Introduction to R Notebooks

Matthew Alampay Davis

May 30, 2023

This is an R Markdown Notebook, which will be the format for your problem set submissions if you choose to use R. These notebooks are great for our purposes because you can edit them like a document and also code within it and view the resulting output right beneath the code. R Notebooks thus conveniently combine word processing, equation type-setting, and coding all within an intuitive user interface and with a very good-looking output.

Let's start by creating headings and subheadings. These will be useful for problem set submissions so you can easily distinguish questions (e.g., "Question 1"), sub-questions (e.g., "1b"), and sub-sub-questions (e.g.,"1b part i"):

Part 1: Introduction to R Notebooks

The pound sign (#) here created this subheading. You can't see it if you're reading this in RStudio, but once you render this notebook (i.e. tell R to convert the notebook into a pdf file), it will appear as bolder and larger than standard text. You can use multiple pound signs to create a subheading (##) or a sub-subheading (###) and so on:

1a: Headings

1a-i: Subheadings

Like so!

To see this difference, click on the *Knit* button at the top and select *Knit to pdf.* Knitting is just what R calls the process of converting a notebook to a properly formatted document. When you knit a notebook, the Notebook file is saved and R attempts to render it into a document (either HTML or pdf). If successful, a document with the same name will appear in the same folder as the Notebook file. You can preview what this document looks like in RStudio by clicking the *Preview* button or pressing *Cmd+Shift+K* (*Ctrl+Shift+K* for PCs).

1b: Typesetting equations

Notice that using asterisks like this makes that text appear as italicized. Another formatting function is the ability to write out equations. To do so, you should run the following expression in the console panel of RStudio:

tinytex::install_tinytex()

That is to say, just copy the above into the console panel of RStudio (you'll notice RStudio's interface is divided into four panels) and press enter to download this LaTeX distribution. You only ever need to do this once per computer you use so no need to think about this again after running this once. This allows us to format equations like the one below:

$$y_i = \beta_0 + \beta_1 x_i + e_i$$

This way of typesetting uses a language called TeX very commonly used in academic writing. For our purposes, this is all we really need to know:

- sandwich the equations with single dollar signs to include an equation in-line and double dollar signs to include an equation as a centered standalone line
- backslash + text for non-italicized text: example text
- backslash + (case-sensitive) name for Greek letters: γ and Γ
- underscore + curly brackets to subscript: γ_{sub}
- carat + curly brackets to superscript: γ^{super} or combine both: γ_3^2
- backslash + 'frac' + two sets of curly brackets for fractions: $\frac{numerator}{denominator}$
- backslash + 'widehat' to add a 'hat' to denote estimators: $\widehat{\beta_1}$
- backslash + 'times' for a multiplication sign
- backaslash + 'leq' or 'geq' produces the "less/greater than or equal to" inequality signs

Here's a meaningless equation using all these commands:

$$\widehat{Example} \geq \beta_0 + 9.7636 \times Dosage_i + \frac{numerator^2}{denominator} + \epsilon$$

1c: Code chunks

To start coding in an R Notebook, add a new chunk by clicking the *Insert Chunk* button on the RStudio toolbar or by pressing *Cmd+Option+I* (*Ctrl+Alt+I* for PCs):

```r
# An example command
2+2
```

```
## [1] 4
```

Here, we've commanded R to compute 2+2 and if you run this command (either by pressing *Cmd+Enter* (*Ctrl+Enter* for PCs) while in a chunk or by clicking on the green *Run* button at the top right of the chunk, you can see the resulting output beneath the chunk. When we convert this notebook into a pdf, we'll be able to see both the code and the output in-line. If you want to omit both, you can replace the first line of the chunk with "r, include = FALSE". See the first chunk of the PS1 Practice Problems for a example doing this.

Part 2: Data

2a: Exploring some data

Let's do something a tiny bit more complicated. "cars" is a practice dataset that's built into R as an example of a dataset. It consists of two variables, "speed" and "dist". We can see a quick preview of this dataset by using the *head* command, which shows us the first six rows of a dataset:

```r
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

Alternatively, we might want to just see the first three observations:

```r
head(cars, 3)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

or the last three observations:

```r
tail(cars, 3)
```

```
##    speed dist
## 48    24   93
## 49    24  120
## 50    25   85
```

Or we might want to get summary statistics about each variable:

```r
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00  
```

In these examples, we are using different functions (head, tail, summary). These functions take "arguments" in parentheses that are separated by commas. The first argument was the dataset we wanted to use (cars) and the second is a number n (3) for the number of observations we wanted to see. Different functions have different arguments. If you ever need to know what arguments a given function has or how to use the function, just type a question mark followed by the function into the console:

```r
?head
```

2b: Creating data

The standard format for datasets in R is a data.frame consisting of variables as columns and observations as rows. Here's an example of another data.frame called ToothGrowth that also comes native to R:

```
head(ToothGrowth)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

For context, these are the results from an experiment studying the effect of vitamin C on tooth growth in 60 Guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods: orange juice (OJ) or ascorbic acid (VC).

We can reproduce this dataset by hand. Let's create the first six observations seen above by defining objects called vectors. We use the "c" command to do this.

```
x1 <- c(4.2, 11.5, 7.3, 5.8, 6.4, 10.0)
x2 <- c('VC', 'VC', 'VC', 'VC', 'VC', 'VC')
x3 <- c(0.5, 0.5, 0.5, 0.5, 0.5, 0.5)
```

Notice that when we input words, we have to put them in quotation marks but numbers are fine as they are.

We can then easily combine these into a data.frame object that we'll call "tooth.data"

```
tooth.data <- data.frame(x1, x2, x3)
tooth.data
```

```
##     x1 x2  x3
## 1  4.2 VC 0.5
## 2 11.5 VC 0.5
## 3  7.3 VC 0.5
## 4  5.8 VC 0.5
## 5  6.4 VC 0.5
## 6 10.0 VC 0.5
```

Which is a nice recreation of the original dataset (at least its first six rows). We also could've very easily assigned names to the variables:

```
tooth.data <- data.frame(len = x1,
                         supp = x2,
                         dose = x3)
tooth.data
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

data.frames make it convenient to perform various statistical commands on the data. It allows us to access data just by referring to the data.frame *tooth.data* and a variable like *len* or *supp* or *dose* by combining them with a dollar sign symbol: for example, *tooth.data$len* extracts all values of the variable *len* as a vector of values.

Let's look at some applications of this, using the full set of observations in the original dataset instead of just the first 6. So first let's just redefine tooth.data as the original dataset:

```
tooth.data <- ToothGrowth
dim(tooth.data) # Vector of two elements: # of rows then # of columns
```

```
## [1] 60  3
```

Now we can compute some basic statistics:

```
# Finding the mean
mean(tooth.data$len)
```

```
## [1] 18.81333
```

```
# Finding the standard deviation and variance
sd(tooth.data$len)
```

```
## [1] 7.649315
```

```
var(tooth.data$len)
```

```
## [1] 58.51202
```

```
# A general summary of the data
summary(tooth.data)
```

```
##       len        supp         dose      
##  Min.   : 4.20   OJ:30   Min.   :0.500  
##  1st Qu.:13.07   VC:30   1st Qu.:0.500  
##  Median :19.25           Median :1.000  
##  Mean   :18.81           Mean   :1.167  
##  3rd Qu.:25.27           3rd Qu.:2.000  
##  Max.   :33.90           Max.   :2.000
```

```
# Correlation between length and dose
cor(tooth.data$len, tooth.data$dose)
```

```
## [1] 0.8026913
```

```
# Covariance between length and dose
cov(tooth.data$len, tooth.data$dose)
```

```
## [1] 3.861299
```

2c: Installing packages

The functions we've used so far are those the basic ones which come pre-installed with "base R". We'll regularly want to use other functions that don't come in-built into R but were written by third-party developers. Collections of these related functions are called a "package" which we'll have to download from the internet. You'll find that we'll rely on these third-party packages for all our problem sets; I'll introduce new ones each week that will be needed to solve the next problem set.

To download a package, for example the the "ggplot2" package for plotting data, run the following commmand in the console:

install.packages('ggplot2') # Make sure the name of the package you want to download is in quotation marks

For a given package, you'll only ever need to do this once per computer.

2d: Plotting data

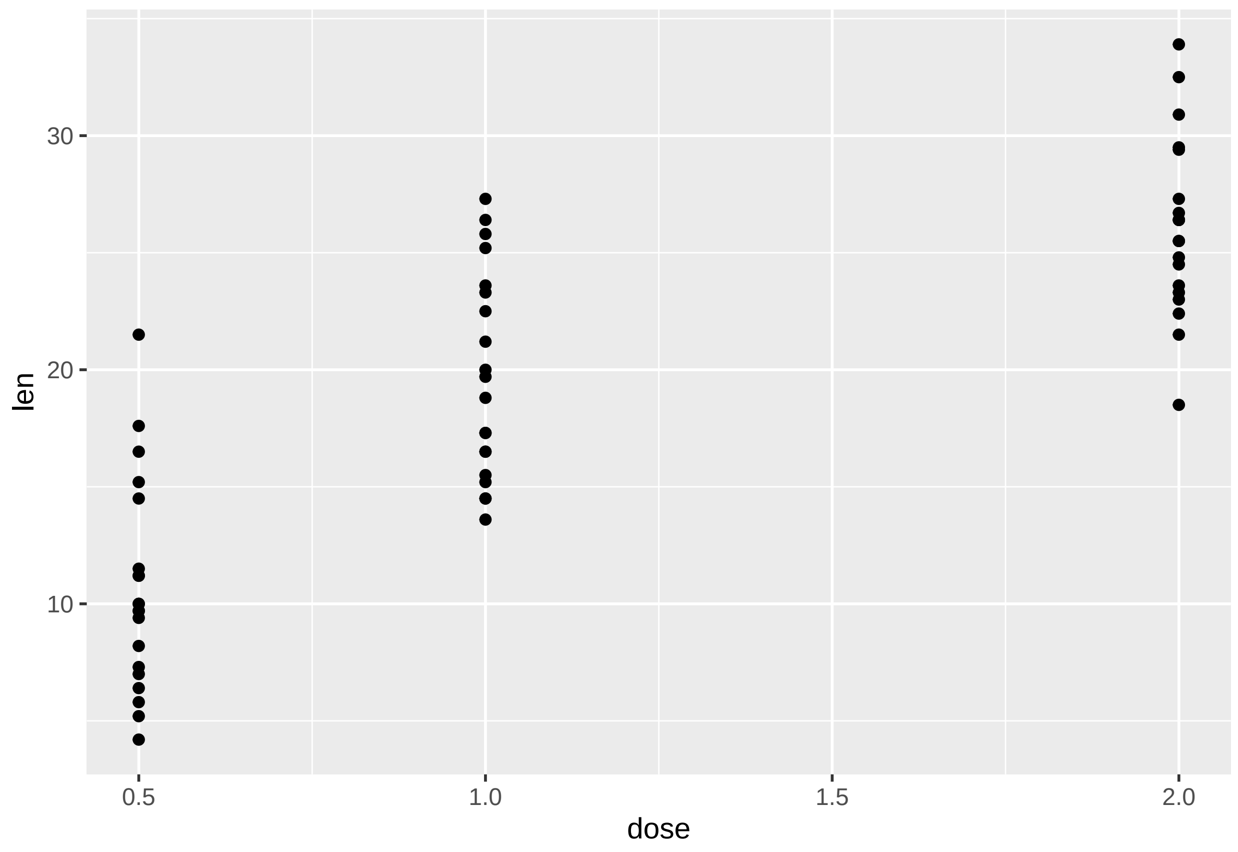
The package we installed specializes in creating easily customizable graphs. To load all the functions that come with ggplot2, use the library command:

```
library(ggplot2) # No need for quotation marks this time
```

Note that the library command only works on packages that we've already installed.

Now look at this command which plots the tooth data we had been looking at:

```
ggplot(data = tooth.data, aes(x = dose, y = len)) +
  geom_point()
```



Let's break this command down:

- ggplot() is a function
- The first argument of this function is the "data" argument which refers to the data.frame we're plotting
- The second argument is "aes" (short for aesthetic) which itself has additional arguments: x = dose and y = len tells it to use the column named 'dose' as the independent variable and 'len' as the dependent variable

ggplot has a unique grammar. The first line inputs the data and variables in that data that we want to plot. We ended this line with a "+" to say we want to add an additional plot element in the next line. In particular, we wanted a scatterplot so in line 2, we used the geom_point() function. This grammar takes some practice to get used to but once you get a hang of this sort of coding grammar, it allows us to make a lot of intuitive customizations as we'll see in the practice problems.

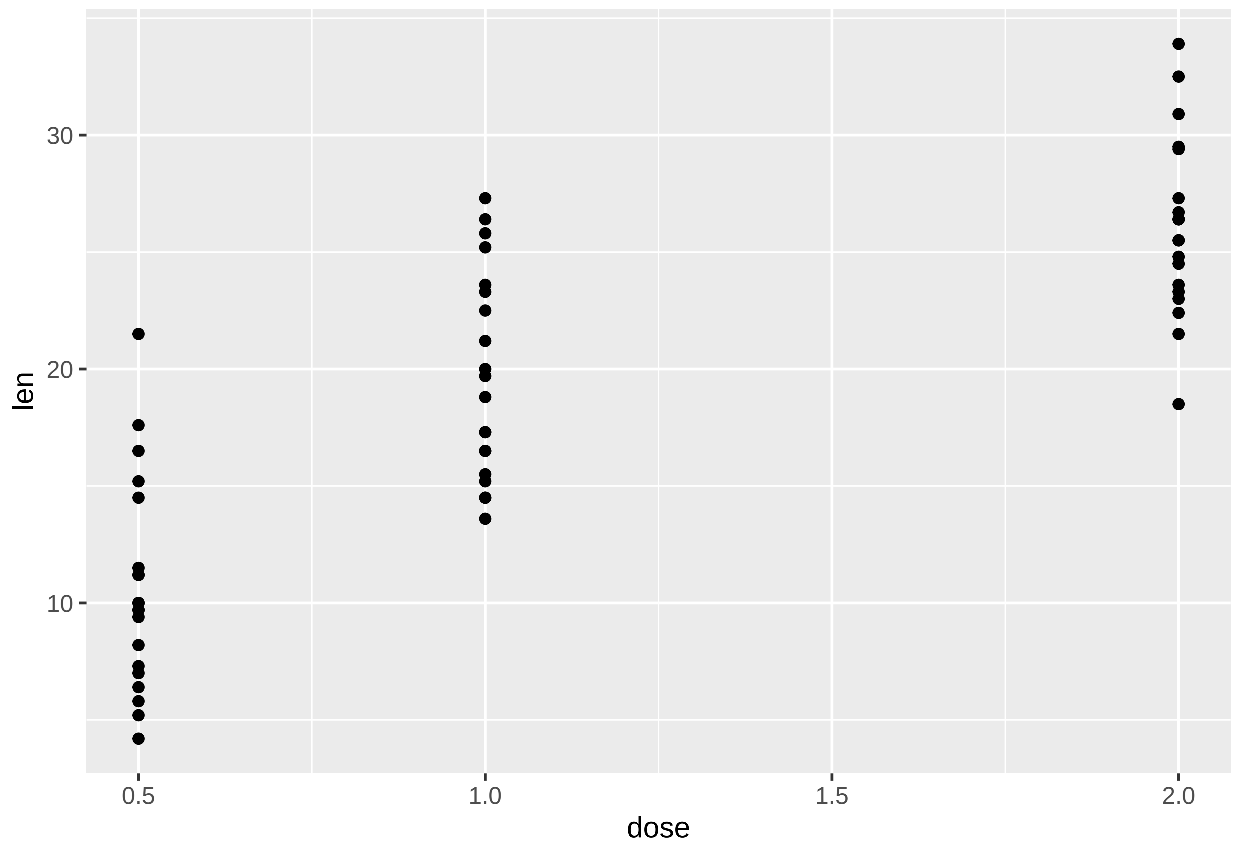
Let us save the graph above as an object. To do this, we come up with some name (let's say "test.plot") and assign the graph to it using the "<-" assignment:

```
test.plot <- ggplot(data = tooth.data, aes(x = dose, y = len)) +
  geom_point()
```

To repeat, the first line calls the ggplot function and tells it what dataset we want to use and which variables we want to use as our x and y variables. The second line choose the kind of graph we want, a scatterplot. The "+" links the two lines as one command. We've assigned this graph object the name "test.plot" using "<-"

Now we have an object called test.plot which is the above graph. You can see a list of all the objects in our 'environment' in the 'Environment' panel in RStudio. We can plot this object:

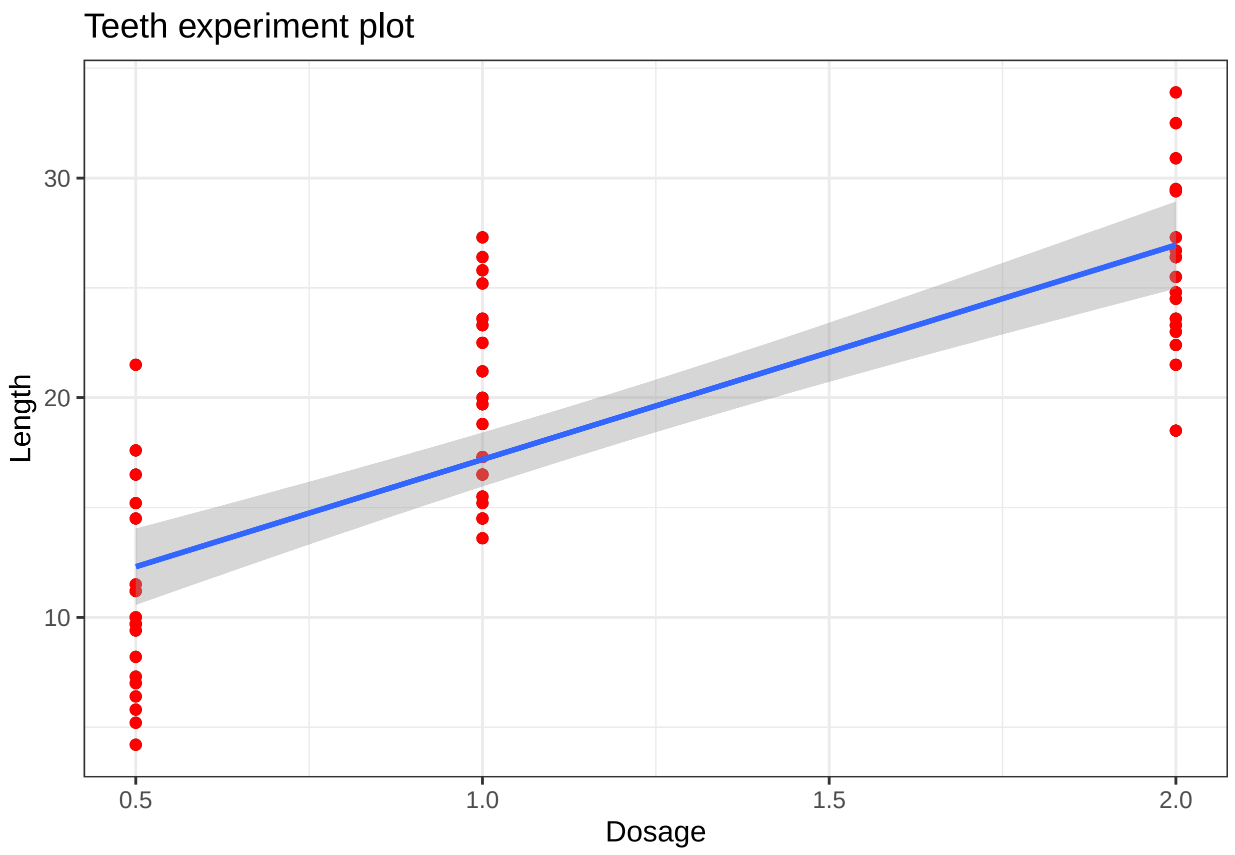
```
test.plot
```



And we can make some new modifications, thanks to the grammar of ggplot2. Let's define the modified plot as a new object called test.plot2:

```
test.plot2 <- test.plot +
  # Change the point colors to red
  geom_point(col = 'red') +
  # Add a line of best fit ('lm' means 'linear model') and include a confidence interval (replace with
  geom_smooth(method = 'lm', se = TRUE) +
  # Modify the axis titles
  ylab('Length') +
  xlab('Dosage') +
  ggtitle('Teeth experiment plot') +
  # Simplify the plot theme to black-and-white
  theme_bw()
test.plot2
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



Notice that we can add comments to our chunks of code by using "#" at the start of a line. Anything after the "#" will be ignored by R until the next line of code. This is useful for communicating to yourself or your reader what each line does.

As another plotting example, see the PS1 Practice Problems for an example where we plot labels instead of points in a scattergraph.

2e: Loading external datasets

More commonly, we'll load pre-existing data and so we'll want to be able to load data we've saved somewhere on our computer. Again, the best way of organizing your data to make this easier is to just include a folder called 'data' in the pset folder. This makes it very easy to locate the data we want to use, as we'll see next.

I have a Stata dataset called "animals.dta" (all Stata datasets end with .dta) which I've saved in a "data" folder located in the same folder as this Notebook. To do so, first note that since R by default cannot open Stata files, we must download/install a package that can. So just like we did with the ggplot package, you'll want to install and load this package if you haven't already:

```
library(readstata13)
```

Then we'll use the "read.dta13" command from this package to read the file in our working directory. Let's name the dataset "animals" by using the assignment notation from before:

```
animals <- read.dta13('data/animals.dta')
```

Note you put the filename/filepath 'data/animals.dta' in quotes since it is not an object in our environment. This tells it to look for a file called *animals.dta* in a folder called *data* located in the same folder as the Notebook file.

Let's get a quick summary of this data:

```
dim(animals) # What are the dimensions (number of rows, number of columns) of this dataset
```

```
## [1] 790   7
```

```
nrow(animals) # Same as above, but just the number of rows
```

```
## [1] 790
```

```
ncol(animals) # The number of columns
```

```
## [1] 7
```

```
head(animals) # The first few observations
```

```
##   village hhn id   animal          type number price
## 1       1   1  1    Goats   Calves-Male      3 15000
## 2       1   1  1 Chickens        Layers      3  3000
## 3       1   1  1    Goats Calves-Female      2 15000
## 4       1   1  1    Goats        Female      4 25000
## 5       1   2  0    Goats Calves-Female      3  9000
## 6       1   2  0    Sheep   Calves-Male      3    dk
```

```
summary(animals) # A brief summary of each variable in the dataset
```

```
##     village           hhn              id            animal
##  Min.   :1.000   Min.   : 1.00   Min.   :0.0000   Length:790
##  1st Qu.:2.000   1st Qu.:16.00   1st Qu.:0.0000   Class :character
##  Median :3.000   Median :33.00   Median :0.0000   Mode  :character
##  Mean   :2.647   Mean   :34.33   Mean   :0.4924
##  3rd Qu.:4.000   3rd Qu.:50.75   3rd Qu.:1.0000
##  Max.   :4.000   Max.   :99.00   Max.   :2.0000
##      type              number             price
##  Length:790         Length:790         Length:790
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
```

We can see that different variables are of different types: village, hhn, and id are numbers (that's why we can compute its mean and maximum) while others are "characters" or "strings", i.e. its values are text entries like "Goats" or "Chickens". We will be interested in both types throughout this course.

Another way we can summarize this data is by tabulation. This lets us look at the frequency of each value of a variable. For example:

```
table(animals$animal)
```

```
##
##      Chickens          Ducks          Goats   Guinea Fowl Other Poultry
##           307             30            251             5             4
##           Pig        Rabbits          Sheep        Turkey
##             4              3            180             6
```

This tells us how often each animal appears in our dataset. Notice again that we can refer to a specific column in the animals dataset by its name using the "$" character. "animals$animal" refers to the column named "animal" in the "animals" object (i.e. dataset in this case) and extracts that specific column as a vector. We could have also referred to the animals$village column. We'll use this often.

2f: Cleaning data by subsetting

One of the packages we'll make most use out of is 'tidyverse', which is really a collection of other packages with similar grammar. 'tidy' here simply refers to cleaning data: manipulating or changing it into a format we find convenient. We'll get to know the 'tidyverse' functions well over the coming weeks but for now, here's a quick first example:

Let's summarize our animals

```
head(animals)
```

```
##   village hhn id   animal          type number price
## 1       1   1  1    Goats   Calves-Male      3 15000
## 2       1   1  1 Chickens        Layers      3  3000
## 3       1   1  1    Goats Calves-Female      2 15000
## 4       1   1  1    Goats        Female      4 25000
## 5       1   2  0    Goats Calves-Female      3  9000
## 6       1   2  0    Sheep   Calves-Male      3    dk
```

```
dim(animals)
```

```
## [1] 790   7
```

Suppose we don't want to use the full 790 observations of data here. In particular, maybe we are only interested in the subsample from village 1 and 3. We can create this subset using the *filter* function:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v tibble  3.2.1     v dplyr   1.1.2
## v tidyr   1.3.0     v stringr 1.5.0
## v readr   2.1.3     v forcats 1.0.0
## v purrr   1.0.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
table(animals$village)
```

```
##
##   1   2   3   4
## 196 118 245 231
```

```
animals.filt <- filter(animals, village %in% c(1,3))
table(animals.filt$village)
```

```
##
##   1   3
## 196 245
```

Here, the first argument is the dataset we want to subset and the second argument is the condition that must be satisfied for an observation to remain in the new dataset. The expression provided means we keep all observations with a value of 1 or 3 for the variable 'village'.

Here's an example filtering on a character vector instead of a numerical vector:

```
table(animals$animal)
```

```
##
##       Chickens          Ducks          Goats    Guinea Fowl Other Poultry
##            307             30            251              5             4
##            Pig        Rabbits          Sheep         Turkey
##              4              3            180              6
```

```
animals.filt2 <- filter(animals, animal != 'Chickens')
table(animals.filt2$animal)
```

```
##
##          Ducks          Goats    Guinea Fowl Other Poultry            Pig
##             30            251              5             4              4
##        Rabbits          Sheep         Turkey
##              3            180              6
```

The "!=" notation means "not equal to". We've removed all chicken observations from our dataset.

Equivalently, we can also reduce our dataset by keeping all observations but subsetting the number of columns/variables. We use the *select* function, also from *tidyverse*:

```
head(animals)
```

```
##   village hhn id   animal           type number price
## 1       1   1  1    Goats    Calves-Male      3 15000
## 2       1   1  1 Chickens         Layers      3  3000
## 3       1   1  1    Goats  Calves-Female      2 15000
## 4       1   1  1    Goats         Female      4 25000
## 5       1   2  0    Goats  Calves-Female      3  9000
## 6       1   2  0    Sheep    Calves-Male      3    dk
```

```
animals.select <- select(animals, village, hhn, id, animal, price)
head(animals.select)
```

```
##   village hhn id   animal price
## 1       1   1  1    Goats 15000
## 2       1   1  1 Chickens  3000
## 3       1   1  1    Goats 15000
## 4       1   1  1    Goats 25000
## 5       1   2  0    Goats  9000
## 6       1   2  0    Sheep    dk
```

Equivalently, we can instead identify the variables we want to remove:

```
animals.select2 <- select(animals, -c(type, number))
head(animals.select2)
```

```
##   village hhn id   animal price
## 1       1   1  1    Goats 15000
## 2       1   1  1 Chickens  3000
## 3       1   1  1    Goats 15000
## 4       1   1  1    Goats 25000
## 5       1   2  0    Goats  9000
## 6       1   2  0    Sheep    dk
```

Part 3: Regressions

3a: Regression models with homoskedastic standard errors

The main thing we'll learn in this course is how to run regression models of various types. Let's use a dataset called 'cars' that comes in-built into R. Here's what it looks like:

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00  
```

Suppose we want to run a very simple univariate regression of speed on distance. Let's create a 'model object' called "cars.model" and use the "lm" function (short for linear model) to estimate a regression:

```
cars.model <- lm(speed ~ dist, data = cars)
```

Here, the first argument is a formula "speed ~ dist" meaning speed is our outcome variable on the LHS and distance is our single regressor on the RHS. The 'data' argument tells us which object/dataset to look for these variables. This command then gives us a new type of object named "cars.model". If we wanted to print the regression output, we would use the "summary" function on this object:

```
summary(cars.model)
```

```
##
## Call:
## lm(formula = speed ~ dist, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.5293 -2.1550  0.3615  2.4377  6.4179
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  8.28391    0.87438   9.474 1.44e-12 ***
## dist         0.16557    0.01749   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.156 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

We can clearly see this gives us a y-intercept (i.e. a constant) of 8.28391 and a coefficient estimate of 0.16557 on distance. We interpret this as saying "an increase in distance by 1km (or whatever unit distance is in) is associated with an increase in speed of 0.16557." The intercept and distance coefficients each have standard errors, t-values, and p-values clearly associated with them, which we care about for inference.

We can do some more things with our model object using the 'lmtest' package (for running tests on these linear models). For example:

```
library(lmtest)
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```r
# If we only cared about the coefficients:
coef(cars.model)
```

```
## (Intercept)        dist
##   8.2839056   0.1655676
```

```r
# If we want to extract the residuals
cars.model$residuals
```

```
##           1           2           3           4           5           6
## -4.61504079 -5.93958139 -1.94617594 -4.92639228 -2.93298684 -0.93958139
##           7           8           9          10          11          12
## -1.26412199 -2.58866258 -3.91320318 -0.09855441 -1.91979773  1.39814831
##          13          14          15          16          17          18
##  0.40474287 -0.25752743 -0.91979773  0.41133742 -0.91320318 -0.91320318
##          19          20          21          22          23          24
## -2.90001408  1.41133742 -0.24433833 -4.21796012 -7.52931161  3.40474287
##          25          26          27          28          29          30
##  2.41133742 -2.22455467  2.41793197  1.09339137  3.41793197  2.09339137
##          31          32          33          34          35          36
##  0.43771563  2.76225622  0.44431018 -2.86704131 -4.19158191  4.75566167
##          37          38          39          40          41          42
##  3.09998592 -0.54250072  6.41793197  3.76885078  3.10658048  2.44431018
##          43          44          45          46          47          48
##  1.11976958  2.78863443  5.77544533  4.12636413  0.48387749  0.31830992
##          49          50
## -4.15201460  2.64285051
```

```r
# If we want to extract the predicted/fitted values
cars.model$fitted.values
```

```
##         1         2         3         4         5         6         7         8
##  8.615041  9.939581  8.946176 11.926392 10.932987  9.939581 11.264122 12.588663
##         9        10        11        12        13        14        15        16
## 13.913203 11.098554 12.919798 10.601852 11.595257 12.257527 12.919798 12.588663
##        17        18        19        20        21        22        23        24
## 13.913203 13.913203 15.900014 12.588663 14.244338 18.217960 21.529312 11.595257
##        25        26        27        28        29        30        31        32
## 12.588663 17.224555 13.582068 14.906609 13.582068 14.906609 16.562284 15.237744
##        33        34        35        36        37        38        39        40
## 17.555690 20.867041 22.191582 14.244338 15.900014 19.542501 13.582068 16.231149
##        41        42        43        44        45        46        47        48
## 16.893420 17.555690 18.880230 19.211366 17.224555 19.873636 23.516123 23.681690
##        49        50
## 28.152015 22.357149
```

```r
# If we want to extract the R_squareds
cars.model.summary <- summary(cars.model)
cars.model.summary$r.squared
```

```
## [1] 0.6510794
```

```
cars.model.summary$adj.r.squared
```

```
## [1] 0.6438102
```

```
# And if you want to save any of these as separate objects to be referred to later:
cars.coefs <- coef(cars.model)
cars.resid <- cars.model$residuals
cars.fit <- cars.model$fitted.values

summary(cars.model)
```

```
##
## Call:
## lm(formula = speed ~ dist, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.5293 -2.1550  0.3615  2.4377  6.4179
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  8.28391    0.87438   9.474 1.44e-12 ***
## dist         0.16557    0.01749   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.156 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

3b: Linear models with heteroskedasticity-robust standard errors

Let's return to our tooth data again and now use it to estimate another non-robust regression model:

```
tooth.model <- lm(len ~ dose, data = tooth.data)
# Look at the model output
summary(tooth.model)
```

```
##
## Call:
## lm(formula = len ~ dose, data = tooth.data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.4496 -2.7406 -0.7452  2.8344 10.1139
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   7.4225     1.2601    5.89 2.06e-07 ***
## dose          9.7636     0.9525   10.25 1.23e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.601 on 58 degrees of freedom
## Multiple R-squared:  0.6443, Adjusted R-squared:  0.6382
## F-statistic: 105.1 on 1 and 58 DF,  p-value: 1.233e-14
```

For 'lm' model objects, we can extract the residuals and fitted/predicted values from this regression to the original dataset:

```
# The original
head(tooth.data)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

```
# Now let's create variables called residuals and predictions
tooth.data$residuals <- tooth.model$residuals
tooth.data$predictions <- tooth.model$fitted.values

# New dataset
head(tooth.data)
```

```
##    len supp dose  residuals predictions
## 1  4.2   VC  0.5 -8.1042857    12.30429
## 2 11.5   VC  0.5 -0.8042857    12.30429
## 3  7.3   VC  0.5 -5.0042857    12.30429
## 4  5.8   VC  0.5 -6.5042857    12.30429
## 5  6.4   VC  0.5 -5.9042857    12.30429
## 6 10.0   VC  0.5 -2.3042857    12.30429
```

We also could've done the same thing in the following way:

```
tooth.residuals <- tooth.model$residuals
tooth.predictions <- tooth.model$fitted.values
tooth.data <- data.frame(ToothGrowth,
                         residuals = tooth.residuals,
                         predictions = tooth.predictions)
head(tooth.data)
```

```
##    len supp dose  residuals predictions
## 1  4.2   VC  0.5 -8.1042857    12.30429
## 2 11.5   VC  0.5 -0.8042857    12.30429
## 3  7.3   VC  0.5 -5.0042857    12.30429
## 4  5.8   VC  0.5 -6.5042857    12.30429
## 5  6.4   VC  0.5 -5.9042857    12.30429
## 6 10.0   VC  0.5 -2.3042857    12.30429
```

If we want robust standard errors (which we often will), we will need another package called estimatr (so install this if you haven't already). This allows us to produce another type of model object using its "lm_robust" function:

```r
library(estimatr)
tooth.model.robust <- lm_robust(len ~ dose,
                                data = tooth.data,
                                se_type = 'HC1')
summary(tooth.model.robust)
```

```
##
## Call:
## lm_robust(formula = len ~ dose, data = tooth.data, se_type = "HC1")
##
## Standard error type:  HC1
##
## Coefficients:
##             Estimate Std. Error t value  Pr(>|t|) CI Lower CI Upper DF
## (Intercept)    7.423      1.285   5.775 3.196e-07     4.85    9.995 58
## dose           9.764      0.881  11.082 6.011e-16     8.00   11.527 58
##
## Multiple R-squared:  0.6443 ,    Adjusted R-squared:  0.6382
## F-statistic: 122.8 on 1 and 58 DF,  p-value: 6.011e-16
```

Notice the coefficient estimates are identical to those in the lm model, but the standard errors are different reflecting that we are using the more conservative heteroskedasticity-robust standard errors. The "SE type" argument here makes sure we use 'HC1' robust standard errors, the same as Stata's default.

lm_robust models are similar to lm models but not exactly identical. For example, it is easier to extract some statistics from an lm_robust model than lm models:

```r
# You do not need to use the summary() function to extract the R2s
tooth.model.robust$r.squared
```

```
## [1] 0.6443133
```

```r
tooth.model.robust$adj.r.squared
```

```
## [1] 0.6381807
```

But unlike lm models, we cannot extract residuals from an lm_robust model as easily. You'll have to produce them yourself:

```r
residuals.robust <- tooth.data$len-tooth.model.robust$fitted.values
```

That being said, the residuals from a model with robust standard errors is going to be identical to those from a non-robust model since standard errors don't affect residuals if the estimates are the same. You could simply use the same residuals as those from the lm model:

```r
# Residuals are just the difference between true and predicted values
all.equal(residuals.robust, tooth.model$residuals)
```

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

3c: Typesetting regression equations

We might also find it convenient to translate a regression model into an equation that we can use for typesetting (in fact, PS1 asks for this a couple of times). For lm models, you can do this through a package called 'equatiomatic'. To install this particular package, you'll want to run the following two commands:

install.packages('remotes') remotes::install_github('datalorax/equatiomatic')

Then load it as usual. Here's how it works:

```r
library(equatiomatic)

# Basic equation
extract_eq(cars.model)
```

$$\operatorname{speed} = \alpha + \beta_{1}(\operatorname{dist}) + \epsilon \tag{1}$$

```r
# Equation with coefficient estimates
extract_eq(cars.model,
           use_coefs = TRUE)
```

$$\widehat{\operatorname{speed}} = 8.28 + 0.17(\operatorname{dist}) \tag{2}$$

```r
# Equation with coefficient estimates and standar errors beneath them
extract_eq(cars.model,
           use_coefs = TRUE,
           se_subscripts = TRUE)
```

$$\widehat{\operatorname{speed}} = \underset{(0.874)}{8.28} + \underset{(0.017)}{0.17}(\operatorname{dist}) \tag{3}$$

Then you can simply copy and paste these outputs in between dollar signs like with our equation typesetting. The above outputs respectively produce:

$$\operatorname{speed} = \alpha + \beta_{1}(\operatorname{dist}) + \epsilon$$

$$\widehat{\operatorname{speed}} = 8.28 + 0.17(\operatorname{dist})$$

$$\widehat{\operatorname{speed}} = \underset{(0.874)}{8.28} + \underset{(0.017)}{0.17}(\operatorname{dist})$$

Note however that this works on lm models but not lm_robust models. For lm_robust models, you can simply run the equivalent non-robust lm model, copy the equatiomatic output, and just change the standard errors to the robust standard errors.

Part 4: Hypothesis testing

Let's return to the first linear model:

```
summary(tooth.model)
```

```
##
## Call:
## lm(formula = len ~ dose, data = tooth.data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.4496 -2.7406 -0.7452  2.8344 10.1139
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   7.4225     1.2601    5.89 2.06e-07 ***
## dose          9.7636     0.9525   10.25 1.23e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.601 on 58 degrees of freedom
## Multiple R-squared:  0.6443, Adjusted R-squared:  0.6382
## F-statistic: 105.1 on 1 and 58 DF,  p-value: 1.233e-14
```

We can also summarize it through

```
coeftest(tooth.model)
```

```
##
## t test of coefficients:
##
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  7.42250    1.26008  5.8905 2.064e-07 ***
## dose         9.76357    0.95253 10.2501 1.233e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Let's create a 95% confidence interval for our coefficient estimates using the *confint* function:

```
confint(tooth.model)
```

```
##                2.5 %    97.5 %
## (Intercept) 4.900171  9.944829
## dose        7.856870 11.670273
```

```
# The function works for robust regression models too:
confint(tooth.model.robust)
```

```
##                2.5 %    97.5 %
## (Intercept) 4.849592  9.995408
## dose        7.999997 11.527146
```

We could also do this for different confidence levels. For example, a 99% confidence interval:

```
confint(tooth.model.robust, level = 0.99)
```

```
##                0.5 %   99.5 %
## (Intercept) 3.999243 10.84576
## dose        7.417134 12.11001
```

Final notes

Preamble chunks

For organizational purposes, we'll usually want to begin all our Notebooks with a chunk dedicated solely to loading all the packages used in the Notebook instead of loading them one at a time and only when we first need them. We call this opening chunk the preamble For this notebook, it would have looked like this:

```
library(ggplot2) # No need for quotation marks this time
library(readstata13)
library(tidyverse)
library(lmtest)
library(estimatr)
library(equatiomatic)
```

You'll notice two things in the chunk output: first, some packages "mask" one another's functions. This means you've loaded two or more packages which each have a function with the same name and so they conflict. By default, R assigns the conflicting name to the function from the package that was loaded last. Something to keep in mind; masked functions are one of the most frustrating sources of coding problems for both new and experienced R users.

If the preamble fails to run, it usually means that you have not installed one of the packages and the error will usually tell you which ones they are.

Debugging

The ordering of chunks is important. When we render the Notebook, it opens a blank environment with no loaded libraries or defined objects then runs the chunks in sequence. This means it can only use packages, objects, or data that have been loaded, assigned, or defined in a prior chunk. The most common knitting error is referring to an object in your Notebook before you've even defined it. For example, if you've defined a data.frame called 'data' with a variable called 'variable' and at some point you run a command like 'mean(data$variable)', you have to make sure the object 'data' is defined earlier than the command mean(data) is run or else R won't know what 'data' is and knitting will result in an error. They can be in different chunks, it's just the order of appearance that matters. Similarly, if you defined an object in the console but you didn't do so in your Notebook, R won't know what that object is when knitting and it will also flag an error. So a command may work in the console or in your interface but still fail to run when knitting.

In general, there is a learning curve when using a new software for the first time. 90% of coding is getting errors and Googling how to fix them. If you run into any coding difficulties, especially in these first weeks, don't hesitate to post a question on Ed Discussion for your classmates or me to help you out. Online resources like StackOverflow and ChatGPT are your friends here.