Lab Session 2: Vectors and Data Types

Solutions

9/11/2020

We expect you to review the `class 2` material, here prior to lab. If you find yourself in lab without R installed, try using RStudio cloud: https://rstudio.cloud/.[1]

1 Warm-up: Vector creation.

1. Create a new R script and add code to load the tidyverse.

```
library(tidyverse)
```

2. In the lecture, we covered `c()`, `:`, `rep()`, `seq()`, `rnorm()`, `runif()` among other ways to create vectors. Use each of these functions once as you create the vectors required below.

   a. Create an integer vector from seven to seventy.

```
c(7:70)
```

```
##  [1]  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
## [26] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
## [51] 57 58 59 60 61 62 63 64 65 66 67 68 69 70
```

```
7:70
```

```
##  [1]  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
## [26] 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
## [51] 57 58 59 60 61 62 63 64 65 66 67 68 69 70
```

   b. Create a numeric vector with 60 draws from the random uniform distribution

```
runif(60)
```

```
##  [1] 0.5683538846 0.0205104277 0.7092157681 0.4487249316 0.0002471774
##  [6] 0.2868282734 0.5839155971 0.2633793848 0.9604981067 0.8940514720
## [11] 0.2358707148 0.9758889992 0.6337525947 0.8955348674 0.1349425027
## [16] 0.0859924695 0.0491934228 0.0522113051 0.6824128947 0.3250433479
## [21] 0.5572530243 0.4599688153 0.0945807202 0.8509030649 0.8275658034
## [26] 0.3589847651 0.2538955167 0.8556171078 0.5817513417 0.2503475142
## [31] 0.5410516174 0.5344028412 0.0531174964 0.9740717690 0.0605288467
## [36] 0.8098904958 0.2918797559 0.5099244302 0.7104848460 0.0988144758
## [41] 0.6721906255 0.0533076327 0.1418984905 0.6560813647 0.6183298128
## [46] 0.2348305057 0.3418257367 0.3758054264 0.4249662606 0.5451701770
## [51] 0.8617312748 0.4677271657 0.7368820217 0.5178881630 0.2817808683
## [56] 0.6993886526 0.2380420147 0.9594222449 0.4187634010 0.0489111154
```

   c. Create a character vector with the letter "x" repeated 1980 times.

[1] Sign up for the free tier which should be sufficient for camp. You will have to install packages.

```
rep("x", 1980)
```

```
##  [1] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
## [20] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
## [39] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
##  [ reached getOption("max.print") -- omitted 1930 entries ]
```

d. Create a character vector of length 5 with the items "Nothing" "works" "unless" "you" "do". Call this vector `angelou_quote` using `<-`.

```
angelou_quote <- c("Nothing", "works", "unless", "you", "do")
angelou_quote
```

```
## [1] "Nothing" "works"   "unless"  "you"     "do"
```

e. Create a numeric vector with 1e4 draws[2] from a standard `normal` distribution.

```
rnorm(1e4)
```

```
##  [1] -0.436303444 -0.482488158  0.039970426  1.279981174 -0.787241440
##  [6] -1.519359929 -0.009617635  0.495570789 -0.075985588  1.296903180
## [11]  0.636153306  0.987832825 -1.086653444 -2.349647694 -1.816436398
## [16] -0.186139825 -0.282793248 -0.507015715  0.709704927  2.340045521
## [21]  1.252637619 -0.938060012  0.151655372 -1.321915461  0.981088795
## [26]  0.992958496  0.705343373  1.830513000  0.601438250  0.195001911
## [31] -2.126440120  0.518555387 -0.395562931  0.663818755 -1.445574327
## [36]  0.318088487 -1.010829211  1.680292918 -0.152018218  1.058605556
## [41]  0.492064783 -0.227639527 -0.404583393 -0.964539740 -0.281017648
## [46] -0.800033674  1.365069419 -1.681602792 -0.403600545  1.788670808
##  [ reached getOption("max.print") -- omitted 9950 entries ]
```

f. Create an integer vector with the numbers 0, 2, 4, ... 20.

```
seq(from= 0 , to = 20, by = 2)
```

```
##  [1]  0  2  4  6  8 10 12 14 16 18 20
```

3. Run this code and explain why we get an error.

```
# make sure you followed direction in part d above.
sum(angelou_quote)
```

Solution:
We get the following error:
"Error in sum(angelou_quote) : invalid 'type' (character) of argument"
This is because the sum function is unable to add character values together.

4. If we want `angelou_quote` to be a single string, we can use `paste0`.

```
paste0(angelou_quote, collapse = " ")
```

```
## [1] "Nothing works unless you do"
```

a. We gave collapse the argument " " i.e. a character string that is a blank space. Try a different character string.

```
paste0(angelou_quote, collapse = ". ")
```

```
## [1] "Nothing. works. unless. you. do"
```

[2] This is scientific notation. Try `1e4 - 1 + 1` in the console.

```
paste0(angelou_quote, collapse = "#")
```

```
## [1] "Nothing#works#unless#you#do"
```

5. Try these lines of code using `paste0` (or it's `tidyverse` synonym `str_c`)[3] .

```
paste0(angelou_quote, ".com")
```

```
## [1] "Nothing.com" "works.com"   "unless.com"  "you.com"     "do.com"
```

```
paste0(angelou_quote, c("!", "!", "?", " :(", "!!"))
```

```
## [1] "Nothing!" "works!"   "unless?"  "you :("   "do!!"
```

```
str_c(c("bob", NA, "maya"), "@gmail.com")
```

```
## [1] "bob@gmail.com"  NA                "maya@gmail.com"
```

```
paste0(c("bob", NA, "maya"), "@gmail.com")
```

```
## [1] "bob@gmail.com"  "NA@gmail.com"   "maya@gmail.com"
```

   a. Explain to your partner what paste0 is doing.
   Solution:
   paste0 converted the NA value into a character, which is unexpected behaviour.

6. Common error alert. Run the following code and explain why it throws an error.

```
c(1, 2) + c(1 2)
```

   This is an example where the error is not so helpful. I get this one a lot, because I forget to put a comma where it should be.

2 Calculating Mean and Standard Deviation with vectors

2.1 Is the coin fair?

In this exercise, we will calculate the mean of a vector of random numbers. To get started, we'll generate some fake data using built-in random[4] sampling functions. Let's start by flipping coins.

```
(coin_flips <- sample(c("Heads", "Tails"), 10, replace = TRUE))
```

```
##  [1] "Tails" "Tails" "Tails" "Heads" "Heads" "Tails" "Heads" "Tails" "Heads"
## [10] "Tails"
```

`sample()` is a function that requires two arguments.

- In the first position, we have a vector of any type. We sample *from* this vector.
- In the second position, we have `size` which is the number of items to choose.

If we want to have independent draws from our sampling vector, we say `replace = TRUE`. By default `replace` is `FALSE`.

1. Run the following code and get an error.

   a. Interpret the error **Solution:** We get an error because this code is taking samples without replacement. After 6 draws, there are no more numbers to sample from.

[3] `tidyverse` synonyms are usually preferrable since they have fewer quirky behaviors. For example, try `str_c(c("bob", NA, "maya"), "@gmail.com")` vs `paste0(c("bob", NA, "maya"), "@gmail.com")`

[4] Techinically, "pseudo-random", but who's asking.

b. Adjust the code so that you simulate 100 independent die rolls.
Solution:

```
die_rolls <- sample(c(1, 2, 3, 4, 5, 6), 100, replace = TRUE)
die_rolls
```

```
##   [1] 3 5 2 1 5 1 5 6 5 1 2 3 6 4 6 3 2 5 3 5 4 2 5 1 1 2 2 5 1 1 3 3 5 6 1 3 6
##  [38] 5 2 5 1 6 4 5 6 2 2 3 4 3 4 5 1 4 5 5 3 2 3 2 1 4 3 2 3 1 2 5 3 5 2 6 5 5
##  [75] 1 6 1 3 6 5 6 4 1 4 1 2 2 3 1 2 5 4 6 6 3 1 5 2 3 3
```

2. In my coin-toss simulation above, I sample from a character vector. Doing so, makes it easier to interpret the outcome, but difficult to do stuff with the results. Replace the characters with 1 and 0. Now, you'll be able to do math, but the results are more abstract. You can choose whether 1 represents heads or tails, just be consistent. Collect samples of size 10, 1000 and 1000000.[5]

Solution:
(I chose 1 to represent heads, and 0 to represent tails.)

```
# replace ... with suitable code
ten_flips <- sample(c(0, 1), 10, replace = TRUE)
thousand_flips <- sample(c(0, 1), 1000, replace = TRUE)
million_flips <- sample(c(0, 1), 1e6, replace = TRUE)
```

a. What data type are your `xxx_rolls` vectors?
Solution: They are of the `double` type.

```
typeof(ten_flips)
```

```
## [1] "double"
```

a. Use `sum()` on your vectors. What does this represent?
Solution: This represents the number of "heads" that I flipped.

```
sum(ten_flips)
```

```
## [1] 3
```

a. Use `length()` on your vectors to verify the vectors are the right length. What does this represent?
Solution: This represents the total number of times that I "flipped" the coin.

```
length(ten_flips)
```

```
## [1] 10
```

3. A fair coin assigns equal probability to heads and tails. Thus, the probability of heads or tails is 50 percent or .5. We can run experiments or simulations to see if our "coins" are fair. In particularly, we can calculate an estimate of the probability of heads by computing estimated probability heads $= \hat{p}(\text{heads}) = \frac{\text{n heads}}{\text{n flips}}$. The estimated probability is often called $\hat{p}$. Use the starter code to calculate the estimated probability of heads from your `ten_flips` sample.

Solution:

```
n_heads <- sum(ten_flips)
n_flips <- length(ten_flips)
p_hat_ten <- n_heads / n_flips
```

4. Repeat the code from part 3 to find the estimated probability of heads from your `thousand_flips` sample and `million_flips` sample.[6]
Solution:

[5] Note: you can use scientific notation `1e6` is short for 1 with 6 zeros.
[6] In the fall, we'll discuss ways to write this type of code more efficiently without copy paste.

```
p_hat_thousand <- sum(thousand_flips) / length(thousand_flips)
p_hat_mil <- sum(million_flips) / length(million_flips)
```

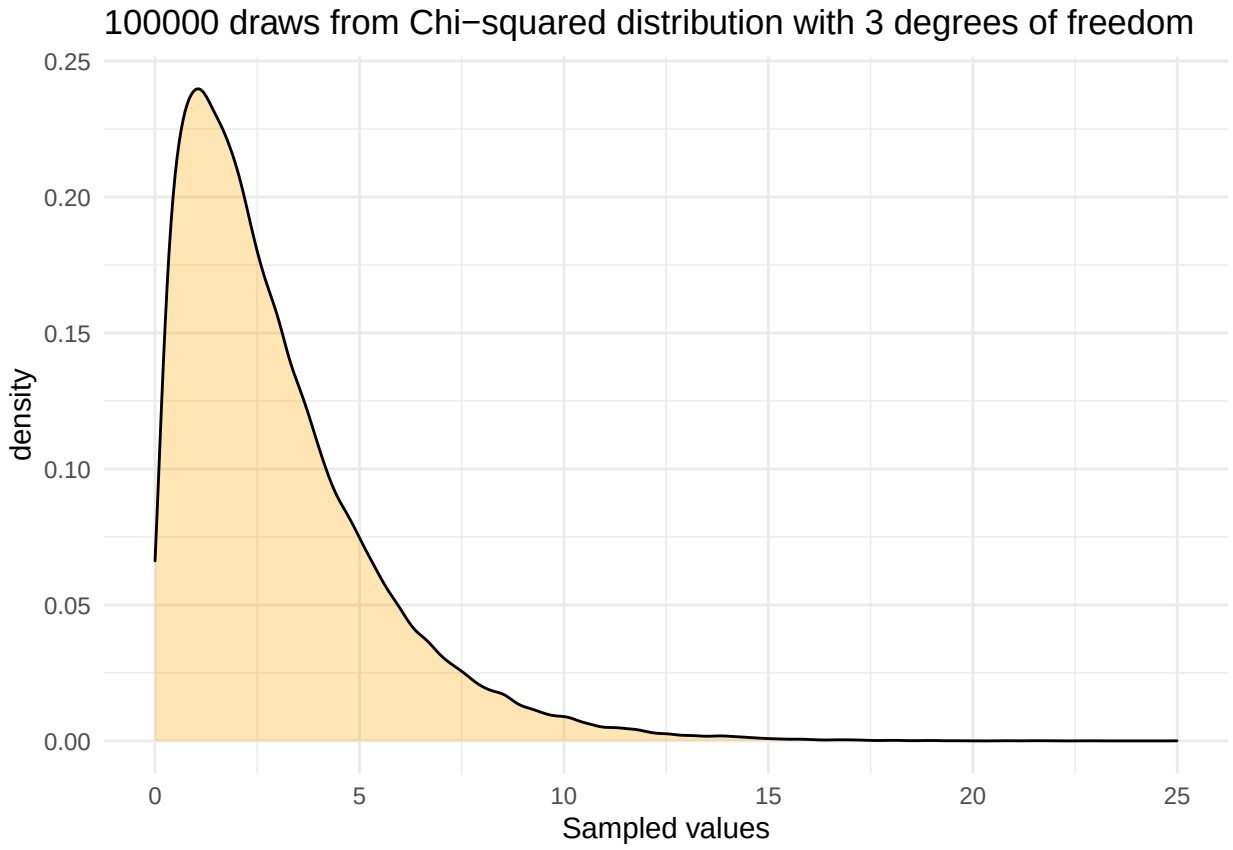
a. Re-run all the code from parts 2 through 4 a few times. Notice that the random number generator will give a different sequence of flips each time.
b. What do you notice about the estimated probabilities as the sample size gets larger? (This is an example of the "Law of Large Numbers")
Solution: The estimated probabilities approach 0.5 as the sample size gets larger.

5. We had you calculate the estimated probability with `sum() / length()`. R also has a function `mean()` built in. Simplify the computation for `p_hat_xxx` by using `mean()`. **Solution:**

```
p_hat_ten <- mean(ten_flips)
```

2.2 A new distribution.

Now we are going to take random samples from a chi-squared distribution with 3 degrees of freedom. Do not worry about what the distribution's name means, but be aware of that it looks something like the picture below. It's possible–but highly unlikely–to get values up to `Inf`, which is R for infinity.



We are going to calculate the mean, variance and standard deviation of the distribution using vectors in three different ways.

1. First, we'll do it "by hand". The formula for sample variance is $Var(x) = \frac{\sum(x-\bar{x})^2}{n-1}$. where
   - $\bar{x}$ is the sample mean.
   - n is the sample size and
   - $\sum$ means we add up

Solution:

```
# fill in the ... with appropriate code.
x <- rchisq(100000, 3)

# this one should be straight forward!
# (See what we did with coin flips)
x_bar <- mean(x)
n <-  length(x)
# The formula in R will be exactly the same as the
# fomula in math thanks to vectorization!
# If you aren't sure the code will work the way you want
# try with a simpler x. x <- c(1, 0, 1, 1)
var_x <- sum((x - x_bar)^2) / (n-1)
var_x
```

```
## [1] 5.991139
```

2. Standard deviation is the square root of Variance, i.e. $sd(x) = \sqrt{Var(x)}$. Calculate the standard deviation.[7]
Solution:

```
sd_x <- sqrt(var_x)
sd_x
```

```
## [1] 2.44768
```

3. Now, we'll check your work using built in R functions. To calculate variance use `var()`. To calculate standard deviation use `sd()`. Try them out. If you disagree with your previous results, it's most likely a coding error in the definition of `var_x`.[8]
Solution:

```
var(x)
```

```
## [1] 5.991139
```

```
sd(x)
```

```
## [1] 2.44768
```

4. Finally, we can do this in a `tibble` setting and use summarize. You may need to load a package.[9] Using a tibble provides two services 1) the results print as an organized table. 2) We can do further `tidyverse` processing with it.
Solution:

```
# replace the ... with suitable code.
tibble(x = x) %>%
summarize(mean = mean(x),
          variance = var(x),
          `standard deviation` = sd(x))
```

```
## # A tibble: 1 x 3
##    mean variance `standard deviation`
##   <dbl>    <dbl>                <dbl>
## 1  3.00     5.99                 2.45
```

[7] Hint: we have the function `sqrt()`

[8] The most common errors are about where you put your parentheses. The second most common error is where you put the power i.e. ^.

[9] Hint: it's the `tidyverse`.

5. Copy your code from the previous problem, but replace `summarize` with `mutate`. Explain the result to your group.
Solution: Using mutate instead of summarize retains the original vector x, and the aggregate calculated column vectors (mean, variance, and SD) are the same length as vector x instead of becoming reduced to single values like they are when we use summarize.

```
tibble(x = x) %>%
mutate(mean = mean(x),
          variance = var(x),
          `standard deviation` = sd(x))
```

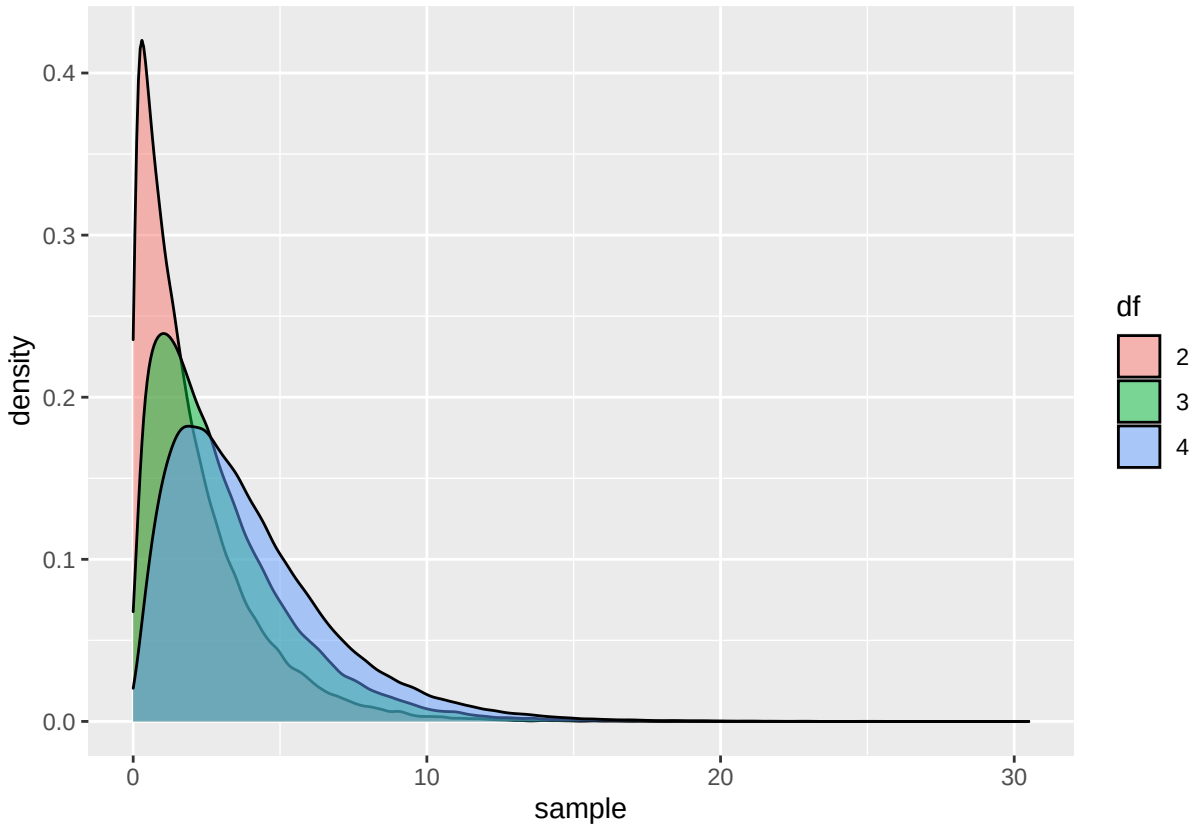
```
## # A tibble: 100,000 x 4
##         x  mean variance `standard deviation`
##     <dbl> <dbl>    <dbl>                <dbl>
##  1  3.72   3.00     5.99                 2.45
##  2 13.0    3.00     5.99                 2.45
##  3  7.99   3.00     5.99                 2.45
##  4  1.93   3.00     5.99                 2.45
##  5  8.16   3.00     5.99                 2.45
##  6  1.22   3.00     5.99                 2.45
##  7  5.47   3.00     5.99                 2.45
##  8  0.439  3.00     5.99                 2.45
##  9  1.23   3.00     5.99                 2.45
## 10  0.858  3.00     5.99                 2.45
## # ... with 99,990 more rows
```

2.3 Challenge problems

1. Run the code below. The resulting graph shows three chi-sq distribtions determined by their degrees of freedom.

```
chi_sq_samples <-
 tibble(x = c(rchisq(100000, 1) + rchisq(100000, 1),
              rchisq(100000, 3),
              rchisq(100000, 4)),
        df = rep(c("2", "3", "4"), each = 1e5))

chi_sq_samples %>%
  ggplot(aes(x = x, group = df, fill = df)) +
  geom_density( alpha = .5) +
  labs(fill = "df", x = "sample")
```



2. How many rows are in the tibble? Explain how the code that defines `x` and the code that defines `df` make vectors that are the right length.
Solution: There are 300,000 rows in the tibble. The code that defines x creates the first column, which is made up of three 100,000 length chi-squared draws that are combined into a single vector. The code that defines `df` (degrees of freedom) repeats each of the strings `2`, `3` and `4` 100000 times,

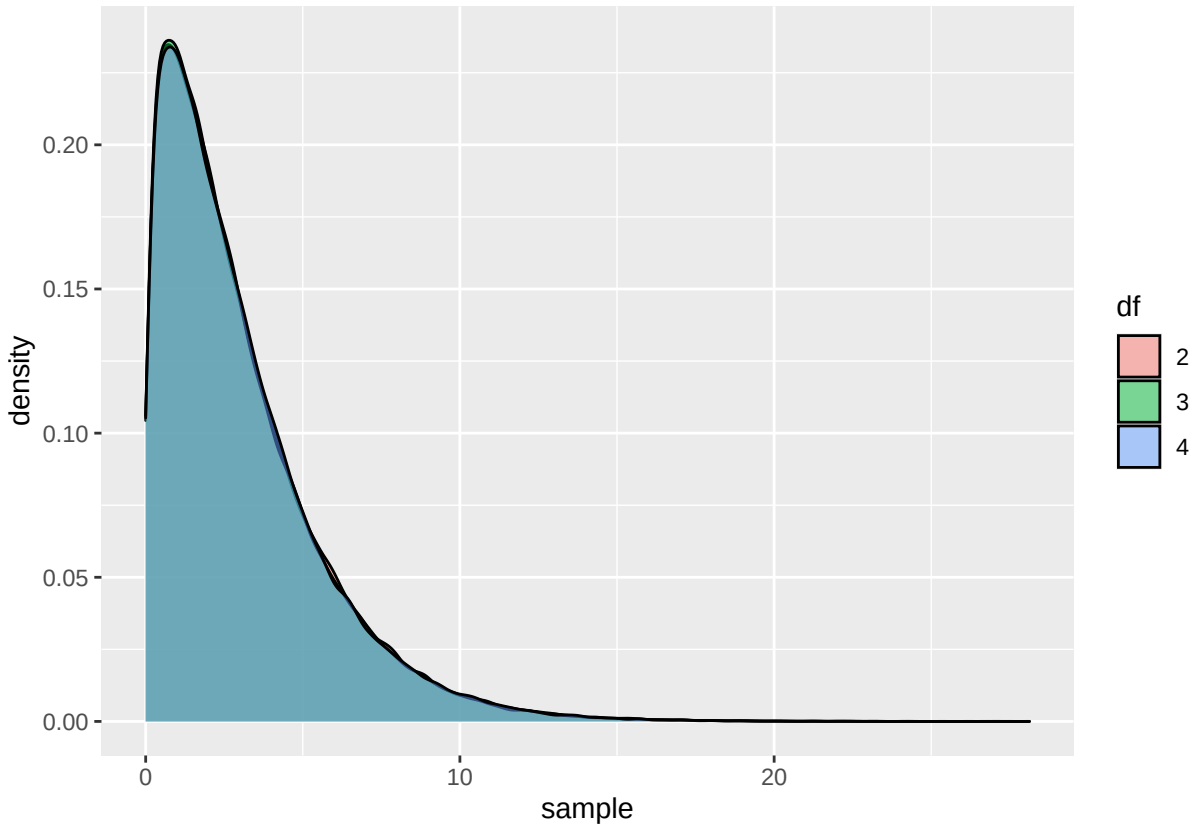
```
nrow(chi_sq_samples)
```

```
## [1] 300000
```

3. Temporarily delete `each =` (keep `1e5`). Explain why the resulting graph looks the way it does. Make sure you put `each =` back. **Solution:** The resulting graph has three very similar draws which are overlaid on each other. Without specifying `each = 1e5`, the vector that we are trying to create for the df column will repeat `c("2", "3", "4")` over and over again until it is the correct length. By saying `each = 1e5`, `df` column will instead repeat the first value (2) 100000 times before adding values for 3 and 4.

```
chi_sq_samples_bad <-
 tibble(x = c(rchisq(100000, 1) + rchisq(100000, 1),
              rchisq(100000, 3),
              rchisq(100000, 4)),
        df = rep(c("2", "3", "4"), 1e5))

chi_sq_samples_bad %>%
  ggplot(aes(x = x, group = df, fill = df)) +
  geom_density( alpha = .5) +
  labs(fill = "df", x = "sample")
```


0.20
0.15
0.10
0.05
0.00
density
0
10
20
sample
df
2
3
4

->