -2.6632801 -4.8081303 -2.26008043  1.1419761  1.363778
## 2 -2.5331554  2.8135597  4.3871383  0.67977714  0.1979198 -2.478426
## 3  1.2641686 -1.4489704  2.2681508  0.28349963 -4.2684292 -1.437570
## 4 -0.6851658 -1.7090189  0.7120512  0.07955778 -4.1953347  3.519224
## 5  1.6465592 -2.4299403  0.4584426  0.85682207  3.5304385 -6.040620
```

tidyverse

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

```
library(tidyverse)
pd <- sim_d %>%
    gather(sd, sim) %>%
    mutate(sd =
      factor(parse_number(sd)))

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



Base R Method

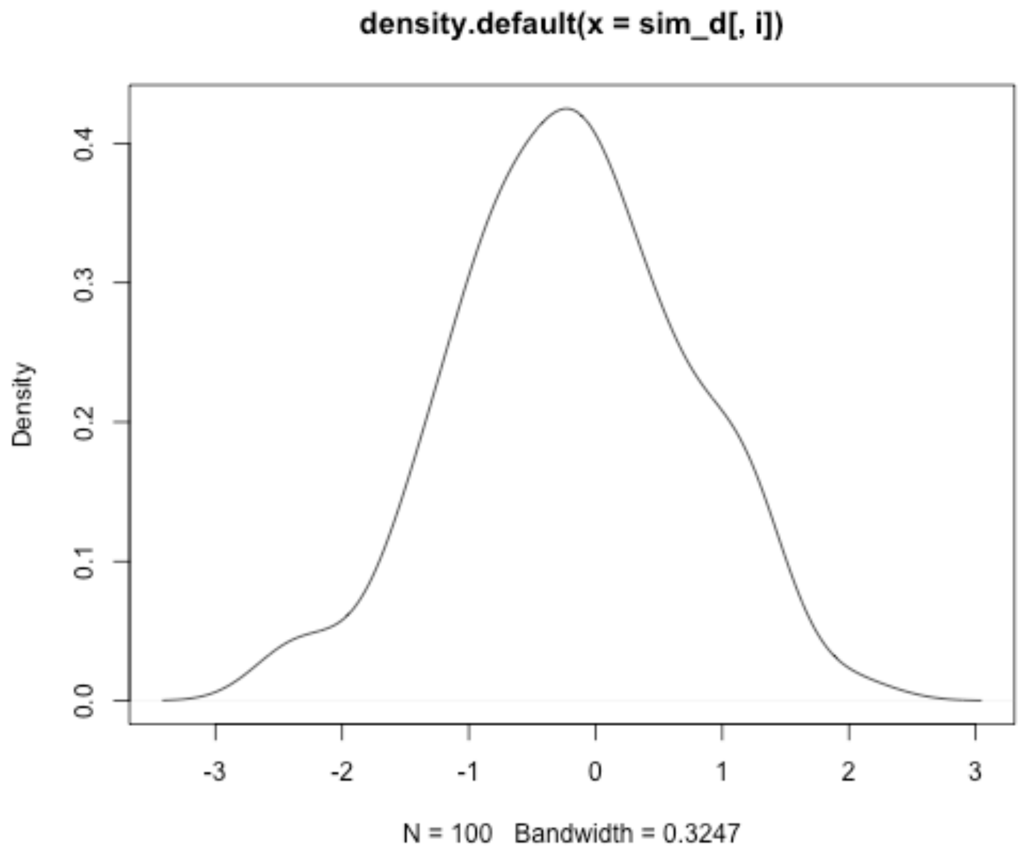
- Calculate all the densities

```
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] -3.42 -3.41 -3.39 -3.38 -3.37 ...
##   ..$ y        : num [1:512] 0.000251 0.000283 0.000319 0.000359 0.000402 ...
##   ..$ bw       : num 0.325
##   ..$ 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] -4.03 -4.01 -4 -3.98 -3.96 ...
##   ..$ y        : num [1:512] 0.000117 0.000132 0.000149 0.000168 0.000188 ...
##   ..$ bw       : num 0.39
##   ..$ n        : int 100
```

- Next, plot the first density

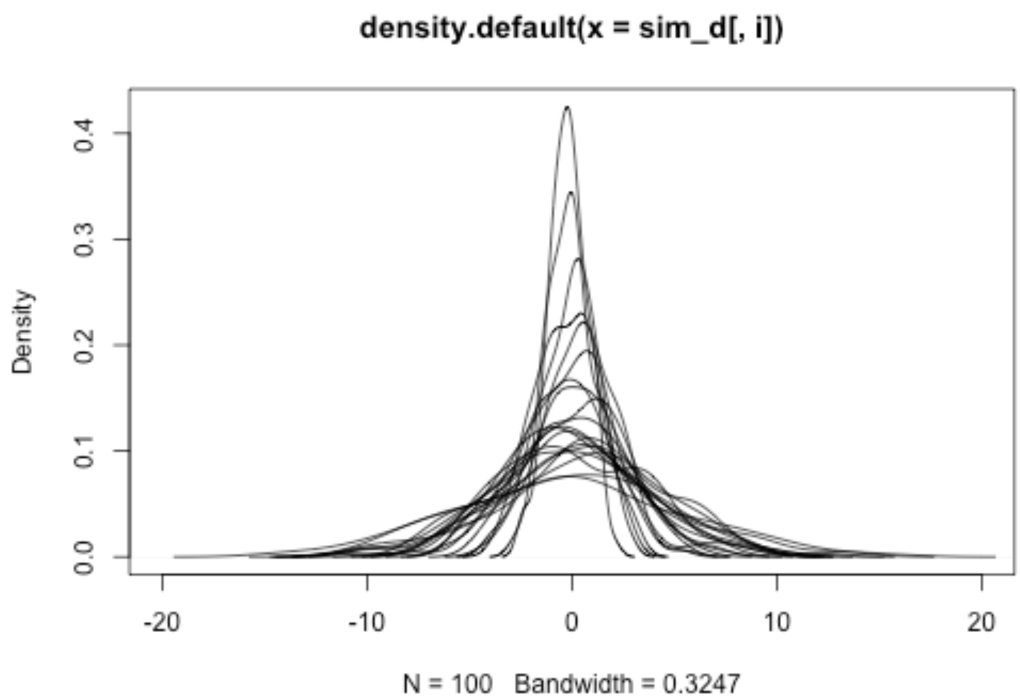
```
plot(densities[[1]])
```



- 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

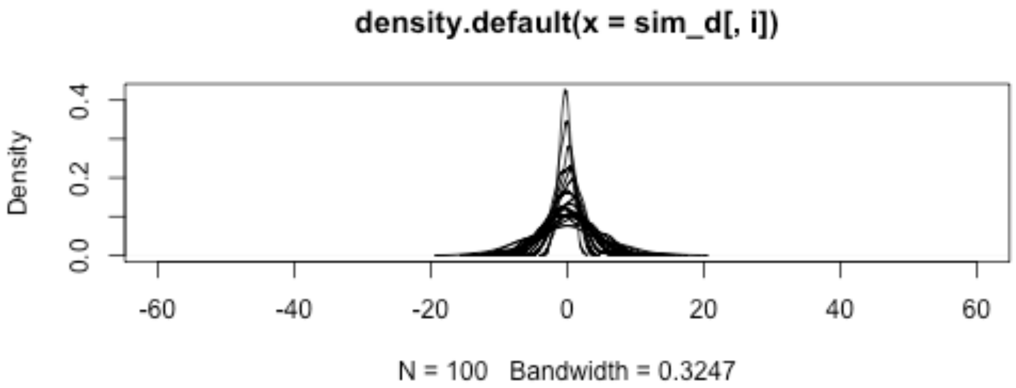
- On the prior slide, I set the index to skip over the first by using `seq(2, length(densities))`

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(-60, 60))

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
##  [8] 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`

- 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
##  [6] -0.294720447 -0.005767173  2.404653389  0.763593461 -0.799009249
##
## [[2]]
##  [1] -1.3771884 -0.3473539 -0.3590581 -0.4938130  0.3026681 -1.0703054
##  [7]  0.5228200 -1.4850461 -0.2691215  0.4528748
##
## [[3]]
##  [1]  0.18667091  1.12586531 -0.07994948  0.70505116  1.52007711
##  [6] -0.96733538 -1.79843910  0.06541664 -0.32998918 -0.76004356
```

The `lapply` version

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

```
## [[1]]
##  [1] -1.06620017 -0.23845635  1.49522344  1.17215855 -1.45770721
##  [6]  0.09505623  0.84766496 -1.62436453  1.40856336 -0.54176036
##
## [[2]]
##  [1]  0.33439767 -0.23276729  1.89138982 -1.77065716 -0.17352985
##  [6] -1.14384377  0.48785128  2.67511464 -1.81739641 -0.07404891
##
## [[3]]
##  [1] -0.2061791  2.1582303 -1.3745979  0.6952094  2.3757270 -0.3650308
##  [7] -0.9883000 -0.2256499  0.7018506 -1.4189555
##
## [[4]]
##  [1]  2.583603577  0.009027176 -4.647838497 -1.771463710  2.476107092
##  [6] -1.562928561 -0.162405516  0.068240400 -2.554748823  0.785547796
##
## [[5]]
##  [1]  0.7588861  3.3730270  1.8621258  0.1472586 -0.1485428  1.0909322
##  [7] -1.5973563  0.1897585  0.6351741  0.9907080
##
```

Some more examples

Loop through a data frame

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

Some more examples

Loop through a data frame

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

```
lapply(iris, is.double)
```

```
## $Sepal.Length
## [1] TRUE
##
## $Sepal.Width
## [1] TRUE
##
## $Petal.Length
## [1] TRUE
##
## $Petal.Width
## [1] TRUE
##
## $Species
## [1] FALSE
```

```r
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
```

Add a condition

```r
lapply(iris, function(x) {
    if(is.double(x)) {
        mean(x)
    }
})
```

```
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
##
## $Species
## NULL
```

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
```

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
##  [6]  2.06134181 -1.79664494  0.58407712 -0.72275312 -0.62916466
##
## [[2]]
##  [1] -2.1794473 -0.3111469  0.4015587 -1.7126011  2.3263539 -0.9114363
##  [7] -2.7345314 -0.1368609  2.8222280  1.9155850
##
## [[3]]
##  [1]  1.7884237  1.1045592  0.6460515 -0.6132968 -2.1109298 -3.1121246
##  [7] -1.6501414 -2.4958643 -1.3830868  1.0198842
##
## [[4]]
##  [1] -1.4154959 -2.4615063 -1.6710007 -2.7490179  1.2860121 -2.4028595
##  [7] -0.2327985  0.9271338  1.9224409  3.0302573
##
## [[5]]
##  [1] -3.1684074  1.6641842 -1.0017759 -0.3250514  2.6053925 -1.0928366
##  [7]  1.2228524 -0.1684038 -0.8821553  2.5391869
##
## [[6]]
##  [1] -0.4491476 -0.4249910  1.3927569  1.8303650 -1.8467486  2.2937465
```

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 ...
##   ..$ 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 ...
```

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

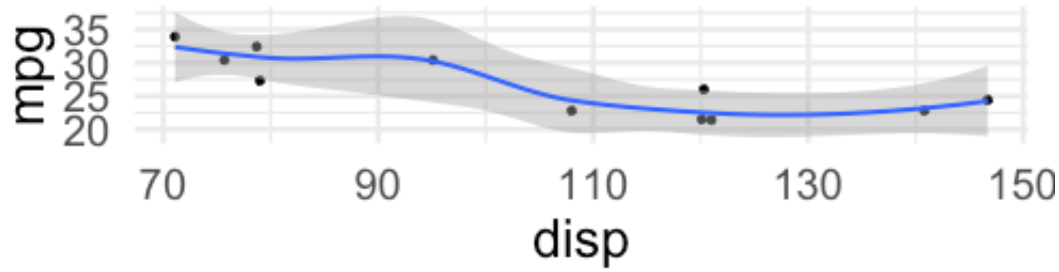
```
## $`4`
## [1] 26.66364
##
## $`6`
## [1] 19.74286
##
## $`8`
## [1] 15.1
```

But more power

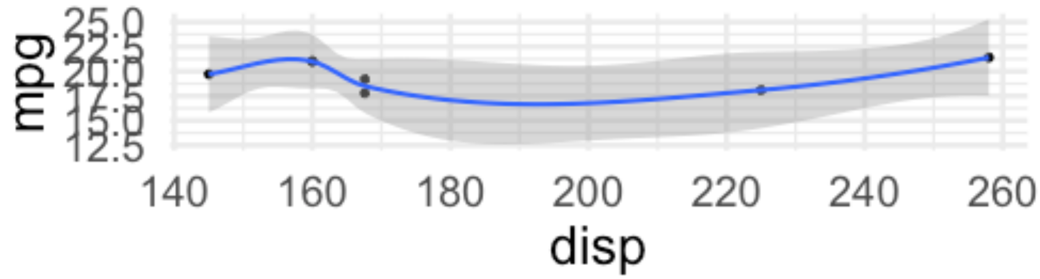
Produce separate plots

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

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


mpg
35
30
25
20
70
90
110
130
150
disp

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


mpg
25.0
22.5
20.0
17.5
15.0
12.5
140
160
180
200
220
240
260
disp

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

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.

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)

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(disp, mpg)) +
        geom_point() +
        geom_smooth()
})
```

Saving w/ `for` loop

Save plots to an object (list)

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

Specify file names/directory

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

```
## [1] "/Users/Daniel/Teaching/data_sci_specialization/c3-fun_program_r/plots/cyl4
## [2] "/Users/Daniel/Teaching/data_sci_specialization/c3-fun_program_r/plots/cyl6
## [3] "/Users/Daniel/Teaching/data_sci_specialization/c3-fun_program_r/plots/cyl8
```

Saving

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

Equivalent with `lapply`

You can actually use `lapply` just like a for loop if the vector you feed it is just an index

```
lapply(seq_along(plots), function(i) {
    ggsave(filenames[i],
           plots[[i]],
           device = "png",
           width = 6.5,
           height = 8)
})
```

- Generally I would not do this - it's somewhat outside of the spirit of functional programming (which I still haven't fully defined, but I will next week)

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)

Variants of `lapply`

- `sapply`
  - Will try to simplify 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, it throws an error if output doesn't match
  - 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"
```

```
dim(sim_s)
```

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

```
sim_s
```

```
##             [,1]         [,2]        [,3]        [,4]       [,5]
##  [1,]  0.5941965  0.779306321  2.74940128 -0.68645388 -2.0738827
##  [2,] -0.2703152 -0.769341787  0.98914028  0.81404203 -1.7553302
##  [3,]  1.5540761  0.557351725  0.05554497 -2.31502359 -1.2805445
##  [4,] -0.5107425 -0.371040537 -0.22192603  1.63122053  0.6694425
##  [5,] -0.2918427  1.531224851 -0.22716057  1.88567516 -1.6983260
##  [6,]  1.1014382 -0.057133464  0.61435068 -0.01641402 -0.4938612
##  [7,]  0.4586652  0.036121399  1.09297901  0.42979979  0.2780433
##  [8,]  0.8519080 -0.004296324 -1.37368787  2.14724620 -2.3195758
##  [9,]  1.4661999 -2.441487172 -0.19822002 -0.93377031  2.5543842
```

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

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

```r
sapply(iris, is.double)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

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

```r
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
head( iris[ ,sapply(iris, is.double)] )
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
## 4          4.6         3.1          1.5         0.2
## 5          5.0         3.6          1.4         0.2
## 6          5.4         3.9          1.7         0.4
```

Challenge

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

Challenge

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

```
head( iris[ ,!sapply(iris, is.double), drop = FALSE] )
```

```
##   Species
## 1  setosa
## 2  setosa
## 3  setosa
## 4  setosa
## 5  setosa
## 6  setosa
```

`vapply`

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

`vapply`

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

BUT

`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`

- 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.

```r
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
##       qsec         vs         am       gear       carb
##  17.848750   0.437500   0.406250   3.687500   2.812500
```

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

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

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

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##         TRUE         TRUE         TRUE         TRUE        FALSE
```

Coercion with `vapply`

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

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

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##            1            1            1            1            0
```

Count missing data

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

```
##    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

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

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

- `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
- The `lapply` family of functions help put the focus on a given function, and what values are being looped through the function

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
- 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

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
- 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

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
- 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