In this section we'll explore basic hypothesis testing in `R`, along with learning a bit about some elements that make a well-coded function. After adding our trusty `auto.dta` dataset to the package we made last week, we'll work through calculating t and F-statistics.

Last section

Hopefully you're now comfortable using a package to store your functions nicely. One thing I think I got wrong last week was what the `LazyData` field in the `DESCRIPTION` file does. If we specify `LazyData: TRUE`, `R` will load any package specific datasets when the `library()` function is called. If we specify `LazyData: FALSE`, we would have to manually load any datasets using the `data()` function.

Because we're going to use it again, let's add the `auto.dta` dataset to our respective packages. To start, open your package project in RStudio and add a folder called `data` to the package folder. Next, call the following from the package directory.

```
library(foreign)
auto <- read.dta("<path to>/auto.dta")
names(auto) <- c("price", "mpg", "weight")
save(auto, file = "data/auto.rda")
```

Now, make sure you have the `LazyData: TRUE` field specified in your `DESCRIPTION` file, and rebuild your package. You've now added the `auto` dataset to your package! It will be loaded each time you call your package with `library()`.

Calculating t-tests and F-tests

First, a basic overview in conducting t and F-tests. We've got a lot of what we need already in our package. Let's start by calling it into the `R`-environment.

```
library(misc212)
```

Remember, this contains our `OLS()` function and the `auto data.frame`, which we'll use soon. For reference, consider the regression output from `lm()`:

```
results <- lm(price ~ 1 + mpg + weight, data = auto)
coef(summary(results))

##                Estimate   Std. Error    t value    Pr(>|t|)
## (Intercept) 1946.068668 3597.0495988  0.5410180 0.590188628
## mpg          -49.512221   86.1560389 -0.5746808 0.567323727
## weight         1.746559    0.6413538  2.7232382 0.008129813

summary(results)$fstatistic

##    value    numdf    dendf
## 14.73982  2.00000 71.00000
```

Now we'll run OLS and define some useful elements for hypothesis testing using the definitions in lecture notes:

```
X <- cbind(1, auto$mpg, auto$weight)
y <- auto$price
n <- NROW(X)
k <- NCOL(X)
b <- OLS(y, X)
e <- y - X %*% b
s2 <- t(e) %*% e/(n - k)
XpXinv <- solve(t(X) %*% X)
se <- sqrt(s2 * diag(XpXinv))
```

By the way, it's good practice to define intermediate variables like `XpXinv`, and `s2`. This can be useful for debugging and making your code more readable. For example, I could have defined `se` as

```
sqrt((t(y - X %*% b) %*% (y - X %*% b) / (n-k)) * diag(solve(t(X) %*% X)))
```

(or worse!), which would have been a nightmare to debug or understand.

There's a cool trick I recently learned in RStudio to turn code like this into functions. Ultimately, we'd like to turn this into a function of `y` and `X`, just like the `OLS()` function. Let's highlight the rows from `n <- ...` to `se <- ...` and hit `Ctrl/Command + Alt + X`. Name the function something appropriate. I named mine `se`. Cool huh?! The only modification I had to make was to add a return line.

We can now use the vector of standard errors to calculate our `t` and `p` values for the individual t-tests:

```
seResults <- se(y, X)
b <- OLS(y, X)
n <- NROW(X)
k <- NCOL(X)
t <- (b - 0) / seResults
p <- apply(t, 1, function(t) {2 * pt(-abs(t), df = (n - k))})
```

Great! We have replicated the `t value` and `Pr(>|t|)` columns of the canned output. Now let's try to replicate the full regression F-statistic.

This is a joint test of coefficient significance; are the coefficients jointly different from a zero vector? Note well that the "regression F-statistic" almost always refers to a test of joint significance of the coefficients on everything but the intercept. Max has a great description as to why the F test is different from two separate tests of significance. Going from the notation in the notes, we are testing joint significance by setting:

$$\boldsymbol{R} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \boldsymbol{r} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{1}$$

We will work from equation (2.81), which is fairly daunting, but, because $\boldsymbol{r} = 0$, we can omit it:

$$F = \frac{(\boldsymbol{Rb})'[\boldsymbol{R}(\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{R}']^{-1}(\boldsymbol{Rb})/J}{s^2} \tag{2}$$

Let's start calculating:

```
R <- rbind(c(0, 1, 0), c(0, 0, 1))
J <- 2
RXXInvR <- solve(R %*% solve(t(X) %*% X) %*% t(R))
F <- t(R %*% b) %*% RXXInvR %*% (R %*% b) / (s2 * J)
```

It worked! This is, of course, one of the simplest possible F-tests we could conduct, but you can see how it would be easy to construct your own F-tests using this framework.

Let's turn this into a function. A good start might be:

```
FCalc <- function(R, J, b, s2, X) {
  RXXInvR <- solve(R %*% solve(t(X) %*% X) %*% t(R))
  F <- t(R %*% b) %*% RXXInvR %*% (R %*% b) / (s2 * J)
  F
}
```

which we could call with:

```
R <- rbind(c(0, 1, 0), c(0, 0, 1))
J <- 2
FCalc(R, J, b, s2, X)

##          [,1]
## [1,] 14.73982
```

This gets the job done. But, it seems like it could be better. For a moment, let's just think about what I got right here. The first line has the right spacing between all the elements; I start the first piece of calculation on a new line and the last line is a lone '}'. The body of the function is indented with 2 spaces. All of this is consistent with normal `R` style.[1] The only functions that don't need to conform to this style are short one line functions where the function body can also be put on the first line:

```
xSq <- function(x) x^2
xSq(3)

## [1] 9
```

Good practice is to use the minimum inputs to a function, to make its use easier. Let's do this with the `FCalc()` function:

[1] See Google's style guide for more.

```r
FCalc <- function(R, y, X) {
  n <- NROW(X)
  k <- NCOL(X)
  b <- OLS(y, X)
  e <- y - X %*% b
  s2 <- t(e) %*% e / (n - k)
  J <- NROW(R)
  RXXInvR <- solve(R %*% solve(t(X) %*% X) %*% t(R))
  F <- t(R %*% b) %*% RXXInvR %*% (R %*% b) / (s2 * J)
  F
}
```

We could also add a p-value and return the results as a `list` or `data.frame`. One thing we should add is a check that our R matrix is the correct size:

```r
FCalc <- function(R, y, X) {
  if (NCOL(R) != NCOL(X)) {
    stop("Dimensions of R and X do not match")
  }
  n <- NROW(X)
  k <- NCOL(X)
  b <- OLS(y, X)
  e <- y - X %*% b
  s2 <- t(e) %*% e / (n - k)
  J <- NROW(R)
  RXXInvR <- solve(R %*% solve(t(X) %*% X) %*% t(R))
  F <- t(R %*% b) %*% RXXInvR %*% (R %*% b) / (s2 * J)
  F
}
```

This still isn't perfect, especially if we're worried about computational time.[2] We would likely use these functions something like this:

```r
OLSResults <- OLS(y, X)
seResults <- se(y, X)
FResults <- FCalc(R, y, X)
```

Here, `OLS()` is called three times, once in each function, which is inefficient. A simple modification could be to give the results of the `OLS()` function as argument to the other functions. Nonetheless, many other lines of code are repeated in the `se()` and `FCalc()` functions. I'll leave it to you to work out a nice solution to this. As another exercise, you can extend the function to include the case where $\boldsymbol{r} \neq \mathbf{0}$.

[2] This can be an issue if you are performing many simulations, or using very large data sets.

Puzzle

- **Partitioned regression:** Generate a 100×4 matrix $\boldsymbol{X}$ *including* a column of ones for the intercept. Additionally, generate a vector $\boldsymbol{y}$ according to the generating process:

$$y_i = 1 + x_{1i} + 2x_{2i} + 3x_{3i} + \varepsilon_i,$$

where $\varepsilon_i \sim N(0, 1)$. Let $\boldsymbol{Q}$ be the first three columns of $\boldsymbol{X}$ and let $\boldsymbol{N}$ be the final column. In addition, let

$$\begin{aligned}
\hat{\gamma}_1 &= (\boldsymbol{Q}'\boldsymbol{Q})^{-1}\boldsymbol{Q}'\boldsymbol{y} \quad \text{and} \quad \boldsymbol{f} = \boldsymbol{y} - \boldsymbol{Q}\hat{\gamma}_1 \\
\hat{\gamma}_2 &= (\boldsymbol{Q}'\boldsymbol{Q})^{-1}\boldsymbol{Q}'\boldsymbol{N} \quad \text{and} \quad \boldsymbol{g} = \boldsymbol{N} - \boldsymbol{Q}\hat{\gamma}_2 \\
\hat{\gamma}_3 &= \boldsymbol{f} \cdot \boldsymbol{g}/||\boldsymbol{g}||^2 \quad \text{and} \quad \boldsymbol{e} = \boldsymbol{f} - \boldsymbol{g}\hat{\gamma}_3
\end{aligned}$$

Show that $\hat{\beta} = [(\hat{\gamma}_1 - \hat{\gamma}_2\hat{\gamma}_3) \quad \hat{\gamma}_3]$. Note that the total dimension of $\hat{\beta}$ is 4.