For Loops & Bootstrapping

The Problem

- Copying and pasting code is not an efficient use of your time
- Often you want to do the same thing but with different inputs:
  - Get the mean of a variable for 20 different groups
  - Plot different variables as your x-axis against the exact same y-axis
  - Do the exact same analysis and make the same plots for the levels of an independent variable (e.g., patients and controls)
- In sum, you're trying to **iterate** (perform repeatedly)

How do we address this?

In this class, we are going to talk specifically about `for loops`. Why?

- They are *general purpose*. You will find them in nearly every single programming language. If you decide to not use R and instead go to Python or Matlab or whatever, you'll still come across them.
- This is a fundamental component of programming. I would guess that you learn this within the first 2 weeks of CS 131.

There are other ways to do this that are `R`-specific. We are NOT going to cover these (counterintutive, I know).

- the `apply` family of functions, including `lapply`
- using the `purr` package from the `tidyverse`

Lists

Lists are basically vectors but where every element can be a totally different data class. Ex:

```
myList <- list(6, head(iris), "hello world", c(1, 3, 5, 7, 9))
myList
```

```
## [[1]]
## [1] 6
##
## [[2]]
##   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
##
## [[3]]
## [1] "hello world"
##
## [[4]]
## [1] 1 3 5 7 9
```

Lists

You can also name the elements in your list:

```
myList <- list(Numberz = 6, DFs = head(iris), Chars = "hello world", VectorKing = c(1, 3, 5, 7, 9))
myList
```

```
## $Numberz
## [1] 6
##
## $DFs
##   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
##
## $Chars
## [1] "hello world"
##
## $VectorKing
## [1] 1 3 5 7 9
```

Lists

The weirdest thing about lists is accessing the elements within a list. Think of it like a book:

- The element itself is like the chapter of a book
- But there's a page that just says "Chapter 4" but doesn't contain any text
- You need to tell R to go to the actual chapter

We can do this with double brackets 👇

```
myList[1]
```

```
## $Numberz
## [1] 6
```

```
myList[[1]]
```

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

Lists

Why does this matter? Say we wanted to take our number 6 from our list and add it to the number 12

```
myList[1] + 12
```

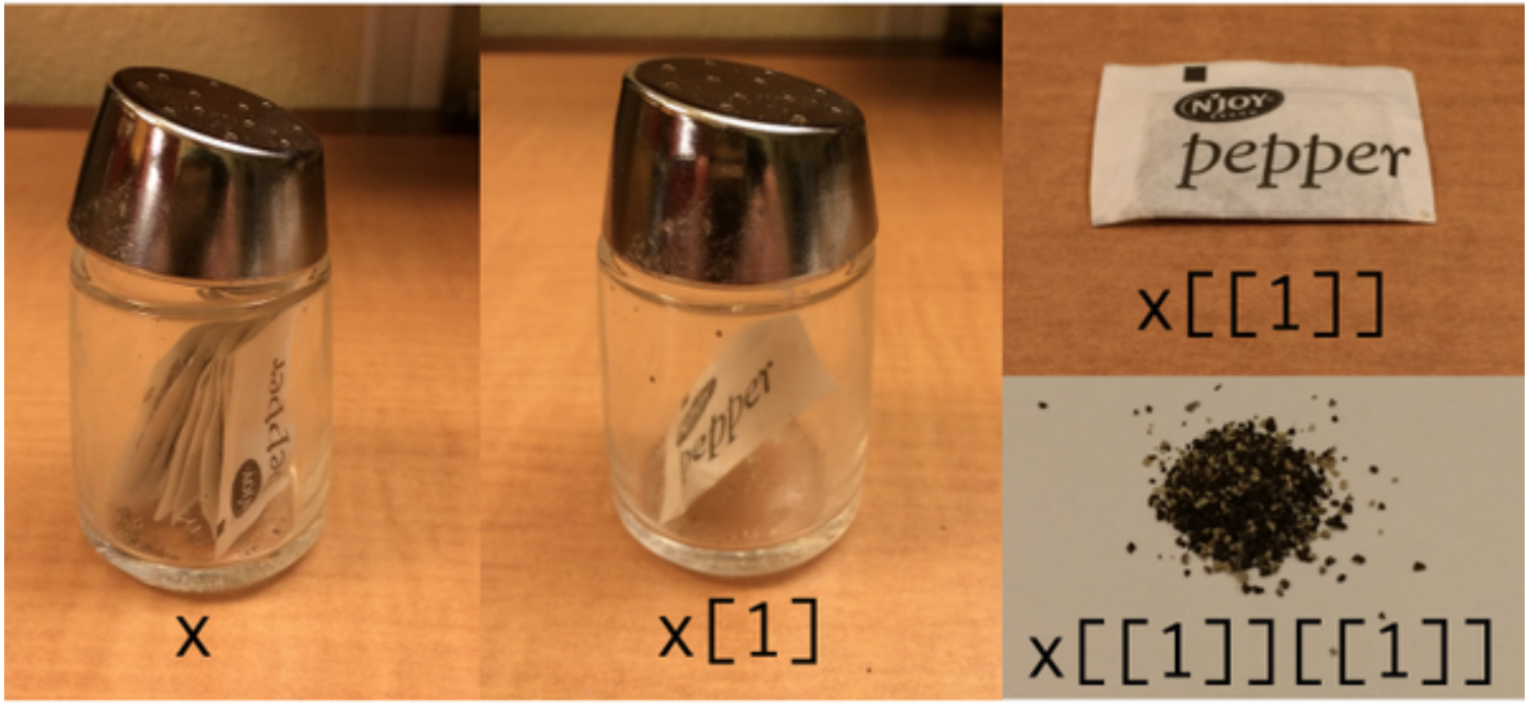
```
## Error in myList[1] + 12: non-numeric argument to binary operator
```

```
myList[[1]] + 12
```

```
## [1] 18
```

Lists are weird

From Hadley Wickham, creator of `tidyverse`



Now on to `for loops`

```
for (i in 1:some number) {
  do something
}
```

"For each element in 1 through _, perform some function"

"Perform the function contained in this for loop on every single element in some list of elements"

for loops

```
for (i in 1:some number) {
  do something
}
```

The important parts:

- You can iterate over anything: rows of a data.frame, columns of a data.frame, lists, or simple 1-dimensional vectors. The `1:some number` portion are the elements you're iterating over. If you wanted to do something 5 times, you could say `1:5`. Most of the time, we don't know that second number, though. So we can use a function. `1:nrow(data.frame)` or `1:length(list)` or `1:length(vector)`.
- The `i` stands for "each" and is the same type of `i` seen in equations. The top line then reads: "For each item/element in 1 through some number".
- The part between the curly brackets `{ }` is what you want to do (it's the body of the for loop).

A Simple Example

For each number in 1 through 10, print the following: *"(number) squared is (that number squared)"*

So for the number 2, the output should be *"2 squared is 4"*

The functions we are going to use are `print()` and `paste0()`.

```
for (i in 1:10) {
  print(paste0(i, " squared is ", i^2))
}
```

```
## [1] "1 squared is 1"
## [1] "2 squared is 4"
## [1] "3 squared is 9"
## [1] "4 squared is 16"
## [1] "5 squared is 25"
## [1] "6 squared is 36"
## [1] "7 squared is 49"
## [1] "8 squared is 64"
## [1] "9 squared is 81"
## [1] "10 squared is 100"
```

Storing the output

Is there any way for me to access any of the squared numbers from above? No!

If you want to store the output of a loop (which in almost all cases we do), you need to **initialize** an empty object. This means make a blank object *before* running your loop that will contain your stored results.

Let's run the same loop as last time, but this time let's store the squared numbers, rather than just printing out some lines.

```
squaredNumbers <- NULL

for (i in 1:10) {
  squaredNumbers[i] <- i^2
}

squaredNumbers
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Recap

The basic steps of constructing a `for loop`:

1. Figure out what it is that you want to iterate through (numbers, columns of a data.frame, data.frames, a list, etc.)

2. Think about what you want your output to look like, and initialize an empty object that can store the output

3. Type out all of the steps within the body of the loop

Applied Examples

What if you want to run the following models where the `y` variable stays the same, but the `x` variable changes? For this first example, let's say we only care about the p-value.

- `lm(Sepal.Length ~ Sepal.Width, data = iris)`
- `lm(Sepal.Length ~ Petal.Length, data = iris)`
- `lm(Sepal.Length ~ Petal.Width, data = 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
```

Applied Examples

Thinking through what we need to happen:

1. We're iterating through specific columns of the `iris` data.frame -- so we need to get those column names into a form we can iterate through.

2. The output is a vector that contains p-values only

3. The body of the loop needs to contain the `lm()` functions

Applied Example 1

```
varsToIterate <- colnames(iris)[2:4]
sigValues <- NULL

for (i in 1:length(varsToIterate)) {
  model <- lm(Sepal.Length ~ iris[,i], data = iris)
  model <- tidy(model)
  sig <- model[[2,5]]
  sigValues[i] <- sig
}

sigValues
```

```
## [1] 0.000000e+00 1.518983e-01 1.038667e-47
```

Applied Example 1

You can make comments within `for loops`

```
# get the column names in a format that we can iterate through
varsToIterate <- colnames(iris)[2:4]

# initialize an empty output vector
sigValues <- NULL

for (i in 1:length(varsToIterate)) {
  # run the model where i is the column that is varying
  model <- lm(Sepal.Length ~ iris[,i], data = iris)

  # the tidy function comes from the `broom` package
  model <- tidy(model)

  # find the p-value from the tidied model
  sig <- model[[2,5]]

  # store the p-value in the output
  sigValues[i] <- sig
}

# print the output
```

Applied Example 2

Now, what if you want to store the entire output of the model, not just the p-value? The output of `tidy(model)` is a data.frame. So the result of our loop will now be a **list of data.frames**, rather than a vector of p-values. Note the double brackets in `modelList[[i]]`!

```
varsToIterate <- colnames(iris)[2:4]

modelList <- list()

for (i in 1:length(varsToIterate)) {
  model <- lm(Sepal.Length ~ iris[,i], data = iris)
  model <- tidy(model)
  modelList[[i]] <- model
}
```

Applied Example 2

```
print(modelList)
```

```
## [[1]]
## # A tibble: 2 x 5
##   term        estimate std.error statistic p.value
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 (Intercept)        0  3.79e-17   0.            1
## 2 iris[, i]          1  6.43e-18   1.56e17       0
##
## [[2]]
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    6.53      0.479     13.6  6.47e-28
## 2 iris[, i]     -0.223     0.155     -1.44 1.52e- 1
##
## [[3]]
## # A tibble: 2 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)    4.31     0.0784      54.9 2.43e-100
## 2 iris[, i]      0.409    0.0189      21.6 1.04e- 47
```

Applied Example 2.1

Let's modify this slightly so that each output data.frame has a name associated with it (rather than 1, 2, 3)

```
varsToIterate <- colnames(iris)[2:4]

modelList <- list()

for (i in 1:length(varsToIterate)) {
  name <- paste0(varsToIterate[i])

  model <- lm(Sepal.Length ~ iris[,i], data = iris)
  model <- tidy(model)
  modelList[[name]] <- model
}
```

Applied Example 2.1

```
print(modelList)
```

```
## $Sepal.Width
## # A tibble: 2 x 5
##   term        estimate std.error statistic p.value
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 (Intercept)        0  3.79e-17   0.             1
## 2 iris[, i]          1  6.43e-18   1.56e17        0
##
## $Petal.Length
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    6.53      0.479     13.6  6.47e-28
## 2 iris[, i]     -0.223     0.155     -1.44 1.52e- 1
##
## $Petal.Width
## # A tibble: 2 x 5
##   term        estimate std.error statistic   p.value
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)    4.31     0.0784      54.9 2.43e-100
## 2 iris[, i]      0.409    0.0189      21.6 1.04e- 47
```

Applied Example 3

Let's do the same thing but with some modifications:

- Let's also plot X & Y so we have a figure that corresponds with each model
- Instead of the output being a list of the different models, what if we wanted the information from all of the models to be contained within a data.frame?

When it comes to looping through plots, there are a few odd things:

- the `sym()` function will take the quotes off of a string so that it can be evaluated properly
- `!!` says "actually evaluate what a variable stands for". This will make sense when you see it in the code.

Applied Example 3

We want to store the model outputs AND a list of plots. So we need to intialize 2 things

```
modelDF <- data.frame() # for models
plotList <- list() # for plots

for (i in 1:length(varsToIterate)) {
  nameX <- paste0(varsToIterate[i]) # a character string for labels
  nameY <- "Sepal.Length" # doesn't change in the loop!

  # make the models
  model <- lm(Sepal.Length ~ iris[,i], data = iris)
  model <- tidy(model)

  # add a column that repeats whatever nameX is
  # for us, this will make it easier to keep track of what "i" is
  model$predictor <- rep(nameX, times = nrow(model))

  # now bind the current model underneath the previous model
  # so that it's all contained within the same data.frame
  modelDF <- rbind(modelDF, model)

  # now make the plots
  nameXplot <- sym(varsToIterate[i])

  plotList[[i]] <- ggplot(data = iris,
                          aes(x = !! nameXplot,
                              y = Sepal.Length)) +
    geom_point(color = "cornflowerblue") +
    labs(title = paste0(nameX, " by ", nameY),
         x = nameX,
         y = nameY)
}
```

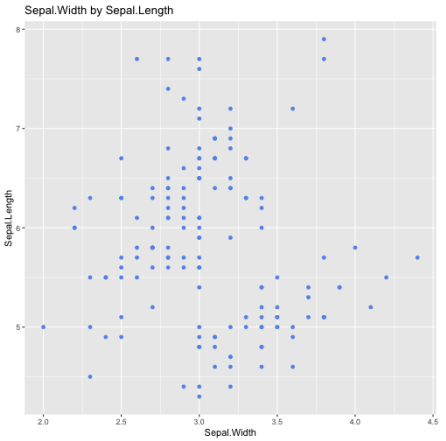
Applied Example 3

```
modelDF
```

```
## # A tibble: 6 x 6
##   term        estimate std.error statistic   p.value predictor
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl> <chr>
## 1 (Intercept)    0      3.79e-17   0.      1.00e+  0 Sepal.Width
## 2 iris[, i]      1      6.43e-18   1.56e17 0.        Sepal.Width
## 3 (Intercept)    6.53   4.79e- 1   1.36e 1 6.47e- 28 Petal.Length
## 4 iris[, i]     -0.223  1.55e- 1  -1.44e 0 1.52e-  1 Petal.Length
## 5 (Intercept)    4.31   7.84e- 2   5.49e 1 2.43e-100 Petal.Width
## 6 iris[, i]      0.409  1.89e- 2   2.16e 1 1.04e- 47 Petal.Width
```

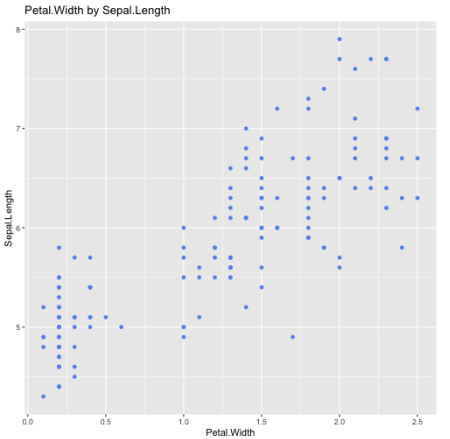
Applied Example 3

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




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

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



Bootstrapping

> to get oneself out of a situation using existing resources

In statistics...any test or metric that uses random sampling with replacement

- a bootstrapped mean
- bootstrapped confidence intervals
- bootstrap anything your heart desires!

Bootstrapped Mean

Let's say we have a sample of 100 participants that complete an IQ test. IQ tests typically have a mean of 100 and a standard deviation of 15. We want to get the mean IQ of our sample of 100 participants, but we want it to be a **robust** mean -- that is, we want to be pretty darn confident in our mean. What should we do?

We can bootstrap our mean! And we can do that using a `for loop`:

- *for each iteration of ## of iterations...*
  - Randomly choose (sample) 50% of your participants
  - Now calculate the mean of just this 50%
  - On the next iteration (e.g., `i + 1`) choose a new set of 50% of participants
  - Recalculate the mean of just this new 50%
  - etc.
- After we run all of our iterations, then we will get the **mean of means**

Bootstrapping

This whole process sounds like something you are already familiar with -- **sampling distributions**

You can think of bootstrapping as building up your sampling distribution for whatever statistic you want. But instead of repeating your experiment 1000x, you're using a random sample of your current experiment.

Bootstrapped Means Example

```
iqs
```

```
##   [1]  85 101 191 117 119 199 136  50  79  91 174 131 133 119  85 202 138
##  [19] 148  89  60 102  62  77  82  29 155 121  56 176 134  98 158 157 154 1
##  [37] 141 110  98  94  78 103  50 221 173  57  93  90 152 109 126 112 111 1
##  [55] 102 189  36 142 119 124 132  88  96  62  59 128 135 116 159 216  88
##  [73] 163  78  79 164  99  52 122 106 113 132  94 145 102 130 168 135  97 1
##  [91] 163 140 125  82 181  83 222 190 101  62
```

Bootstrapped Means Example

Things you need to decide:

- How many iterations? This is really a personal decision. The more iterations you have, the longer your code will take to run. 1000 & 5000 are common, but you could pick 3756 iterations and that would be fine, too.
- How much of the total sample do you want to comprise your sample? 50% of your participants? 75%? 80%? I typically see things within the 50-75%.
- For now, lets go with 100x iterations (to minimize how long it takes for the code to run)
- For now, lets do 50%

Bootstrapped Means Example

```
bsMeans <- NULL

for (i in 1:100) {
  subsample <- sample(x = iqs,
                      size = 50)

  bsMeans[i] <- mean(subsample)
}
```

Bootstrapped Means Example

Let's look at our vector of means

```
bsMeans
```

```
##   [1] 117.06 115.76 122.66 118.02 114.18 111.00 116.16 107.64 119.70 119.12
##  [11] 111.98 116.10 113.56 115.82 116.18 124.68 119.64 111.04 114.98 116.14
##  [21] 123.44 111.96 126.50 121.18 116.62 120.36 119.00 117.02 113.68 117.54
##  [31] 110.44 120.18 118.82 120.00 114.66 111.66 113.24 108.20 121.62 104.24
##  [41] 122.48 122.54 124.28 127.58 110.68 110.62 122.46 105.44 115.46 120.18
##  [51] 114.60 124.78 112.78 117.22 105.92 112.28 124.66 115.20 119.02 113.14
##  [61] 118.86 118.96 112.08 113.48 118.04 116.92 114.26 106.74 120.54 117.58
##  [71] 110.38 122.30 115.20 122.26 112.74 121.52 118.64 117.10 118.76 122.96
##  [81] 118.86 116.48 112.76 114.88 120.54 117.34 124.02 118.18 114.28 117.14
##  [91] 113.14 112.96 113.82 110.18 123.36 110.12 112.84 116.04 118.46 111.90
```

Bootstrapped Means Example

Now, get the mean of our vector of means...this is our **bootstrapped mean**

```
mean(bsMeans)
```

```
## [1] 116.4972
```

When I made these fake IQ scores, I set the "true" population mean to be 113.

Our mean of the original sample was `mean(iqs)` = **117.56**

Our bootstrapped mean is `mean(bsMeans)` = **116.4972**

The bootstrapped mean is closer to the true population mean -- it's more robust, and therefore we trust it more.

What can you bootstrap?

Any statistic!

- central tendencies (means, medians, modes)
- dispersions (variances, standard deviations)
- other estimates (confidence intervals, reliability coefficients, correlations, etc.)

Models!

- take a subsample and run a model
- then take the mean of all the coefficients (like regression coefficients, t-values, etc.)

what can't you bootstrap?

Final Thoughts

- Code is supposed to make your life EASIER. Use `for loops` to your advantage! That means if you find yourself copying/pasting the same thing a billion times with only minor changes, there's likely a much simpler way of doing everything you need all in one go.
  - check out nested `if/else` statements
  - for `R` specific functions, check out the `lapply()` function and the `purr` package
- The biggest piece of advice I have is to think carefully about what you want the end result to look like and then work backwards. Don't just start doing stuff to your dataset because you think that's what your supposed to do. Think "in order to make this plot, I need my data in this particular format, what do I need to do to get there?"
- Bootstrapping is just a `for loop`. You can bootstrap anything your heart desires.
  - when you test out a for loop, use really small iterations!