PSTAT 10 Worksheet 1

Due 6/24/24 11:59pm

Using RMarkdown

Your TA will introduce R Markdown's basic formatting syntax along with how to run R code in an R Markdown document. For PSTAT 10, you should "knit" your result to pdf to upload to Gradescope. To knit to pdf, you must install the tinytex package:

```
install.packages('tinytex')
tinytex::install_tinytex()
```

You must also have the following in the metadata header in your RMarkdown document:

```
output: pdf_document
```

Problem 1: R Projects and `here()`

In my experience, even technologically capable students sometimes struggle with their computer's filesystem. For this class, I highly recommend you to pick a location on your computer in which to store all class files. The best practice is to have a dedicated spot for your code and not to use the desktop or downloads folder.

Traditionally, students would tell R where to run their code by getting (`getwd()`) and setting (`setwd()`) their working directory. You will find many R scripts that start with `setwd("absolute/file/path")`. Let's do this as a rite of passage:

```
getwd()
```

By default, the working directory is set to wherever you have your R Markdown file saved. If you haven't saved your file yet, the working directory will be your user folder.

You can change your working directory by typing

```
# Only run one of these, replace the path by a real one
setwd("/path/to/my/directory") # For Mac OS and Linux
setwd("c:/Documents/my/working/directory") # For Windows
```

It is in this directory that R will look for any files that it needs to run your code.

However, there is a better, more modern way to do this, which makes R projects easier to share.

Create an `R Project` by selecting `File > New Project...`. Save the project to a dedicated folder of your choice (not in `Documents` or `Downloads`). Save this R Markdown file in the same folder as your project.

Next, install and load the `here` package.

```
# install.packages("here")
library(here)
```

```
## here() starts at "path/to/your/project"
```

With this package you can use relative file paths instead of absolute file paths. This can be very helpful when you collaborate on projects. Your collaborator will not have to change their working directory. As soon as

they have loaded this project, it will *just run*.

Now, tell R where the root directory (the "lowest" of your folders) is by running `set_here()`. This reates an invisible file .here in your project directory, which tells R where your root directory is every time you open this project.

```
set_here()
```

```
## File .here already exists in "path/to/your/project"
```

Now test `here`. It converts your relative file path to an absolute one:

```
here()
```

```
## [1] "path/to/your/project"
```

```
here("week1_files")
```

```
## [1] "path/to/your/project/week1_files"
```

Yay! `here` has set our working directory without us having to get into the depths of our file system. By adding folders as arguments to `here`, we can easily navigate our folders without ever having to explicitly change the working directory.

Admittedly, this may seem a bit technical for now (and it is!) but it will help you in the long run as you pursue a career in data science where you will collaborate with others.

Problem 2: Importing data

Download the file heights.csv from the course website and place it into your working directory. Use read.csv() to read in the data from the file into an R object. `here` will look for the file `heights.csv` in the `week1_files` folder.

```
heights_df <- read.csv(here("week1_files", "heights.csv"))
summary(heights_df)
## id_. gender age height
## Min. : 1.0 Length:506 Min. :18.0 Min. :143.0
## 1st Qu.:127.2 Class :character 1st Qu.:20.0 1st Qu.:163.0
## Median :253.5 Mode :character Median :21.0 Median :171.0
## Mean :253.5 Mean :22.5 Mean :170.8
## 3rd Qu.:379.8 3rd Qu.:23.0 3rd Qu.:179.0
## Max. :506.0 Max. :61.0 Max. :200.0
```

The object `heights_df` is a *data frame*, which we will talk much more about later. For now, know that summary summarizes the data in a data frame and that the *vector* called "height" can be accessed with the dollar sign '$':

```
heights_df$height
```

Find the sum of all the heights in this data frame.

Problem 3: String concatenation

1. Print `"hello world"` to the console.
2. Run the following code

```
x <- "hello"
y <- "world"
```

In some languages, adding two strings will concatenate them. In other words, `x + y` would return `"helloworld"`. Does this work in R? Explore the functions `paste` and `paste0` to see how to concatenate strings in R.

Problem 4: Vector coercion

You might know from linear algebra that a *vector* represents change in the form of a magnitude and a direction, often visualized with an arrow drawn in space. While this idea is useful in R programming as well, in this class a vector is simply a sequence of values.

Atomic vectors are vectors in which all values are the same data type. The other type of vector is the *list*, which we may talk about later. Lists can hold elements of different data types. Create an atomic vector of the elements 1 through 10.

```
x <- 1:10
```

Now, change the fifth element of x to be the string `"cat"`. What happens to the other elements of x? This is called coercion. Try this with logical data types (TRUE, FALSE) as well and establish the coercion hierarchy (or just look it up on StackOverflow).

Fun fact: Programming languages that automatically coerce values to a given data type when incompatible data types are mixed are called *dynamically typed* (e.g. R, Python), whereas languages which will not coerce values are called *statically typed* (e.g. Java, C, C++).