Exploring Week 1 Content

Thursday

Introduction to Today

The goal for today is to put into practice the lecture topics we have gone over thus far.

R Tip of the Day: Copying & pasting is your best friend! If I have done something similar before, I copy code and adapt it to my new purpose. This will end up saving you lots of time in the long run.

Loading Our Libraries

First, we have to load our "libraries". A **library** in R, as a reminder, is an open-source package created by a very kind individual that contains functions (short cuts) to get things done in R.

```r
library(car)
library(ggplot2)
library(pastecs)
library(psych)
library(gridExtra)
library(kableExtra)
```

Loading in Our Festival Data

Next We will load in some data! I have uploaded the data files for you into the cloud, so you can just run the below code.

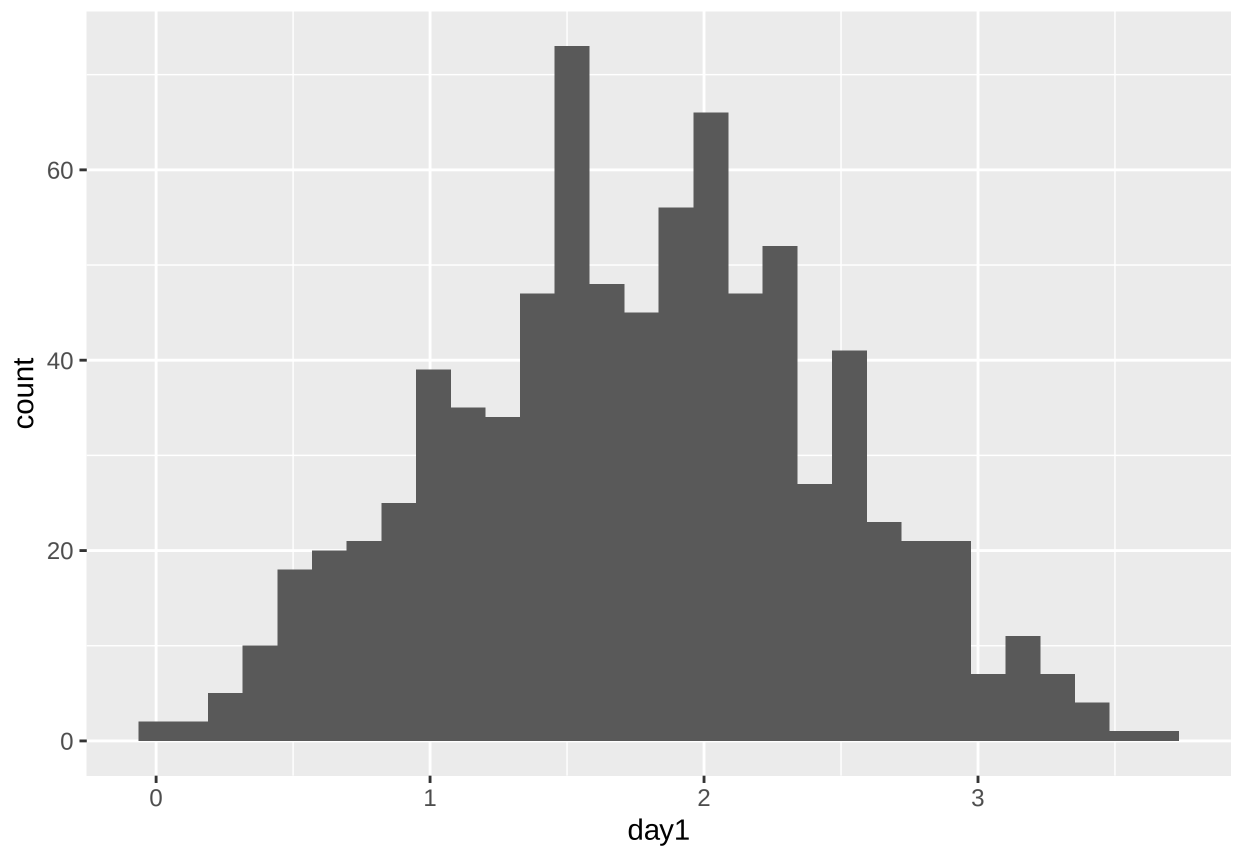
```r
#Read in the download data:
dlf <- read.delim("DownloadFestival.dat", header=TRUE)

#Remove the outlier from the day1 hygiene score
dlf$day1 <- ifelse(dlf$day1 > 20, NA, dlf$day1)
```

Plotting our data

Last time we worked with this dataset, we noticed an outlier (extreme value) on our hygiene variable. Let's comfirm with a histogram

```r
ggplot(dlf,(aes(day1))) + geom_histogram()
```



Yikes! Looks like someone scored 20 on hygeine. Let's go ahead and remove the outlier.

Here I am going to break down the below code cunk:

dlf$day1: This piece of code displays *indexing*. We use the $ to tell R we want to grab the day1 data from the dlf dataset. Meaning, we want to access the day 1 hygiene data from our Download Festival dataset.

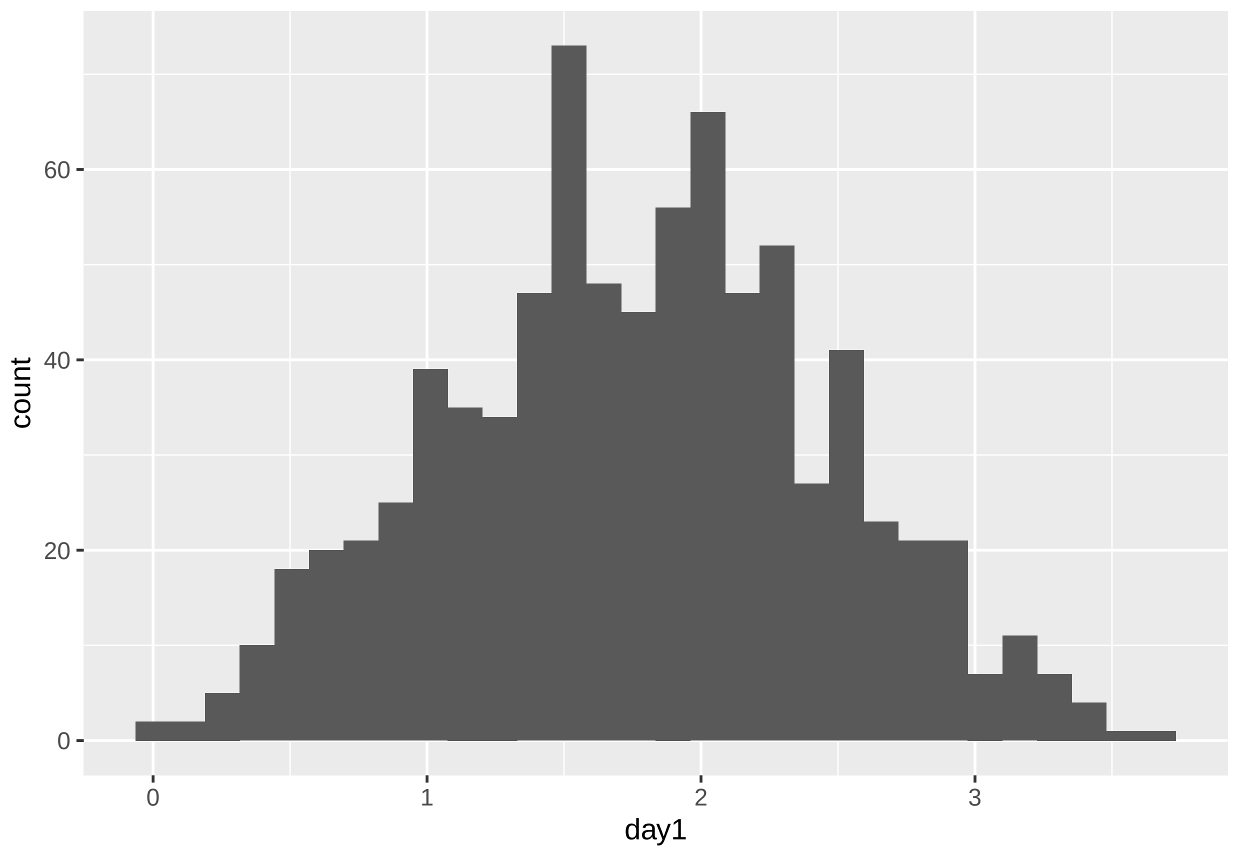
<- is what we call an assignment operator. We currently have an outlier (score 20) in our day1 variable. We are using this arrow to assign whatever is of the right of the arrow into whatever is on the left of the arrow (you could also use a '=' sign). This means what we are going to overwrite our original data with whatever code we have on the right. (More on the <- down below at the end of the script)

ifelse is a function that has to parts, an "if" and an "else". Here, it is saying, if the value of day 1 hygiene is greater than or equal to 20 (which our outlier is), then assign the number value (originally 20) an *"NA"*. This will effectively remove it for the dataset. The final part says, for everything else keep the original value of day 1 hygiene.

```r
dlf$day1 <- ifelse(dlf$day1 >= 20, NA, dlf$day1)
```

Let's run the same histogram code again to verify we got rid of the outlier.

```r
ggplot(dlf,(aes(day1))) + geom_histogram()
```



Woohoo! No more outlier. Now I want to introduce a nifty plotting function. I want to plot 3 histograms all side by side. By default, the plotting window is 1x1 meaning it will plot one plot at a time. The *grid.arrange(p1, p2, ncol = X)* function allows us to adjust the plotting window.

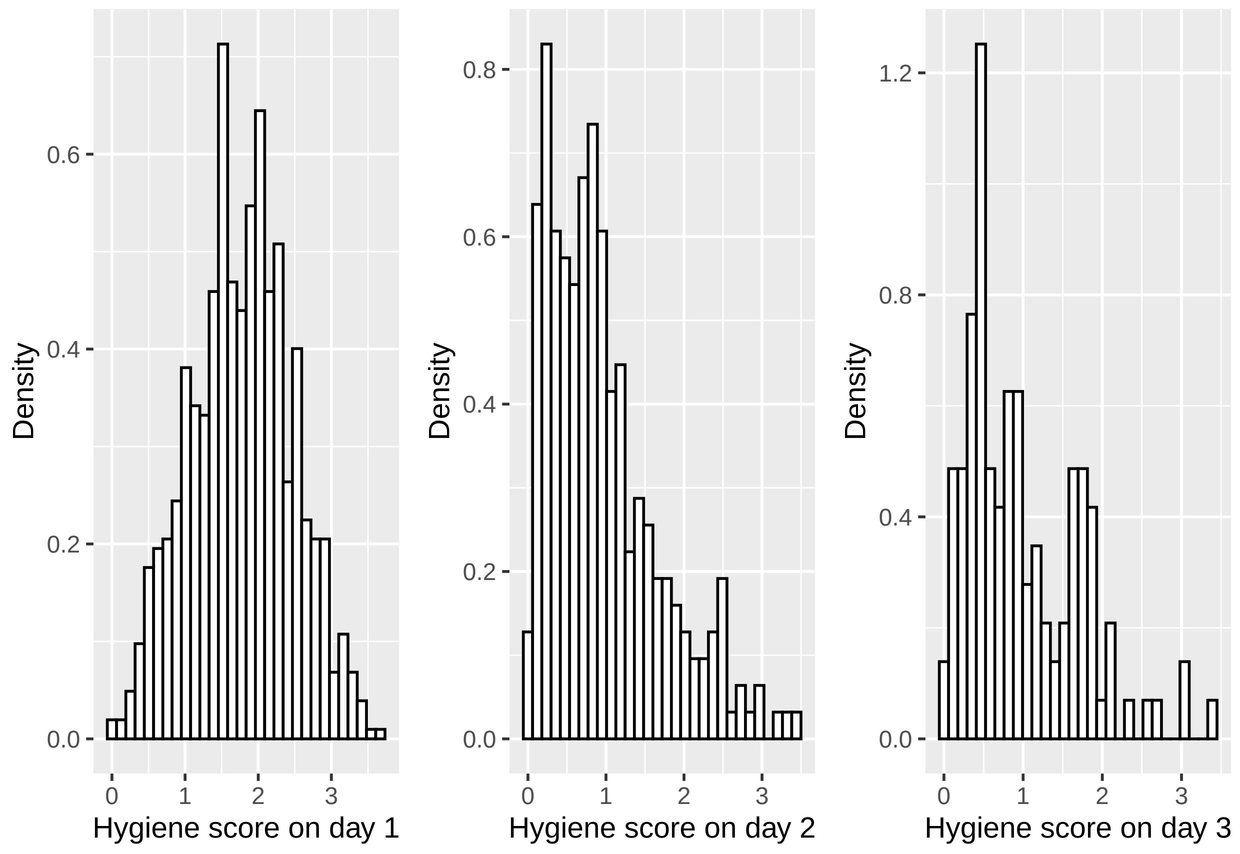
Adjusting the Plotting Window

```r
#Histogram for day 1:
hist.day1 <- ggplot(dlf, aes(day1)) + theme(legend.position = "none") +
  geom_histogram(aes(y=..density..), colour="black", fill="white") +
  labs(x="Hygiene score on day 1", y = "Density")


#Histogram for day 2:
hist.day2 <- ggplot(dlf, aes(day2)) + theme(legend.position = "none") +
  geom_histogram(aes(y=..density..), colour="black", fill="white") +
  labs(x="Hygiene score on day 2", y = "Density")


#Histogram for day 3:
hist.day3 <- ggplot(dlf, aes(day3)) + theme(legend.position = "none") +
  geom_histogram(aes(y=..density..), colour="black", fill="white") +
  labs(x="Hygiene score on day 3", y = "Density")


#This tell us what we want our plotting window to have 1 row with 3 columns.
grid.arrange(hist.day1, hist.day2, hist.day3, ncol = 3)
```



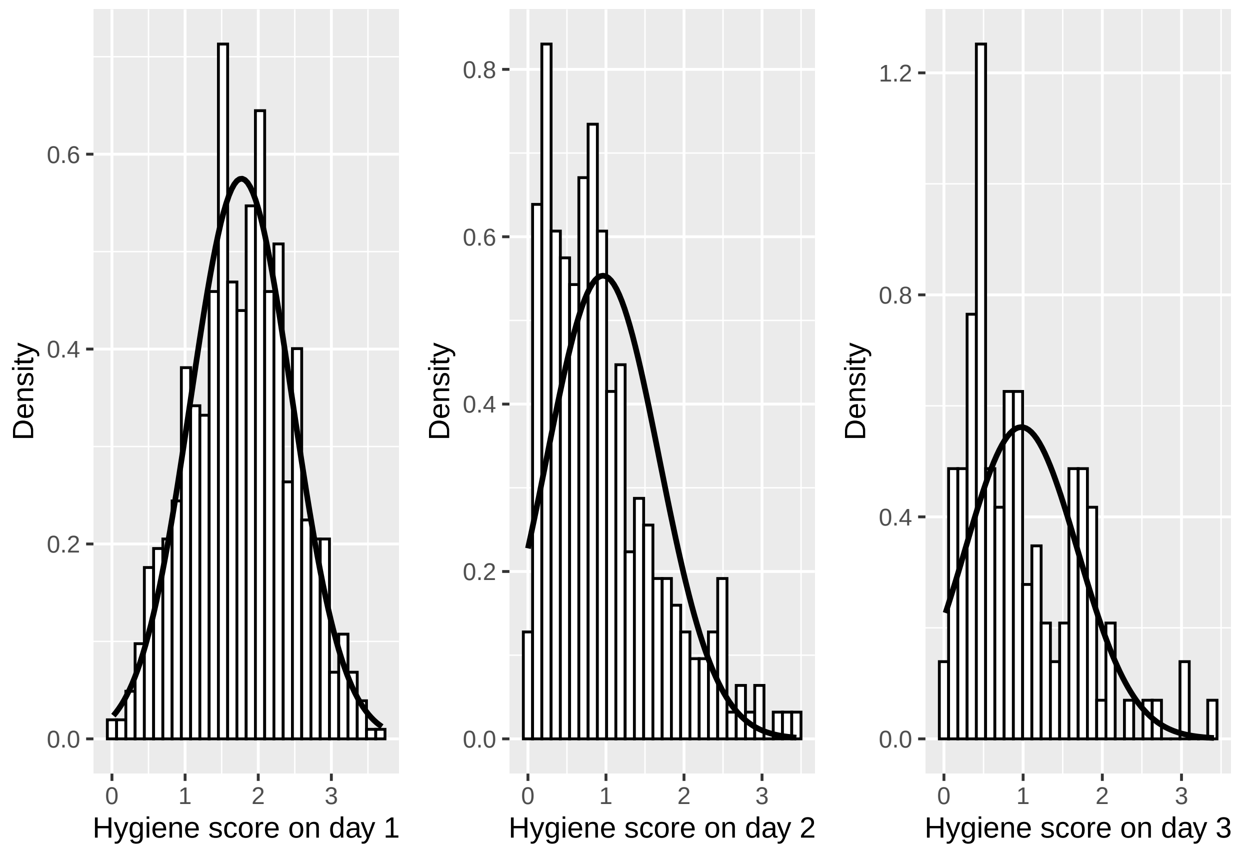
Adding Curves to our graph

```r
#Add the curves to the Histograms:
hist.day1.curve <- hist.day1 +
  stat_function(fun = dnorm, args = list(mean = mean(dlf$day1, na.rm = TRUE),
                                         sd = sd(dlf$day1, na.rm = TRUE)),
                colour = "black", size = 1)


hist.day2.curve <- hist.day2 +
  stat_function(fun = dnorm, args = list(mean = mean(dlf$day2, na.rm = TRUE),
                                         sd = sd(dlf$day2, na.rm = TRUE)),
                colour = "black", size = 1)


hist.day3.curve <- hist.day3 +
  stat_function(fun = dnorm, args = list(mean = mean(dlf$day3, na.rm = TRUE),
                                         sd = sd(dlf$day3, na.rm = TRUE)),
                colour = "black", size = 1)

#This tell us what we want our plotting window to have 1 row with 3 columns.
grid.arrange(hist.day1.curve, hist.day2.curve, hist.day3.curve, ncol = 3)
```



Numerically: Describe the Data

Using the describe() function for a single variable.

```
describe(dlf$day1)
```

```
##    vars   n mean   sd median trimmed mad  min  max range skew kurtosis
## X1    1 809 1.77 0.69   1.79    1.77 0.7 0.02 3.69  3.67    0    -0.42
##      se
## X1 0.02
```

Pro-tip, use “kable” to “prettify” your tables.

```
kable(describe(dlf$day1)) %>% kable_styling(latex_options="scale_down")
```

	vars	n	mean	sd	median	trimmed	mad	min	max	range	skew	kurtosis	se
X1	1	809	1.770828	0.6939127	1.79	1.771495	0.696822	0.02	3.69	3.67	-0.0031554	-0.4239914	0.0243967

For normality, we would look at the skew and kurtosis.

A general guideline for skewness is that if the number is greater than +1 or lower than −1, this is an indication of a substantially skewed distribution. For kurtosis, the general guideline is that if the number is greater than +1, the distribution is too peaked.

Using the describe() and stat.desc() function to describe multiple variables

```
#Using describe
kable(describe(dlf[,c("day1", "day2", "day3")])) %>% kable_styling(latex_options="scale_down")
```

	vars	n	mean	sd	median	trimmed	mad	min	max	range	skew	kurtosis	se
day1	1	809	1.7708282	0.6939127	1.79	1.7714946	0.696822	0.02	3.69	3.67	-0.0031554	-0.4239914	0.0243967
day2	2	264	0.9609091	0.7207801	0.79	0.8687264	0.607866	0.00	3.44	3.44	1.0828112	0.7554615	0.0443609
day3	3	123	0.9765041	0.7102770	0.76	0.9021212	0.607866	0.02	3.41	3.39	1.0078127	0.5945454	0.0640435

```
#Using stat.desc

kable(round(stat.desc(dlf[, c("day1", "day2", "day3")], basic = FALSE, norm = TRUE), digits = 3))
```

	day1	day2	day3
median	1.790	0.790	0.760
mean	1.771	0.961	0.977
SE.mean	0.024	0.044	0.064
CI.mean.0.95	0.048	0.087	0.127
var	0.482	0.520	0.504
std.dev	0.694	0.721	0.710
coef.var	0.392	0.750	0.727
skewness	-0.003	1.083	1.008
skew.2SE	-0.018	3.612	2.309
kurtosis	-0.424	0.755	0.595
kurt.2SE	-1.235	1.265	0.686
normtest.W	0.996	0.908	0.908
normtest.p	0.032	0.000	0.000

Z-Scores

A *fun* example adapted from YaRrr (an R Book).

A common task in statistics is to standardize variables – also known as calculating z-scores. The purpose of standardizing a vector (vector = r lingo for a simple data structure) is to put it on a common scale which allows you to compare it to other (standardized) variables. To standardize a vector, you simply subtract the vector by its mean, and then divide the result by the vector's standard deviation.

If the concept of z-scores is new for you in this class, don't worry. In the next worked example, you'll see how it can help you compare two sets of data. But for now, let's see how easy it is to standardize a vector using basic arithmetic (and R as our super fancy super calculator!).

Let's say you have a vector a containing some data. We'll assign the vector to a new object called a then calculate the mean and standard deviation with the *mean()* and *sd()* functions:

```
a <- c(5, 3, 7, 5, 5, 3, 4)
mean(a)
```

```
## [1] 4.571429
```

```
sd(a)
```

```
## [1] 1.397276
```

Ok. Now we'll create a new vector called *a.z* which is a standardized version of a. To do this, we'll simply subtract the mean of the vector, then divide by the standard deviation.

```
a.z <- (a - mean(a)) / sd(a)
```

Now let's look at the standardized values:

```
a.z
```

```
## [1]  0.3067192 -1.1246370  1.7380754  0.3067192  0.3067192 -1.1246370
## [7] -0.4089589
```

Based on what we know about z-scores, the mean of a.z should now be 0, and the standard deviation of a.z should now be 1. Let's make sure:

```
mean(a.z)
```

```
## [1] 1.982309e-16
```

```
sd(a.z)
```

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

Sweet, it worked!. Oh, don't worry that the mean of a.z doesn't look like exactly zero. Using non-scientific notation, the result is 0.000000000000000198. For all intents and purposes, that's 0. The reason the result is not exactly 0 is due to computer science theoretical reasons that I cannot explain (because…I don't understand them).

Z-scores in Practice: Evaluating a Pirate Competition

Your gluten-intolerant first mate just perished in a tragic soy sauce incident and it's time to promote another member of your crew to the newly vacated position. Of course, only two qualities really matter for a pirate: *rope-climbing*, and *grogg drinking*.

Therefore, to see which of your crew deserves the promotion, you decide to hold a climbing and drinking competition. In the climbing competition, you measure how many feet of rope a pirate can climb in an hour. In the drinking competition, you measure how many mugs of grogg they can drink in a minute. Five pirates volunteer for the competition – here are their results:

```
names.pirates <- c("Heidi","Andrew", "Becki", "Madison", "David")
grogg <- c(12, 8, 1, 6, 2)
climbing <- c(100, 520, 430, 200, 700)

cbind(names.pirates, grogg, climbing)
```

```
##      names.pirates grogg climbing
## [1,] "Heidi"       "12"  "100"
## [2,] "Andrew"      "8"   "520"
## [3,] "Becki"       "1"   "430"
## [4,] "Madison"     "6"   "200"
## [5,] "David"       "2"   "700"
```

Now we've got the data, but there's a problem: the scales of the numbers are very different. While the grogg numbers range from 1 to 12, the climbing numbers have a much larger range from 100 to 700. This makes it difficult to compare the two sets of numbers directly.

To solve this problem, we'll use standardization. Let's create new standardized vectors called *grogg.z* and *climbing.z*

```
#subtracting each individual grogg score from the mean and then dividing by the standard deviation.
grogg.z <- (grogg - mean(grogg)) / sd(grogg)

#subtracting each individual climbing score from the mean and then dividing by the standard deviation.
climbing.z <- (climbing - mean(climbing)) / sd(climbing)
```

Now, let's look at the final results

```
grogg.z
```

```
## [1]  1.37948189  0.48949358 -1.06798598  0.04449942 -0.84548890
```

```
climbing.z
```

```
## [1] -1.1969581  0.5365674  0.1650977 -0.7842139  1.2795069
```

It looks like there were two outstanding performances in particular.

In the grogg drinking competition, the first pirate (Heidi) had a z-score of 1.4. We can interpret this by saying that Heidi drank 1.4 more standard deviations of mugs of grogg than the average pirate.

In the climbing competition, the fifth pirate (David) had a z-score of 1.3. Here, we would conclude that David climbed 1.3 standard deviations more than the average pirate.

But which pirate was the best on average across both events?

To answer this, let's create a combined z-score for each pirate which calculates the average z-scores for each pirate across the two events. We'll do this by adding two performances and dividing by two. This will tell us, how good, on average, each pirate did relative to her fellow pirates.

```
average.z <- (grogg.z + (climbing.z)) / 2
```

Let's look at the result:

```
round(average.z, 1)
```

```
## [1]  0.1  0.5 -0.5 -0.4  0.2
```

The highest average z-score belongs to the second pirate (Andrew) who had an average z-score value of 0.5. The first and last pirates, who did well in one event, seemed to have done poorly in the other event.

Moral of the story: promote the pirate who can drink and climb

Extra Tips

Missing (NA) values

In R, missing data are coded as *NA*. In real datasets, NA values turn up all the time. Unfortunately, most descriptive statistics functions will freak out if there is a missing (NA) value in the data. For example, the following code will return NA as a result because there is an NA value in the data vector:

```
a <- c(1, 5, NA, 2, 10)
mean(a)
```

```
## [1] NA
```

Thankfully, there's a way we can work around this. To tell a descriptive statistic function to ignore missing (NA) values, include the argument *na.rm = TRUE* in the function. This argument explicitly tells the function to ignore NA values. Let's try calculating the mean of the vector a again, this time with the additional *na.rm = TRUE* argument:

```
mean(a, na.rm = TRUE)
```

```
## [1] 4.5
```

Now, the function ignored the NA value and returned the mean of the remaining data.

While this may seem trivial now (why did we include an NA value in the vector if we wanted to ignore it?!), it will be become very important when we apply the function to real data which, very often, contains missing values.

Objects and functions

To understand how R works, you need to know that R revolves around two things: **objects** and **functions**. Almost everything in R is either an object or a function. In the following code chunk, I'll define a simple object called tattoos using a function *c()* (stands for concatonate but I think of it as 'combine'):

Create a vector object called tattoos:

```
tattoos <- c(4, 67, 23, 4, 10, 35)
```

Apply the mean() function to the tattoos object:

```
mean(tattoos)
```

```
## [1] 23.83333
```

Objects

What is an object? An **object** is a thing – like a number, a dataset, a summary statistic like a mean or standard deviation, or a statistical test. Objects come in many different shapes and sizes in R. There are simple objects like which represent single numbers, vectors (like our tattoos object above) which represent several numbers, more complex objects like dataframes which represent tables of data, and even more complex objects like hypothesis tests or regression which contain all sorts of statistical information.

Different types of objects have different attributes. For example, a vector of data has a length attribute (i.e.; how many numbers are in the vector), while a hypothesis test has many attributes such as a test-statistic and a p-value. Don't worry if this is a bit confusing now – it will all become clearer the more we play with R. For now, just know that objects in R are things, and different objects have different attributes.

Functions

What is a function? A **function** is a procedure that typically takes one or more objects as arguments (aka, inputs), does something with those objects, then returns a new object. For example, the mean() function we used above takes a vector object, like tattoos, of numeric data as an argument, calculates the arithmetic mean of those data, then returns a single number (a scalar) as a result. Thankfully, R has hundreds (thousands?) of built-in functions that perform most of the basic analysis tasks you can think of.

99% of the time you are using R, you will do the following: 1) Define objects. 2) Apply functions to those objects. 3) Repeat!. Seriously, that's about it. However, as you'll soon learn, the hard part is knowing how to define objects they way you want them, and knowing which function(s) will accomplish the task you want for your objects.

Numbers versus Characters

For the most part, objects in R come in one of two flavors: *numeric* and *character*. It is very important to keep these two separate as certain functions, like mean(), and max() will only work for numeric objects, while functions like grep() and strtrim() only work for character objects.

A numeric object is just a number like 1, 10 or 3.14. You don't have to do anything special to create a numeric object, just type it like you were using a calculator.

These are all numeric objects:

```
1
```

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

```
10
```

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

```
3.14
```

```
## [1] 3.14
```

A character object is a name like "Madison", "Brian", or "University of Konstanz". To specify a character object, you need to include quotation marks "" around the text.

These are all character objects:

```
"Madison"
```

```
## [1] "Madison"
```

```
"Brian"
```

```
## [1] "Brian"
```

```
"10"
```

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

If you try to perform a function or operation meant for a numeric object on a character object (and vice-versa), R will yell at you. For example, here's what happens when I try to take the mean of the two character objects "1" and "10":

```
# This will return an error because the arguments are not numeric!
mean(c("1", "10"))
```

```
## Warning in mean.default(c("1", "10")): argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

If I make sure that the arguments are numeric (by not including the quotation marks), I won't receive the error:

```r
# This is ok!
mean(c(1, 10))
```

```
## [1] 5.5
```

Creating new objects with <-

By now you know that you can use R to do simple calculations. But to really take advantage of R, you need to know how to create and manipulate objects. All of the data, analyses, and even plots, you use and create are, or can be, saved as objects in R. For example the download festival dataset which we've used before is an object stored in our workspace. Once the object was loaded, we could use it to calculate descriptive statistics, hypothesis tests, and to create plots.

To create new objects in R, you need to do object assignment. Object assignment is our way of storing information, such as a number or a statistical test, into something we can easily refer to later. This is a pretty big deal. Object assignment allows us to store data objects under relevant names which we can then use to slice and dice specific data objects anytime we'd like to.

To do an assignment, we use the almighty <- operator called assign To assign something to a new object (or to change an existing object), use the notation: 'object <- …', where "object" is the new (or updated) object, and "…" is whatever you want to store in object. Let's start by creating a very simple object called a and assigning the value of 100 to it.

Good object names strike a balance between being easy to type (i.e.; short names) and interpret. If you have several datasets, it's probably not a good idea to name them a, b, c because you'll forget which is which. However, using long names like March2015Group1OnlyFemales will give you carpal tunnel syndrome.

```r
# Create a new object called a with a value of 100
a <- 100
```

Once you run this code, you'll notice that R doesn't tell you anything. However, as long as you didn't type something wrong, R should now have a new object called a which contains the number 100. You can see it in your "Enrivonment" window. If you want to see the value, you need to call the object by just executing its name. This will print the value of the object to the console:

```r
# Print the object a
a
```

```
## [1] 100
```

Now, R printed the value of a (in this case 100). If you try to evaluate an object that is not yet defined, R will return an error. For example, let's try to print the object b which we haven't yet defined:

```r
b
```

```
## Error in eval(expr, envir, enclos): object 'b' not found
```

As you can see, R yelled at us because the object b hasn't been defined yet.

Once you've defined an object, you can combine it with other objects using basic arithmetic. Let's create objects a and b and play around with them.

```
a <- 1
b <- 100

# What is a + b?
a + b
```

```
## [1] 101
```

Assign a + b to a new object (c)

```
c <- a + b
```

What is c?

```
c
```

```
## [1] 101
```

To change an object, you must assign it again!

Normally I try to avoid excessive emphasis, but because this next sentence is so important, I have to just go for it. Here it goes...

To change an object, you assign it again!

No matter what you do with an object, if you don't assign it again, it won't change. For example, let's say you have an object z with a value of 0. You'd like to add 1 to z in order to make it 1. To do this, you might want to just enter z + 1 - but that won't do the job. Here's what happens if you don't assign it again:

```
z <- 0
z + 1
```

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

Ok! Now let's see the value of z

```
z
```

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

Damn! As you can see, the value of z is still 0! What went wrong? Oh yeah...

To change an object, you must assign it again!

The problem is that when we wrote z + 1 on the second line, R thought we just wanted it to calculate and print the value of z + 1, without storing the result as a new z object. If we want to actually update the value of z, we need to reassign the result back to z as follows:

```
z <- 0
z <- z + 1  # Now I'm REALLY changing z
z
```

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

Phew, z is now 1. Because we used assignment, z has been updated. About freaking time.

How to name objects

You can create object names using any combination of letters and a few special characters (like . and _). Here are some valid object names

```
# Valid object names
group.mean <- 10.21
my.age <- 32
FavoritePirate <- "Jack Sparrow"
sum.1.to.5 <- 1 + 2 + 3 + 4 + 5
```

All the object names above are perfectly valid. Now, let's look at some examples of invalid object names. These object names are all invalid because they either contain spaces, start with numbers, or have invalid characters:

```
# Invalid object names!
famale ages <- 50 # spaces
5experiment <- 50 # starts with a number
a! <- 50 # has an invalid character
```

If you try running the code above in R, you will receive a warning message starting with *Error: unexpected symbol*

Anytime you see this warning in R, it almost always means that you have a naming error of some kind.

R is case-sensitive!

Like a text message, you should probably watch your use of capitalization in R. Like English, R is case-sensitive – it R treats capital letters differently from lower-case letters. For example, the four following objects Plunder, plunder and PLUNDER are totally different objects in R:

```
# These are all different objects
Plunder <- 1
plunder <- 100
PLUNDER <- 5
```

I try to avoid using too many capital letters in object names because they require me to hold the shift key. This may sound silly, but you'd be surprised how much easier it is to type mydata than MyData 100 times.

Applying these tips, Example: Pirates of The Caribbean

Let's do a more practical example – we'll define an object called blackpearl.usd which has the global revenue of Pirates of the Caribbean: Curse of the Black Pearl in U.S. dollars. A quick Google search showed me that the revenue was $634,954,103. I'll create the new object using assignment:

```r
blackpearl.usd <- 634954103
```

Now, my fellow European pirates might want to know how much this is in Euros. Let's create a new object called *blackpearl.eur* which converts our original value to Euros by multiplying the original amount by 0.88 (assuming 1 USD = 0.88 EUR)

```r
blackpearl.eur <- blackpearl.usd * 0.88
blackpearl.eur
```

```
## [1] 558759611
```

It looks like the movie made 558,759,611 in Euros. Not bad. Now, let's see how much more Pirates of the Caribbean 2: Dead Man's Chest made compared to "Curse of the Black Pearl." Another Google search uncovered that Dead Man's Chest made $1,066,215,812 (that wasn't a mistype, the freaking movie made over a billion dollars).

```r
deadman.usd <- 1066215812
```

Now, I'll divide deadman.usd by blackpearl.usd:

```r
deadman.usd / blackpearl.usd
```

```
## [1] 1.679201
```

It looks like "Dead Man's Chest" made 168% as much as "Curse of the Black Pearl" - not bad for two movies based off of a ride from Disneyland. (Note…I personally love these movies!)