# Exploring Week 2 in R

Thursday

## Introduction to Today

The goal for today is to put into practice the lecture topics we have gone over thus far, namely, exploring and evaluating assumptions in R. Today we will focus on the assumption of homogeneous variances.

**R Tip of the Day:** R comes with many free datasets. Run the command *data()* to see what there is.

```
data(package = .packages(all.available = TRUE))
```

## Loading Our Libraries & Data

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.

```
library(car)
library(ggplot2)
library(psych)
library(dplyr)

dlf <- read.delim("DownloadFestival.dat", header=TRUE)
dlf[dlf$day1 >20,] <- NA #getting rid of outliers
```

## Levene's test

**Levene's test** will check if our data and compare the variances between groups in our data. Equal Variances is an assumption of many many many statistical tests.

The **null hypothesis** is that all populations variances are equal ($p > 0.05$)

The **alternative hypothesis** is that at least two of them differ ($p < 0.05$)

Like the Shapiro-Wilks test, we want the $p > 0.05$. If it is $< .05$, then we are violating the assumption of equal variances

### Example 1: Festival Data

If p > 0.05, assumption is violated.

```
#Levene's test day 1:
leveneTest(day1 ~ gender, na.omit(dlf))
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value Pr(>F)
## group   1  0.3621 0.5485
##       121
```

```
#Levene's test for day 2:
leveneTest(day2 ~ gender, na.omit(dlf))
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value  Pr(>F)
## group   1  5.3411 0.02252 *
##       121
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

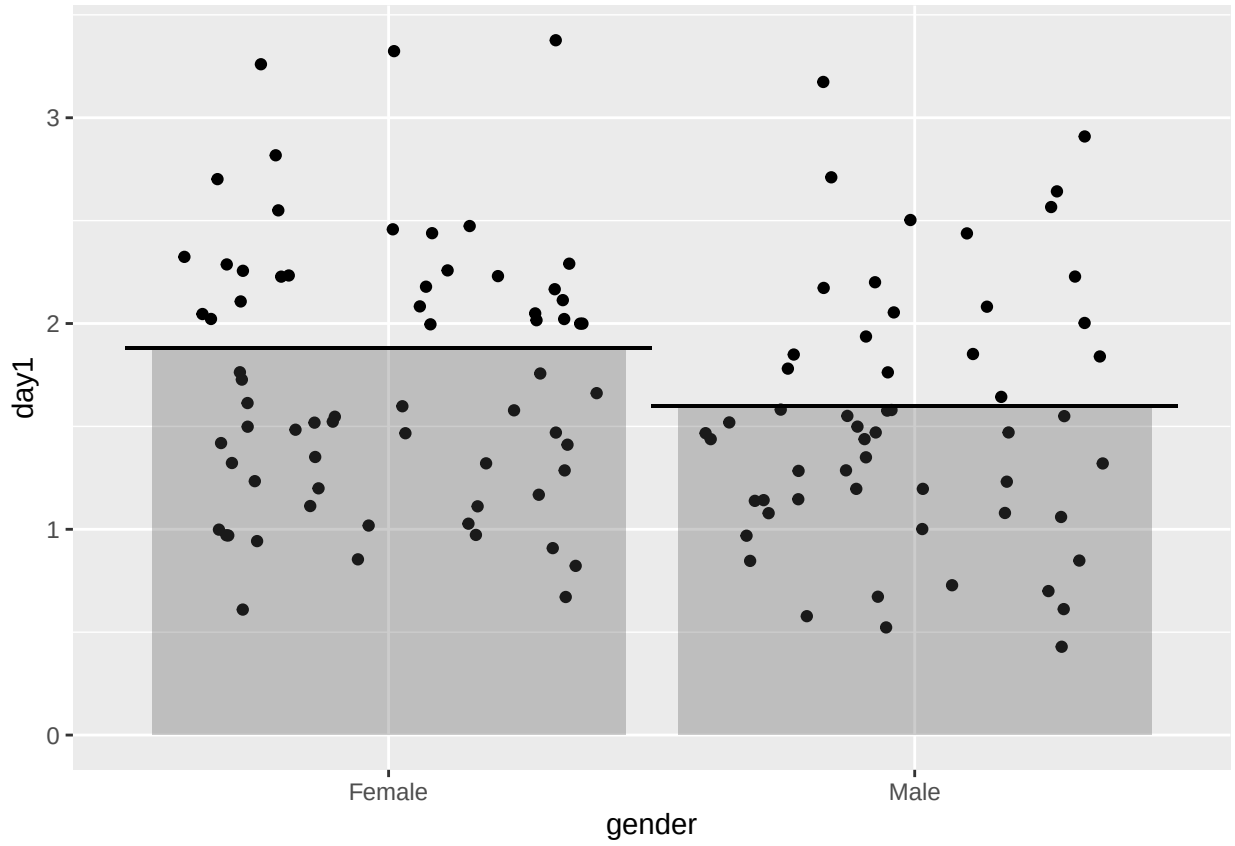
```
#Levene's test of the hygiene scores on day 3:
leveneTest(day3 ~ gender, na.omit(dlf))
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value   Pr(>F)
## group   1  8.3407 0.004594 **
##       121
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Assumption is violated for all three days across the levels of our gender variable.

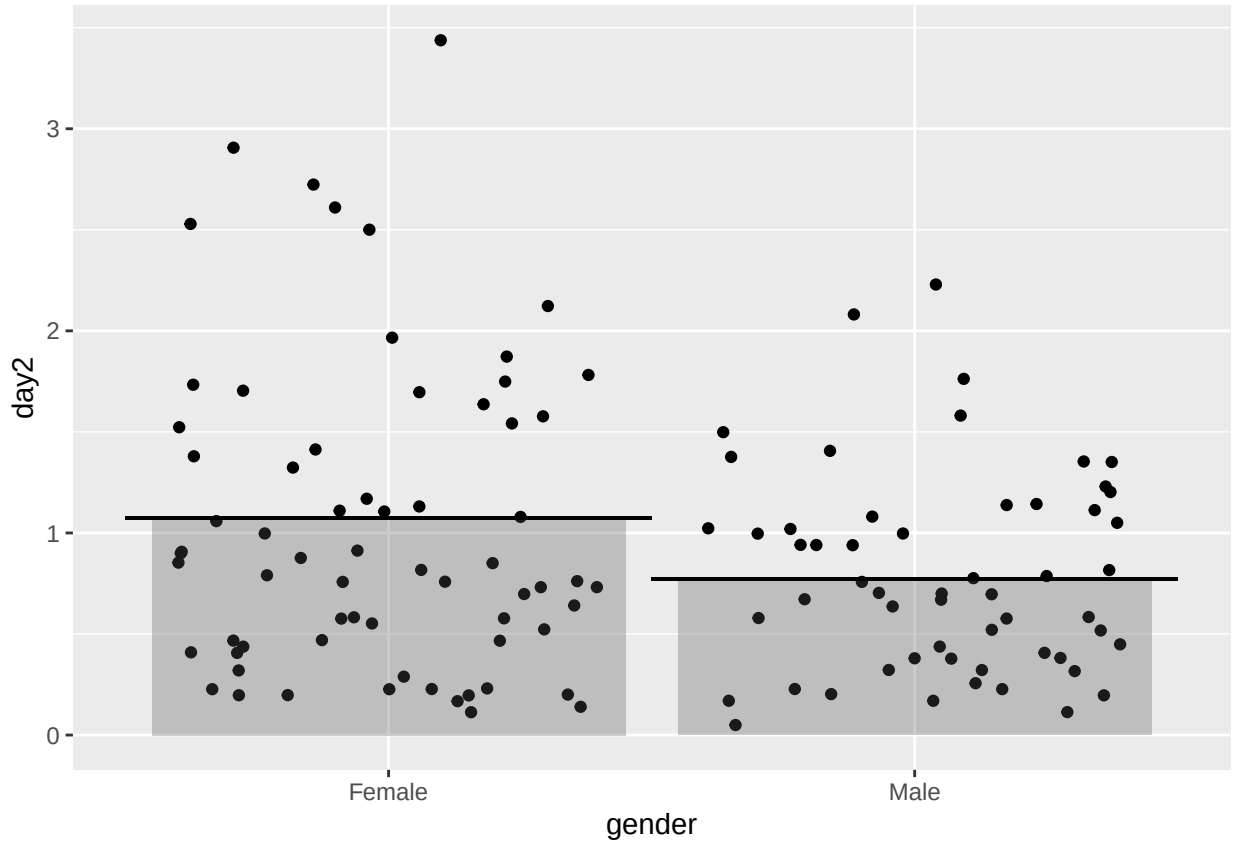
### Plotting Day 1

```
gen.mean.1 <- dlf %>%
        group_by(gender) %>%
        summarise(day1 = mean(day1, na.rm  = T))

ggplot(na.omit(dlf), aes(gender, day1)) + geom_point(position = "jitter") +
  geom_bar(data = gen.mean.1, stat = "identity", alpha = .3) +
  geom_segment(aes(x = 0.5, y = 1.88, xend = 1.5, yend = 1.88)) +
  geom_segment(aes(x = 1.5, y = 1.6, xend = 2.5, yend = 1.6))
```



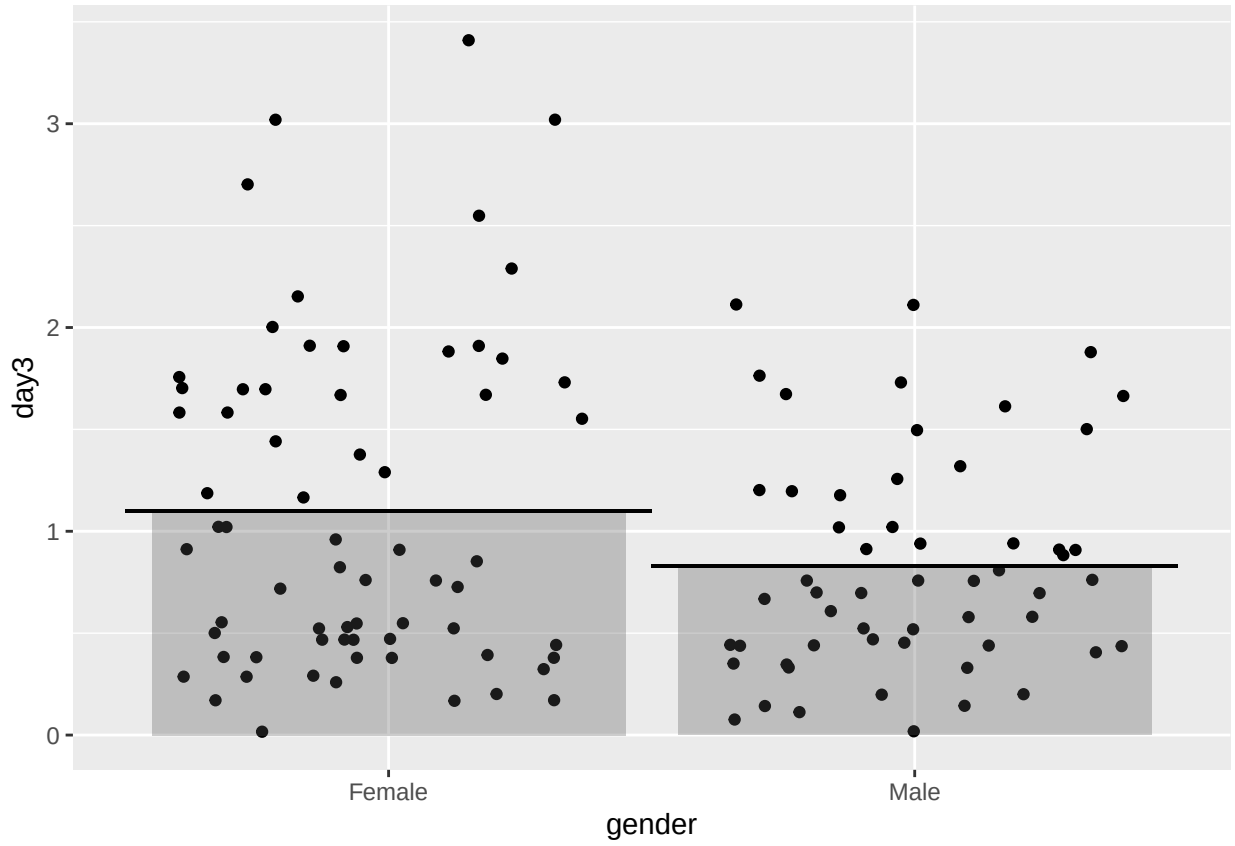
### Plotting Day 2

```
gen.mean.2 <- dlf %>%
        group_by(gender) %>%
        summarise(day2 = mean(day2, na.rm  = T))

ggplot(na.omit(dlf), aes(gender, day2)) + geom_point(position = "jitter") +
  geom_bar(data = gen.mean.2, stat = "identity", alpha = .3) +
  geom_segment(aes(x = 0.5, y = 1.074, xend = 1.5, yend = 1.074)) +
  geom_segment(aes(x = 1.5, y = .773, xend = 2.5, yend = .773))
```



### Plotting Day 3

```
gen.mean.3 <- dlf %>%
        group_by(gender) %>%
        summarise(day3 = mean(day3, na.rm  = T))

ggplot(na.omit(dlf), aes(gender, day3)) + geom_point(position = "jitter") +
  geom_bar(data = gen.mean.3, stat = "identity", alpha = .3) +
  geom_segment(aes(x = 0.5, y = 1.1, xend = 1.5, yend = 1.1)) +
  geom_segment(aes(x = 1.5, y = .829, xend = 2.5, yend = .829))
```



## Example 2: Tooth Growth Data

If p > 0.05, assumption is violated.

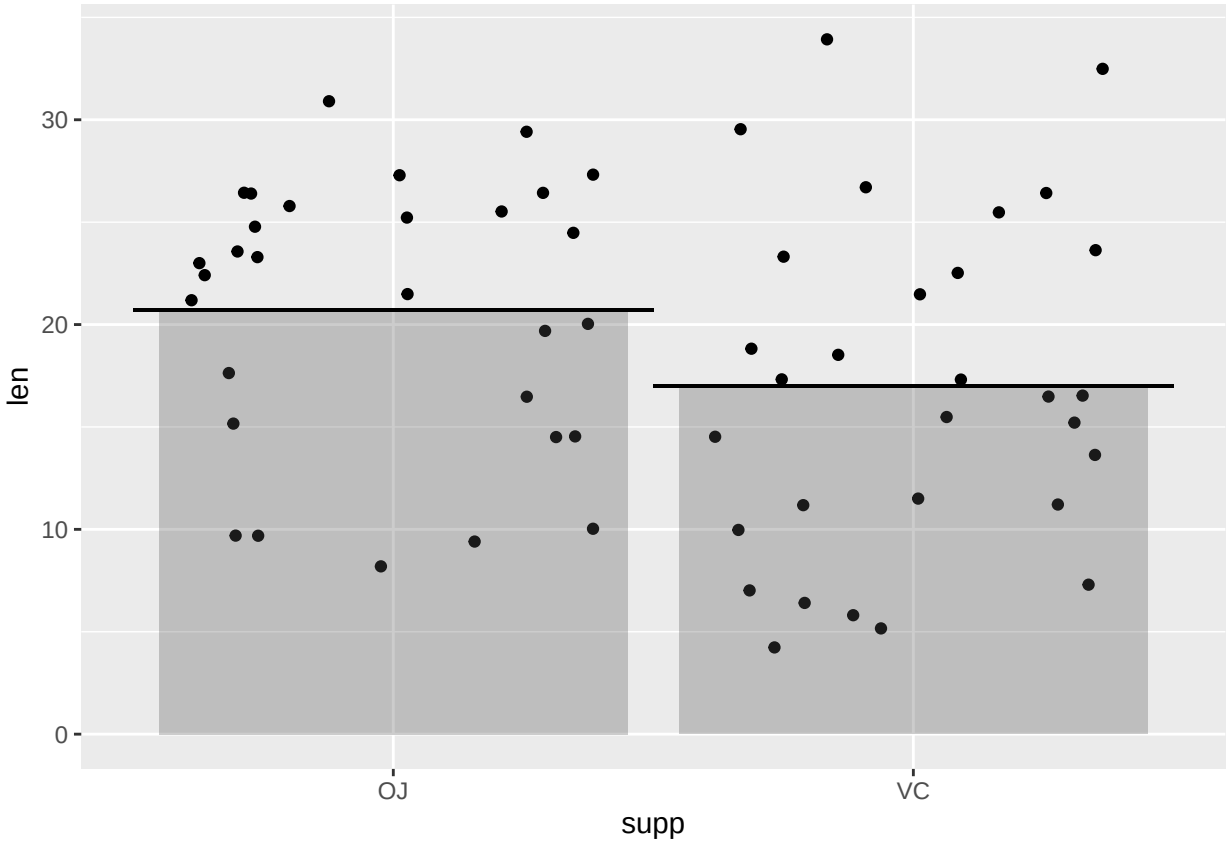
```
#Levene's test
leveneTest(len ~ supp, ToothGrowth)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##       Df F value Pr(>F)
## group  1  1.2136 0.2752
##       58
```

Assumption is not violated.

### Plotting Tooth Growth Data

```
len.mean.1 <- ToothGrowth %>%
        group_by(supp) %>%
        summarise(len = mean(len, na.rm  = T))

ggplot(ToothGrowth, aes(supp, len)) + geom_point(position = "jitter") +
  geom_bar(data = len.mean.1, stat = "identity", alpha = .3) +
  geom_segment(aes(x = 0.5, y = 20.7, xend = 1.5, yend = 20.7)) +
  geom_segment(aes(x = 1.5, y = 17, xend = 2.5, yend = 17))
```



## Example 3: Plant Growth Data

If p > 0.05, assumption is violated.

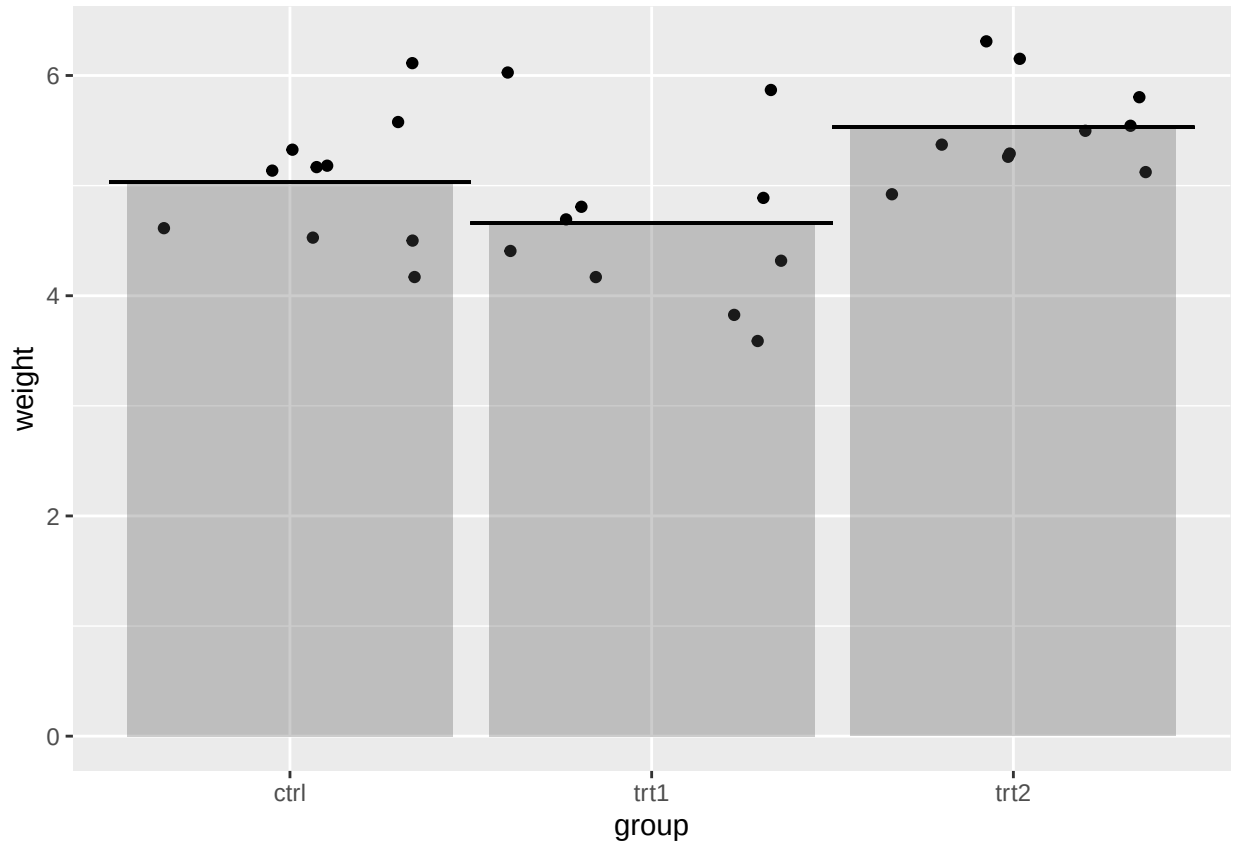
```
#Levene's test
leveneTest(weight ~ group, PlantGrowth)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##       Df F value Pr(>F)
## group  2  1.1192 0.3412
##       27
```

Assumption is not violated.

### Plotting Plant Growth Data

```
weight.mean.1 <- PlantGrowth %>%
        group_by(group) %>%
        summarise(weight = mean(weight, na.rm  = T))

ggplot(PlantGrowth, aes(group, weight)) + geom_point(position = "jitter") +
  geom_bar(data = weight.mean.1, stat = "identity", alpha = .3) +
  geom_segment(aes(x = 0.5, y = 5.03, xend = 1.5, yend = 5.03)) +
  geom_segment(aes(x = 1.5, y = 4.66, xend = 2.5, yend = 4.66))+
  geom_segment(aes(x = 2.5, y = 5.53, xend = 3.5, yend = 5.53))
```



## Example 4: Motor Trends Data

If p > 0.05, assumption is violated.

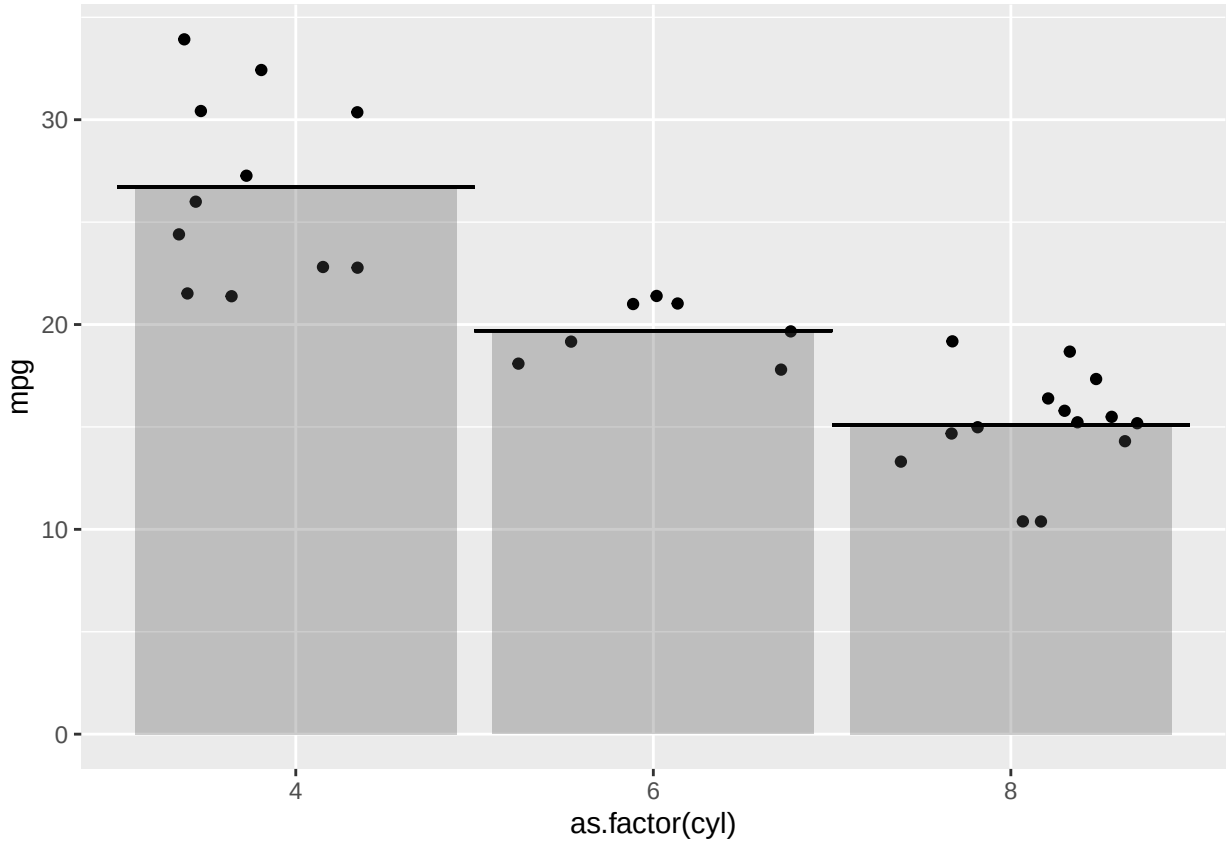
```
#Levene's test
leveneTest(mpg ~ as.factor(cyl), mtcars)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##       Df F value  Pr(>F)
## group  2  5.5071 0.00939 **
##       29
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Assumption is violated.

### Plotting Motor Trends Data

```
mpg.mean.1 <- mtcars %>%
        group_by(cyl) %>%
        summarise(mpg = mean(mpg, na.rm  = T))

ggplot(mtcars, aes(x = as.factor(cyl), mpg)) + geom_point(position = "jitter") +
  geom_bar(data = mpg.mean.1, stat = "identity", alpha = .3) +
  geom_segment(aes(x = .5, y = 26.7, xend = 1.5, yend = 26.7)) +
  geom_segment(aes(x = 1.5, y = 19.7, xend = 2.5, yend = 19.7))+
  geom_segment(aes(x = 2.5, y = 15.1, xend = 3.5, yend = 15.1))
```




## Example 5: Beaver Data

If p > 0.05, assumption is violated.

```
#Levene's test
leveneTest(temp ~ as.factor(activ), beaver1)
```

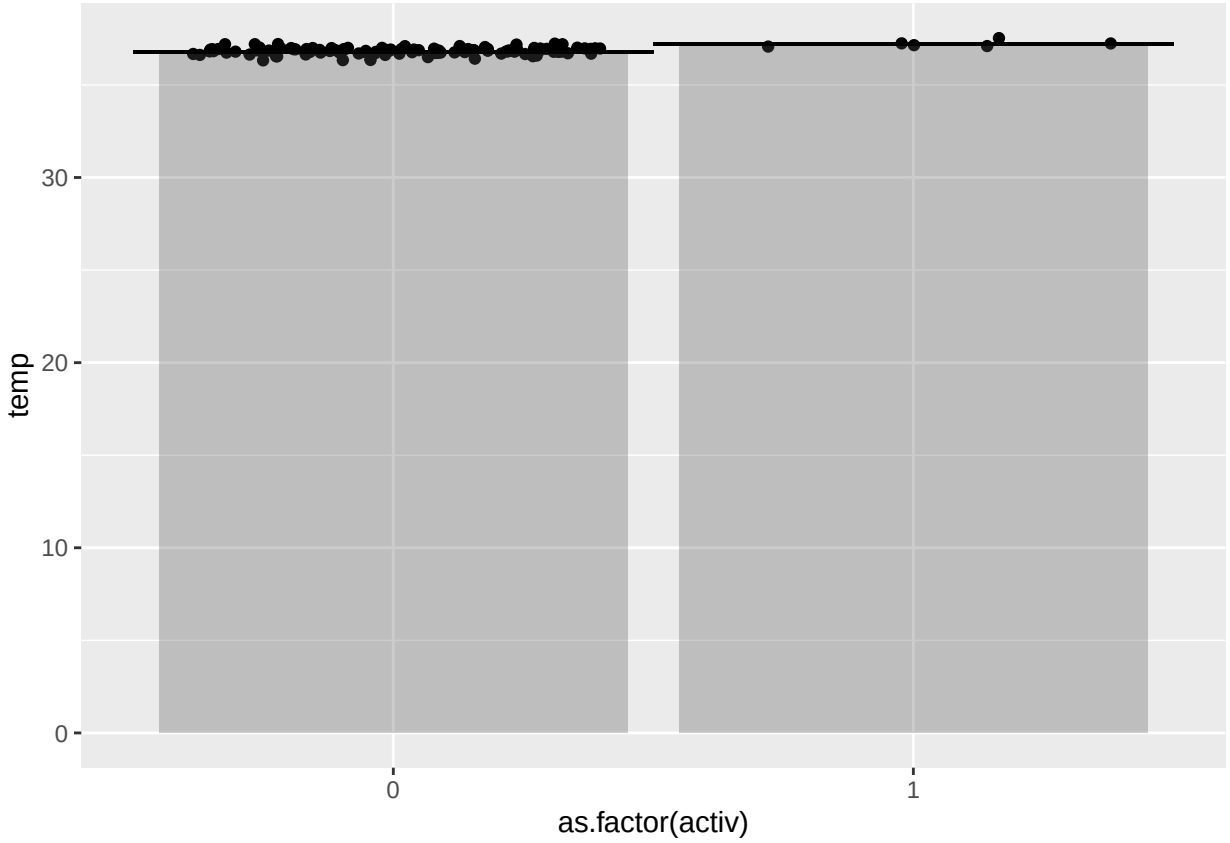
```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value Pr(>F)
## group   1  0.0513 0.8213
##       112
```

Assumption is not violated.

### Plotting Beaver Data

```
temp.mean.1 <- beaver1 %>%
        group_by(activ) %>%
        summarise(temp = mean(temp, na.rm  = T))

ggplot(beaver1, aes(x = as.factor(activ), temp)) + geom_point(position = "jitter") +
  geom_bar(data = temp.mean.1, stat = "identity", alpha = .3) +
```

```
  geom_segment(aes(x = .5, y = 36.8, xend = 1.5, yend = 36.8)) +
  geom_segment(aes(x = 1.5, y = 37.2, xend = 2.5, yend = 37.2))
```




# ExtRa R

## Styling Guide: Commenting and Spacing

Like all programming languages, R isn't just meant to be read by a computer, it's also meant to be read by other humans – which is why we are learning/working in this wonderful R Markdown format. For this reason, it's important that your code looks nice and is understandable to other people and, most importantly, your future self. Today I'll focus on the two most critical aspects of good style: commenting and spacing.

### Commenting code with the # (hash/hashtag/pound) sign

Within a code chunk, comments are completely ignored by R and are just there for whomever is reading the code.

You can use comments to explain what a certain line of code is doing (and you'll notice in my code chunks, I try and add comments throughout). You could also use comments to simply visually separate meaningful chunks of code from each other.

Comments in R are designated by a *# (pound)* sign. Whenever R encounters a # sign, it will ignore all the code after the # sign on that line. Additionally, R Studio will display comments in a separate color than standard R code to remind you that it's a comment. The color will depend on the theme you have.

Here is an example of a short script that is nicely commented. Try to make your scripts look like this!

```
# Step 1: Load the car package
library(car)

# Step 2: See the column names in the mtcars
names(mtcars)

# Step 3: Calculations

# What is the mean mpg?
mean(mtcars$mpg, na.rm = T)

#What are the row names?
row.names(mtcars) #datasets don't usually have row names, but this one does!

# How many mpg does the Mazda RX4 get?
mtcars$mpg[row.names(mtcars) == 'Mazda RX4']
```

I cannot stress enough how important it is to comment your code! Trust me, even if you don't plan on sharing your code with anyone else, keep in mind that your future self could be reading it in the future. When using an *.rmd* R Markdown File, you are able to elaborate in text form outside of the chunks. However, it is good to get into the habit of commenting your code.

## Spacing

Howwouldyouliketoreadabookiftherewerenospacesbetweenwords? I'mguessingyouwouldn't. Soeverytimeyouwritecodewithoutproperspacing,rememberthissentence.

Commenting isn't the only way to make your code legible. It's important to make appropriate use of spaces and line breaks. For example, I include spaces between arithmetic operators (like =, + and -) and after commas (which we'll get to later). For example, look at the following code:

```
# BAD looking code
a<-(100+3)-2
mean(c(a/100,642564624.34))
t.test(formula=day1~gender,data=dlf)
plot(x=mdlf$day1,y=dlf$gender,main="myplot")
```

That code looks HORRIBLE. Don't write code like that. It makes my eyes hurt. Now, let's use some liberal amounts of commenting and spacing to make it look less better.

```
#Nicely Spaced (and commented!) Code
# Some meaningless calculations. Not important
a <- (100 + 3) - 2
mean(c(a / 100, 642564624.34))

# t.test comparing day1 of male v female
t.test(formula = day1 ~ gender, data = dlf)

# A scatterplot of day1 and gender.
#  Hard to see a relationship this way
plot(x = dlf$day1,
```

```
     y = dlf$gender,
     main = "myplot")
```

See how much better that second chunk of code looks? Not only do the comments tell us the purpose behind the code, but there are spaces and line-breaks separating distinct elements.

There are a lot more aspects of good code formatting. For a list of recommendations on how to make your code easier to follow, check out Google's own company R Style guide here

## Indexing Vectors with [ ]

By now we've applied functions like mean() and table() to vectors. However, in many analyses, you won't want to calculate statistics of an entire vector. Instead, you will want to access specific subsets of values of a vector based on some criteria. For example, you may want to access values in a specific location in the vector (i.e.; the first 10 elements) or based on some criteria within that vector (i.e.; all values greater than 0), or based on criterion from values in a different vector (e.g.; All values of age where sex is Female).

To access specific values of a vector in R, we use indexing using brackets []. In general, whatever you put inside the brackets, tells R which values of the vector object you want. There are two main ways that you can use indexing to access subsets of data in a vector: *numerical* and *logical* indexing. Let's go through a broad example.

Creating Boat Sale Data Vectors

```
boat.names <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
boat.colors <- c("black", "green", "pink", "blue", "blue",
                 "green", "green", "yellow", "black", "black")
boat.ages <- c(143, 53, 356, 23, 647, 24, 532, 43, 66, 86)
boat.prices <- c(53, 87, 54, 66, 264, 32, 532, 58, 99, 132)
boat.costs <- c(52, 80, 20, 100, 189, 12, 520, 68, 80, 100)
```

What was the price of the first boat?

```
boat.prices[1]
```

```
## [1] 53
```

What were the ages of the first 5 boats?

```
boat.ages[1:5]
```

```
## [1] 143  53 356  23 647
```

What were the names of the black boats?

```
boat.names[boat.colors == "black"]
```

```
## [1] "a" "i" "j"
```

What were the prices of either green or yellow boats?

```
boat.prices[boat.colors == "green" | boat.colors == "yellow"]
```

```
## [1]  87  32 532  58
```

Change the price of boat "s" to 100

```
boat.prices[boat.names == "s"] <- 100
```

What was the median price of black boats less than 100 years old?

```
median(boat.prices[boat.colors == "black" & boat.ages < 100])
```

```
## [1] 115.5
```

How many pink boats were there?

```
sum(boat.colors == "pink", na.rm = T)
```

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

What percent of boats were older than 100 years old?

```
mean(boat.ages < 100, na.rm = T)
```

```
## [1] 0.6
```

## Numerical Indexing

With numerical indexing, you enter a vector of integers corresponding to the values in the vector you want to access in the form a[index], where a is the vector, and index is a vector of index values. For example, let's use numerical indexing to get values from our boat vectors.

What is the first boat name?

```
boat.names[1]
```

```
## [1] "a"
```

What are the first five boat colors?

```
boat.colors[1:5]
```

```
## [1] "black" "green" "pink"  "blue"  "blue"
```

What is every second boat age?

```
boat.ages[seq(1, 5, by = 2)]
```

```
## [1] 143 356 647
```

You can use any indexing vector as long as it contains integers. You can even access the same elements multiple times:

What is the first boat age (3 times)

```
boat.ages[c(1, 1, 1)]
```

```
## [1] 143 143 143
```

If it makes your code clearer, you can define an indexing object before doing your actual indexing. For example, let's define an object called my.index and use this object to index our data vector:

```
my.index <- 3:5
boat.names[my.index]
```

```
## [1] "c" "d" "e"
```

## Logical Indexing

The second way to index vectors is with logical vectors. A logical vector is a vector that only contains TRUE and FALSE values. In R, true values are designated with TRUE, and false values with FALSE. When you index a vector with a logical vector, R will return values of the vector for which the indexing vector is TRUE. If that was confusing, think about it this way: a logical vector, combined with the brackets [ ], acts as a filter for the vector it is indexing. It only lets values of the vector pass through for which the logical vector is **TRUE**.

You could create logical vectors directly using c(). For example, I could access every other value of the following vector as follows:

```
a <- c(1, 2, 3, 4, 5)
a[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
```

```
## [1] 1 3 5
```

As you can see, R returns all values of the vector a for which the logical vector is TRUE.

### Logical comparison operators in R

However, creating logical vectors using c() is tedious. Instead, it's better to create logical vectors from existing vectors using comparison operators like < (less than), == (equals to), and != (not equal to). For example, let's create some logical vectors from our boat.ages vector:

Which ages are > 100?

```
boat.ages > 100
```

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

Which ages are equal to 23?

```
boat.ages == 23
```

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

Which boat names are equal to c?

```
boat.names == "c"
```

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

You can also create logical vectors by comparing a vector to another vector of the same length. When you do this, R will compare values in the same position (e.g.; the first values will be compared, then the second values, etc.).

For example, we can compare the boat.cost and boat.price vectors to see which boats sold for a higher price than their cost:

Which boats had a higher price than cost?

```
boat.prices > boat.costs
```

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

Which boats had a lower price than cost?

```
boat.prices < boat.costs
```

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

Once you've created a logical vector using a comparison operator, you can use it to index any vector with the same length. Here, I'll use logical vectors to get the prices of boats whose ages were greater than 100:

What were the prices of boats older than 100?

```
boat.prices[boat.ages > 100]
```

```
## [1]  53  54 264 532
```

Here's how logical indexing works step-by-step:

1. Which boats are older than 100 years?

```
boat.ages > 100
```

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

2. Writing the logical index by hand (you'd never do this!. Show me all of the boat prices where the logical vector is TRUE:

```
boat.prices[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE)]
```

```
## [1]  53  54 264 532
```

Doing it all in one step! You get the same answer:

```
boat.prices[boat.ages > 100]
```

```
## [1]  53  54 264 532
```

## & (and), | (or), %in%

In addition to using single comparison operators, you can combine multiple logical vectors using the OR (which looks like | and AND & commands. The OR | operation will return TRUE if any of the logical vectors is TRUE, while the AND & operation will only return TRUE if all of the values in the logical vectors is TRUE. This is especially powerful when you want to create a logical vector based on criteria from multiple vectors.

For example, let's create a logical vector indicating which boats had a price *greater than 200* OR *less than 100*, and then use that vector to see what the names of these boats were:

Which boats had prices greater than 200 OR less than 100?

```
boat.prices > 200 | boat.prices < 100
```

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

What were the NAMES of these boats?

```
boat.names[boat.prices > 200 | boat.prices < 100]
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

You can combine as many logical vectors as you want (as long as they all have the same length!):
Boat names of boats with a color of black OR with a price $> 100$

```
boat.names[boat.colors == "black" | boat.prices > 100]
```

```
## [1] "a" "e" "g" "i" "j"
```

Names of blue boats with a price greater than 200

```
boat.names[boat.colors == "blue" & boat.prices > 200]
```

```
## [1] "e"
```

You can combine as many logical vectors as you want to create increasingly complex selection criteria. For example, the following logical vector returns TRUE for cases where the boat colors are black OR brown, AND where the price was less than 100:

Which boats were eithe black or brown, AND had a price less than 100?

```
(boat.colors == "black" | boat.colors == "brown") & boat.prices < 100
```

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

What were the names of these boats?

```
boat.names[(boat.colors == "black" | boat.colors == "brown") & boat.prices < 100]
```

```
## [1] "a" "i"
```

When using multiple criteria, make sure to use parentheses when appropriate. If I didn't use parentheses above, I would get a different answer.

The *%in%* operation helps you to easily create multiple OR arguments.Imagine you have a vector of categorical data that can take on many different values. For example, you could have a vector x indicating people's favorite letters.

```
x <- c("a", "t", "a", "b", "z")
```

Now, let's say you want to create a logical vector indicating which values are either a or b or c or d. You could create this logical vector with multiple | (OR) commands:

```
x == "a" | x == "b" | x == "c" | x == "d"
```

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

However, this takes a long time to write. Thankfully, the %in% operation allows you to combine multiple OR comparisons much faster. To use the %in% function, just put it in between the original vector, and a new vector of possible values. The %in% function goes through every value in the vector x, and returns TRUE if it finds it in the vector of possible values – otherwise it returns FALSE.

```
x %in% c("a", "b", "c", "d")
```

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

As you can see, the result is identical to our previous result.

## Counts and percentages from logical vectors

R will interpret TRUE values as 1 and FALSE values as 0. This allows us to easily answer questions like "How many values in a data vector are greater than 0?" or "What percentage of values are equal to 5?" by applying the sum() or mean() function to a logical vector.

We'll start with a vector x of length 10, containing 3 positive numbers and 5 negative numbers:

```
x <- c(1, 2, 3, -5, -5, -5, -5, -5)
```

We can create a logical vector to see which values are greater than 0:

```
x > 0
```

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

Now, we'll use sum() and mean() on that logical vector to see how many of the values in x are positive, and what percent are positive.

```
sum(x > 0)
```

```
## [1] 3
```

```
mean(x > 0)
```

```
## [1] 0.375
```

This is a really powerful tool. Pretty much any time you want to answer a question like "How many of X are Y" or "What percent of X are Y", you use sum() or mean() function with a logical vector as an argument.