TA Session 13: Iteration and Loops

Harris Coding Camp

Summer 2022

General Guidelines

You may encounter some functions we did not cover in the lectures. This will give you some practice on how to use a new function for the first time. You can try following steps:

1. Start by typing `?new_function` in your Console to open up the help page
2. Read the help page of this `new_function`. The description might be a bit technical for now. That's OK. Pay attention to the Usage and Arguments, especially the argument `x` or `x,y` (when two arguments are required)
3. At the bottom of the help page, there are a few examples. Run the first few lines to see how it works
4. Apply it in your questions

It is highly likely that you will encounter error messages while doing this exercise. Here are a few steps that might help get you through it:

1. Locate which line is causing this error first
2. Check if you have a typo in the code. Sometimes your group members can spot a typo faster than you.
3. If you enter the code without any typo, try googling the error message. Scroll through the top few links see if any of them helps
4. Try working on the next few questions while waiting for help by TAs

Warm up

Recall, for-loops are an iterator that help us repeat tasks while changing inputs. The most common structure for your code will look like the following code. Complete and run the code.

```
# what are you iterating over? The vector from -10:10
items_to_iterate_over <- c(-10:10)

# pre-allocate the results
results <- ...

# write the iteration statement:
# we'll use indices so we can store the output easily
for (i in ...) {

  # we capture the median of three random numbers from normal distributions
  # with various means (from -10 to 10)
  ...[[i]] <- median(rnorm(n = 3, mean = items_to_iterate_over[[i]]))

}
```

I. Writing for-loops

1. Write a for-loop that iterates over the indices of `x` and prints the `i`th value of `x`.

   ```
   x <- c(5, 10, 15, 20, 25000)

   # replace the ... with the relevant code

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

2. Write a for-loop that prints the numbers `5, 10, 15, 20, 25000` by iterating over the vector directly.

3. Write a for-loop that simplifies the following code so that you don't repeat yourself! Don't worry about storing the output yet. Use `print()` so that you can see the output. What happens if you don't use `print()`?

   ```
   sd(rnorm(5))
   sd(rnorm(10))
   sd(rnorm(15))
   sd(rnorm(20))
   sd(rnorm(25000))
   ```

   a. adjust your for-loop to see how the `sd` changes when you use `rnorm(n, mean = 4)`
   b. adjust your for-loop to see how the `sd` changes when you use `rnorm(n, sd = 4)`

4. Now store the results of your for-loop above in a vector. Pre-allocate a vector of length 5 to capture the standard deviations.

II. Vectorization vs for loops

Recall, vectorized functions operate on a vector item by item. It's like looping over the vector!

The following for-loop is better written vectorized.

Compare the loop version

```
names <- c("Alysha", "Fanmei", "Paola")

out <- character(length(names))

for (i in seq_along(names)) {
  out[[i]] <- paste0("Welcome ", names[[i]])
}
```

to the vectorized version

```
names <- c("Alysha", "Fanmei", "Paola")
out <- paste0("Welcome ", names)
```

The vectorized code is preferred because it is easier to write and read, and is possibly more efficient.

1. Rewrite this for-loop as vectorized code:

```
radii <- c(0:10)
area <- double(length(radii))

for (i in seq_along(radii)) {

  area[[i]] <- pi * radii[[i]] ^ 2

}
```

2. Rewrite this for-loop as vectorized code:

```
radii <- c(-1:10)
area <- double(length(radii))

for (i in seq_along(radii)) {

  if (radii[[i]] < 0) {
    area[[i]] <- NaN
  } else {
    area[[i]] <- pi * radii[[i]] ^ 2
  }

}
```

3. Rewrite this for loop as vectorized code:

```
set.seed(5)
visitors <- rpois(20, lambda = 1)
total_visitors <- 0

for (v in visitors) {
  total_visitors <- total_visitors + v
}
```

4. a. Vectorize this code.

```
big <- c()
# slow loop
tictoc::tic()
for (x in 2:10000) {
  big <- c(big, x * 10)
}
tictoc::toc()
```

```
## 0.153 sec elapsed
```

Compare the run times of the loop and vectorized code. It should be noticeable without using any timers. To time the code you can install the `tictoc` package[1] There are a lot of ways to measure code efficiency. `tictoc` wins because it is easy to remember and easy to use.

4. b. The loop is slow because we didn't pre-allocate space. Re-write the loop with preallocated integer vector.

III. Simulating the Law of Large Numbers

The Law of Large Numbers says that as sample sizes increase, the mean of the sample will approach the true mean of the distribution. More details will be covered in Stats 1, but now we are going to simulate this phenomenon!

We'll start by making a vector of sample sizes from 1 to 50, to represent increasing sample sizes.

1. Create a vector called `sample_sizes` that is made up of the numbers 1 through 50.

We'll make an empty tibble to store the results of the for loop:

```
estimates <- tibble(n = integer(), sample_mean = double())
```

2. Write a loop over the `sample_sizes` you specified above. In the loop, for each sample size you will:

a. Calculate the mean of a sample from the random normal distribution with mean = 0 and sd = 5.
b. Make an intermediate tibble to store the results
c. Append the intermediate tibble to your tibble using bind_rows().

[1] (No relation to the social media company.)

```
set.seed(60637)
for (___ in ___) {
  # Calculate the mean of a random sample from normal distribution with mean = 0 and sd = 5
  ___ <- ___
  # Make a tibble with your estimates
  this_estimate <- tibble(n = ___, sample_mean = ___)
  # Append the new rows to your tibble
  # If not sure how bind_rows works, use ?bind_rows
  ___ <- bind_rows(estimates, ___)
}
```

3. We can use `ggplot` to view the results. Fill in the correct information for the data and x and y variables, so that the `n` column of the `estimates` tibble is plotted on the x-axis, while the `sample_mean` column of the `estimates` tibble is plotted on the y-axis.

```
# your data goes in the first position
___ %>%
  ggplot(aes(x = ___, y = ___)) +
  geom_line()
```

4. As the sample size (`n`) increases, does the sample mean becomes closer to 0, or farther away from 0?

Rewrite the loop code without looking at your previous code and use a wider range of sample sizes. Try several different sample size combinations. What happens when you increase the sample size to 100? 500? 1000? Use the `seq()` function to generate a sensibly spaced sequence. How does this compare to before?

```
set.seed(60637)
sample_sizes <- ___
estimates_larger_n <- ___

for (___ in ___) {
  ___ <- ___
  ___ <- ___
  ___ <- ___
}

___ %>%
  ggplot(___(___ = ___, ___ = ___)) +
  geom_line()
```

5. Looking at your results, you might think a small sample size is sufficient for estimating a mean, but your data had a relatively small standard deviation compared to the mean. Let's run the same simulation as before with different standard deviations.

Do the following:

a. Create a vector called `population_sd` of length 4 with values 1, 5, 10, and 20 (you're welcome to add larger numbers if you wish).
b. Make an empty tibble to store the output. Compared to before, this has an extra column for the changing population standard deviations.
c. Write a loop inside a loop over `population_sd` and then `sample_sizes`.

d. Then, make a `ggplot` graph where the x and y axes are the same, but we facet (aka we create small multiples of individual graphs) on `population_sd`.

```
set.seed(60637)
population_sd <- ___
# use what every you came up with in the previous part
sample_sizes <- ___
estimates_adjust_sd <- ___

for (___ in ___){
  for (___ in ___) {
    ___ <- ___
    ___ <- ___
    ___ <- ___
  }
}

___ %>%
  ggplot(___) +
  geom_line() +
  facet_wrap(~population_sd) +
  theme_minimal()
```

How do these estimates differ as you increase the standard deviation?

6. At this point in your coding career, efficiency is minimally important. However, we discussed that loops in R are much slower if we build our output item by item (or row by row in this case). We do better if we pre-allocate space.

a. Rewrite your loop from part 1 where we start with `estimates <- vector("list", n)`. This creates a list of length `n` "filled" with NULL.

b. During each iteration of your loop, fill one of the spaces in `estimates` with a named vector (which we think of as a row). Use `[[` to index into your list.

c. Finally, when your loop terminates, you'll have `estimates` filled with `n` "rows". `bind_rows(estimates)` will convert the list to a tibble.