TA Session 12: Writing Functions

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

Writing functions requires a new level of debugging

- You want to be confident that the code works the way you think it does. So try to test the code with various input types. Professionals sometimes write tests for the code before they even write the code!
- As you test, add temporary `print()` calls within your functions to figure out if and where the code is doing something unintended.
- Break up your problem into bite sized pieces, try the components separately, and then put them together. You'll see some of this philosophy in this lab.
  - When individual functions are less complicated, bugs should be easier to find.
  - The "bite-sized pieces" should encapsulate some logical set of actions that make sense together. For example, tidyverse takes `subset()` and encapsulates row subsetting with `filter()` and column subsetting with `select()`. If they went any further,[1] the functions would be harder to use.
  - So there's a balancing act between making the functions too complicated and not capturing a cohesive logical step.

This might feel more complicated than previous ones, but with some patience you'll get there!

[1] e.g. a function for evaluating the filter call and another for doing the row subsetting

Warm up

1. Let's write a function that takes n-th power of some numeric input.

Recall the form of a function looks like:

```
name <- function(arg1, arg2) {
  body
}
```

Let's start writing:

a. What's a good name for the function? Replace "name" with your chosen name.
b. The function requires a numeric input and a power. What are good names for these two arguments? Replace the args with your argument names.
c. Finally, write the body.
d. Test that the code works.

2. If you write a function without an explicit `return()` call, what does R return?

I. Writing quadratic functions

Recall a function has the following form:

```
name <- function(args) {
  # body
  do something (probably with args)
}
```

1. Write a function called `calc_quadratic` that takes an input `x` and calculates $f(x) = x^2 + 2x + 1$. For example:

```
calc_quadratic(5)
```

```
## [1] 36
```

a. What are the arguments to your function? What is the body of the function?
b. This function is vectorized! (Since binary operators are vectorized). Show this is true by running `calc_quadratic` with an input vector that is -10 to 10.

2. You realize you want to be able to work with any quadratic. Update your functions so that it can work with any quadratic in standard form $f(x) = ax^2 + bx + c$.

- Your new function will take arguments `x`, `a`, `b` and `c`.
- Set the default arguments to `a = 1`, `b = 2` and `c = 1`

3. Write a function called `solve_quadratic` that takes arguments `a`, `b` and `c` and provides the two roots using the quadratic formula.

In our outline, we suggest you: - Calculate the determinant ($\sqrt{b^2 - 4ac}$) and store as an intermediate value. - Return two values by putting them in a vector. If you stored the roots as `root_1` and `root_2`, then the final line of code in the function should be `c(root_1, root_2)` or, if you prefer, `return(c(root_1, root_2))`.

```
# fill in the ... with appropriate code
solve_quadratic <- function(...){

  determinant <- ...
  root_1 <- ...
  root_2 <- ...

  c(root_1, root_2)

  }
```

The code should work as follows:

```
solve_quadratic(a = -4, b = 0, c = 1)
```

```
## [1] -0.5  0.5
```

4. We "normalize" a variable by subtracting the mean and dividing by the standard deviation $\frac{x-\mu}{\sigma}$. Write a function called `normalize` that takes a vector as input and normalizes it.

You should get the following output.

```
normalize(1:5)
```

```
## [1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
```

a. What output do you get when the input vector is `0:4`? How about `-100:-96`? Why?
b. What happens when your input vector is c(1,2,3,4,5, NA)? Rewrite the function so the result is:[2]

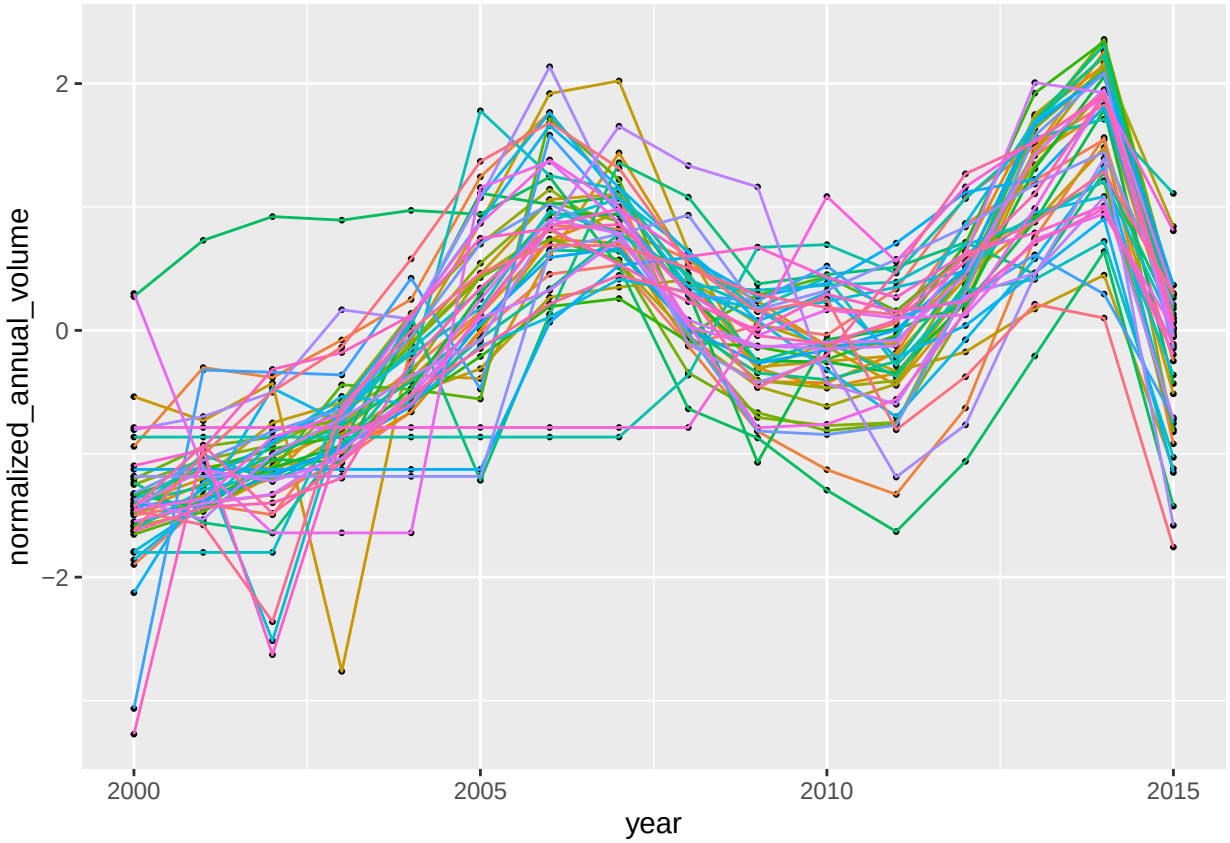
```
## [1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111         NA
```

5. The `txhousing` data set is comes with `ggplot`. Use your normalize function in `mutate` to create `normalized_annual_volume` to make the following graph.

```
library(tidyverse)

# replace the ... with the appropriate code.
txhousing %>%
  group_by(year, city) %>%
  summarize(annual_volume = sum(volume, na.rm = TRUE)) %>%
  group_by(city) %>%
  mutate(...) %>%
  ggplot(aes(x = ..., y = ...)) +
  geom_...(size = .5) +
  geom_...(aes(color = city),
           show.legend = FALSE)
```

[2] Hint: take advantage of `mean` and `sd` NA handling.


normalized_annual_volume
2
0
−2
2000
2005
2010
2015
year

II. Simulating Data with Monte Carlo Simulations

This is a preview of material you'll see in Stats I where you will be asked to investigate statistical concepts using Monte Carlo simulations. We'll try not to get too technical in the main body of the lab. There are some "technical notes" which you can ignore!

In a Monte Carlo simulation, you repeatedly:

1. Generate random samples of data using a known process.
2. Make calculations based on the random sample.
3. Aggregate the results.

Functions and loops help us do these repetitious acts efficiently, without repeatedly writing similar code or copying and pasting.

Today's problem: Let us investigate how good the random number generator in R is.[3] We hypothesize that `rnorm(n, mean = true_mean)` provides random sample of size `n` from the normal distribution with mean = `true_mean` and standard deviation = 1.

The lesson is organized as follows.

1. Do a single simulation.
2. Take the logic of the simulation, encapsulate it in functions and then run 1000s of simulations!

A single simulation

Recall our hypothesis is that `rnorm()` faithfully gives us random numbers from the normal distribution. If we test this with a single random draw, we might be misled. For example, let's draw 30 numbers from a normal distribution with true mean of 0.5 and see if the observed mean appears statistically different from the true mean.

Q1. Complete and run the code. What is the mean of this random sample? Hint: The answer is provided below! Double-check if you have the same one.

```
# Setting a seed ensures replicability
set.seed(4)

# we set our parameters
true_mean <- .5
N <- 30

# We simulate and observe outcomes
simulated_data <- rnorm(...)    # the standard deviation is 1 by default
... <- which_function(simulated_data)
...
```

```
## [1] 0.9871873
```

Q2. Wow! The observed mean is almost twice what we expected given `true_mean`! Let's calculate a z-score to put that in perspective. (Focus on the formulas, you'll learn the intuition in stats class.[4]

[3] Contrived? yes. This is similar to the problem where you have a coin and want to tell if it's fair. On the other hand, there's a deeper statistical idea here related to the difference between t-tests and z-tests!

[4] **Techincal note**: If you can't help but be curious about the stats, what we are doing is logically equivalent to a z-test.

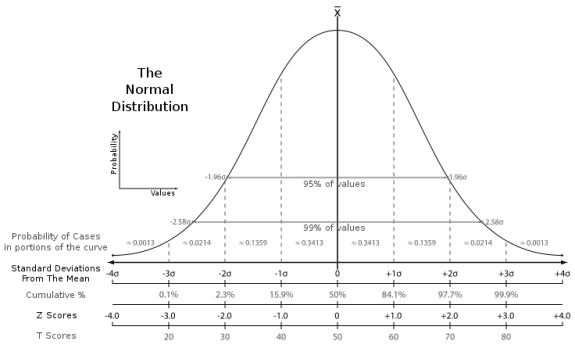


Figure 1: The normal distribution

A z-score is calculated $\frac{\bar{X}-\mu}{\frac{s_n}{\sqrt{N}}}$ where $\bar{X}$ is the sample mean, μ is the true mean, s_n is the sample standard deviation and N is the number of observations. Complete and run the code. What is the z-score of this random sample?

```
obs_sd <- sd(simulated_data)
zscore <- ...
zscore
```

```
## [1] 3.303849
```

Q3. We expect the observed mean of this simulated data will be within 1.96 standard deviations of μ 95 out of 100 times.[5] This observation is 3.3 standard deviations from μ. The probability of that happening by chance is very small. To be more formal about this probability, we can calculate a "p-value"[6]. Plug in the z-score and run the code below. What do you get?

```
(1 - pnorm(abs(zscore))) * 2
```

```
## [1] 0.000953672
```

This says that the probability of getting this draw by chance is less than 0.1 percent or 1 in 1000. [7]

That outcome seems surprising, but we could also just have made an unusual draw. In this TA session, we want to see how often we get such extreme results. We will repeat the steps above 1000 times each, but first we'll write functions that will make this process smooth!

Writing Helper Functions to Make Our Monte Carlo Simulation

[5] Technical note: This is only approximately correct as N gets large.

[6] For more details, see https://en.wikipedia.org/wiki/P-value.

[7] **Technical note:** Again, you'll learn why this is the formula in stats 1! For the curious, `pnorm()` takes a z-score as the input and returns the probability of observing a value less than or equal to the z-score. So if X is distributed standard normal, `pnorm(z)` $= P(X \leq z)$. This is an integral that measures the area under the probability density function. To see this in action, you can checkout this link

```r
# You might need some tidy functions later.
library(tidyverse)
```

We want to develop functions that automate repeated steps in our Monte Carlo. In that way, we can define a few important parameters and run the entire process without rewriting or copying and pasting code over and over again.

As you saw in the motivating example, we must do the following 1000 times or `B` times if we parameterize the number of iterations with `B`:

1. Simulate data and calculate sample statistics.
2. Determine z-scores.
3. Test whether the z-score is outside the threshold.
4. Measure to what extent our simulations match the theory.

To proceed, we'll write the steps into their own functions, then call them in the right order in the function `do_monte_carlo()`. We are breaking a complicated process into smaller chunks and tackling them one by one!

Q4. Before following our road map, think about how you would set up functions to automate this process. What would the inputs and outputs be of each step/function?

Now check out `do_monte_carlo` below. It's our road map. Your processes will be different from ours, but that doesn't mean ours is better.

```
do_monte_carlo <- function(N, true_mean, B = 1000, alpha = .05){
  # step 1: Simulate B random samples and calculate sample statistics
  sample_statistics <- make_mc_sample(N, true_mean, B)
  # step 2: Determine z-scores
  z_scores <- get_zscores(sample_statistics$mean, true_mean,
                          sample_statistics$sd, N)
  # step 3: Test whether the z-scores are outside the threshold.
  significance <- test_significance(z_scores, alpha)
  # step 4: Measure to what extent our simulations match the theory.
  # (We expect a number close to alpha)
  mean(significance)
}
```

Check if you realize how each step works. It takes a sample-size `N`, a `true_mean`, number of iterations `B` (1000 by default) and a significance level `alpha` (.05 by default). It returns the proportion of observed means that are significantly different from the `true_mean` with 95 percent confidence level[8]

Determine z-scores

We'll start with step 2 determine z-scores. Recall the formula for a z-score is $\frac{\bar{X}-\mu}{\frac{s_n}{\sqrt{N}}}$.

Q5. Write a function called `get_zscores` that takes the observed means and standard deviations, the true mean and the sample size as inputs and returns a z-score as an output. Name the arguments `obs_mean`, `true_mean`, `obs_sd`, and `N`.

If your functions works, it should get the same results as we do for `test`.

```
test <- get_zscores(obs_mean = 4.4, true_mean = 4.3, obs_sd = 0.25, N = 100)
test
```

```
## [1] 4
```

Q6. The function you wrote should also work on vectors. Run the following code which takes estimates of the mean and standard deviation from 5 random draws and returns their associated z-scores:

```
made_up_means <- c(4.4, 4.1, 4.2, 4.4, 4.2)
made_up_sd <- c(.25, .5, .4, 1, .4)
made_up_zscores <- get_zscores(obs_mean = made_up_means,
                               true_mean = 4.3,
                               obs_sd = made_up_sd,
                               N = 100)
made_up_zscores
```

We next ask: Which observation from `made_up_zscores` is **not** statistically different from 4.3 with 95 percent confidence? In other words, which observed mean and standard deviation return |z-score| < 1.96?

[8] technically, `1 - alpha` percent confidence level

Check for Significance

Now we write code for step 3: Test whether the z-scores are outside the threshold.

The threshold depends on `alpha` and the formula is `abs(qnorm(alpha/2))`.[9]

Q7. For example, for a two-tailed z-test at the 95% confidence level, the cutoff is set at 1.96. Verify this using the formula above.[10]

Q8. Write a function `test_significance()` that takes `zscores` and a given `alpha` and determines if there is a significant difference at the given level.

Run the following code, and check that your code matches the expected output:

```
test_significance(zscores = 2, alpha = 0.05)
```

Should return TRUE. $|2| \geq 1.96$ And:

```
test_significance(zscores = c(1.9, -0.3, -3), alpha = 0.05)
```

Should return FALSE, FALSE, and TRUE.

Building `make_mc_sample()`

Now we do step 1: simulate B random samples and calculate sample statistics.

Our goal is `make_mc_sample(N, true_mean, B)` a function that produces sample statistics from `B` random samples from the normal distribution with mean `true_mean` of size `N`. When you think of doing something `B` times it suggest we need a loop. Let's start with the body of the loop. And because we're in a lesson about functions, let's write a function.

Q9. Write a function called `calc_mean_and_sd_from_sample()` that

a. Generates a random sample with `rnorm()` of size `N` centered around `true_mean`;
b. Calculate the `mean()` and `sd()` of the random sample, and return the mean and sd in a tibble with column names `mean` and `sd`.[11]

Idea: To return two values from a function, we need to put those values into a data structure like a vector or tibble.

Here's a test! Verify your function works. Remember, what guarantees that you get the same numbers from a random number generator as we did is that we're setting a seed to 5.[12]

```
set.seed(5)
calc_mean_and_sd_from_sample(N = 30, true_mean = 0.5)
```

```
## # A tibble: 1 x 2
##    mean    sd
##   <dbl> <dbl>
## 1 0.511 0.992
```

[9] **Technical Note** `qnorm()` takes a probability level and return the cutoff, e.g. `qnorm(0.975)` returns 1.96, the critical value associated with 95% confidence or $\alpha = 0.05$. Why is 0.975 used to get the cutoff associated with 95% confidence? It's a two-tailed test, so we check if our z-score is from below the 2.5th percentile or above the 97.5th percentile. We will assume we always do a two-tailed test.

[10] Hint: `alpha = .05`

[11] Hint: Here's the outline of the code you need `tibble(mean = ___, sd = ___)`. As always, test out new code ideas in the console to build confidence.

[12] This code was tested in R 3.6, 4.0 and 4.1.

Now, this function only does what we need **once**, while we'll need it to do it `B` times. This is an appropriate time for a loop!

Q10. We've produced the function for you in two ways; one is a loop and the other uses a `map` function which is something akin to a vectorized way of doing repeated actions. We can get `B` samples

```
make_mc_sample <- function(N, true_mean, B) {
  # pre-allocate output
  out <- vector("list", B)

  for (i in 1:B) {
  # you fill in the code here
    out[[i]] <- calc_mean_and_sd_from_sample(N, true_mean)
  }
  # leave the following code unchanged
  bind_rows(out)
}

make_mc_sample_ <- function(N, true_mean, B){
  map_dfr(1:B, ~ calc_mean_and_sd_from_sample(N, true_mean))
}
```

Here's a test. Attempt different inputs. What happens if `N` is 0, 1, 2, 1000?

```
set.seed(24601)
make_mc_sample(N = 30, true_mean = 100, B = 3)
```

```
## # A tibble: 3 x 2
##    mean    sd
##   <dbl> <dbl>
## 1  99.8 0.920
## 2  99.8 0.952
## 3 100.  1.04
```

Functions, Assemble

Now you have all the helper functions that are critical for our simulation. We want to simulate 1000 sets of 30 data points drawn from a normal distribution with true mean 0.5 and then see how often our random sample mean is significantly different from the true mean at a significance level of 0.05. If everything is working as expected, we should see about 5% of the random means to be statistically different.

```
do_monte_carlo <- function(N, true_mean, B = 1000, alpha = .05){
  # step 1: Simulate B random samples and calculate sample statistics
  sample_statistics <- make_mc_sample(N, true_mean, B)
  # step 2: Determine z-scores
  z_scores <- get_zscores(sample_statistics$mean, true_mean, sample_statistics$sd, N)
  # step 3: Test whether the z-scores are outside the threshold.
  significance <- test_significance(z_scores, alpha)
  # step 4: Measure to what extent our simulations match the theory. (We expect a number close to alpha
  mean(significance)
}
```

Q11. Test out your function with `N` equals 30 and `true_mean` equals 0.5. The resulting number should be close to .05 (`alpha`).[13] (The code will take a few seconds to run.)

Q12. Try again with a different `alpha` and verify that `do_monte_carlo` returns a number in the ball park of `alpha`.

Challenge (optional):

Technical Note In the technical notes, we hinted again and again that our Monte Carlo experiment should only real "work" if `N` is "very large" (or "approaches infinity"). When `N` is not very large, the distribution of our z-scores is not a "normal distribution" but rather they're distributed according to "Student's t-distribution"[14].

In this challenge, you will update your Monte Carlo experiment to use t-tests instead of z-tests.

Q13. Rewrite one of your functions to accommodate the `t`-distribution. To do this you have to replace `qnorm()` with `qt()` with `N - 1` degrees of freedom.[15]

Q14. Run the Monte Carlo experiment several times with both versions of the code to verify that your result is closer to `alpha` on average.

[13] For technical reasons discussed in the challenge, we expect the output of `do_monte_carlo` to be slightly larger than `alpha` on average.

[14] we might call them t-statistics instead of z-scores!

[15] We still use `rnorm()` because our data generating process assumes independent observations from the normal distribution!