Lesson 7-Key!

YOUR NAME

2023-10-25

Learning goals

- Using R's built-in loop functions: `apply`, `lapply` and contrasting them with `for` loops
- Conditional statements
- Investigate sampling error
- Visualize confidence intervals.
- Adding error bars to plots

Apply and lapply

Last week you learned about loops, which are standard in all programming languages. However, R has some built in functions that perform similar operations and are a bit more simple to use. They belong to the "apply" family. Here are some differences between apply and lapply Differences between lapply and apply: Data Types:

- `lapply` works on lists and vectors.
- `apply` works on matrices, arrays, and data.frames.

Return Type:

- `lapply` always returns a list.
- `apply` can return a vector, matrix, or array, depending on the function being applied and the input's dimensions.

Use Case:

- Use `lapply` when you have a list or vector and you want to apply a function to each of its elements.
- Use `apply` when you have a matrix or array or data.frame and you want to apply a function across its rows or columns.

There are many other functions in the apply family but we are just focusing on these two. I recommend typing ? apply and ?lapply into the console before starting. The second argument, margin, can be tricky.

1. Let's start super basic. Say you have a `matrix mat <- matrix(1:12, nrow=3)`. You want to find the sum of each column. paste the matrix first then use apply to find the sum. Name it something then print the result

2. Use lapply to find how many characters are in each name. Save as an object and then print. To count characters, you can use the `nchar` function. Here are your names, get counting `names_list <- list("Alessandra", "Penelope", "Meigui", "Tatiana", "Keisuke", "Nadia", "Amy")`

What is different about this output? Correct, it's a list!

3. Comparing `apply` and `for` loops You're a teacher and you've just given your students a test. The test scores are stored in a matrix where each row represents a student and each column represents a test question. Calculate the average score for each student using both apply and a for loop. Your matrix is called scores and has all the information you need.

```
scores <- matrix(c(85, 90, 78, 88,
                   92, 80, 95, 90,
                   60, 55, 78, 84),
                 nrow=3, byrow=TRUE)
colnames(scores) <- c("Math", "English", "Biology", "History")
rownames(scores) <- c("Alice", "Bob", "Charlie")
print(scores)
```

```
##         Math English Biology History
## Alice     85      90      78      88
## Bob       92      80      95      90
## Charlie   60      55      78      84
```

First use the new functions you learned to find the average test score for each student.

"'{r. ex3.1 } #use apply

```
Now use a for loop. Remember to create blank variables first to store your information. You can call th
```

```
```r
#first count the number of scores in scores and save it as num_students

#create blank variable called averages. Make it numeric with as many elements as num_students

#create your for loop to iteratively calculate the mean score for each student and store it in differen

#name the object averages using the rownames of object scores
```

- Which of these was easier for you? Which student is doing the best?
- Which class do you think is hardest? Lets do the process again to find the average (mean) for each class

First use `apply` and save the results in an object called `class_averages_apply`:

"'{r. ex3.3 } #create object

#print your object
```

Then use a `for` loop and save the results in an onject called `class_averages_loop`:

```r
#first count the number of subjects in the scores matrix and save it as num_subjects:

#create blank variable called class_averages_loop. Make it numeric with as many elements as num_subjects

#create your for loop to iteratively calculate the mean score per subject



#name the object class_averages_loop using the colnames of object scores

#print your object
```

And now you understand the joy of the apply family!

Conditional Statements

Conditional Statements are a key part of any programming language.

1. if Statement:

The `if` statement evaluates a condition, and if that condition is `TRUE`, it executes the code inside the statement. Change `x` to a few different numbers to see how the output changes!

```r
x <- 10

if (x > 5) {
  print("x is greater than 5")
}
```

```
## [1] "x is greater than 5"
```

In the above code, since `x` is indeed greater than 5, the message "x is greater than 5" will be printed.

Change `x` to a few different numbers to see how the output changes!

2. if-else Statement:

There might be cases where you want to execute one set of instructions if the condition is `TRUE` and another set if it's `FALSE`. This is where the `if-else` statement comes in.

```r
x <- 3

if (x > 5) {
  print("x is greater than 5")
```

```
} else {
  print("x is not greater than 5")
}
```

```
## [1] "x is not greater than 5"
```

You can keep adding else statements!

3. if-else if-else Ladder:

For multiple conditions, we can use an `if-else` ladder.

```
x <- 5

if (x > 10) {
  print("x is greater than 10")
} else if (x > 5) {
  print("x is greater than 5 but less than or equal to 10")
} else {
  print("x is less than or equal to 5")
}
```

```
## [1] "x is less than or equal to 5"
```

And so on and so on. This is really useful for making code do complicated things!

Change `x` to a few different numbers to see how the output changes!

4. Write a basic if statement that prints "this number is positive" when that is true. Try some numbers!

Now add an `else if` statement that tells you if the number is negative, and an else that tells you if its 0.

Amazing! You now understand basic conditionals! Let's combine this with loops. Here is an example of a loop that contains an if statement Lets say you have a bunch of numbers and you want to know how many are positive.

```
# a vector of numbers
numbers <- c(-5, 3, 8, -2, 0, 7, -9)

# a counter variable that will change each time the loop cycles
positive_count <- 0
for (num in numbers) {
  if (num > 0) {
    positive_count <- positive_count + 1 #add 1 to positve_count if number if positive.
↪  Otherwise do nothing
  }
}
```

This loop adds one to the count for each positive number.

```
# I gave you a bunch of random numbers.
numbers <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# start your code here
# create a counter variable called sum_even
```

5. Write a loop telling me the sum of all the even numbers.

More work in sampling

Now we are going to return to our example from last class, the human genes data set. We are going to graph more sampling distributions and add error bars to some plots.

6. Last class we compared the means of differently sized samples. Generally, the larger your sample, the closer it will be to the true population mean. The human gene data set is a bit weird though. Because it has a few genes that are huge, a large sample size might be more likely to include one of those giant genes and mess up your mean.

For a real experiment you would just deal with that. For our purposes, you should filter the data set to only include genes smaller than 15000 base pairs. Load in your data and do this. You can keep it's name the same

Note: This issue with the long genes is explained in the key for lab 6.

```
#load the readr library

#read in human_genes.csv into human_genes using read_csv.


#filter dataset to only contain genes of size <=15000 and save it as human_genes2
```

Run the loop you wrote last week and investigate the means now that we have filtered the data. This was the loop where you took 100 samples of three different sizes: 10, 100, 1000, with replacement, and calculated their means. Do this again here with this filtered dataset and look at sample sizes of 10, 50, and 500. What do you see?

```
#calculate the mean gene length for human_genes


#create empty numeric vectors with 1000 positions for the means. Call them
↪  mean_values_10, mean_values_50, mean_values_500



#start your loop here. Take samples with replacement, sizes 10, 50, 500 and take the
↪  median of those samples


#take the mean of means (the grand mean) of these three vectors and save them to a new
↪  vector called grand_means
```

```
#compare these three means with the real mean of the filtered dataset
```

7. What's up with these samples?

1. Start by finding the standard error of the mean, which is your the standard deviation of the sampling distribution. Your three sampling distributions were created above and are called mean_values_10, mean_values_50, and mean_values 500.

```
#call them sem_10, sem_50, sem_500
```

Congratulations! You just found the standard error by taking the standard deviation of the sampling distributions!

Now, think about which one of them should have the lowest SEM? Now check if that's true. Does it match your expectations?

2. Now lets find the 95% confidence intervals. Type ?quantile into your console to learn how to use this neat function. Hint: We want 95% of the data to be within the intervals, so what should the quantiles be?

```
#call them CI_10, CI_500, CI_500
```

Now we have these confidence intervals, and you didn't need a crazy formula or assumptions about the mean being normally distributed! You simply took samples with replacement and calculated your confidence intervals empircally - aka, you used bootstrapping!

8. Lets add the CIs to a graph! Lets first make a data frame of all the information we will need for the chart. You need the following columns: SampleSize, MeanValues, UpperCI and LowerCI. You can use the ones you've calculated above.

```
#create a data.frame called df containing the columns listed above.
```

Amazing! Now make a strip chart using this data frame. Don't worry about CI yet. Just make a simple plot using geom_point() for each of the means, grouped by sample size. Add labels though. See the expected output document for inspiration.

```
#first, make sure SampleSize is treated as a factor


#now create your plot code here and save it as p1


#print p1, aka, your plot
```

Gorgeous. Now add your error bars. Use `geom_errorbar` to see the difference in those sample sizes. Add geom_errorbar after geom_point(). Inside aes(), use ymin and ymax with the lower and upper CIs, respectively.

```
#create your plot code here and save it as p2


#print p2, aka, your plot
```

Those are super different! As you can see, larger sample sizes create narrower confidence intervals. This is **CRUCIAL** for your own data analysis.

9. Another graph Above we plotted only the grand means of the sampling distributions. But what do these sample distributions look like compared to a normal distribution? R has a built in plot (gasp it's not ggplot2) called a density plot that will show us. First, use the density function to find the density of each of your sample means. Name them something you'll remember. PS: ggplot also has an option for density plots.

Now use the R function "plot" to plot each of these.

You see as the n increases they get more normal!

The end!