# Statistics M2 Assignment 2

**Required readings**

- Phillip Good textbook (in the Zotero): Chapter 1 and Chapter 2; if you choose to do any of the exercises, please let me know, I can help you check your answers
- GGplot tutorial (parts 1, 2, and 3): http://bit.ly/2ke0mX8

**NOTE:** Chapter 1, section 1.7.2, of the Good textbook talks about a resampling procedure called "bootstrapping." That is **not** what we did in class. What we did in class is called a "permutation test," and that's what you're going to do more of in Exercise 3 here. If, after looking over last week's in-class notes, you're not clear on permutation tests, read over **Chapter 3, Section 3.6** of the Good textbook.

## Assignment 2

Create a fresh repository on Github for this assignment, and a fresh R project, in a fresh folder. Do not use your Assignment 1 folder, repository, or R project. As in Assignment 1, you should put your work in an Rmd file. We will add an additional file in this assignment, to store your functions.

You're going to be writing R code, in the form of functions. I've provided you test code, in the form of code chunks that you can insert into your document. **Do not modify these code chunks.** Copy them wholesale. When the test code gives the right output, your function is working (at least, as far as I can tell). I strongly recommend the following strategy: paste the code chunk into your document **before you start writing the function.** When you try and knit your document, you'll get an error, and the Rmd document won't work. Then start writing the function. Gradually, you'll go from a document that doesn't work to a working document with a wrong function to a working document with a correct function. This is good practice.

If you have to change your code many times, you might be tempted to type or paste code into the R console to try things out, rather than putting it directly into the document (especially if you're working on the class server, where there might be some delays due to the Internet connection). You will probably run into the problem that objects that you created elsewhere in the document won't exist. This is a good thing. Your Rmd document is a sealed, self-contained environment, which you can distribute to others and be content that, if it worked for you, it will probably work for them. When it is compiled, the variables created in the code have nothing to do with the outside world (including the R console). The R console, on the other hand, can be contaminated with variables you created and forgot about, and your code might wind up depending on them (in ways you forget about). This is a bad thing. But it is still annoying to have to reload your document every time you want to mess around with something. This is why Rstudio gives you the "Run" dropdown menu, from which you can select the useful options "Run Current Chunk" (which will run all the code in the selected chunk in the console), and "Run All Chunks Above" (which will run all the code in the chunks up to but not including the selected chunk—these chunks might be doing something that the current chunk depends on). With these, you can re-create the same environment on the console as you'd find sealed inside your compiled Rmd document.

You can of course submit your written answers in English or French.

### Exercise 1: Writing R functions

After you have created your Rmd file, create a second file (an R script: go to File > New File > R script), and save it in your Assignment 2 folder as "functions.R". Instead of writing your functions in a chunk at the top of the Rmd like we did in class, you're going to write your functions in this file, and load the contents of this file at the top of the Rmd. This will save you some mess.

The chunk that loads the contents of this file should contain only one line:

```
source("functions.R")
```

**Suggested additional readings:**

Do these if you want to understand functions better. If you get stuck with these readings, I suggest you try out the exercises, see how far you get, and then come back to the readings.

- http://bit.ly/2yCkQv8
- http://bit.ly/2g3BTiK
- http://bit.ly/2wtdQA2
- http://bit.ly/2r1Q22u

**Exercise 1a: filling in the blanks**

Here is a function that's partially written for you. It has a comment at the top to explain what it does. To insert comments, you simply include the symbol # on a line of R code. Everything that comes after the # is a comment. That means it will be treated by R as if it didn't exist. It is there purely for the reader. You should always write a comment at the beginning of your function to describe what it does, as I've done here.

Your job is to replace the comment lines **inside** the function with a line or lines of code, so that the function does what it says it does. (Remember to write this function inside `functions.R`, and not in your main Rmd file.)

```
# Sum values in a column of a data frame.
#
# ARGUMENTS:
# d: a data frame or tibble
# var: the name of a column of d, provided as a string
#
# RETURN VALUE:
# if the specified column exists and contains numbers, returns the sum of
# all values in the column; otherwise, returns NULL
sum_column <- function(d, var) {
  # Set a default value to return
  result <- NULL
  x <- d[[var]] # Remember, as we showed in class: there are two ways
                # to access a column, and this is one; you should try
                # and figure out why the other way won't work here,
                # but that's not part of the homework
  if (!is.null(x)) { # This tests to see whether the column exists in
                     # d; again, you should try and find a way to test
                     # this out for yourself to verify that it's true,
                     # but that's not part of the homework
    # YOUR CODE HERE: if x contains numbers, set the variable
    # result to be the sum of the values in x
    #
    # You will need to do a bit of research in order to figure out how
    # to test whether a vector contains numbers.
  }
  # YOUR CODE HERE: return the result
}
```

In order to verify that your function works, add a new chunk to your Rmd that contains the following code:

```
print(sum_column(iris, "Sepal.Length"))
print(sum_column(iris, "Species"))
print(sum_column(warpbreaks, "breaks"))
```

The output you get should be:

```
## [1] 876.5
```

```
## NULL
```

```
## [1] 1520
```

If it isn't, there's something wrong with your function. Of course, you can also add another chunk to test other cases, if you're concerned.

**Exercise 1b: a function with one argument and one return value**

Write a function called `my_sum` that adds up the values in a vector. You can do anything you want, except using the built-in function that adds up the numbers in a vector (which you probably used in the last question)! This would be cheating. You can probably also find many examples of functions like this on the Internet. This would be sort of helpful if you get stuck, but you have to try yourself first. Otherwise you won't learn. To make sure that you understand your function, I'm requiring you to **write a comment above every line explaining what it does**. Don't do this normally. In general, if it's not obvious from your variable names what your function does, you should change the function to make it clearer. But this is an exercise.

I've written the initial comment for you. (Keep it in your code!)

```
# Sum values in a vector.
#
# ARGUMENTS:
# x: a vector
#
# RETURN VALUE:
# if the vector contains numbers, returns the sum of
# all values; otherwise, returns NULL
#
# [YOUR FUNCTION HERE]
```

In order to verify that your function works, add a new chunk to your Rmd that contains the following code:

```
print(my_sum(iris$Sepal.Length))
print(my_sum(iris$Species))
print(my_sum(warpbreaks$breaks))
```

The output should be the following:

```
## [1] 876.5
```

```
## NULL
```

```
## [1] 1520
```

**Exercise 1c: a function with two arguments and one return value**

Write a function called `sum_divided_by` that takes two arguments: a vector `x` and a number `k`. It returns the sum of the elements of `x` **divided by** the number `k`. If either `x` or `k` are non-numeric, it should return `NULL`. You can do anything you want, with one requirement: **to calculate the sum of the elements in `x`, you must use your function `my_sum`**. Make sure you write a comment at the top, and, in this case too, write a comment above every line explaining what it does.

In order to verify that your function works, add a new chunk to your Rmd that contains the following code:

```
print(sum_divided_by(iris$Sepal.Length, 12))
print(sum_divided_by(iris$Species, 22))
print(sum_divided_by(iris$Sepal.Length, "Not numeric"))
print(sum_divided_by(warpbreaks$breaks, -12))
```

The output should be the following:

```
## [1] 73.04167
```

```
## NULL
```

```
## NULL
```

```
## [1] -126.6667
```

(Technically, in R, all numbers are vectors that simply happen to be of length one. This means that you could accidentally put a longer vector in place of `k`, instead of just a number, and you wouldn't know there was a problem until you looked at the strange result. You don't need to worry about this problem. Just make sure that your comment at the top of the function explains what `k` is supposed to be.)

**Exercise 1d: one more function**

Write your own function `my_mean` to calculate the mean of a vector (the sum divided by the number of elements), making use of your function `sum_divided_by`, and not making use of the built-in functions that calculate the sum or the mean. It should return `NULL` for non-numeric vectors. As in every case in this exercise, I remind you to document it. That means: include a comment at the top like the ones in my examples. In the case of this particular example, I'd also like you to put line by line comments explaining each step.

Here is the code chunk that you need to include to test the behaviour of the function:

```
print(my_mean(iris$Sepal.Length))
print(my_mean(iris$Species))
print(my_mean(warpbreaks$breaks))
```

It should give the output:

```
## [1] 5.843333
```

```
## NULL
```

```
## [1] 28.14815
```

**Exercise 2: Working with ggplot**

First, make sure you've done the required readings on ggplot (above).

**Exercise 2a: Creating violin plots**

A violin plot is similar to a histogram. In this exercise, you'll fill in the blanks in the function on the next page to produce a violin plot, split up by group. Notice that I'm explicitly referring to the `ggplot2` package to access its functions using `ggplot2:: ...`. In all the examples that you saw in the tutorial, you didn't have to do this, because you were told to type `library(ggplot2)`, which allows you to skip the `ggplot2:: ...` . For this exercise, continue to use the `ggplot2:: ...` and we will talk about the benefits and hazards of `library(...)` in the future.

You can and should use the GGplot help! You can access the help pages in R, or access the illustrated reference online.

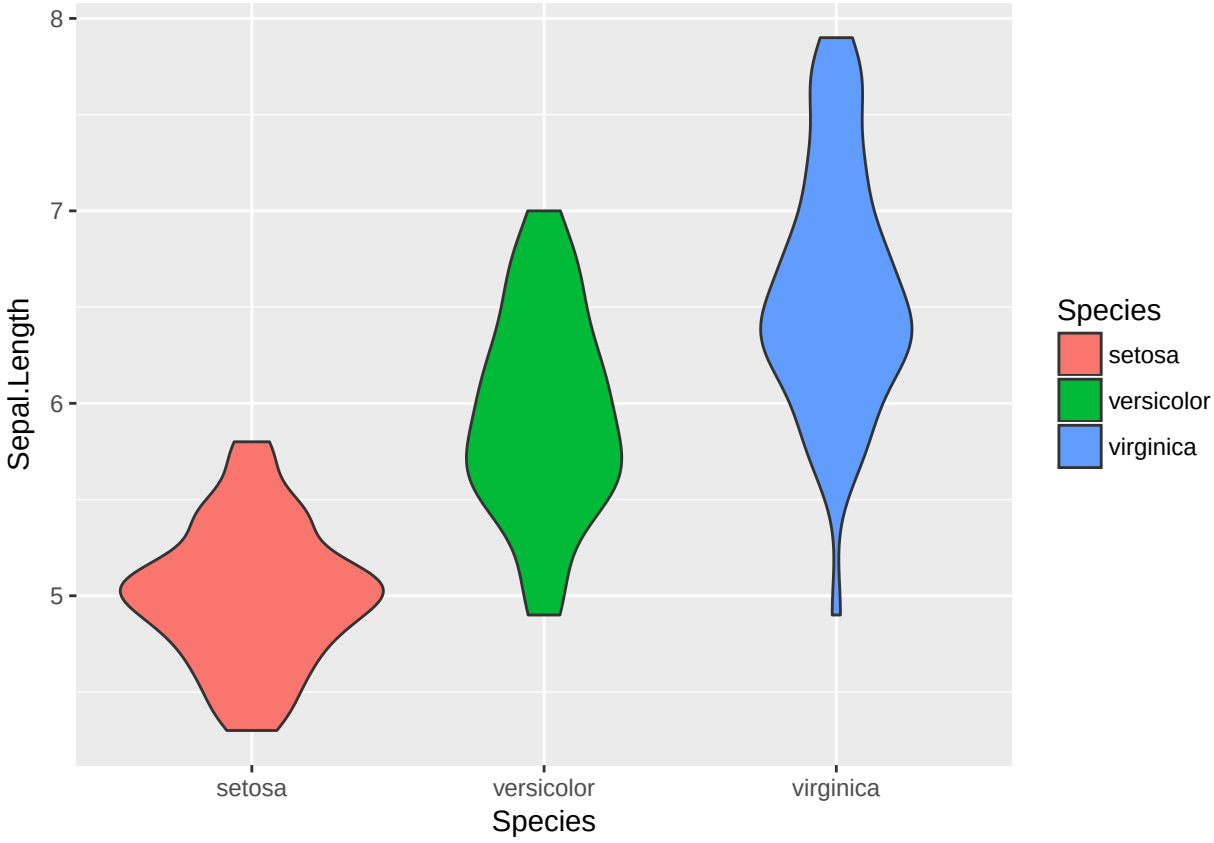
Notice that you shouldn't change my code. The variable you need to return is indeed `p`. Part one of the tutorial you read explains the idea of ggplot "plot objects". If you're having trouble updating a plot object after reading the tutorials, feel free to do some Googling, or look in the in-class notes from last class.

```
# Return a violin plot.
#
# ARGUMENTS:
# d: a data frame or tibble
# var: the name of a column of d containing the dependent variable, provided as a
#     string
# grouping_var: the name of a column of d containing a grouping variable,
#               provided as a string
#
# RETURN VALUE:
# A ggplot plot object containing a violin plot, grouped by the values
# of the grouping variable.
#
grouped_violin_plot <- function(d, var, grouping_var) {
  # Create the base ggplot object
  p <- ggplot2::ggplot(d, ggplot2::aes_string(y=var,
                                              x=grouping_var,
                                              fill=grouping_var))
  # YOUR CODE HERE: Create a violin plot
  return(p)
}
```

Add this chunk to test:

```
print(grouped_violin_plot(iris, "Sepal.Length", "Species"))
```

The output should be the following:

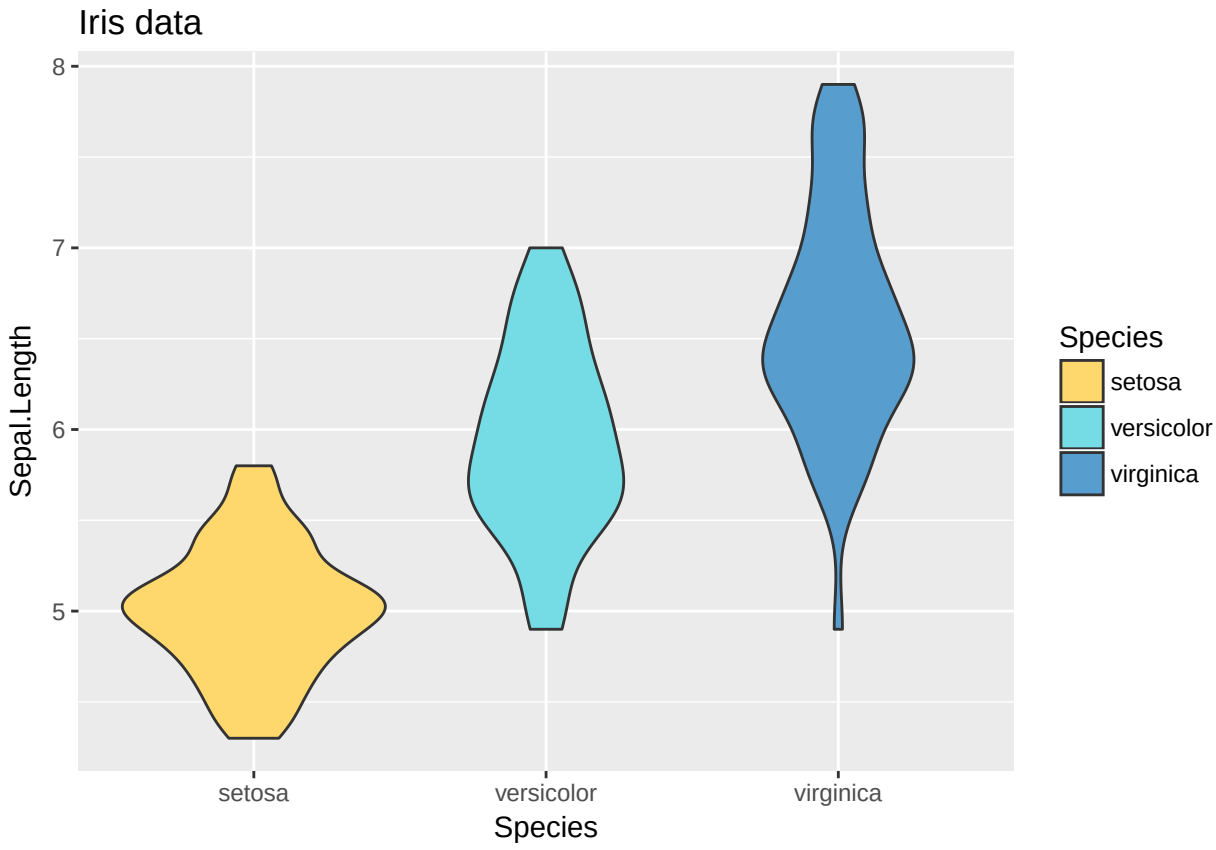


**Exercise 2b: Modifying plots**

You don't need to write a new function in this exercise. You just need to figure out how to modify the code given here to change the colour scheme and add a main title. Add this code in a chunk, modify it.

```
p <- grouped_violin_plot(iris, "Sepal.Length", "Species")
# YOUR CODE HERE: Change the colour scheme for the interior of the three violin plots
#                 to anything else at all.
# YOUR CODE HERE: Add a main title that says "Iris data".
print(p)
```

You should get something like this (not necessarily with these colours):




**Exercise 3: Permutation tests**

This part of the exercise is based on the second part of what we did in class.

**Exercise 3a: Writing a more generic function for taking a test statistic**

Based on what we did in class, your first task is to replace the code that takes the test statistic - in this case the difference in the medians - with something more generic.

Here's an extract from what we had in class (this is not code that will run):

```
for (i in 1:N_SAMPLES) {
  ...
  statistics[i] <- median(spr_fake_gram$logRT) - median(spr_fake_ungram$logRT)
  ...
}
```

Fill in the blanks in the following function.

```
# Difference in the medians between two groups.
#
# ARGUMENTS:
# d: a data frame or tibble
# var: the name of a column of d containing the dependent variable, provided as a string
# grouping_var: the name of a column of d containing a grouping variable, provided as a string
# group1: the value of grouping_var that corresponds to the first group
# group2: the value of grouping_var that corresponds to the second group
#
# RETURN VALUE:
# The median value of var for the first group, minus the median value of var for the second
# group.
#
difference_in_medians <- function(d, var, grouping_var, group1, group2) {
  d_1 <- dplyr::filter(d, get(grouping_var) == group1)
  d_2 <- dplyr::filter(d, get(grouping_var) == group2)
  # YOUR CODE HERE: assign the difference in the medians to to the variable 'result'
  return(result)
}
```

Paste in the following chunk and use it to test your code.

```
difference_in_medians(iris, "Sepal.Width", "Species", "versicolor", "virginica")
difference_in_medians(iris, "Sepal.Width", "Species", "virginica", "virginica")
```

It should give the following output:

```
## [1] -0.2
```

```
## [1] 0
```

**Exercise 3b: Writing a more generic function of the randomization (i.e., permutation) function**

We'd previously written a function called `randomize_rts` to shuffle the reaction times in our data frame. It looked like this:

```
randomize_rts <- function(d) {
  result <- d
  n <- nrow(d)
  d$logRT <- sample(d$logRT, n)
  return(d)
}
```

Fill in the blanks in the following function to replace it:

```
# Randomize the order of a column.
#
# ARGUMENTS:
# d: a data frame or tibble
# var: the name of a column of d containing the variable to randomize,
#      provided as a string
#
# RETURN VALUE:
# A data frame or tibble exactly the same as d, except with the order of
# var permuted randomly.
#
randomize <- function(d, var) {
  d[[var]] <- # YOUR CODE HERE: generate a shuffled version of d[[var]]
  return(d)
}
```

Notice that, because of the way functions work in R, shuffling the column inside the table called `d` in the function doesn't change the original table that you passed in. In the following code chunk, the original `iris` table will not change after you apply `randomize` to it.

Copy in the following chunk and use it to test your code.

```
iris$Sepal.Width[1:10]
if(!exists(".Random.seed")) set.seed(NULL)
previous_seed <- .Random.seed
set.seed(1)
randomize(iris, "Sepal.Width")$Sepal.Width[1:10]
randomize(iris, "Species")$Species[1:10]
randomize(iris, "Species")$Sepal.Width[1:10]
set.seed(previous_seed)
```

You should get the following output:

```
##  [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1
```

```
##  [1] 3.4 2.8 3.0 2.8 3.2 2.8 3.4 2.7 2.5 2.9
```

```
##  [1] versicolor versicolor setosa     versicolor versicolor setosa
##  [7] versicolor setosa     setosa     setosa
## Levels: setosa versicolor virginica
```

```
##  [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1
```

(The part about the random seed is something we saw last time in the assignment, and that I mentioned in class. I'm setting the random seed so that you'll see the same shuffle as me, and then I'm setting it back to whatever it was before I changed it. The output contains something about "Levels", which you can safely ignore for now.)

**Exercise 3c: A function to get a statistic for multiple permutations**

Last time we used the following code to get the difference in the medians for a sequence of 200 new permuted samples.

```
N_SAMPLES <- 200
statistics <- rep(0, N_SAMPLES)
for (i in 1:N_SAMPLES) {
  spr_fake <- randomize_rts(spr)
  spr_fake_gram <- dplyr::filter(spr_fake, Condition == "Grammatical")
  spr_fake_ungram <- dplyr::filter(spr_fake, Condition == "Ungrammatical")
  statistics[i] <- median(spr_fake_gram$logRT) - median(spr_fake_ungram$logRT)
}
```

You can now replace all the code within the `for` loop using the two functions you've just written. Now, go farther. Fill in the blanks in the following function.

```
# Perform a permutation test for two groups.
#
# ARGUMENTS:
# d: a data frame or tibble
# var: the name of the column in d on which the test statistic will be calculated,
#      provided as a string
# grouping_var: the name of the column in d which gives the grouping
# group1: the value of grouping_var corresponding to the first group
# group2: the value of grouping_var corresponding to the second group
# statistic: a function yielding a test statistic, which takes as input
#            a data frame, the name of a variable on which to calculate the
#            test statistic, the name of a grouping variable, the value of
#            the grouping variable corresponding to the first group, and
#            the value of the grouping variable corresponding to the second
#            group
# n_samples: the number of permutation samples to draw (default: 9999)
#
# RETURN VALUE:
#
# A list containing two elements:
#
#  - observed: the value of statistic() in d
#  - permuted: a vector containing the values of statistic() under n_samples
#              permutations
#
permutation_twogroups <- function(d, var, grouping_var, group1, group2, statistic,
                                  n_samples=9999) {
  observed_statistic <- statistic(d, var, grouping_var, group1, group2)
  permutation_statistics <- rep(0, n_samples)
  for (i in 1:n_samples) {
    # YOUR CODE HERE: use randomize(...) to create a permutation and then
    #                 fill in the vector permutation_statistics with the
    #                 value of statistic(...) for this new permutation
  }
  result <- list(observed=observed_statistic,
                 permuted=permutation_statistics)
  return(result)
}
```

Before you start, there are three surprising and previously unseen things here.

First, the result (the return value) is a list. We have not seen lists before, but they are simply ways of bundling up several objects together. In this case, I use a list because R only lets you return one object from a function—whereas, when we're doing a permutation test, it's useful to get back several different pieces of information. You can access the named elements of a list in the same way you access columns of a table, using `...[["ELEMENT_NAME"]]` or `...$ELEMENT_NAME`.

Second, one of the arguments, `statistic`, needs itself to be a function. When you use the function `permutation_twogroups`, it will be by telling it what test statistic to calculate (up to now, we've been using the difference in medians), and it will then calculate this given test statistic for all `n_samples` permutations.

Third, one of the arguments, `n_samples`, has a default value. That means you don't need to specify it. By default, it will be 9999 (which is also a reasonable value for many permutation tests). If you do wish to specify it (for example, to test your code without waiting too long), you can simply provide a different value. Whenever you're calling a function and providing a value for an optional argument, you should always explicitly tell R which argument you're targeting in the function call, using `[NAME_OF_PARAMETER]=[VALUE]`. (Why? Think about what would happen if there were **two** optional parameters. You don't need to answer this question explicitly in the homework, just think about it and experiment.)

The following code chunk should concretize these concepts for you. Insert it to test your code.

```
if(!exists(".Random.seed")) set.seed(NULL)
previous_seed <- .Random.seed
set.seed(1)
ptest_1 <- permutation_twogroups(iris, "Sepal.Width", "Species", "versicolor",
                                 "virginica", difference_in_medians, n_samples=10)
ptest_2 <- permutation_twogroups(iris, "Sepal.Width", "Species", "versicolor",
                                 "virginica", difference_in_medians, n_samples=10)
ptest_3 <- permutation_twogroups(randomize(iris, "Sepal.Width"), "Sepal.Width",
                                 "Species", "versicolor", "virginica",
                                 difference_in_medians, n_samples=10)
set.seed(previous_seed)
print(ptest_1)
print(ptest_2)
print(ptest_3)
print(ptest_3$observed)
print(ptest_3[["observed"]])
```

You should get the following output.

```
## $observed
## [1] -0.2
##
## $permuted
##  [1]  0.10 -0.05  0.00  0.00  0.00  0.00  0.00  0.00  0.10 -0.10

## $observed
## [1] -0.2
##
## $permuted
##  [1]  0.00  0.00  0.00  0.00  0.05  0.00 -0.10  0.05  0.00 -0.05

## $observed
## [1] 0
##
## $permuted
##  [1]  0.00  0.00  0.00  0.00  0.00  0.00 -0.05  0.00  0.00  0.00

## [1] 0

## [1] 0
```

**Exercise 3d**

Does it matter for your permutation test whether you permute `var` or `grouping_var`? Why or why not?

**Exercise 3e: Plotting the sampling distribution**

The set of test statistic values from the permuted samples represent the hypothesis that the two groups of observations are the same (follow the same distribution: i.e., come from the same underlying source of variability). The many permuted samples represent a good way of simulating that hypothesis. The set of test statistic values provide a window into what we would expect the test statistic to look like under that hypothesis. Given that the samples won't always be the same (because there is variability), the test statistic won't always be the same. The set of test statistic values from the permuted samples thus give us a window into the **sampling distribution** of the test statistic under this hypothesis. Taking only ten samples isn't very representative of the sampling distribution; a usually reasonable number is 10,000, which explains why (almost explains why) I used 9999 as the default value.

Start with the following code chunk to generate a larger sample of permutation test statistic values. When you compile the document, it will take a little while. Now, at the beginning of your code chunk, you will see `{r}` in curly brackets. Change this for this code chunk, and only for this code chunk, so that it says `{r, cache=T}`. This will save the result so that it doesn't get recalculated every time you preview the document. That way, it will only be slow the first time.

```
if(!exists(".Random.seed")) set.seed(NULL)
previous_seed <- .Random.seed
set.seed(1)
ptest <- permutation_twogroups(iris, "Sepal.Width", "Species", "versicolor",
                               "virginica", difference_in_medians)
set.seed(previous_seed)
```

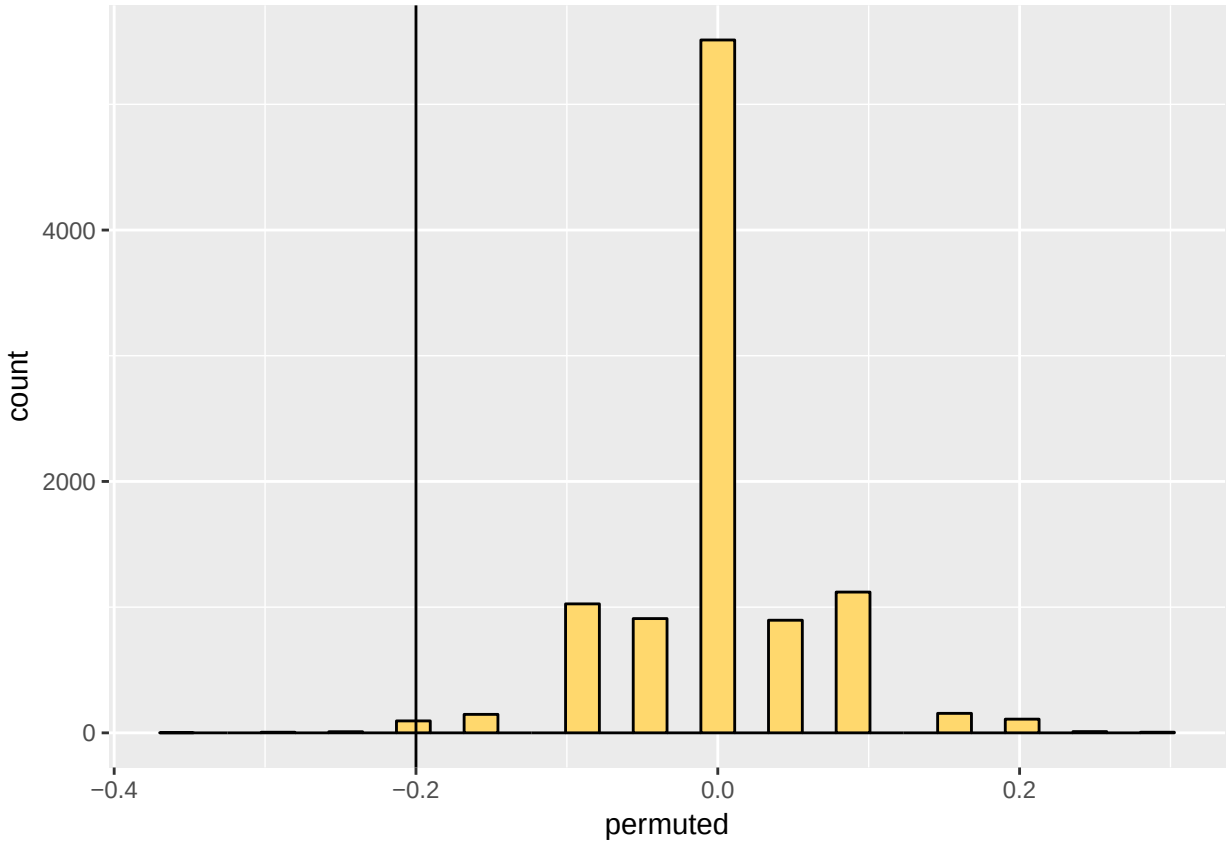
Now, add the following code in a separate chunk (underneath the previous), and fill in the blanks, using ggplot to create a histogram of the sampling distribution with a vertical line indicating the observed value of the test statistic. (A note on the first line: `ggplot` can only read from tables, and so I need to do a conversion from the list I got back from `permutation_twogroups`. That's why that line is there, and yes, there's magic in here with the `[]` that we haven't talked about, and, no, `[]` is not the same as `[[]]`. You can safely ignore the way this line works if you want.)

```
ptest_d <- tibble::as_tibble(ptest["permuted"])
# YOUR CODE HERE: plot a histogram with a vertical line at the observed value
```

Don't use my `plot_hist_fancy` function. You don't even need to write a function (although it would not be stupid to do so). Your previous `plot_hist` function won't work, since there's only one group.

You should get output something like this.

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




What do you conclude from your plot?

**Exercise 3f: Test statistics**

Write a new test statistic function that can stand in the place of `difference_in_medians`. Call it `new_test_statistic`. It can be absolutely anything you want. It can be well thought out, or it can be absurd or trivial (or it can be all three, or none of these). It doesn't need to use the grouping in the data. I suppose it doesn't even need to use the data. It just needs to take the same arguments as `difference_in_medians` and return one number. Don't forget to comment it properly. Insert this code chunk (remember: **exactly** this code chunk, variable names and everything), to make use of your new invention:

```
if(!exists(".Random.seed")) set.seed(NULL)
previous_seed <- .Random.seed
set.seed(1)
ptest_new <- permutation_twogroups(iris, "Sepal.Width", "Species", "versicolor",
                                   "virginica", new_test_statistic)
set.seed(previous_seed)
```

Then, make a histogram, as in the previous question, showing the sampling distribution and the observed value of the test statistic. Write a discussion comparing and contrasting the two histograms, and see if you can explain any differences you see.

**Exercise 3g: Calculating p-values**

Insert the following functions in your functions file.

```
permutation_pvalue_right <- function(p) {
  n_above <- sum(p$permuted >= p$observed)
  n_samples <- length(p$permuted)
  return((n_above + 1)/(n_samples + 1))
}

permutation_pvalue_left <- function(p) {
  n_below <- sum(p$permuted <= p$observed)
  n_samples <- length(p$permuted)
  return((n_below + 1)/(n_samples + 1))
}
```

Intuitively, these functions are just taking a list, of the kind returned by our `permutation_twogroups` function, and giving us back the proportion of observations that are greater than or less than the observed value of the test statistic—plus a slight adjustment (adding one in the numerator and denominator) to resolve a slight inaccuracy that would otherwise exist, which we will discuss at a later date.

Insert a code chunk here applying these two functions to the `ptest` object you saved in an earlier chunk, from which you constructed your first histogram.

If the value we observed in the real data is "extreme," in the positive direction (for the first function), or in the negative direction (for the second function), then the number we get back (the *p-value*) is a very, very small fraction (for example, 0.0001, which R will print in scientific notation as 1e-04). "Extreme" means "extreme as compared with the predictions of a hypothesis that the two groups are drawn from equivalent sources of variability." (Call that hypothesis **Hypothesis A**.) It is a not-so-small fraction (for example, 0.5) if the value we observed in the real data is entirely expected under **Hypothesis A**.

A test statistic should be constructed to respond to the needs of a specific confirmatory question, contrasting a second hypothesis, **Hypothesis B**, against **Hypothesis A**. (The test statistic you constructed in the last exercise was not constructed with any such purpose in mind, but that's an exception, because this is an exercise.) The test statistic will depend on what **Hypothesis B** is. We construct test statistics so as to give really, really small *p*-values using simulations of **Hypothesis A** when **Hypothesis B** is true, and to give not-small fractions using simulations of **Hypothesis A** when **Hypothesis A** is true. This way we can assess "how true" we think either of the two hypotheses is.

The difference in the medians statistic—as written in your function—is a test statistic that one might use under certain **Hypothesis B**s but not others. In other words, for certain **Hypothesis B**s, we would expect that, if we simulate **Hypothesis A** (this particular **Hypothesis A**, the hypothesis of same distributions), this statistic will be extreme in a certain direction when **Hypothesis B** is true.

Fill in the blanks in this discussion:

```
Let's take the example of ...
... [SOME IMAGINED DATA WITH TWO GROUPS: IT COULD BE REACTION TIMES IN A SELF-PACED
READING EXPERIMENT, OR IT COULD BE FLOWERS, IF YOU THINK YOU CAN MAKE UP HYPOTHESES
ABOUT FLOWERS].

We'd predict that the difference in the medians statistic would be extreme on the right
side if our *Hypothesis B* were that ... [EXPLAIN A HYPOTHESIS UNDER WHICH THE PREDICTION
WOULD BE TO HAVE LARGE VALUES OF difference_in_medians()].

In which case, we would use the function ... [EITHER permutation_pvalue_left OR
`permutation_pvalue_right`] ...
... because ... .
```

**Exercise 3h**

Here is some code that creates a two-group subset of the `iris` data set containing six data points in each group, and another containing thirteen data points in each group.

```
iris_subset_1 <- iris[c(89:94, 108:112),]
iris_subset_2 <- iris[88:114,]
```

Do the same permutation test as before, on `Sepal.Width`, looking at the difference in the medians for `versicolor` minus `virginica`, using each of these two subsets. Use 9999 permutations, as before. For each, plot a histogram with three vertical lines: one for each of the observed values (from each of the three iris (sub-) datasets). Put them in different colours so that we can tell the three datasets apart.

What do you notice? Compare the three histograms (you might consider putting them all on the same `x` scale using `ggplot2::xlim()` in order to better see the differences) and explain why there are the differences in the sampling distributions, and what is systematic about them.

Now compare the position of the vertical lines across the three datasets, and explain why there are differences.

Finally: given the result of the permutation test on the full dataset, I think it's reasonable to conclude that the two groups of observations don't come from the same source of variability (don't you?). Yet, the two new histograms would lead us to different conclusions about how extreme the observed statistics are. You can confirm this by calculating the appropriate p-values. One would tell us that they are all "fairly extreme," and the others would tell us that they are not so extreme. Should we change our minds? Why or why not?