TA Session 13: Iteration and Loops (solutions)

Harris Coding Camp

Summer 2022

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)
  ... <- median(rnorm(n = 3, mean = items_to_iterate_over[[i]]))

}
```

SOLUTION:

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

# pre-allocate the results
out <- rep(0, length(items_to_iterate_over))

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

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

}
```

I. Writing for-loops

1. Write a for-loop that prints the numbers `5, 10, 15, 20, 25000`.

SOLUTION:

```
x <- c(5, 10, 15, 20, 250000)

for (number in x){

  print(number)

}
```

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
## [1] 250000
```

2. 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[[...]])
}
```

SOLUTION:

```
x <- c(5, 10, 15, 20, 250000)

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

```
## [1] 5
## [1] 10
## [1] 15
## [1] 20
## [1] 250000
```

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))
```

SOLUTION: if we don't use `print()` the for loop is going to run silently, but won't display any results at the console.

a. adjust your for-loop to see how the `mean` 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)`

SOLUTION:

```
x <- c(5, 10, 15, 20, 250000)

# replace the ... with the relevant code

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

```
## [1] 1.331669
## [1] 1.122355
## [1] 1.271678
## [1] 1.251056
## [1] 1.000207
```

```
for (i in seq_along(x) ){
  n <- x[[i]]
  print(sd(rnorm(n, mean = 4)))
}
```

```
## [1] 1.156682
## [1] 0.7196148
## [1] 0.783999
## [1] 0.9465416
## [1] 0.9992382
```

```
for (i in seq_along(x) ){
  n <- x[[i]]
  print(sd(rnorm(n, sd = 4)))
}
```

```
## [1] 3.26172
## [1] 2.515515
## [1] 4.184381
## [1] 3.560519
## [1] 4.002527
```

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.

```
x <- c(5, 10, 15, 20, 250000)

sd_of_samples <- rep(0, length(x))

for (i in seq_along(x) ){
```

```
  sd_of_samples[[i]] <- sd(rnorm(x[[i]]))

}
```

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

}
```

SOLUTION:

```
radii <- c(0:10)

area <- pi * radii ^ 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
  }

}
```

SOLUTION:

```
radii <- c(-1:10)

area <- ifelse(radii >= 0, pi * radii ^ 2, NaN)
```

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
}

# SOLUTION
total_visitors <- sum(visitors)
```

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

```
#SOLUTIONS

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

```
## 0.124 sec elapsed
```

```
# a

tictoc::tic()
big_vectorized <- 2:10000 * 10
tictoc::toc()
```

```
## 0 sec elapsed
```

```r
# b
# preallocating space.
input <- 2:10000
big_alloc <- integer(length(input))

tictoc::tic()
for (i in seq_along(2:10000)) {
  big_alloc[[i]] <- input[[i]] * 10
}
tictoc::toc()
```

```
## 0.002 sec elapsed
```

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. (Hint: You can use `seq()` or `:` notation).

```
set.seed(60637)
sample_sizes <- 1:50
```

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().

```
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, ___)
}
```

SOLUTION:

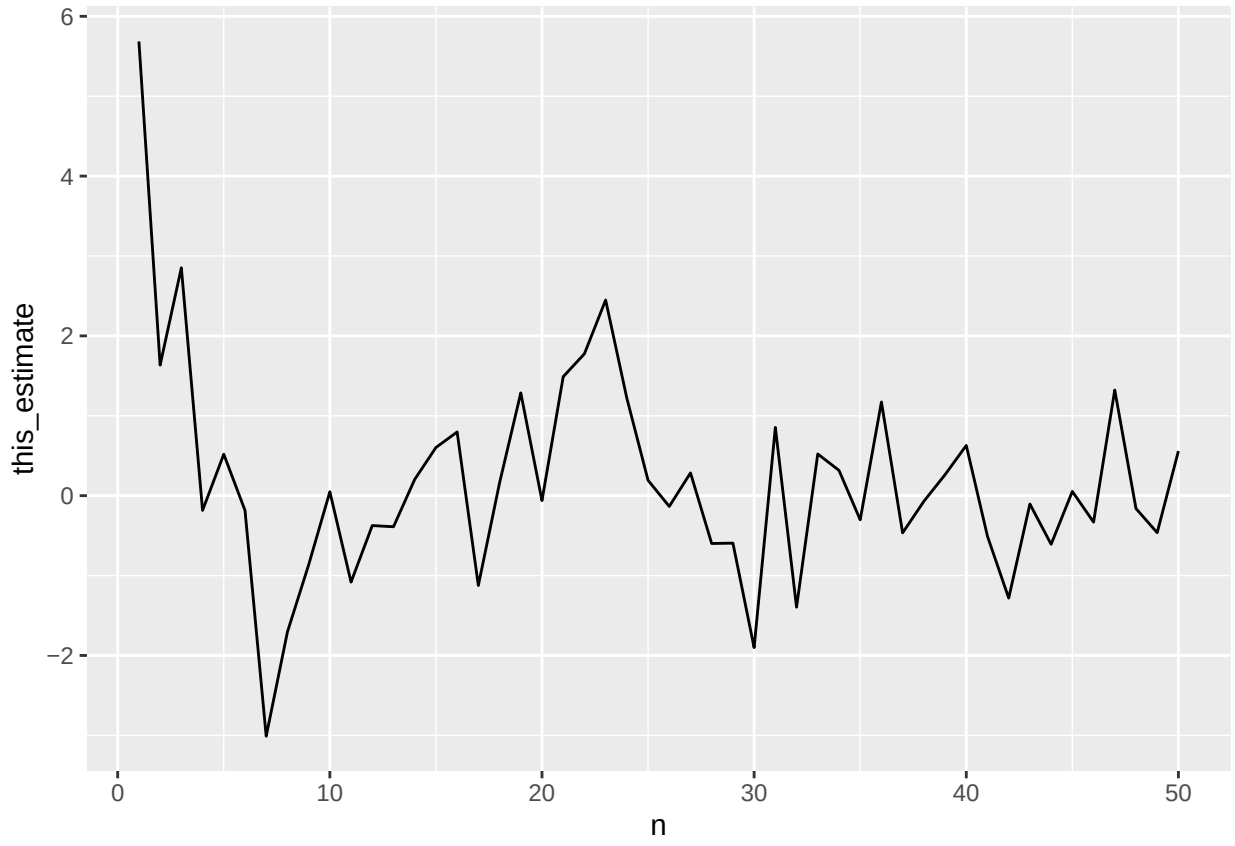
```
set.seed(60637)

for (n in sample_sizes) {

  this_estimate <- mean(rnorm(n, mean = 0, sd = 5))

  estimates <- bind_rows(estimates, c(n = n, this_estimate = this_estimate))
}
```

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()
```

SOLUTION:

```
estimates %>%
  ggplot(aes(x = n, y = this_estimate)) +
  geom_line()
```



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

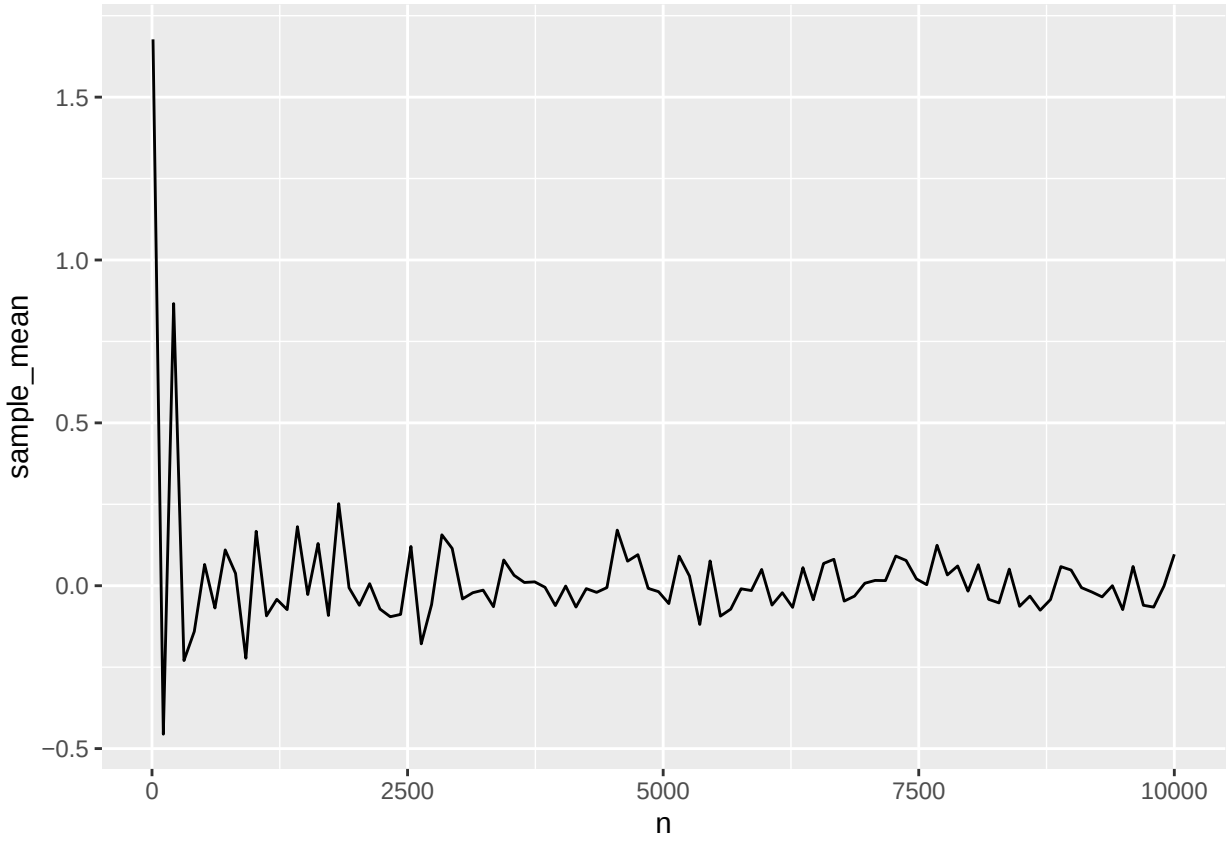
SOLUTION: closer

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()
```

SOLUTION: Just like before, we can see that for small samples, the mean fluctuates quite a bit. However, we can also see that the data becomes more centralized around zero as the **n** increases.

```
set.seed(60637)
sample_sizes <- seq(10, 10000, length.out = 100)
estimates <- tibble(n = integer(), sample_mean = double())

for (n in sample_sizes) {
  sample_mean <- mean(rnorm(n, mean = 0, sd = 5))
  estimates <- bind_rows(estimates, c(n = n, sample_mean = sample_mean))
}

estimates %>%
  ggplot(aes(x = n, y = sample_mean)) +
  geom_line()
```


1.5
1.0
0.5
0.0
−0.5
sample_mean
0
2500
5000
7500
10000
n

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.

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`.

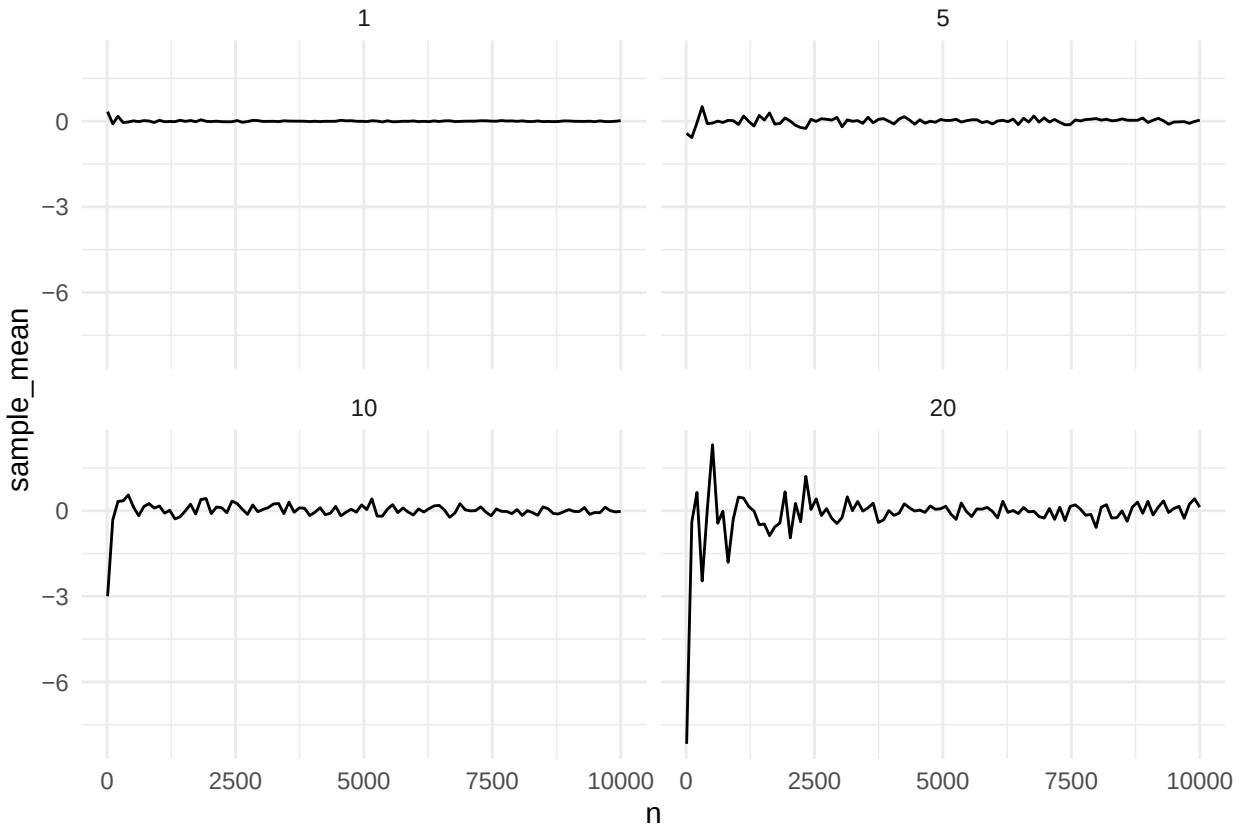
How do these estimates differ as you increase the standard deviation?

```
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()
```

SOLUTION: we can see that as the standard deviation increases, the spread of our mean calculations also increases—the wider the distribution, the larger the draw we require for the sample mean to converge towards the true mean. This makes sense, because as the population standard deviation of the distribution we draw from increases, the likelier it is we draw extreme numbers.

```
set.seed(60637)

population_sd <- c(1, 5, 10, 20)
estimates <- tibble(n = integer(), sample_mean = double(), population_sd = integer())

for (sd in population_sd){

  for (n in sample_sizes) {

    sample_mean <- mean(rnorm(n, mean = 0, sd = sd))

    estimates <- bind_rows(estimates, c(n = n, sample_mean = sample_mean, population_sd = sd))
```

```
  }
}

estimates %>%
  ggplot(aes(x = n, y = sample_mean)) +
  geom_line() +
  facet_wrap(~ population_sd) +
  theme_minimal()
```



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.

SOLUTION:

```
set.seed(60637)
estimates <- vector("list", 10)
n <- 1:10

for (i in n) {

  this_estimate <- mean(rnorm(n[i], mean = 0, sd = 5))

  estimates[[i]] <- bind_rows(estimates, c(n = n[i], this_estimate = this_estimate))

}
```