R Homework #3

PUT YOUR NAME HERE

In this homework, we are going to explore R's control structures and loops (if, for, while, etc.) and related functions.

Let's start with the `if else` control structure! Like many programming languages, R has the basic `if-else` conditional control structure. Below is an example:

```
winnings <- 6

if (winnings < 0) {
    "You Lost Money!"
} else {
    "You Didn't Lose Money"
}
```

```
## [1] "You Didn't Lose Money"
```

You can also chain `if-else` statements together to add in more conditions!

```
winnings  <- 0

if (winnings < 0) {
    "You Lost Money!"
} else if(winnings == 0) {
    "You Came Out Even"
} else {
    "You Won Money!"
}
```

```
## [1] "You Came Out Even"
```

Exercise: Write an `if-else` loop to tell if a number is even or odd! Hints: Roughly, a number is even if it divisible by 2 and an odd number if it isn't. Thus, we are looking to see if there is any remainder after the division (by 2) happens. This is done using the modular operator (and has to do with modular arithmetic). The modulus operator in R is `%%`. To test out the operator, try lines of code like `3 %% 2`, `4 %% 2`, `10 %% 2` and other even and odd numbers to get a sense of what the operator does. Then assign some even or odd number to a variable and check if it is even or odd.

Related to the `if-else` structure, there is an `ifelse()` function that can be used if you don't want to take up as much space. The function isn't used that much though.

Since the `if-else` loops contain boolean tests, it will be helpful to give a quick overview. `&` and `|` are "and" and "or" in R. We've already had some experience with `==`, `>`, `!`, and other basic boolean operators. An example of using this would be

```
x <- 5

# note that you can't chain this together to get something like 0 <= x <= 10
0 <= x & x <= 10
```

```
## [1] TRUE
```

```
# the & is vectorized
c(T, T) & c(T, F)
```

```
## [1]  TRUE FALSE
```

There are also the `&&` and `||` operators which operate like their single version counterparts, except they are "short-circuting" operators. There are two concrete differences betweent the single and double versions. First, the single versions are vectorized (we'll see what that means in a little bit)and the double versions are not. Second, the double versions are smart in the evaluation. For example, the code `1 == 2 && 1 == 1` is going to evaluate to `FALSE` because 1 does not equal 2 and the `&&` operator does not even bother to evaluate the right-hand side.

Other functions that you should be aware of are `any()`, `all()`, and `which()`. `any()` and `all()` take in a set of logical tests and return whether any or all of them return `TRUE` while `which()` takes in a logical test/vector and returns the elements that are `TRUE`.

Exercise: Make a vector that is the sequence of integers from 1 to 20 and then use `which` to find the elements that are divisible by 3. Hint: Use the modulus operator again!

Now let's turn our view to the `while` loop! Like in other languages, a `while` loop will keep iterating until the logical condition is no longer met! Here is a simple example!

```
i <- 5
while (i < 10) {
    i <- i + 1
    print(i)
}
```

```
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

There are a few things to note about the code above. The first is that R does not have a special iteration operator like many other languages do (`++` or `+=`) so you have to use the code `i <- i + 1` (where you replace `i` with whatever iterator you are using.) The second thing to note about a `while` loop is that it can be dangerous in the sense that it will keep running if you condition is never met, so be careful in how you use it. There is a `break` command that you can include into the different loops if you want to break the loop.

Exercise: The `runif` function will generate a random number from a continuous uniform distribution between two numbers `a` and `b`. I want you to make a while loop that counts how many draws it takes to get a number lower than .1 from a uniform distribution with a lower bound of 0 and an upper bound of 1. Hint: The `runif` function I want you to use is `runif(1)`, which says generate 1 uniform random variate from the uniform distribution with a lower bound of 0 and upper of 1 (the 0 and 1 are defaulted.)

Now let's move onto the most popular control loop, the `for` loop! The `for` loop in R will iterate along a vector and execute code for each iteration. Here is a simple `for` loop in R that prints out the iterator at each step of the vector:

```
for (i in c(2,4,6,8)) {
    print(i)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
```

If you are trying to perform an operation along a vector, but you don't care about the vector itself, a helper function is `seq_along()`. The function will create an integer vector from 1 to the length of the vector. This is helpful when you don't necessarily know the length of the vector.

Exercise: I want you to create a vector that contains the integers from 1 to 20. Then, use a for loop to add 2 to each number. You'll have to combine your knowledge of `for` loops and vector subsetting to accomplish this.

If you read about `for` loops in R, you'll likely see that they are slow and that you shouldn't use them. That is true to a certain extent, especially when the operations you are performing inside the loop are simple. For this reason, we may want to turn something else that is faster or more descriptive than `for` loops. For us, those are special functions that are called functionals. Functionals are functions that take in a function as one of its arguments. The functionals that we will use are `lapply()` and `sapply()`. Each of these functions will take in a vector and a function, and will apply the function to each element of the vector. `lapply()` will return a list of the results and `sapply()` will try to simplify the result to a simpler data structure. Here is a simple example where I square every element of a numbers 1 through 10 with both `lapply()` and `sapply()`. Note the differences in the outputs.

```
f <- function(x) x^2

lapply(1:10, f)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
##
## [[5]]
## [1] 25
##
## [[6]]
## [1] 36
##
## [[7]]
## [1] 49
##
## [[8]]
## [1] 64
##
## [[9]]
## [1] 81
##
## [[10]]
## [1] 100
```

```
sapply(1:10, f)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

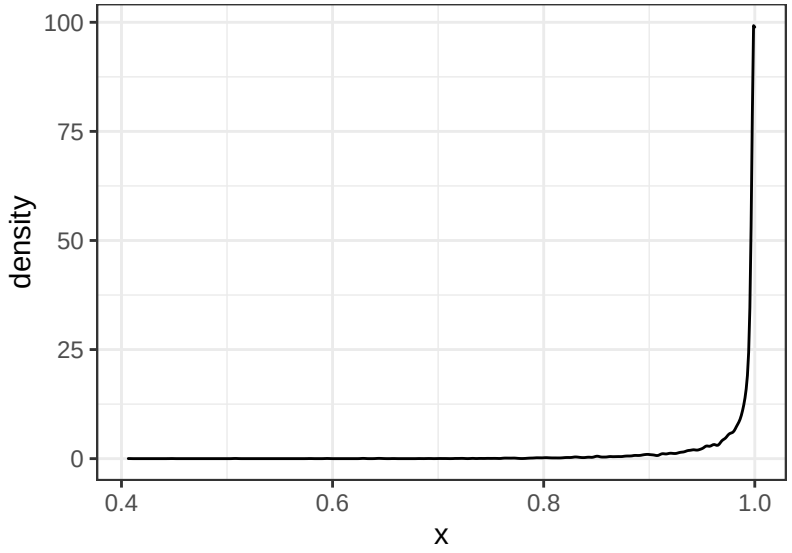
Exercise: The class of theorems that are traditionally called the Central Limit Theorem is the most important

theorem in statistics. The CLT states that, under certain conditions, the distribution of means of samples from any distribution will be approximately normal if the sample size is large enough. We can use functionals to help us test this behavior! To do so I want you to complete the following:

- Use a functional to create 10000 samples of size 100 from a beta distribution with an alpha parameter of 10 and a beta parameter of .2. To do so, you want to use `lapply()` where the first argument is `1:10000` and the second argument is `function(x) rbeta(50, 10, .2)`. Notice that the actual function does not depend on the numbers from the first argument. This is not a mistake. Also this particular function, where no actual function is defined, is called an anonymous function or a lambda function.
- Use `sapply()` to calculate the mean of each sample. The result from the first part is the first argument and the second argument is `mean`.
- Plot the result using a histogram. You'll need to make the result from step 2 a data frame in order plot using ggplot. You can use `as_tibble()` or `as.data.frame()` to automatically do the conversion or do it manually.

Here is what a $\beta(10, .2)$ distribution looks like for reference:

```
ggplot(data.frame(x=rbeta(10000, 10, .2))) + geom_density(aes(x))
```



Place you answer below: