# EEP/IAS 118 - Coding Bootcamp Solutions

## Introduction to R and Jupyter Notebooks

### August 2021

# Jupyter Notebook Basics

We're going to start off this term by using *Jupyter Notebooks* to run **R**, our preferred programming language for statistical computing. Jupyter notebooks are an interactive computing environment that lets us combine both text elements and active **R** code. Importantly for us, notebooks let us

- Write and run **R** code through our web browsers, and
- Add narrative text that describes our code and the output from said code

This means that we can use **R** without having to install any software on our personal computers, and can ignore errors that might pop up from local conflicts or issues with packages on personal machines. Further, it lets us answer written exercises and coding problems all in one place.

When you clicked the link on bCourses, you were taken to a folder in your web browser. This folder is hosted on a remote web server through Berkeley's **Datahub** and should look something like this:

Datahub Folder

Today we are doing Coding Resources, so you will want to click on that folder.

You should see the following files.

1. The first file *Coding Bootcamp Session.ipynb* is our notebook for today

2. *autos.csv* is a data file in comma separated value (csv) format

3. *autos.dta* is the same data file, but formatted for Stata in .dta format

We'll be using both csv and dta data this term, so we'll practice reading in both formats.

(There is also a folder of images...don't worry about those!)

## Running Jupyter Notebooks

Double clicking on the *Coding Bootcamp.ipynb* notebook will open up the notebook in another browser window. The notebook that opens should look... well, like this!

At the top of the page you'll find the menubar and toolbar:

From this menu we can run our code, display our text, save our notebook, and download a pdf or other format of our notebook.

Clicking on **File** in the menu bar lets us save the notebook and build a checkpoint. You can use **Revert to Checkpoint** to return to previous checkpoints if you want to go back to a previous version of the notebook.

**Print Preview** lets you view the notebook as it would look when printed.

We will be using the **Download As > PDF via HTML** command to export a pdf copy of the notebook after running all text/code for submission. This is the first option on the menu.

## Editing Cells

This and all notebooks are comprised of a linear collection of boxes, called *cells*. For the sake of this class, we'll be working with two types of cells: *Markdown* cells for text, and *Code* for writing and executing **R** code.

Markdown cells support plain text as well as markdown code, html, and LaTeX math mode. For this class, plain text answers are totally fine. If you want to dive into Markdown formatting, this cheat sheet has information on formatting text, building tables, and adding html. If you want to add pretty math equations, see this LaTeX Math Mode Guide.

Select a cell by right clicking on it. A grey box with a blue bar to the left will appear around the cell, like so:

This means you are in **command mode**. In command mode you can see the cell type (whether it is markdown for text or code for **R** commands) in the toolbar but can't edit the content of the cell.

To edit the content of cells, double click on the cell to enter **edit mode**. When in edit mode, the box surrounding the cell will turn green - as will the left margin.

A text (markdown) cell in edit mode should look like this:

and a code (R) cell like this:

Use edit mode to type in all your text and code. When you are done with your paragraph or want to try out your code, it's time to run the cell.

Try selecting this cell by right clicking on it, then double click to enter edit mode.

## Running Cells

When you are ready to run a cell (done typing your text in a cell and want to display it in formatted mode, or want to run **R** code), hit `shift + enter`, `control + enter`, or hit `Run` in the toolbar.

Running a text cell will exit edit mode, format the cell's text, and select the next cell down.

You can also run a cell from edit or command mode - try this by right clicking on the above cell that you were typing in, and hitting `shift + enter`.

We'll see later that running code cells will oftentimes add output to our notebook. This will be how we follow what we're doing in **R** and how we'll get our output.

## R

**R** is our programming language for statistical programming. It is open source and free, handles lots of different types of data, and has tons of different packages that allow it to a ton of different things (regression analysis, plotting, working with spatial data, web scraping, creating applications, machine learning to name a few).

## R in Code Cells

Thanks to *Datahub*, **R** is running in the background of our Jupyter notebooks. As a result, we can type **R** code into code cells, run the cell, and get results all in one place.

Let's try it: in the code cell below, type '2 + 2' and run the cell.

In [1]:
```
2+2
```

4

**R** took our code as input, and spit out the result - in this case the value of our summation, 4.

## R Syntax

*Note of caution:* **R** is a stickler for typos. Precise syntax is essential. Capital letters, commas, or parentheses must be in the correct place. Any deviation from required syntax will lead your code to either fail or produce unintended (and incorrect) results. You will spare yourself a lot of aggravation if you take the time to go slowly and carefully as you're getting started until you have gotten more familiar with the commands and their required syntax.

## Packages, Libraries, and Paths

One of **R**'s best features is that it's based on packages. Base **R** does have some stats functions, but **R** plus packages is incredibly powerful. Nearly any time you do anything in **R**, you'll be using functions contained within a package.

Every time we want to use a function contained in a package, we must first call that package using the `library()` function. For example, we can load the **haven** package by typing and running the command `library(haven)` in the below cell:

In [2]:
```
library(haven)
```

The package was loaded correctly in the background. Since there was no output to display nor an error to show, we received no input. We can confirm that the package was loaded correctly by running the `sessionInfo()` function below.

In [3]:
```
sessionInfo()
```

```
R version 4.0.5 (2021-03-31)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.2 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
```

```
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] haven_2.4.3

loaded via a namespace (and not attached):
 [1] fansi_0.5.0       digest_0.6.27     utf8_1.2.2        crayon_1.4.1
 [5] IRdisplay_1.0     repr_1.1.3        lifecycle_1.0.0   jsonlite_1.7.2
 [9] magrittr_2.0.1    evaluate_0.14     pillar_1.6.2      rlang_0.4.11
[13] uuid_0.1-4        vctrs_0.3.8       ellipsis_0.3.2    IRkernel_1.1.1
[17] forcats_0.5.1     tools_4.0.5       hms_1.1.0         compiler_4.0.5
[21] pkgconfig_2.0.3   base64enc_0.1-3   pbdZMQ_0.3-5      htmltools_0.5.1.1
[25] tibble_3.1.3
```

This shows us information about our **R** version, local settings, active packages, ones we've manually loaded, and others that are loaded through our system.

If we had written the package name wrong, and instead tried to load a nonexistent package "hevon", we would receive an error in the notebook:

In [4]:
```
library(hevon)
```

```
Error in library(hevon): there is no package called ‘hevon’
Traceback:

1. library(hevon)
```

## Paths

When we start loading in data files we will need to account for the location of files in our file paths. **R** handles paths by basing itself in the *working directory*, and all file paths are defined relative to that working directory.

We can view the working directory by using the `getwd()` function.

In [5]:
```
getwd()
```

'/home/jovyan/ENVECON-118-FA21/Coding Resources'

The above path is the same as the folder our linked opened to. When calling files, we will start our paths in this folder - if the file is located in the same folder as our notebook, we do not need to include all the "/home/rstudio..." path, and instead can reference it by name. If the file we want to reference (let's call it *enviro.csv*) is in a subfolder (say called *Data*), we will need to include that subfolder in our path: "Data/enviro.csv"

We'll deal more with working directories once we move into **RStudio** - for now all the files we need will be in the same folder as our template, so we can call the files directly by name.

## Loading Files in R

We're going to read in our data, but before we can read in our dataset we should refresh ourselves on what the file is called. To do this, we can use the `list.files()` command to see all the files in our current working directory.

In [6]:
```
list.files()
```

'autos.csv' · 'autos.dta' · 'Coding Bootcamp Session.ipynb' · 'images'

The file we're interested in first is the **autos.dta** file. Since this is a stata-formatted file (.dta), we'll need to use the `read_dta()` function in the **haven** package (which we already loaded). To use the function we need to include the filename with quotes in the parentheses: `read_dta("autos.dta")`.

Here we run into an important feature about **R**: it is an object-oriented language. When we load a dataset or want to store an object to memory (such that we can do things with it later), we have to *assign it a name*. To do this, we use the syntax

```
name <- function(arguments)
```

The arrow tells us we are assigning the output of the function (given the function's arguments/inputs) to *name*. The arrow is read as "gets," so if we wanted to store a vector of integers between 1 through 10 as `integers` using the command

```
integers <- 1:10
```

We read it as "integers 'gets' 1 through 10."

Okay, let's load in the **autos** data and save it to the object `carsdata`:

In [7]:
```
carsdata<-read_dta("autos.dta")
```

The dataset has now been stored to memory under the name `carsdata`. To view it in our notebook, we can just type the name and hit run... or if we want to view only the first few lines of the dataset we can use the `head()` command to see only a few observations.

In [8]:
```
head(carsdata)
```

A tibble: 6 × 12

| make | price | mpg | rep78 | headroom | trunk | weight | length | turn | displacement | gear_ratio | foreign |
|---|---|---|---|---|---|---|---|---|---|---|---|
| <chr> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl+lbl> |
| AMC Concord | 4099 | 22 | 3 | 2.5 | 11 | 2930 | 186 | 40 | 121 | 3.58 | 0 |
| AMC Pacer | 4749 | 17 | 3 | 3.0 | 11 | 3350 | 173 | 40 | 258 | 2.53 | 0 |
| AMC Spirit | 3799 | 22 | NA | 3.0 | 12 | 2640 | 168 | 35 | 121 | 3.08 | 0 |
| Buick Century | 4816 | 20 | 3 | 4.5 | 16 | 3250 | 196 | 40 | 196 | 2.93 | 0 |
| Buick Electra | 7827 | 15 | 4 | 4.0 | 20 | 4080 | 222 | 43 | 350 | 2.41 | 0 |
| Buick LeSabre | 5788 | 18 | 3 | 4.0 | 21 | 3670 | 218 | 43 | 231 | 2.73 | 0 |

The very first bit of information tells us the format of the data (data frame). The first row tells us the variable names, and the second row tells us the variable type (character string or type of numeric variable). Each row in the table is a different observation - here giving us info on a car make and model, its price, mpg, and other characteristics.

If we forget the info in our dataframe, we can check the names of the variables:

In [9]:
```
names(carsdata)
```

'make' · 'price' · 'mpg' · 'rep78' · 'headroom' · 'trunk' · 'weight' · 'length' · 'turn' · 'displacement' · 'gear_ratio' · 'foreign'

If we want to interact with just one column of our data frame, we can refer to it using the `$` command. This is useful if we're interested in, say, obtaining the mean of only miles per gallon for the cars in our sample.

In [10]:
```
avg_mpg <- mean(carsdata$mpg)
avg_mpg
```

21.2972972972973

### Other File Formats

If we wanted to load a .csv file, we'd want to use the `read.csv()` command to load it, once again using quotation marks around the file name and assigning it a name.

## Manipulating Data Frames

We have a lot of flexibility to select certain observations, certain variables, or certain values within our data frame. We can also perform a lot of operations to variables - change their values or create new variables. One of the packages we'll be using for this is the **tidyverse** package. It's actually a collection of packages designed for data science, and includes a number that we'll use throughout this term.

To start, let's load **tidyverse** and use it to perform transformations on our dataset.

In [12]:
```
library(tidyverse)

carsdata_low_price <- filter(carsdata, price <= 10000)
head(carsdata_low_price)

carsdata_low_price <- arrange(carsdata_low_price, desc(price))
head(carsdata_low_price)

carsdata_low_price <- select(carsdata_low_price, make, price, mpg, weight)
head(carsdata_low_price)
```

A tibble: 6 × 12

| make | price | mpg | rep78 | headroom | trunk | weight | length | turn | displacement | gear_ratio | foreign |
|---|---|---|---|---|---|---|---|---|---|---|---|
| <chr> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl+lbl> |
| AMC Concord | 4099 | 22 | 3 | 2.5 | 11 | 2930 | 186 | 40 | 121 | 3.58 | 0 |
| AMC Pacer | 4749 | 17 | 3 | 3.0 | 11 | 3350 | 173 | 40 | 258 | 2.53 | 0 |
| AMC Spirit | 3799 | 22 | NA | 3.0 | 12 | 2640 | 168 | 35 | 121 | 3.08 | 0 |
| Buick Century | 4816 | 20 | 3 | 4.5 | 16 | 3250 | 196 | 40 | 196 | 2.93 | 0 |
| Buick Electra | 7827 | 15 | 4 | 4.0 | 20 | 4080 | 222 | 43 | 350 | 2.41 | 0 |
| Buick LeSabre | 5788 | 18 | 3 | 4.0 | 21 | 3670 | 218 | 43 | 231 | 2.73 | 0 |

A tibble: 6 × 12

| make | price | mpg | rep78 | headroom | trunk | weight | length | turn | displacement | gear_ratio | foreign |
|---|---|---|---|---|---|---|---|---|---|---|---|
| <chr> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl> | <dbl+lbl> |
| BMW 320i | 9735 | 25 | 4 | 2.5 | 12 | 2650 | 177 | 34 | 121 | 3.64 | 1 |
| Audi 5000 | 9690 | 17 | 5 | 3.0 | 15 | 2830 | 189 | 37 | 131 | 3.20 | 1 |
| Olds 98 | 8814 | 21 | 4 | 4.0 | 20 | 4060 | 220 | 43 | 350 | 2.41 | 0 |
| Datsun 810 | 8129 | 21 | 4 | 2.5 | 8 | 2750 | 184 | 38 | 146 | 3.55 | 1 |
| Buick Electra | 7827 | 15 | 4 | 4.0 | 20 | 4080 | 222 | 43 | 350 | 2.41 | 0 |
| VW Dasher | 7140 | 23 | 4 | 2.5 | 12 | 2160 | 172 | 36 | 97 | 3.74 | 1 |

A tibble: 6 × 4

| make | price | mpg | weight |
|---|---|---|---|
| <chr> | <dbl> | <dbl> | <dbl> |
| BMW 320i | 9735 | 25 | 2650 |
| Audi 5000 | 9690 | 17 | 2830 |
| Olds 98 | 8814 | 21 | 4060 |
| Datsun 810 | 8129 | 21 | 2750 |
| Buick Electra | 7827 | 15 | 4080 |
| VW Dasher | 7140 | 23 | 2160 |

Phew, that was a lot of stuff! Let's back up and work through it.

The first thing we did after loading the **tidyverse** package was to `filter()` our data frame. Here we are selecting only the observations with price less than or equal to $10,000. We save this to the new `carsdata_low_price` object.

Next, we arrange the new `carsdata_low_price` data frame in descending order of price using the `arrange()` function and overwrite our `carsdata_low_price` with this new arrangement.

Finally, we `select()` only a few variables - here we choose to keep just vehicle make, price, and miles per gallon - and once again overwrite our `carsdata_low_price` object.

Now we'll create new versions of our variables using the `mutate()` command. If I wanted to rescale price to be in units of $1,000, we can do this by typing

In [13]:

```
carsdata_low_price <- mutate(carsdata_low_price, price_thousand = price/1000)
head(carsdata_low_price)
```

A tibble: 6 × 5

| make | price | mpg | weight | price_thousand |
|---|---|---|---|---|
| <chr> | <dbl> | <dbl> | <dbl> | <dbl> |
| BMW 320i | 9735 | 25 | 2650 | 9.735 |
| Audi 5000 | 9690 | 17 | 2830 | 9.690 |
| Olds 98 | 8814 | 21 | 4060 | 8.814 |
| Datsun 810 | 8129 | 21 | 2750 | 8.129 |

| make | price | mpg | weight | price_thousand |
|---|---|---|---|---|
| <chr> | <dbl> | <dbl> | <dbl> | <dbl> |
| Buick Electra | 7827 | 15 | 4080 | 7.827 |
| VW Dasher | 7140 | 23 | 2160 | 7.140 |

Here we are replacing `carsdata_low_price` object with itself after adding in a new variable names `price_thousand` that is `price` divided by 1000.

## Summarizing Variables

There are a number of different ways for us to obtain information about our variables in **R**. While we saw earlier one way to directly get the mean of a variable, we could instead have used the `summarise` command, which lets us obtain a number of different statistics - either one at a time, or multiple together

In [14]:

```
avg_mpg <- summarise(carsdata_low_price, mean(mpg))
avg_mpg

multi_stats <- summarise(carsdata_low_price, max(mpg), min(price), n())
multi_stats
```

A tibble: 1 × 1

| mean(mpg) |
|---|
| <dbl> |
| 22.28125 |

A tibble: 1 × 3

| max(mpg) | min(price) | n() |
|---|---|---|
| <dbl> | <dbl> | <int> |
| 41 | 3291 | 64 |

`summarise()` is useful because we can use it to select specific summary statistics we're insterested in or create truly custom summary stats. If instead we wanted to get a bunch of basic stats all at once, we could use the `summary()` command for a given variable - say to get information on the variable `price`. We can refer to specific variables in a data frame with a `$`. So for example, `carsdata_low_price$price`.

In [15]:

```
summary(carsdata_low_price$price)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   3291    4179    4728    5159    5798    9735 
```

## Defining New Variables

Suppose we want to add new columns to our data frame. This can be done easily, by simply defining them with our `$` notation. Suppose we want to make a new column called price_sq which is the squared price of each car. We can do so by defining `carsdata_low_price$price_sq<-` (with an appropriately defining expression on the right side). To do this, note that you can call other columns and values you already have defined or loaded in to R. R also uses most basic math symbols in the ways you would expect: `+`, `-`, `*`, `\`, and `^`. Try it below!. Note: Be careful with order of operations and use parentheses when necessary.

In [16]:

```
carsdata_low_price$price_sq <- carsdata_low_price$price^2
carsdata_low_price$price_sq
```

94770225 · 93896100 · 77686596 · 66080641 · 61261929 · 50979600 · 46922500 · 42068196 · 40220964 · 39727809 · 39627025 · 38800441 · 38007225 · 34798201 · 34644996 · 33628401 · 33616804 · 33500944 · 32706961 · 32547025 · 29127609 · 28933641 · 27269284 · 26925721 · 26749584 · 26050816 · 25796241 · 24344356 · 23912100 · 23193856 · 22553001 · 22401289 · 22306729 · 22061809 · 21594609 · 21058921 · 20394256 · 20286016 · 20241001 · 20088324 · 19829209 · 19580625 · 19571776 · 19263321 · 18455616 · 17598025 · 17530969 · 17480761 · 17405584 · 16801801 · 16662724 · 16483600 · 16080100 · 15960025 · 15872256 · 15642025 · 15171025 · 14661241 · 14432401 · 14424804 · 14047504 · 13446889 · 10883401 · 10830681

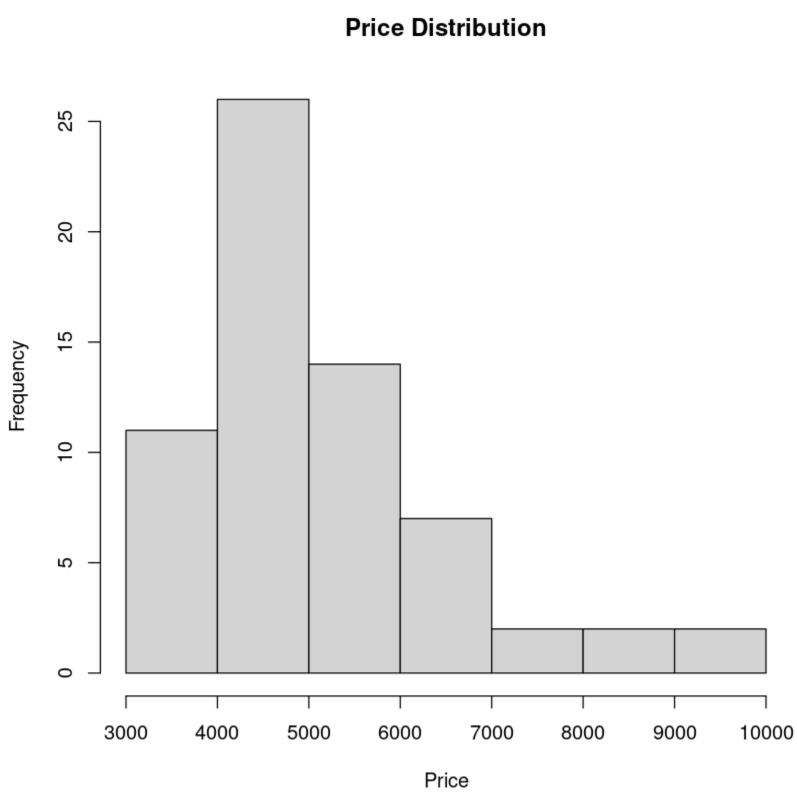
## Plotting in R

Today we're going to learn how to use the basic plot function in **R**. In the coming weeks, we're going to learn **ggplot2**, a fantastic graphics package that is one of the great reasons for using **R** to produce graphics. Its syntax is a little bit more involved, so we're going to start off with some simple functions.

First, we can generate a histogram of vehicle prices using the `hist()` command.

In [17]:

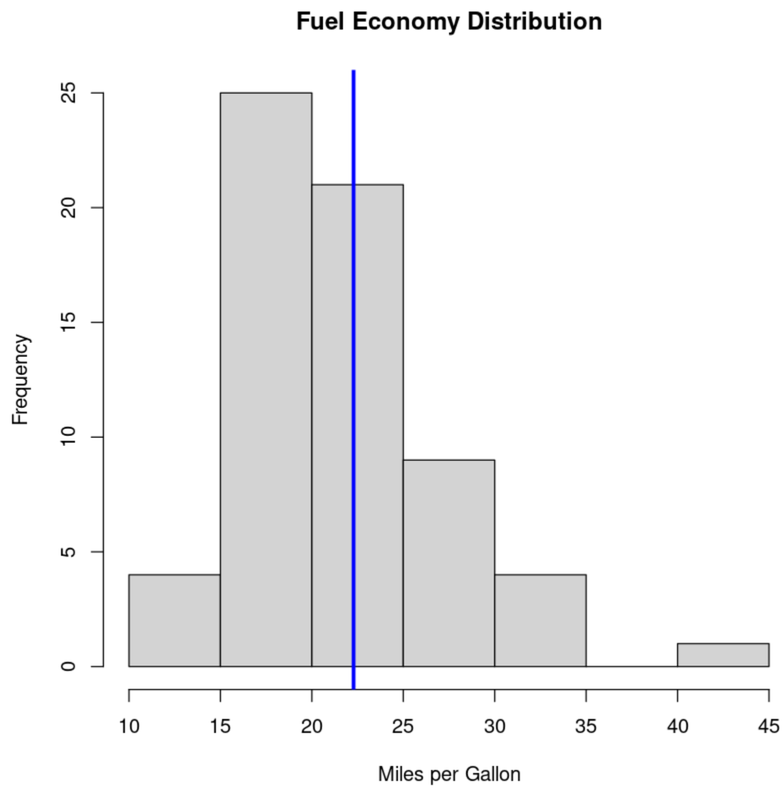
```
hist(carsdata_low_price$price,
    main = "Price Distribution",
    xlab = "Price")
```




Now let's look at fuel economy, and add a blue line of width three at the average mpg:

In [18]:

```
hist(carsdata_low_price$mpg,
    main = "Fuel Economy Distribution",
    xlab = "Miles per Gallon")
abline(v = avg_mpg, col = "blue", lwd = 3)
```




We can also make a scatterplot of vehicle price and fuel efficiency using the `plot()` function.

In [19]:
```
plot(carsdata_low_price$mpg, carsdata_low_price$price, # x-axis first, y-axis second
    main = "Price vs. Fuel Efficiency",
    xlab = "Miles per Gallon",
    ylab = "Price ($)") # don't forget the last parenthesis!
```


There seems to be a negative correlation here: the higher MPG, the lower the price. Is this potentially an SUV effect?

## Regression

Running regressions with **R** is quite easy. Later in the course we'll get into some more complex regression commands, but for now we'll stick with simple linear regression using the `lm()` command.

First, let's take a step back and see how mpg and price are correlated in the data.

In [20]:

```
mpg_price_cor <- cor(carsdata_low_price$mpg, carsdata_low_price$price)
mpg_price_cor
```

-0.258800719701856

What if we used a simple OLS regression instead?

In [21]:

```
mpg_price_regression <- lm(mpg ~ price, data = carsdata_low_price)
# basic syntax: lm(depvar ~ indvar1 + indvar2 + ..., data=data)
mpg_price_regression
```

```
Call:
lm(formula = mpg ~ price, data = carsdata_low_price)

Coefficients:
(Intercept)        price
  27.499087    -0.001011
```

Our `mpg_price_regression` object contains our regression coefficients as well as a bunch of other objects, most notably residuals and fitted values.

In [22]:

```
head(mpg_price_regression$fitted.values)
mean(mpg_price_regression$residuals)
```

**1:** 17.652375663974 **2:** 17.6978920511395 **3:** 18.5839443879621 **4:** 19.2768049481488 **5:** 19.5822704797931 **6:** 20.2771539905204

-7.63278329429795e-17

# Help with R

The internet is your best friend. While qw will introduce you to some R commands during section, you may find that there are times where you are having trouble with a function's syntax or need a new function to perform a certain task. Asking Google, using the R Documentation site as well as R Project Package Reference Manuals/Vignettes for help with functions and syntax will get you quite far. If something is unclear, searching for the R task and perusing answers on StackExchange can be a helpful resource. We will try our best to make sure homework assignments mention the functions/packages needed for a new task, and that we've at least seen them in section or lecture.