R: An Introduction

Introduction to R

Background

Since our time in class will be brief and will involve lots of material, you'll need to access R outside of class, either by visiting a computer lab or on your machine. Provided you have the space on your computer, I strongly recommend you download R and RStudio (you need to download both). You can download R and RStudio by visiting <https://www.r-project.org/> and <https://www.rstudio.com/products/rstudio/download/>.

You can also use R on the internet, although I don't recommend this option. The best online site is RStudio Cloud. As its name suggests, it's RStudio (the application you'll use to run the programming language R), but it's located on the cloud. Aside from already being familiar with the RStudio interface, RStudio Cloud is also valuable because you can save your scripts. The reason I don't recommend using RStudio Cloud or other web-based sites is because they are unstable and prone to crashing, leaving you prone to losing all your work. But, you can access the site here: <https://rstudio.cloud/>.

You'll create and save R scripts while working in and outside of class. In your scripts you'll type and run the code from these notes. Any code executed below, whether it be a simple command that returns the mean of some numbers or generates a plot, needs to be included in your scripts. Make sure to create a folder to save your scripts. These scripts reflect what you've learned over the semester and will be invaluable when you work on your projects near the end of the semester.

Using Comments in R

When you run R code, R interprets everything you command it to run. You'll frequently find yourself needing to run code you've ran in the past, but can't remember exactly what the code means or what it'll do when executed. Using comments to remind yourself (and others reading your script) what the code means and does is, perhaps, the most helpful and underappreciated way to get the hang of R. The best advice I can give you at this early stage is the following: **Comment the hell out of every script you make**. To insert comments into your script use the `#` symbol at the beginning of your comment. For example, run the following lines of code:

```r
# Compute six plus six
6 + 6
```

```
## [1] 12
```

Notice how R simply ignored the line beginning with the `#`; that line is for your understanding and not for R's. I cannot understate the importance of inserting lots of comments in your script, especially when you're doing something that's new or complicated or counterintuitive. Trust me, you'll thank yourself for inserting lots of comments later in the semester when you're working on your projects.

Variables and Assignment

Many circumstances will require you to assign a name to objects[1] for later use. To assign an object a name use the assignment operator, denoted by the `<-` symbol. For example, run the following code:

[1]By way of example, the Consumer Price Index in the FRED database is named CPIAUCSL. Not a very descriptive name. Simplifying its name to `cpi` in R will keep you from constantly asking yourself "What is CPIAUCSL?".

```
multiply <- 5*6
```

Notice that, by assigning `5*6` a name (`multiply`), R didn't return the number 30. Here is what R understood from the above command: The name of `5*6` is `multiply`. To get R to return the result in the Console, you need to enter its name and run it, like this:

```
multiply
```

```
## [1] 30
```

Naming Objects

In a very general sense, there are 2 basic types of commands:

1. Commands that tell you something by returning something (e.g., a number or plot)
2. Commands that do something and don't return anything.

The second type of command occurs when you assign objects names using the assignment operator, `<-`, which stores that object for later use. The names you choose need to be concatenated (no spaces) strings of letters and numbers, each string starting with a letter of the alphabet. I recommend using the underscore symbol, `_`, to separate your lettered-numbered strings and only using lowercase letters (e.g., `gdp_2020_data`). It's also critical to create descriptive names to help you remember what the name refers to. Don't be afraid of creating objects with long names. The benefit of immediately recognizing what an object is outweighs by far the cost of having to type longish names. And anyways, R will try to save you time typing by proposing options once you start typing. On that note, I recommend you take R's proposals, especially when learning R, in order to misspelling an object's name and generating subsequent errors.

(Built-In) Functions

Functions manipulate objects. Like mathematical functions, R functions require arguments (inputs), and these arguments are always enclosed by brackets, `()`. One of the most basic and frequently used functions is the combine function, written as `c()`. You'll use the combine function to create vectors (an ordered sequence of elements). For example, create a vector called `x` consisting of whatever numbers you like:

```
x <- c(3, 6, -2, 0, -27)
```

What's the mean of `x`? Use the `mean()` function to find out:

```
mean(x)
```

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

Many of the functions you'll encounter have multiple arguments that then need to separated by commas. Further, functions often have default values. We can specify values by putting the name of the argument followed by a single equals sign, `=`. For example, the `head()` function shows you the first *n* rows of a dataframe, the name of the dataframe being its only non-default argument. The default number of rows is 6, but you can alter R's default value by inserting `n =`, followed by the number of rows you want to see. Compare the output from these two commands:

```
head(mtcars)

head(mtcars, n = 3)
```

Packages

You'll eventually create your own functions. Fortunately, though, most of the functions you'll need are already built into base R or else can be accessed through packages. An example of a function built into base R is the `head()` function you just used. But, there are many functions that are not built into base R, and so we'll need to install and load packages.

Packages are collections of functions, data and code that are created and made available by other R users. To load a package into R, use the `library()` function, like this: `library(name_of_package)`. If you haven't already installed the package, you first need to run `install.packages()` with the name of the package *in quotation marks* inside the function. That is, run `install.packages("name_of_package")` before loading the package. But after you've installed a package on your machine, you only have to run the `library()` function to load the package into R.

One package we'll use later on is the **car** package. To get some practice, let's install and load **car** as explained above:

```r
# Install the car package
install.packages("car")

# Load the car package into R
library(car)
```

To verify if a package has been loaded into R, click Global Environment in the Environment tab in RStudio's upper-right corner. A list of all packages you loaded into R will then drop down.

Dataframes

The term *dataframe* refers to datasets, and a dataframe is essentially a dynamic Excel spreadsheet. Dataframes consist of rows (or observations) and columns (or variables). There are some classic dataframes built into base R and so you don't need to install a package or import them in order to access them. You saw an example of this with the `mtcars` dataframe.

A typical work flow may start with us importing data and saving it as a variable, like this:

```r
# Rename mtcars as df
df <- mtcars
```

It wasn't necessary to assign the `mtcars` dataframe a new name, but it's frequently convenient to do so. The reason why, and one you'll encounter multiple times, is that when you download data from websites (e.g., the Federal Reserve's FRED database), the file names are nondescript, clumsy alphanumeric strings.[2]

The first thing you should do after importing a new dataset is to check whether the data are in proper order and in the proper format. To view the data use the `View()` function:

```r
# View data frame df
View(df)
```

The `View()` function gives you access to the entire dataframe; it's like looking at an Excel spreadsheet, but it's only for viewing purposes and not for manipulating data.

In addition to viewing your data to verify it looks like it should, you also need to verify the data are in the proper format. That is, you need to check to see if R interpreted numbers as numbers, dates of the year as dates of the year, and so on. To verify the data types of your data, use the `glimpse()` function from the **tidyverse** package as follows:

[2]By way of example, the Consumer Price Index in the FRED database is named CPIAUCSL. Not a very descriptive name. Simplifying its name to `cpi` in R will keep you from constantly asking yourself "What is CPIAUCSL?".

```r
# Verify data types of dataframe df
glimpse(df)
```

We'll have much more to say about data types and dataframes later on.

Identifying Columns in a Dataframe

You frequently need access to a single column of a dataframe. For example, you may need to use the `mean()` function to get the mean of a column of numbers. Working with a particular column requires that you *extract* it from its dataframe. To extract a column use the `$` symbol as in the following example:

```r
# Take the mean mpg
mean(df$mpg)
```

```
## [1] 20.09062
```

Notice what's happening: We want to find the mean of the `mpg` column, but we need to first specify which dataframe (here `df`) the `mpg` column belongs to. So when we want to work with a specific column from a dataframe:

1. Specify the dataframe that column belongs to.
2. Use the `$` symbol to indicate you want extract a column from this dataframe.
3. Specify the column you want from the dataframe.

Thus, the argument inside the last command, i.e., `df$mpg`, means: *extract the column* `mpg` *from the dataframe* `df`.

Subsetting Dataframes

Often you'll need to work with a subset of a dataframe. R offers several different ways of subsetting dataframes. In this example, we will use base R, so you don't have to load any packages.

The basic subsetting problem we'll consider below deals with how to extract some data from a particular column based on a specific characteristic of interest. We might want to know, for example, the miles per gallon of cars weighing over 3,000 pounds using the `mtcars` dataframe. Here, the column is miles per gallon (`mpg`) and the characteristic of interest concerns the weight of cars (`wt`).

In the code below we'll use square brackets, `[]`, to subset columns. Outside the square brackets put the column of interest using the `dataframe_name$column_name` syntax. And inside `[]` put the characteristic of interest. You can think of the code inside `[]` as a general instruction to R that says "only if these criteria are met." Here are some examples with the `mtcars` dataset, which we renamed above `df`:

```r
# Return miles per gallon (mpg) of cars with weight (wt) greater than 3
df$mpg[df$wt > 3]

# Calculate mean miles per gallon of 4 cylinder (cyl) cars
mean(df$mpg[df$cyl == 4])

# State multiple conditions using 'and' (&) and 'or' (|)
df$mpg[df$wt <= 2.5 & df$hp > 100]
df$mpg[df$wt <= 2.5 | df$hp > 100]
```

In calculating the mean miles per gallon of 4 cylinder cars, I wrote `df$cyl == 4` with a double equals sign, `==`, instead of the conventional single `=`. In R a single `=` is used inside functions when assigning values to arguments, as in the earlier example, `head(mtcars, n = 3)`. In the present case, we're subsetting data and using `==` to mean "is exactly equal to" as a testing criterion.

Graphing with `ggplot()`

The standard for graphing and for data visualization in R is a function called `ggplot()`. It comes with the **tidyverse** package. The **tidyverse** is a package we'll use every class (so load it into R at the beginning of every class). There are multiple ways to generate the same plot using `ggplot()`, and you'll eventually need to become more flexible with `ggplot()` commands as your plots become more sophisticated. But, to begin we'll use the following same basic syntax:

```r
# Generic syntax for ggplot()
ggplot(data = name_of_dataframe, mapping = aes(x = column_name, y = column_name)) +
  geom_???()
```

Steps to generate plots with `ggplot()`:

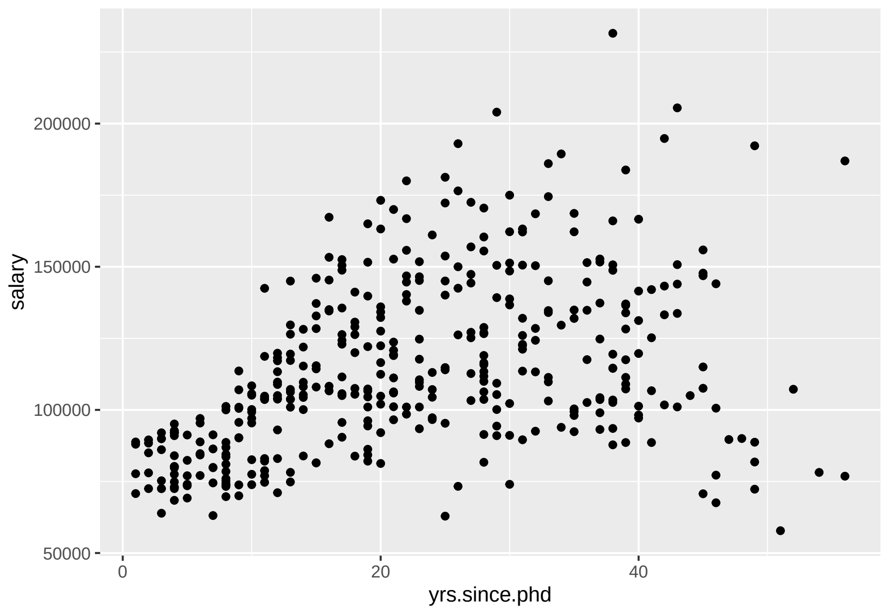
1. Begin by calling the `ggplot()` function.
2. The name of your dataframe is the first argument of the `ggplot()` function. That's the `data = name_of_dataframe` part.
3. After specifying your dataframe you need to decide which variable(s) you want to appear in the plot. That's what the `aes()` function is for. The *aes* in `aes()` is short for "aesthetics," meaning in this context the visual *appearance* of the plot. Accordingly, you'll indicate which variable you want displayed on the x-axis and, if necessary, which you want on the y-axis inside the `aes()` function.
4. The final step for a basic plot is to determine its geometry. Do you want the graph to be a scatterplot? A histogram? A bar chart? That's what the `geom_???()` function is for. The `???` is just a placeholder for the type of geometry you decide on (e.g., `geom_point()` or `geom_histogram()` or `geom_bar()`.) (Notice also that I put `geom_???()` on a new line. If you move to a new line you must end the previous line with a `+`. This tells R there's more code coming on the next line. If, instead, you put `+ geom_???()` on the next line, then R will either return a message indicating errors were made or else generate a plot you didn't intend to create.)

Let's create plots using the data in the **car** package that you've already installed and loaded. Also, install and load **tidyverse** if you haven't already.

```r
# Load the tidyverse package
library(tidyverse)
```

Now, we'll use the `Salaries` dataset in the **car** package, which you should have already loaded into R. To see what kind of data we're dealing with use `View()`. Let's create a scatterplot of the years since professors received their PhD versus their salaries:

```r
# Create a scatterplot
ggplot(data = Salaries, mapping = aes(x = yrs.since.phd, y = salary)) +
  geom_point()
```



Notice that, when specifying the x and y variables, we didn't use the `dataframe_name$column_name` syntax. That's because R already knows which dataframe we want to use, `Salaries`, and so R then knows which columns make up the dataframe.

From personal experience I realize that `ggplot()` has a steeper learning curve than other functions in R. But there are 2 decisive reasons for using it: (1) It's the most powerful graphing function in R and (2) the graphs are easy to customize.

Let's consider some of the customization options of `ggplot()`. First, let's add a regression line to represent how a professor's salary tends to change the greater the number of years has elapsed since receiving their PhD. To do so, we add the `geom_smooth()` function to the previous code with two arguments, as shown below:

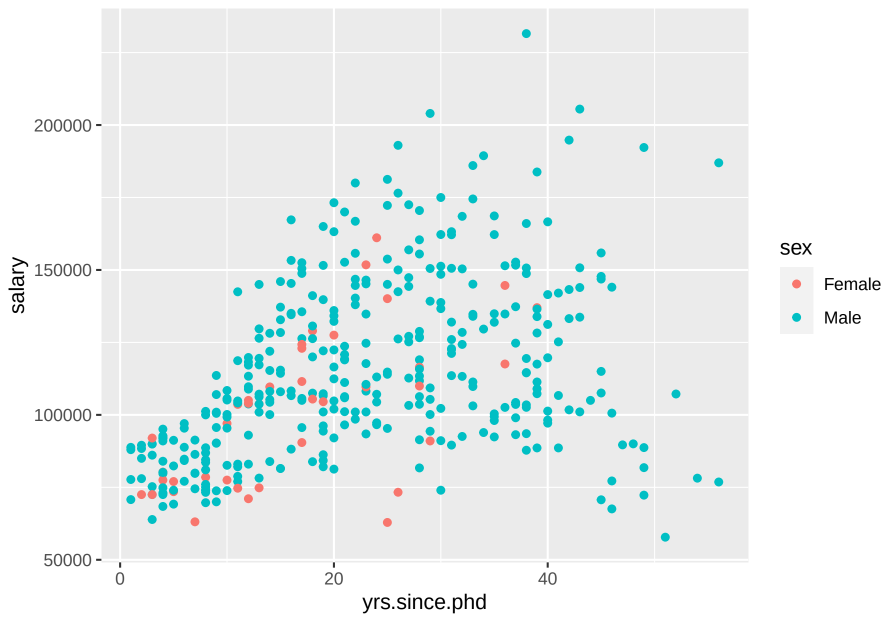
```
# Add a regression line to the scatterplot
ggplot(data = Salaries,
       mapping = aes(x = yrs.since.phd, y = salary)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

Note the 2 arguments inside `geom_smooth()`. I put `"lm"` as its `method` in order to produce a regression *line*. (As you'll soon learn, `lm` stands for linear model in R.) You must always specify the `method` of the smoothing curve you want to fit the data. I added a second argument, `se = FALSE`, because I didn't want confidence intervals around the line. Confidence intervals are the default so if you don't include `se = FALSE` your plot will come with bands around your smoothing curve.

Now, judging from the regression line, salaries tend to be lower for professors who received their PhD more recently. This isn't surprising.

The `Salaries` dataframe also includes a column called `sex` that indicates the gender of each professor in the dataset. Let's return to a simple scatterplot without the regression line, but now let's distinguish between female and male professors. We do this by including a third argument in the `aes()` called `color`:

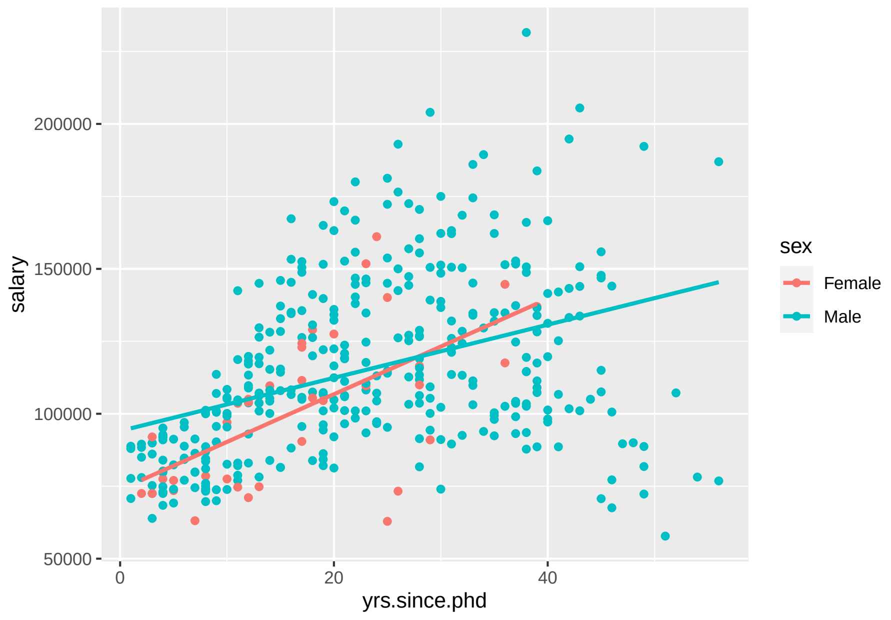
```
ggplot(data = Salaries,
       mapping = aes(x = yrs.since.phd, y = salary, color = sex)) +
  geom_point()
```



Finally, let's add a regression line as before to this new scatterplot:

```
ggplot(data = Salaries,
       mapping = aes(x = yrs.since.phd, y = salary, color = sex)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```

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



Note that by adding the third variable, `sex`, we now have two regression lines: One indicating the average change in salaries for females and a separate one for males.