CreativityV2.r: Numerical and Graphical summaries of data. Explanation of code.
Data manipulation and summaries using tidyverse commands
Graphics using ggplot2 functions.

Lines from creativityV2.r are repeated here so you know what each explanation refers to.

Do not copy and paste code from a pdf file. That works 98% of the time but not 100% of the time. The problem are those characters or character combinations that pdf files make "look pretty". Quotes are one example. They do not copy and paste correctly.

Goals of code:

- Linking add-on R packages
- Get and set the working directory
- Piping)
- Calculate summary statistics
  - For all observations or for individual groups of observations
- Display the creativity data (dotplot, histogram, boxplot)
  - As one group or for each treatment
- Saving results to copy to Word

Confusing things:
Trying to copy code from the pdf file. Don't. Copy from the associated code file or open that file and execute commands from that file.
Setting the working directory so data files can be found and saved plots go where you can locate them.
Remembering that capitalization matters

`# creativity2.r: numerical and graphical summaries of data`
The # symbol starts a comment. All text from the # to the next end-of-line is ignored.

Linking add-on R libraries, also called packages:
R is open-source software with a huge number (currently almost 20,000) of add-on packages contributed by users. We will use various of these in Stat 587. The creativityV2.r code will use functions in two of these packages.

There are two steps to be able to use an add-on packages:
1) `install.packages('dplyr')` copy the package from a repository to your computer. This only needs to be done once (at least until you install a new version of R). Note that the package name is in quotes here. The code in creativity1.r has this command commented out because you only need to run it once. Copy or type the command without the # to do it. and run the command.

2) `library(dplyr)` link that package to your current R session. This needs to be executed every time you start an R session. Functions in the add-on library will only be available if you link that library. Note that the library name **is not in quotes** here.

Why do you have to link a library every time? Sometimes an R library will include functions with the same name as functions in another library or in base R. For example, the function `union{}` is defined in base R and again in dplyr. Using all libraries at the same time would be chaos. By default, R will use the function in the last defined library when there are name conflicts. For example, when you library(dplyr) you (should) see a message that the

`getwd()`
Print the default working directory. If your desired working directory is a folder under this, you can use a relative path in the setwd().

`setwd('name of folder')`
Set the working directory. Can use a relative path or an absolute path, i.e., starting at `c:` or some other drive. R will look for code and data or save files in the specified directory. The Rstart document describes working directories and how to set them.

I strongly recommend creating a folder for all your Stat 587 work. Then create a separate folder for each project you work on.

On a Windows machine, R will start with the working directory in your Documents folder. The full path there is `c:/Users/XX/Documents` where XX is your username on that computer. This may be your netid but it may not. Here's how to change the working directory to a folder in Documents; my example uses Stat587 as that folder name.

```
setwd('Stat587')
```

I do not keep many files in Documents, so my setwd() looks different.

`creativity <- read.csv('creativity.csv', as.is=T)`
Read the csv format file and create the creativity data frame. Details in the Rstart document.

`names(creativity)`
Print names of the variables in creativity. Not essential. Helpful in plain R; the data window in RStudio gives you this information automatically.

`head(creativity)`
Print the first 6 lines of a data frame. Useful to check that it's been read correctly.

`View(creativity)`
RStudio only. Open an interactive window (like a spreadsheet) to view the contents of a data frame.

Summary statistics on a variable:
`summary(creativity$score)`
Print a 6 number summary of the values in creativity$score. See the Rstart document if you don't

recognize creativity$score as the score variable in the creativity data frame. You can also summarize all variables in a data frame by omitting the specific variable name, e.g., `summary(creativity)`.

Piping:
Lines of R code can often become hard to read. Piping is a way to organize code so it's easier to read. The piping operator is `%>%` without any spaces between the 3 symbols. If you're familiar with piping in other computer languages and operating systems, this exactly the same. The pipe evaluates the left-hand side. The result of that is passed as the first argument to the function on operation on the right-hand side. (I'm omitting details about order of arguments because they usually don't matter).

`creativity %>% head()`:
or its two line equivalent in the code. The left-hand side "prints" the contents of the creativity data set. (Remember in the Rstart document, if you don't save something into a named object, it gets printed. Instead of getting printed in the console window, the pipe passes that information into the `head()` function. So these two lines of code do the same thing as `head(creativity)` above.

Important note: If you break the command into multiple lines, it is easier to read. Notice that the `%>%` is at the end of the line. That is crucial. If you tried to run:

```
creativity
%>% head()
```

you'll get a printout of everything in the creativity data set followed by an error that is hard to interpret.

What's going on? Remember that a set of R commands can continue on multiple lines only when R expects more to come. In the bad version, `creativity` is a stand-alone command. `creativity %>%` is not - R expects something on the right-hand side of the pipe. You"ll see a + when the pair of commands is echoed on the console.

Note: If breaking commands onto multiple lines is confusing, you can always put everything on a single line. My experience is that a single line command is much harder to read. Remember you may need to look at your own code again 6 or 12 months after you wrote it.

select():
This dplyr function selects the specified variables in a data frame. It takes a data frame, e.g. piped into it, and returns a new data frame with only the specified variables. So `creativity %>% select(score)` returns a data frame with only the specified variables.

Advanced uses: select() allows all sorts of special ways to specify variables, not just by name. See texts or online resources on data management in R for details.

`creativity %>% select(score) $>$ head()`: See if you can predict what this does before trying it.

`summarize()`:
takes multiple observations and returns a summary statistic, e.g., a mean or a median. Each summary is requested by strings of `new variable name = function( old variable name)`. So `summarize(meanScore = mean(score))` will create a new data frame with one new variable `meanScore` that is the average of all the values of `score`.

You can create summaries of multiple variables or multiple summaries of the same variable by adding additional requests, **separated by commans** inside the summarize( ). Look carefully at where the parentheses are.

Commment: What we've just done using dplyr functions can done as easily or more easily using base R operations. That's going to change right ow.

Summary statistics for subgroups:
The creativity data set has data on 2 treatments. You probably want to see the mean for each treatment. dplyr functions make this easy and return a very interpretable data set.

`group_by{}` Adds grouping information to a data frame. Subsequent operations are done on each group of observations, not on all observations. To get the mean and median score for each treatment, just insert `group_by()` in the chain of pipes between the data frame name and the summarize() operations.

R graphics using ggplot2:
ggplot2 is an add-on library that provides extremely powerful way to produce graphs. Like most powerful systems, it can be a bit intimidating to get started. We'll start with simple things.

I will use ggplot2 and ggplot interchangeably. The language is called ggplot and was first implemented in the ggplot library. ggplot2 is an updated library.

Basic structure of ggplot commands:
Three parts: a data frame name, an aesthetic, and one or more plot command. The first two are arguments to the ggplot() function. The last is done by functions that start with `geom_`. The output from the ggplot() function is passed to the plot function. The two parts are separated by a `+`.

Aesthetics: These are specified by the aes() function. The minimum contents are the names of the x and y variables. The aes() function is also where you specify colors and other options. Examples are below

Advanced note: If this sounds like piping, it is. The `+` does the same thing as the `>` pipe operator. ggplot() was written before piping was written. Folks are revising ggplot to use pipes, but that is in the future.

vertical dotplot of one group of data: `geom_dotplot()`
The setup ggplot() function specifies the data frame: creativity. Since this code produces a vertical dotplot, the `aes()` function names a y variable. Since an x variable is required but not used for a single plot, we specify x = 1.

`geom_dotplot()` draws a dot plot. The best-looking ones produced by binning values and centering them in the plot. Those features are specified by `binaxis='y', stackdir='center'`. You're welcome to play with these arguments, but I recommend you just do them as given. If you want to know more or look at alternatives, look in the documentation.

Important: the +. Notice the ggplot() command ends on the 3rd line of the common before the +. The + tells R that the command is incomplete and to expect more. The plotting command is on the next line.

Separating setup from plotting:
If you will use data + aesthetic for multiple plots, you can save the results of the data + aesthetic. `creativity.setup1` saves the result for the creativity data set with the score as the x axis. An x variable is required (even if ignored), but y variable doesn't need to specified if not needed.

That saved setup can be used for multiple plots. For example:

Histogram: `creativity.setup1 + geom_histogram()`:
draws a histogram with scores on the x axis and frequency on the y.

Horizontal boxplot: `creativity.setup1 + geom_boxplot()`:
draws a horizontal box plot (because score is on the x axis@

Boxplot: `creativity.setup1 + coord_flip() + geom_boxplot()`
flips the x and y axis (so score is now on the y axis) and draws a box plot. This is probably the box plot you want to draw.

Multiple groups: `aes(x=treatment, y = score)`
Goal: Draw side-by-side plots for the two treatments. We want treatment on the X axis and score on the y axis. Once you've defined that aesthetic, the actual plotting is identical to before.

Cleaning up the plot: `theme_classic()`
The default ggplot graphic uses a grey background with white marker lines. Graphs for publications usually don't have these. The ggplot system includes multiple themes that control overall features of a graph. `theme_classic()` is the theme for graphics without bells and whistles. insert that between the setup and the plotting function.

Drawing histograms for multiple groups: `aes( fill=treatment, color=treatment)`
The resulting histograms are best if drawn with different colors for each group. I find the clearest histograms are drawn using filled in bars and coloring them by the treatment. This is done by specifying the fill=treatment and color=treatment options in the aesthetic. I also find it clearest if the two groups are drawn side-by-side instead of overplotting them. This is done by the position='dodge' argument to the `geom_histogram()` function.

Using grey shades instead of colors: `scale_color_grey() + scale_fill_grey()`
This is done using two functions to change the color scheme to grey and the fill color to grey shades.

Moving plots into a Word document

There are two (at least) ways to do this. Some of of the details depend on whether you are using plain R or RStudio.

`ggsave('histogram.emf')`
(either R or RStudio). Saves the current plot in the specified file. File goes in your working directory unless you add a path to the name. You must provide the file extension (e.g., `.emf`), which is what Windows uses to identify the contents. `'emf'` is an enhanced metafile. This and TIFF files (`.'tiff'` file extensionare the two that I find go into Word most easily. In Word, Include / picture will load the picture.

RStudio alternatives:
After plotting something, the lower right window should be a Plots window and show the graph. Look for the Export button in the 2nd menu bar. If not there, click Plots and you should see the Export button.
1) Click Export / Copy to Clipboard. You can then paste the picture into Word.
2) Click Export / Save as image. The dialog box allows you to specify the format, the directory, and the filename. The metafile format is very useful for moving to Word. The directory is your working directory, but you change this by clicking the Directory button. The file name is Rplot by default. You should provide a more descriptive name. You can also change the plot size. You can resize plots in Word, but resizing before saving usually looks better.