Statistical Programming for the Social Sciences Using R

Wesley Stubenbord

7 March 2024

Table of contents

Preface

Welcome! This is the companion website for *Statistical Programming for the Social Sciences Using R*, taught at the Sciences Po Reims campus in the Spring 2024 term. The course is a broad introduction to the general programming skills required for data analysis in the social sciences.

This online textbook contains the relevant tutorials for each week's lesson as well as other resources that you may find helpful throughout the course. The syllabus, slides, assignment submission portals, and other files can be found on the course Moodle site.

Resources

There are a variety of `R` resources out there, many of which have been useful in developing this course.

If you would like to dig deeper into a particular topic or simply want to read other explanations of the concepts discussed in this course, here is a list of helpful resources:

- ***R for Data Science, 2e*** by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund (link): A free introductory textbook on how to use the many features of `R` to make sense of data, co-authored by the creator of the tidyverse package.
- ***Data Visualization*** by Kieran Healy (draft textbook): An introductory textbook on how to make practical and beautiful data visualizations in `R`. Healy has a distinctive and appealing visual style. He also happens to be an accomplished sociologist, known especially for his disdain of nuanced theory and his broader contributions to the study of morals and markets. His examples are especially relevant to the social sciences as a result.
- ***Introduction to Econometrics with R*** by Florian Oswald and colleagues at Sciences Po (companion textbook and slides): you may very well be taking this course because you couldn't get into this other course. For this, I am both sorry, because you are missing out, and not sorry,

because it keeps me gainfully employed and gives me an excuse to write this textbook. Fortunately for you, the course developed by Professor Oswald and colleagues is freely-available online. If you'd like to see material which is more heavily-weighted towards applied statistical methods (especially applied economics), take a look at their extremely well put together course. You may very well rue the day you woke up late for course sign-ups.

- **ggplot2: Elegant Graphics for Data Analysis (3e)** by Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen (link): An in-depth explanation of how the popular `ggplot2` data visualization package works.

Change Log

I'll be making frequent updates to the textbook, often changing the appearance, structure, and/or content. In general, these changes won't be substantive. In other words, they won't change what you are required to know or how you are expected to do things. You might, however, find that certain sections become more detailed as the course progresses. If you would like to see a detailed list of changes (including my frequent struggles with word choice), take a look at the GitHub repository. For those of you with better things to do in your spare time, a brief chronological summary of updates is provided below:

- **2024.03.07** - More minor edits to Chapter 5.
- **2024.03.06** - Minor edits to Chapter 5 for clarity, typos.
- **2024.03.05** - Homework 3 has been posted.
- **2024.03.03** - Chapter 5 has been posted.
- **2024.03.01** - Fixes to Chapter 4 code. A couple of functions were missing `scales::`.
- **2024.02.25** - Chapter 4 has been posted.
- **2024.02.22** - Another resource added to the preface. Minor copy edits.
- **2024.02.18** - I've refactored the course website from R Markdown to Quarto. You may notice some significant changes to the appearance of the website, including the appearance of a new navigation bar on the right-hand side (used for navigating sections within a chapter) and changes to the standard navigation bar on the left-hand side (no more sections and subsections). I'm not completely in love the navigation bar changes, but the other benefits of switching to Quarto are worth it (more functionality, better visual appeal in other areas). Links have been updated as a result (apologies to those of you who may have bookmarked sections). Also, a technical note has been added to Homework #2 (it will not affect the grading of relevant questions).

Chapter 1

An Introduction to R

If you are taking this course, you probably don't need an explanation of why R is useful or why it may be in your best interest to take this course. So I won't beleaguer the point: yes, R will make you fabulously rich; yes, it will help you make new friends; and yes, it will allow you to escape your own mortality. Or, at the very least, it will allow you to do some interesting things with data, which is nearly as nice.

Let's jump into it then. To get started with R, you will need to install two things:

1. R, a programming language
2. **RStudio**, a software program that helps you program in R
    - This type of software program is known as an Integrated Development Environment (IDE)

Truth be told, you don't really need RStudio to program in R, but it certainly makes life easier. The difference between programming in R and programming in R using RStudio might be akin to the difference between driving a Fiat Panda and driving a Ferrari. Both will get you to the same place, but one is more likely to be an enjoyable experience. We will be using RStudio throughout the rest of this journey as a result.

1.1 Installing R

To install R, go to https://cran.irsn.fr/index.html, select the appropriate operating system, and follow the instructions. For example, if you have a Mac, you

will click on "Download R for macOS," followed by the "R-4.3.2-arm64.pkg" link beneath the "Latest release" header. If you have a PC running Windows, you will click on "Download R for Windows" followed by "install R for the first time" then "Download R-4.3.2 for Windows."

In either case, your browser will start downloading an executable installation file which you will then need to run to install R.

> **Caution**
>
> A couple of things you may need to watch out for:
>
> - If you are using an older laptop (>10 years old), you may need to download a different version of R or RStudio. If in doubt, read the instructions on the download page and refer to your operating system version to find the right version.
> - If you have very little hard drive space on your computer, you may need to clear some space before you install RStudio. The latest RStudio version requires 215 MB and you will likely need some additional space for other software and data we will be using in the course later on. Around 2 GB should suffice.

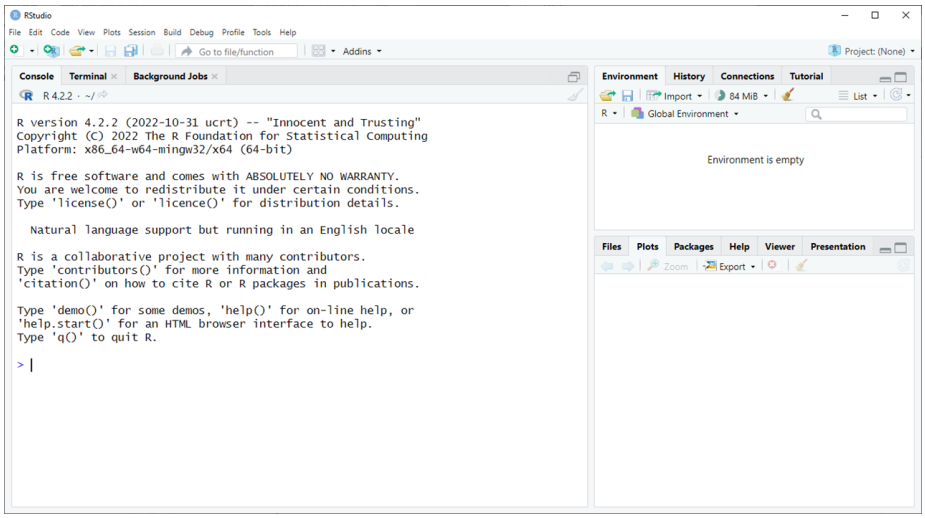
1.2 Installing RStudio

Once you've installed R, go to https://posit.co/download/rstudio-desktop/.

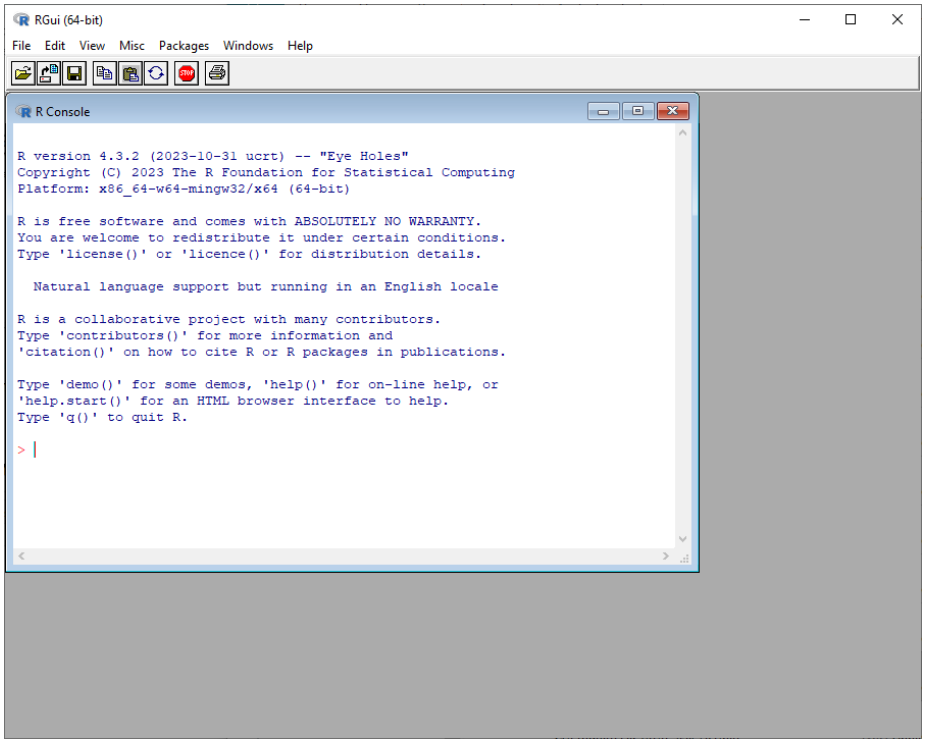
Posit (a company formerly known as RStudio) offers **RStudio Desktop** free of charge. Posit also offers a cloud-hosted version of the software (called Posit Cloud) which has both free and paid tiers. If you have trouble running RStudio Desktop on your computer, you may wish to consider using a Posit Cloud account, as described in the course syllabus.

When you've click on the link above, you'll find that you are ahead of the game. Step 1 is complete, you've already installed R. Here you'll find different versions of RStudio according to your computer's operating system. Select the operating system that corresponds to your particular case (Windows, MacOS, or Linux), download the installer, and then run the installation file from your computer. Next, follow the on-screen steps to complete the set-up.

If all goes well, your screen should look something like this once you have RStudio correctly installed and running:

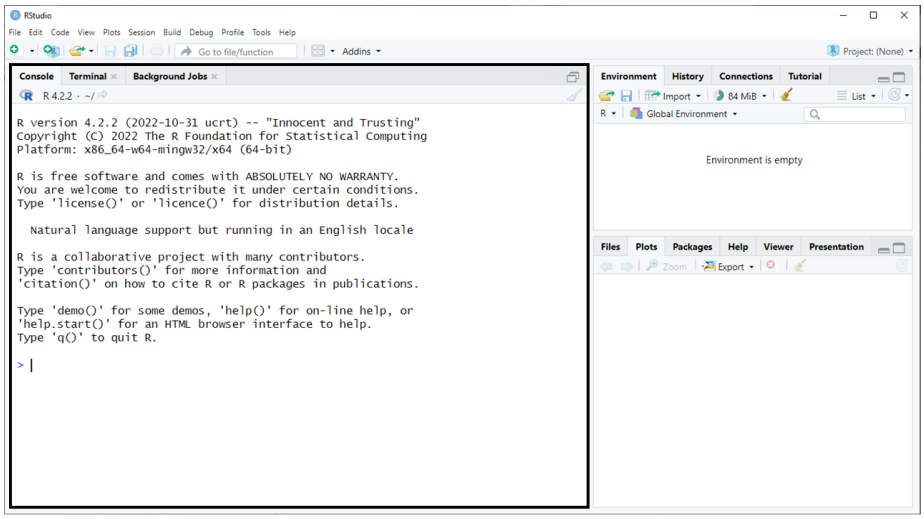


If your screen looks more like the image below, it means that you've accidentally opened RGui, a basic graphical user interface included with R, and not RStudio. We're always going to be working with RStudio for this class, so close out of RGui and open RStudio instead.



1.3 Using the Console

Now the fun begins. The RStudio window you've opened consists of a few different parts. The most important of these right now is the console pane (highlighted with a black square below).



The console allows you to interact with your computer using `R`. So, for example, if you want to use your computer as an over-sized calculator, you can type the following `R` code in the console:

```
1 + 1
```

What happens when you press `Enter` on your keyboard? You get something like this:

```
1 + 1
```

```
[1] 2
```

You've provided an **input**, `1+1`, and received an **output**, `2`. In other words, using the language of `R`, you've told your computer to add one plus one and your computer has correctly interpreted your command and *returned* (or output) an answer, two. When your computer does not know how to interpret a command, usually because you've made a mistake, you will receive an error message as the output instead. Identifying errors and being able to correct them is an essential skill for a programmer and one we will practice, often accidentally, throughout the course.

One more note about outputs: the first number in brackets next to your output, `[1]`, indicates the index number of the output.[1] This is especially helpful when you are running code that generates multiple outputs. See below, for example, where we input `LETTERS` and receive a list of letters in the English alphabet as the output (note, "T" is the 20th letter and our 20th output).

```
LETTERS
```

```
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

Try entering a few more inputs in the console:

1. `10/3`
2. `(10/3) + 1`
3. `(10/3) + 1 + (5.111)\^2`

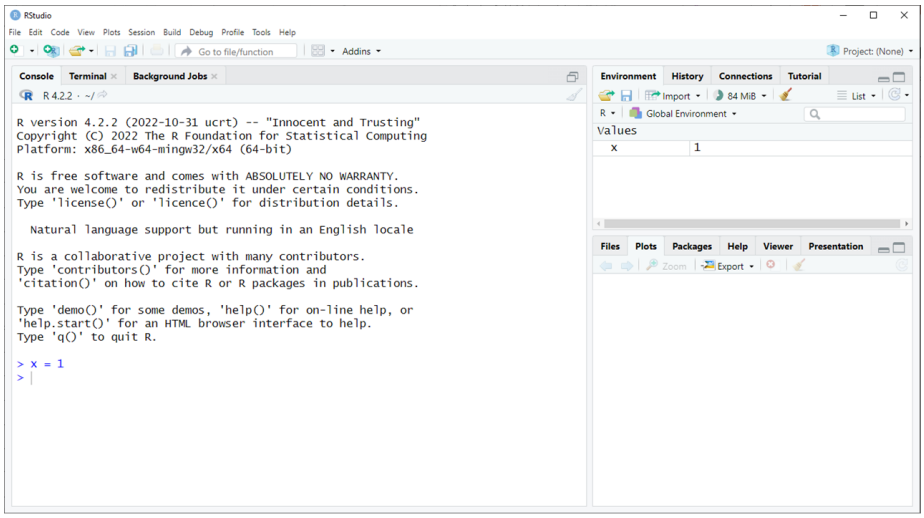
As you can see, `R` is able to handle basic math operations with ease. What about other operations? Can you work with variables in `R`, for example?

Try typing this in the console:

```
x = 1
```

What happens when you press `Enter`?

[1] In `R`, unlike in other programming languages, the first value in any data structure (e.g., a vector) has an index number of `1` rather than `0`. This makes intuitive sense. If you were asked to count people in line at a boulangerie, you would call the next person waiting to place their order the *first* person in line, not the *zero*eth. In Python, you would call them the *zero*eth person and they would have an index number of `0` instead of `1`. For more on where this comes from, see here.



Unlike before, there is no output when you press `Enter`. But, that doesn't mean nothing happened. In fact, something has happened. You've stored a value, `1`, in a variable, `x`, somewhere in your computer's memory or in what we might call the **environment**. You don't receive an output, but RStudio reminds you of your new object's existence via the environment pane in the top right.

We can recall the value we input into our variable, `x`, by entering the object name in the console:

```
x
```

```
[1] 1
```

See! Your computer remembers what you stored in the environment.

Try the following on your own in the console and then take a look at the answer:

1. Can you assign a new value to your variable, `x`?

   Answer

   Yes!

   ```
   x = 3
   x
   ```

   ```
   [1] 3
   ```

2. Can you perform math operations on a variable (e.g., `x*5`)?

Answer

Yes!

```
x * 5
```

```
[1] 15
```

3. Can you create a new variable, `y`, and use it in math operations with `x` (e.g., `x * y`)?

Answer

Yes!

```
y = 2
x * y
```

```
[1] 6
```

4. Can you change the type of variable? What if, for example, I want `x` to be equal to the word `"apple"` instead?

Answer

Yes! For letters and words, you just have to use quotation marks:

```
x = "apple"
x
```

```
[1] "apple"
```

1.4 Calculations with Objects

If you've made it this far, well done! Here's something else you can try. Enter the following in the console:

```
x <- c(1, 2, 3, 4, 5)
```

You'll notice that we're using a different operator here. It's a less than symbol, `<`, followed by a dash, `-`. This is called an **assignment operator** and it has

the same function as the equals sign, `=`. You can use either, but sticking with `<-` when assigning values to objects will make life easier later on.

What happens when you press `Enter`? You have created a **vector**. In `R`, a vector is an **object** that holds a set of values of the same type. In this case, the vector `x` contains a set of numbers: $\{1, 2, 3, 4, 5\}$. You can think of an object as a container which holds things and a "variable" as the name we use to refer to a specific object. Here, `x` is the variable name we've given to our vector of numbers, which is an object. Most things in `R` are objects.

We can do all sorts of things with vectors and other objects in `R`. We can, for example, find the sum of a vector.

```
sum(x)
```

```
[1] 15
```

How did we get an output of 15? We summed each of the elements of our vector `x`: $1 + 2 + 3 + 4 + 5 = 15$. We can also find the mean of a vector:

```
mean(x)
```

```
[1] 3
```

And, we can perform other operations on vectors too. Try each of the following questions on your own in the console and then click on the answer to check your work:

1. Can you find the median of a vector?

   Answer

   You can use `median()` instead of `mean()`!

   ```
   median(x)
   ```

   ```
   [1] 3
   ```

   Some functions are easy to guess, like `median()`, but others are false cognates just like in human languages (e.g., `mode()` won't get you what you might expect in `R` and asking for *pain* in English won't get you bread). We'll talk more about functions and how to figure out what they do in the next chapter.

2. What happens when you multiply a vector by a number?

Answer

Each value in the vector is multiplied by that number!

```
x * 2
```

```
[1]  2  4  6  8 10
```

3. Can you create a new vector which consists only of letters? What about words?

Answer

Yes! Instead of using numbers, you can create a vector using letters enclosed in quotation marks.

```
y <- c('a', 'b', 'c')
y
```

```
[1] "a" "b" "c"
```

The same works for words:

```
y <- c('cat', 'dog', 'parakeet')
y
```

```
[1] "cat"      "dog"      "parakeet"
```

1.5 Saving Your Work

As you've started to see, working with a scripting language like `R` is quite different from working with software like Microsoft Excel or Google Sheets. You work interactively with data using code rather than by changing values directly in a user interface. No more clicking on cells to change values, now you change them programmatically.

One of the great advantages of interacting with data in this way, particularly for the social sciences, is that it allows us to see all of the steps you've taken to produce your analysis and repeat them. We don't have to take your word for how you've calculated something. We can see the code and use it ourselves to produce the same thing.

This means that we leave our source data alone and write the code that produces

the analysis. As with any good recipe, we want the code you write to be clear and easy to follow so that anyone can come back to it and understand what you did. We'll say more about how to do this later on.

There are a couple of different ways to save your code:

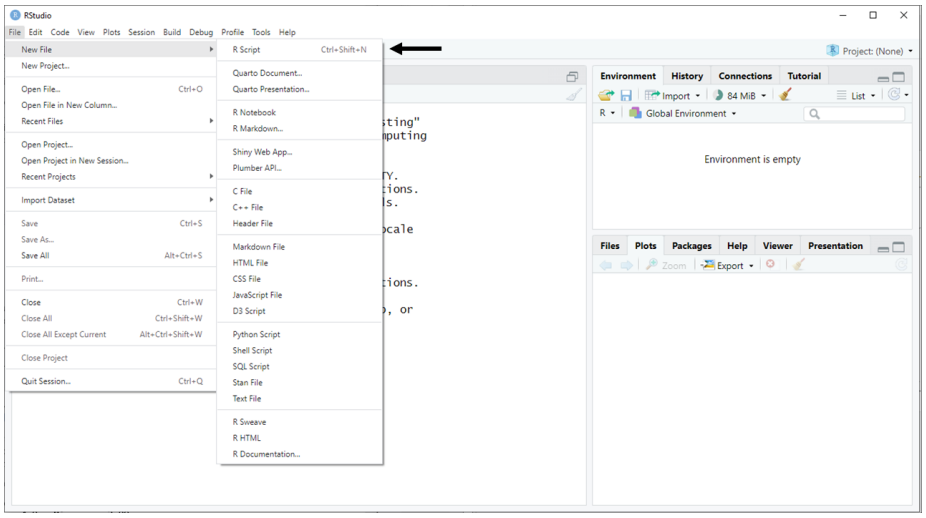
1. In an R Script, a simple text file ending in a `.r` extension

2. In an R Markdown file, an interactive format that allows you to see your code and the results together in the same file

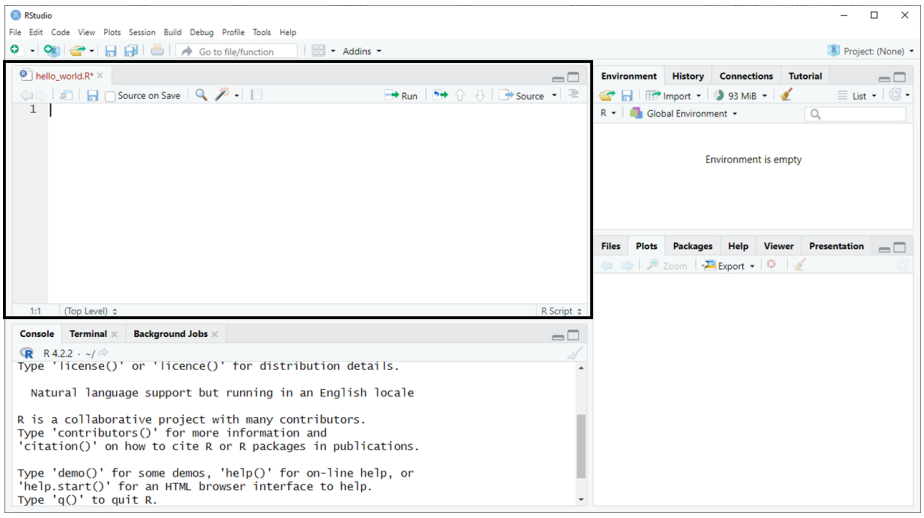
We're going to start with an R Script file and try out R Markdown later on.

1.6 Creating and Saving an R Script

To create an R Script file in RStudio, go to File > New File > R Script.

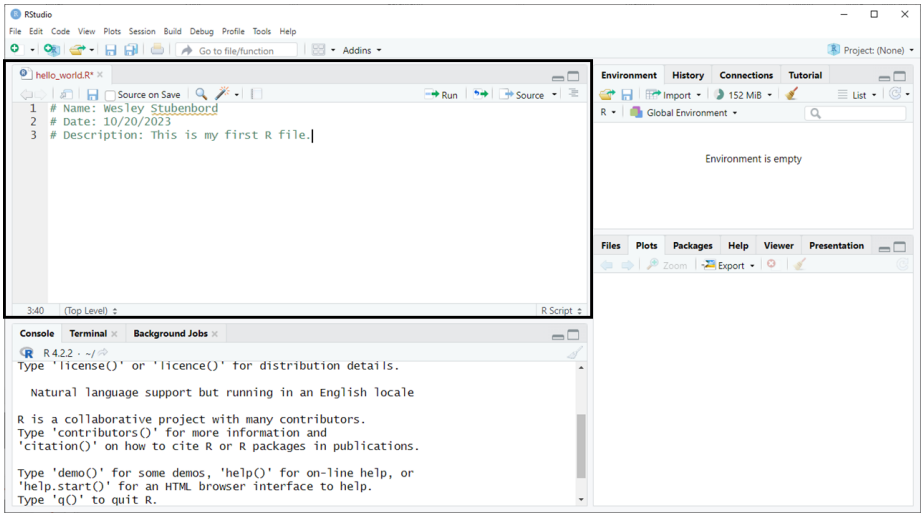


You should now have a window open in RStudio which looks like this:



You can enter comments in your R Script file using a hash tag (`#`) at the beginning of each comment line. A hash tag lets `R` know that this line should not be run as code. Its purpose is to tell us what is happening in a particular section of the code.

I like to start by adding my name, the date, and a description to each file I use. I'll ask that you use a header for each `R` file you submit for this class as well.



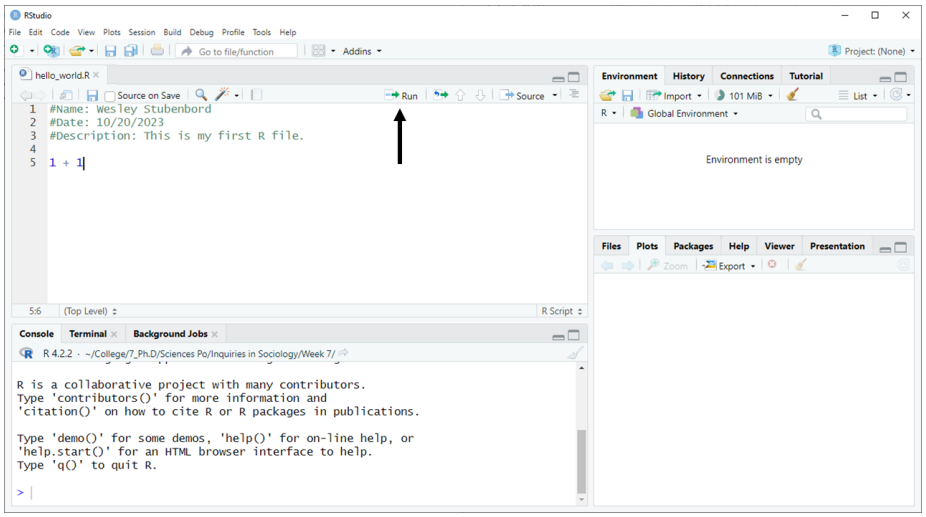
Now, save your R Script somewhere on your computer. Go to File > Save As, then choose a safe place to store it (I recommend creating a folder for this course), give your file a name, and press save. I called mine "hello_world".

1.7 Interacting in an R Script

Interacting in an R Script is slightly different from interacting with the console. Now when you type in code and hit `Enter`, it will not execute the code, it just creates a new line in your file.

To run code in a script in RStudio, you can either:

1. Select the lines you wish to run with your cursor and then press `Ctrl` + `Enter`
2. Or, put your cursor on the line you wish to run and click the `Run` button in the upper-right of the R Script pane



The first option allows you to run multiple lines at a time. The second runs only the line you are currently on. The results of your code will appear in the console pane below your R Script file when run successfully.

After you finish modifying your R Script file, you can save it and close out of RStudio. The next time you wish to access your saved code, you can open your R Script file and your code will be exactly as you left it.

1.8 Summary

Let's briefly recap what you learned this lesson. So far you've learned:

- The difference between `R` and RStudio
- How to interact with the console

- How to create and store values in objects using an assignment operator
- What a vector is and how to create one
- How to use basic functions like `sum()` and `mean()` to perform calculations
- How to make comments using the `#` symbol
- How to create and save R Script files

Chapter 2

Working with Data in R

Before we can get to the nitty-gritty of working with real data, we need to familiarize ourselves with a few more essential concepts.

2.1 Functions

Last class, we assigned a vector to a variable like this:

```
my_vector <- c(1, 2, 3, 4, 5, 6)
```

Where `my_vector` is an object and $1, 2, 3, 4, 5, 6$ is the set of values assigned to it. When you run this code in your console (or in a script file), your new variable and its assigned values are stored in short-term memory and appear in the Environment pane of RStudio.

When we assigned a single value to another variable, however, as in:

```
x <- 1
```

or,

```
first_name = 'Wesley'
```

we didn't use `c()`. So, what exactly is `c()`?

Like `sum()` or `mean()`, `c()` is a **function**. Functions play an important role in all programming languages. They are snippets of code, often hidden in the background, that allow us to accomplish specific tasks, like adding up all of the numbers in a vector, taking the mean, or creating a vector. In `R`, `c()` is a function which ***c***ombines values into a vector or list.

Functions give us the ability to recall previously written code to perform the same task over again. Why re-write code every time you need to use it, after all, when you could use the same code you used last time? Instead of copying and pasting code, we can put it in a function, save it somewhere, and call it when we need it.

2.1.1 Calling a Function

When we want to use a function, or 'call' it as we will sometimes say, we type in the name of the function, enclose **arguments** in a set of parentheses, and run the command. The general form looks something like this:

```
function([arg1], [arg2], ...)
```

2.1.2 Using Arguments in a Function

In some cases, you may just have one argument for a function, as when you want to use the `sum()` function to add the elements of a vector:

```
sum(my_vector)
```

```
[1] 21
```

In other cases, you may have multiple arguments:

```
sum(my_vector, my_vector)
```

```
[1] 42
```

Arguments can be required or optional and the number of arguments and the order in which they are input depends on the specific function you are using and what you are trying to accomplish. The `sum()` function, for instance, returns the sum of all values given as arguments.

Arguments can also be used to specify options for a function. Take a look at the example below:

```
sum(my_vector, NA, my_vector)
```

```
[1] NA
```

Here we are using the `sum()` function to add `my_vector` twice, as in the previous example, but now with a missing value (`NA`). Because the sum of two vectors plus a missing value is unknown, we get an unknown value (`NA`) as the output.

If we want the `sum()` function to ignore the unknown value, we have to provide it with an additional, named argument which tells it to ignore `NA`. We can specify this by adding `, na.rm = TRUE` to our function call. See what happens below:

```
sum(my_vector, NA, my_vector, na.rm = TRUE)
```

```
[1] 42
```

We're back to an answer of 42. The `sum()` function ignored the missing value, as we specified, and added the two vectors.

All functions have named arguments and an ordering to them. If you omit the name of an argument in your function call, the function processes them according to their default ordering. It is generally a good habit to specify argument names, as in the example below where the 'x' argument in the `sum()` function takes the object you are trying to sum, but it is not entirely necessary for simple functions.

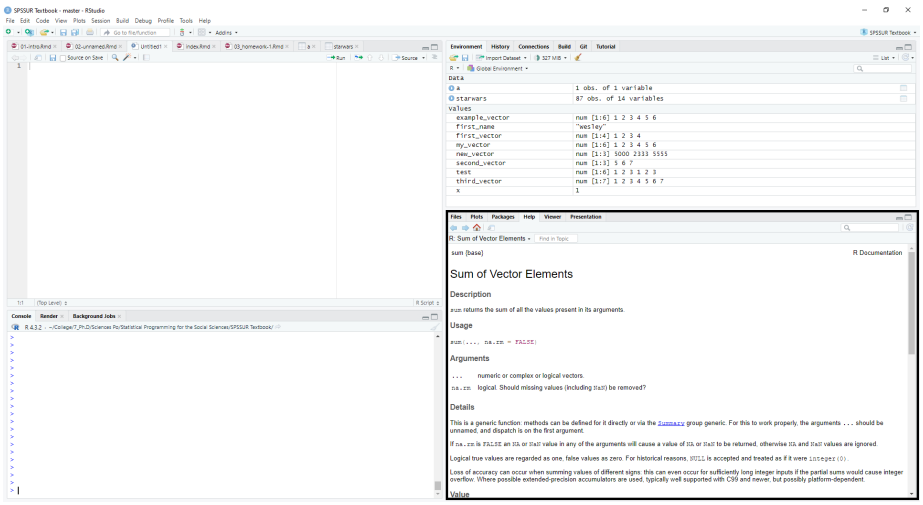
```
sum(x = my_vector)
```

```
[1] 21
```

2.1.3 Getting Help with Functions

As you progress in `R`, you will learn many different functions and it can be difficult to keep track of all of the different arguments. Whenever you want to know more about what a function does or what arguments it takes, simply type `?function_name` into the RStudio console and you will get some useful documentation in the Help pane located in the lower-right of your RStudio window.

```
?sum
```



Check Your Understanding:

Let's take a quick pause to make sure we understand what we've just learned.

1. Create a vector of three numbers and assign it to a variable called `first_vector`. Now use the `mean()` function to find the average of `first_vector`.
2. Now create another vector called `second_vector` which contains the `first_vector` and an `NA` value. Try it on your own first, then click on this footnote to see the answer.[1]
3. Using the `na.rm = TRUE` argument, calculate the mean of `second_vector`.

2.2 Packages

One of the great benefits of `R` is its power and flexibility. We've seen how functions provide us with the ability to reuse code, but functions are common to any programming language or statistical software.

It may sound cliché, but what makes `R` special is its community. `R` is a free and open-source software, which means that anyone can use or contribute to it. If you develop a new statistical method, for instance, you can write the code necessary to implement it and share it with others.

Base `R`, which you installed last class, comes with a number of built-in functions like `mean()`, `sum()`, `range()`, and `var()` . But, `R` users working out of the goodness of their hearts have developed many other functions that accomplish

[1] `second_vector <- c(first_vector, NA)`

an array of tasks, from making aesthetically-pleasing visualizations to executing complex machine learning algorithms.

These functions are put together into what are called **packages**, which can be easily installed and loaded into R. Packages can also contain data and other compiled code.

2.2.1 Installing Packages

We're going to use the `install.packages()` function to install one such package, called **tidyverse**.

```
install.packages('tidyverse')
```

Once you've run this command in your RStudio console, you will have downloaded the tidyverse and saved it to your **library**. The library is simply where your packages are stored.

Tidyverse is actually a set of packages, including `dplyr` and `ggplot2`, all of which are useful for data analysis in R. We'll be using the tidyverse throughout this course and you will find that it's the most commonly used set of packages for data analysis in R.

2.2.2 Loading Libraries

Whenever you start an R session and want to use a package, you have to be sure to load it. Loading a package makes sure that your computer knows what functions and data are inside, so that you can call them at will.

To load an R package, you can use the `library()` function, like this:

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.0     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become er
```

Now, that you've loaded tidyverse, you can access it's special functions like `mutate()` or its data sets, like `starwars`. Try entering `starwars` in your console after you've loaded the tidyverse. What's inside?

2.3 Loading Data

Great, you know what a function is, you have the tidyverse installed, and you've seen that data can be contained in packages, which are easy to install and load.

2.3.1 Using Data from Packages

Let's install and load another package, so that we can take a look at some more data.

```
install.packages('socviz')
```

```
library(socviz)
```

The `socviz` package accompanies a textbook called ***Data Visualization*** written by Kieran Healy, a Professor of Sociology at Duke University, and it contains some interesting datasets including election data from the 2016 U.S. presidential election. This dataset is stored in an object titled `election`. Once you have `socviz` installed and loaded, you can get a preview of its contents by entering the name of the object:

```
election
```

```
# A tibble: 51 x 22
   state     st     fips total_vote vote_margin winner party pct_margin r_points
   <chr>     <chr> <dbl>      <dbl>       <dbl> <chr>  <chr>      <dbl>    <dbl>
 1 Alabama   AL        1    2123372      588708 Trump  Repu~     0.277     27.7
 2 Alaska    AK        2     318608       46933 Trump  Repu~     0.147     14.7
 3 Arizona   AZ        4    2604657       91234 Trump  Repu~     0.035      3.5
 4 Arkansas  AR        5    1130635      304378 Trump  Repu~     0.269     26.9
 5 Californ~ CA        6   14237893     4269978 Clint~ Demo~     0.300    -30.0
 6 Colorado  CO        8    2780247      136386 Clint~ Demo~     0.0491    -4.91
 7 Connecti~ CT        9    1644920      224357 Clint~ Demo~     0.136    -13.6
 8 Delaware  DE       10     443814       50476 Clint~ Demo~     0.114    -11.4
 9 District~ DC       11     311268      270107 Clint~ Demo~     0.868    -86.8
10 Florida   FL       12    9502747      112911 Trump  Repu~     0.0119     1.19
# i 41 more rows
```

```
# i 13 more variables: d_points <dbl>, pct_clinton <dbl>, pct_trump <dbl>,
#   pct_johnson <dbl>, pct_other <dbl>, clinton_vote <dbl>, trump_vote <dbl>,
#   johnson_vote <dbl>, other_vote <dbl>, ev_dem <dbl>, ev_rep <dbl>,
#   ev_oth <dbl>, census <chr>
```

For ease of use, we're going to store a copy of this data in a new object in our environment called `election_2016`.

```
election_2016 <- election
```

Now, we can play around with it. In addition to getting a preview of the data by entering the name of our object in the console, we can also access it through the Environment pane of our RStudio window. Click on `election_2016` and you will see the full dataset.

Just like in a spreadsheet, you can scroll through the full set of columns and rows. Remember, of course, that you cannot edit values in this view tab. This is by design. If we want to make changes to the data or perform calculations, we need to do so *programmatically* by using code.

2.4 Data Types and Data Structures

This seems about as good a point as any to talk about the different types of data you will encounter in `R`.

2.4.1 Data Types

There are six different basic data types in `R`. The most important for our purposes are:

- **character**: letters such as `'a'` or sets of letters such as `'apple'`
- **numeric**: numbers such as `1`, `1.1` or `23`
- **logical**: the boolean values, `TRUE` and `FALSE`

The other types of data are integers (which can only hold integers and take the form `1L`), complex (as in complex numbers with an imaginary component, `1+2i`), and raw (data in the form of bytes). You have already used the previous three and we won't use the latter three in this course.

If you wish to check the data type, or class, of an object, you can use the `class()` function.

```
class(my_vector)
```

```
[1] "numeric"
```

2.4.2 Data Structures

There are many different data structures in R. You've already become familiar with one, vectors, a set of values of the same type. Other data structures include:

- **list**: a set of values of different types
- **factor:** an ordered set of values, often used to define categories in categorical variables
- **data frame**: a two-dimensional table consisting of rows and columns similar to a spreadsheet
- **tibble:** a special version of a data frame from the *tidyverse*, intended to keep your data nice and tidy

Note that data structures are usually subsettable, which means that you can access elements of them. Observe:

```
my_list <- c('a', 'b', 'c', 2)
my_list[2]
```

```
[1] "b"
```

In the example above, we've called an element of the list, `my_list`, using an index number in a set of brackets. Since we entered the number `2` inside brackets next to our list name, we received the second element of the list, the character `'b'`. We can also modify elements of a list in the same way.

Let's now say that we want to change `'b'`, the second element of `my_list`, to the word `'blueberry'`:

```
my_list[2] <- 'blueberry'
my_list
```

```
[1] "a"         "blueberry" "c"         "2"
```

Easy enough. Now try it out yourself:

1. Create a vector with three elements: "sociology", "economics", and "psychology"
2. Call each of them individually.
3. Change the value of the second element to the value of the first element.
4. Change the value of the third element to the value of the first element.

Be sure to do the last two programmatically rather than by re-typing the initial values.

2.5 Using Functions with Data

Back to the elections data. We have our 2016 U.S. Presidential Election data stored in a **tibble** called `election_2016`.

If we want to output a single column from the data, like `state`, we can do so by typing in the name of the data object (in this case, `election_2016`) followed by the `$` symbol, and the name of the column (`state`).

```
election_2016$state
```

```
 [1] "Alabama"              "Alaska"               "Arizona"
 [4] "Arkansas"             "California"           "Colorado"
 [7] "Connecticut"          "Delaware"             "District of Columbia"
[10] "Florida"              "Georgia"              "Hawaii"
[13] "Idaho"                "Illinois"             "Indiana"
[16] "Iowa"                 "Kansas"               "Kentucky"
[19] "Louisiana"            "Maine"                "Maryland"
[22] "Massachusetts"        "Michigan"             "Minnesota"
[25] "Mississippi"          "Missouri"             "Montana"
[28] "Nebraska"             "Nevada"               "New Hampshire"
[31] "New Jersey"           "New Mexico"           "New York"
[34] "North Carolina"       "North Dakota"         "Ohio"
[37] "Oklahoma"             "Oregon"               "Pennsylvania"
[40] "Rhode Island"         "South Carolina"       "South Dakota"
[43] "Tennessee"            "Texas"                "Utah"
[46] "Vermont"              "Virginia"             "Washington"
[49] "West Virginia"        "Wisconsin"            "Wyoming"
```

The `$` is known as a **subset operator** and allows us to access a single column from a table. If we want to perform a calculation on a column, we can use the column as an argument in a function like so:

```
# Sum the total number of votes cast in the 2016 Presidential election.
sum(election_2016$total_vote, na.rm = TRUE)
```

```
[1] 137125484
```

Here, we summed all of the values in the `total_vote` column in the `election_2016` tibble. The `na.rm` argument isn't strictly necessary in this case (since there are no missing or unknown values), but it's good to remember that it's an option in case you need it.

2.6 Exercise

For the remainder of today's session, I'd like you to play around with the election data. In particular:

1. Identify the data type for the `ST`, `pct_johnson`, and `winner` columns.
2. Calculate the mean `vote_margin` across the states.
3. Use the `table()` function to count the number of states won by each presidential candidate.
4. Create a variable which contains the total number of votes received by Hillary Clinton (contained in the column `clinton_vote`) and a variable containing the total number of votes received by Donald Trump (`trump_vote`). Take the difference of the two.
5. Create a variable containing the total number of electoral votes received by Hillary Clinton (contained in `ev_dem`) and another containing the total number received by Donald Trump (`ev_rep`). Take the difference of the two.
6. Try using the `plot(x=, y=)` function to plot a couple of numeric columns.

Chapter 3

Summarizing Data with `dplyr`

In the previous chapter, you learned how to load data from a package, how to access a column from a tibble using the subset operator `$`, and how to use basic functions to answer questions like: what was the total number of votes cast in the 2016 U.S. presidential election?

We've had a couple strokes of luck so far. Our data has been nice and tidy and our questions haven't really required us to poke around in order to find the answers we are interested in. This brings us to *data wrangling* - the art and science of manipulating, distilling, or cajoling data into a format that allows you to find the answers you are seeking.

For this lesson, we are going to continue to maintain the illusion of neat and tidy data and focus on learning the tools necessary to dig deeper into a data set: in particular, `dplyr` and the **pipe operator**. In future lessons, our luck will run out and we will be confronted with the harsh reality of unseemly data.[1]

3.1 Basic Description with Base `R`

Let's use an example to get us started. Last class, you toyed around with the 2016 U.S. presidential election data from the `socviz` package, a helpful collection of data sets and other goodies developed by Kieran Healy.[2]

[1] Sadly, almost all data you encounter out in the wild will be very unseemly for one reason or another. But, maybe after taking this course and ascending the ranks of government/business/academia, you too will become an evangelical for orderly data and help to make the world a tidier place.

[2] The `socviz` package serves as an accompaniment to Healy's textbook, *Data Visualization*, which is highly recommended.

We'll use another data set from the same package in a moment, but, for now, let's return to the `election` data. We're also going to re-load our new best friend, the *tidyverse* package.

```
library(socviz)
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.0     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to
```

Libraries loaded. Remember, once you have the packages installed, you don't need to do it again. So, don't include `install.packages()` in your scripts going forward.[3]

We'll load the data into an object in our environment. This time, we'll use a slightly shorter name for the tibble to spare ourselves some future misery. A longer name means more to retype later.

```
elec_2016 <- election
```

Just like last time, we can do basic calculations on columns using the subset operator and column name. Let's add a few new functions to our repertoire for good measure:

```
# table() gives a contingency table for character variables.
# Here's the number of states (plus D.C.) won by each candidate.
table(elec_2016$winner)
```

```
Clinton   Trump
     21      30
```

[3] Anytime you install packages, do it directly in the console. If someone needs to run your code, they should see the `library()` calls in the beginning of your code after the header and will know whether they need to install additional packages or not. RStudio also has a helpful auto-prompt feature that will let you know if you are missing anything.

Instead of the `library()` function, you can also use `require()`, which has the benefit loading packages if you already have them and installing them if you don't.

```
# Wrapping prop.table() around a contingency table gives relative frequencies.
# i.e., Hillary Clinton won 41.2% (21/51) of states (plus Washington D.C.).
prop.table(table(elec_2016$winner))
```

```
  Clinton     Trump
0.4117647 0.5882353
```

```
# summary() gives us a nice 5-number summary for numeric variables.
# Here we see the min, max, median, mean, and quartiles for the pop. vote margin.
summary(elec_2016$vote_margin)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2736   96382  212030  383997  522207 4269978
```

But, what if we want to do something more specific? What if, for example, we really want to know how much of the popular vote third-party Libertarian candidate Gary Johnson won across the different regions of the United States? Here we need special functions from `dplyr` and the pipe operator.

```
# An illustrative example - no need to try this yet
elec_2016 %>%
  group_by(census) %>%
  summarize(total = sum(johnson_vote))
```

```
# A tibble: 4 x 2
  census      total
  <chr>       <dbl>
1 Midwest   1203062
2 Northeast  676192
3 South     1370056
4 West      1239925
```

We'll learn how to create frequency tables like this and more in a moment.

3.2 The Pipe Operator

The **pipe operator** is a handy tool indeed. It is a specialized operator that comes from the `magrittr` package, which is contained in the tidyverse.

It looks like this: `%>%`. But, it can also look like this: `|>`.

There isn't much of a difference between the two, so you can use whichever you prefer as long as you are consistent.[4] The pipe operator has a straightforward function: it combines a series of steps into a single command and it does this in a way which keeps your code legible. Whenever you see the pipe operator, you should read it as though it is saying, "And then [do this]" (Healy 2019).

So in the previous example provided, you might read the code as saying:

```
elec_2016 %>%                              # Take the election data AND THEN
  group_by(census) %>%                     # group it by census region AND THEN
  summarize(total = sum(johnson_vote))     # sum up the vote for Johnson.
```

Note a couple of things here:

1. The pipe operator always goes at the end of each line, followed by a new line
2. The pipe operator never goes at the end of the command

The first is a convention to make the code more readable and the second is a requirement. If you leave a pipe operator at the end of your statement, `R` will search for the missing code and then give you an unfriendly error when you try to run more code. Don't leave a pipe operator hanging.

3.3 Functions from `dplyr`

`dplyr` (pronounced dee-ply-R) is a set of tools for working with tabular data. It's one of the packages in tidyverse (along with `ggplot2`, `tidyr`, `tibble`, `readr`, and a few others), so you won't have to load it separately.

[4] For more on the differences between the two pipe operators, see here: https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe

`dplyr` has a handful of special functions:

- `group_by()`, which groups data together at some desired level (e.g., states by census region)
- `filter()`, which gives us the rows corresponding to the criteria entered as an argument
- `select()`, which selects columns from the original data
- `summarize()` or `summarise()`, which performs calculations[5]
- `mutate()`, which creates new columns (or variables)
- `arrange()`, which sorts the row order by column values

3.4 Glimpsing GSS Data

Let's load another data set from `socviz` so that we can start testing out these new function. This data set is called `gss_sm` and contains a nice, clean extract from the 2016 General Social Survey.

```
# Loading the data into a new object
gss <- gss_sm
```

The General Social Survey is a nationally representative biennial survey of U.S. adults on sociological topics produced by the National Opinion Research Center (NORC) at the University of Chicago since 1972.

Take a quick look at the data. You can use `glimpse()`, another `dplyr` function, to get a sense of what's inside. You can also inspect it visually using `view()`. Typing in `?gss_sm` (the original name of the data set from the package) will tell you what variables the data contains.[6]

```
view(gss)
glimpse(gss)
```

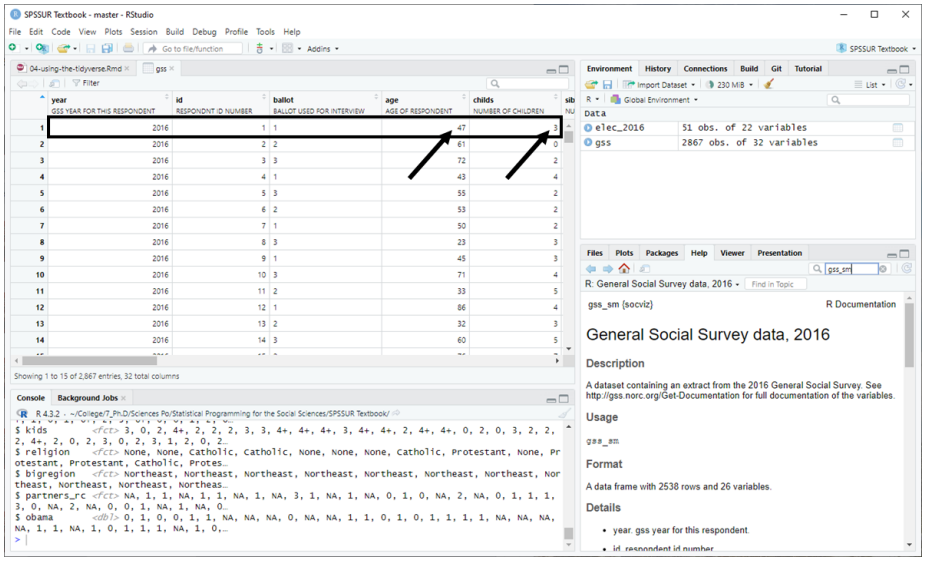
There's a wealth of data in here. As you scroll through the columns and rows, you may have also noticed that the data here is at the *individual-level*. Each row represents an individual respondent (identified by the `id` variable) and

[5] `summarize()` and `summarise()` are the same function, just two different spellings, the choice of which depends on who you've learned English from.

[6] You won't always be able to get documentation on a data set by using the help function, unfortunately. But, in this case, it works because `socviz` comes with documentation that was downloaded when you installed the package. Note that you must refer to the data in your help query by it's original name (`?gss_sm` not `?gss`).

each column consists of a variable (in this case, a coded response to a survey question).

If we click on our data in the environment pane, we can see that the first data row corresponds to respondent #1 who is 47 years old and has 3 children:



3.5 Selecting Columns

There are 32 variables in this data set. Maybe we want to narrow in and look at just a few of them, like: `id`, `sex`, and `religion`. We can use the `select()` function to do this.

```
gss %>%                           # Take the GSS data AND THEN
  select(id, sex, religion)       # take just the ID, sex, and religion columns.
```

```
# A tibble: 2,867 x 3
      id sex    religion
   <dbl> <fct>  <fct>
 1     1 Male   None
 2     2 Male   None
 3     3 Male   Catholic
 4     4 Female Catholic
 5     5 Female None
 6     6 Female None
 7     7 Male   None
 8     8 Female Catholic
```

```
 9     9 Male   Protestant
10    10 Male   None
# i 2,857 more rows
```

In the code above, we told R that we wanted to take the GSS data *and then* (using the pipe operator) only the id, sex, and religion variables. The select function outputs a new tibble containing only those three variables entered as arguments. The number of rows or observations, 2,867, is the same as in the original data.

We can now save a copy of our new three-variable tibble by assigning it to a new object. Let's call this new object gender_relig.

```
gender_relig <- gss %>%
  select(id, sex, religion)
```

Now we have a new object containing our new tibble. If after inspecting this new tibble you decide that you don't need or want it anymore, you can always get rid of it using the rm() function.[7]

```
view(gender_relig)
rm(gender_relig)
```

3.6 Grouping and Summarizing

Let's say we want a table which shows the number of respondents by religious affiliation. There are other ways of doing this, but we're going to use dplyr and the pipe operator.

To do this, we first have to tell R how we would like to group the data. Grouping doesn't visibly change the data, but it prepares R to interpret our next commands according to the groups we specify. We're going to group by the religion variable which contains the respondent's religious affiliation.

```
gss %>%
  group_by(religion)
```

```
# A tibble: 2,867 x 32
# Groups:   religion [6]
    year    id ballot       age childs sibs   degree race  sex   region income16
```

[7] Using the rm() function can help keep your environment a bit more orderly, but it isn't always necessary since your environment will be cleared out each time you close RStudio anyways.

```
   <dbl> <dbl> <labelled> <dbl>  <dbl> <labe> <fct>  <fct> <fct> <fct>  <fct>
 1  2016     1 1             47      3 2      Bache~ White Male  New E~ $170000~
 2  2016     2 2             61      0 3      High ~ White Male  New E~ $50000 ~
 3  2016     3 3             72      2 3      Bache~ White Male  New E~ $75000 ~
 4  2016     4 1             43      4 3      High ~ White Fema~ New E~ $170000~
 5  2016     5 3             55      2 2      Gradu~ White Fema~ New E~ $170000~
 6  2016     6 2             53      2 2      Junio~ White Fema~ New E~ $60000 ~
 7  2016     7 1             50      2 2      High ~ White Male  New E~ $170000~
 8  2016     8 3             23      3 6      High ~ Other Fema~ Middl~ $30000 ~
 9  2016     9 1             45      3 5      High ~ Black Male  Middl~ $60000 ~
10  2016    10 3             71      4 1      Junio~ White Male  Middl~ $60000 ~
# i 2,857 more rows
# i 21 more variables: relig <fct>, marital <fct>, padeg <fct>, madeg <fct>,
#   partyid <fct>, polviews <fct>, happy <fct>, partners <fct>, grass <fct>,
#   zodiac <fct>, pres12 <labelled>, wtssall <dbl>, income_rc <fct>,
#   agegrp <fct>, ageq <fct>, siblings <fct>, kids <fct>, religion <fct>,
#   bigregion <fct>, partners_rc <fct>, obama <dbl>
```

As you can see, our data doesn't appear to have changed in the output above. We still have 32 variables and 2,867 observations. But, there is now a helpful note at the top of our output that says, `Groups: religion[6]`. Our observations have been successfully grouped according to the six religious affiliations in our data.

Next, we have to add another line to our pipe function which specifies how we want to `summarize()` the groups. Remember, for each of these additions to our pipe function we're adding a pipe operator the end of each line, except for the last line. We want it to count up our rows here, so we'll use the `n()` function. The `n()` function counts the number of rows in a data frame.[8]

```
gss %>%
  group_by(religion) %>%      # Group by religion
  summarize(total = n())      # Create a total by counting the rows
```

```
# A tibble: 6 x 2
  religion   total
  <fct>      <int>
1 Protestant  1371
2 Catholic     649
3 Jewish        51
4 None         619
5 Other        159
6 <NA>          18
```

[8]There are other options for counting the number of rows, like the `count()` or `tally()` functions, but I won't use them here.

As you can see, we provided `summarize()` with a new column name, `total`, and a measurement, `n()`. The pipe operator between the two lines ensured that we grouped our data first and then summarized. Now, we have the total number of respondents for each group (religious affiliation).

If we want, we can save a copy of our new tibble in another object, as in the command below. The original data object in our environment (i.e., `gss`) will always remain untouched unless we intentionally re-assign it (i.e., `gss <- gss %>% ...`).

```
relig <- gss %>%
  group_by(religion) %>%
  summarize(total = n())
```

Another quick example, let's say we want to see the count of our 2016 GSS respondents by sex:

```
gss %>%
  group_by(sex) %>%
  summarize(total = n())
```

```
# A tibble: 2 x 2
  sex    total
  <fct>  <int>
1 Male    1276
2 Female  1591
```

In this code snippet, we took the GSS data *and then* grouped it by `sex` *and then* summarized it by creating a `total` which holds a count of the number of rows. Since the number of rows corresponds to the number of respondents who took the 2016 GSS, we can see that 1,276 respondents were male and 1,591 were female.

3.6.1 Grouping by Two Variables

We can also create the equivalent of what is called a two-way contingency table by grouping with two variables at the same time. To do this, we add the second variable as another argument in the `group_by()` function. We can find religious affiliation by sex like this:

```
gss %>%
  group_by(religion, sex) %>%
  summarize(total = n())
```

```
`summarise()` has grouped output by 'religion'. You can override using the
`.groups` argument.

# A tibble: 12 x 3
# Groups:   religion [6]
   religion   sex    total
   <fct>      <fct>  <int>
 1 Protestant Male     559
 2 Protestant Female   812
 3 Catholic   Male     287
 4 Catholic   Female   362
 5 Jewish     Male      22
 6 Jewish     Female    29
 7 None       Male     339
 8 None       Female   280
 9 Other      Male      58
10 Other      Female   101
11 <NA>       Male      11
12 <NA>       Female     7
```

In the output above, we can now identify the number of Protestants who are male, 559, and the number who are female, 812.

3.6.2 Ordering `group_by()` Arguments

It is worth noting that the ordering of arguments in the `group_by()` function sometimes matters (i.e., `group_by(religion, sex)` as opposed to `group_by(sex, religion)`.

Because religion came first in our argument order, our results show us the number of Protestants who are male and the number of Protestants who are female. But, we could have very easily shown the number of males who are Protestant and the number of females who are Protestant.

For a count, these are the same thing. The *number of Protestants who are male* is the same as the *number of males who are Protestant*. But when we start calculating relative frequencies and percentages using `group_by()`, the order will matter.[9] You'll get a sense for this in a moment.

3.7 Calculating with `mutate()`

Is there an equal proportion of male and female Protestants in the GSS? Let's add a relative frequency column to find out.

[9] The *percentage of Protestants who are male* is not the same as the *percentage of males who are Protestant*.

```
gss %>%
  group_by(religion, sex) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
`summarise()` has grouped output by 'religion'. You can override using the
`.groups` argument.
```

```
# A tibble: 12 x 5
# Groups:   religion [6]
   religion   sex    total  freq   pct
   <fct>      <fct>  <int> <dbl> <dbl>
 1 Protestant Male     559 0.408  40.8
 2 Protestant Female   812 0.592  59.2
 3 Catholic   Male     287 0.442  44.2
 4 Catholic   Female   362 0.558  55.8
 5 Jewish     Male      22 0.431  43.1
 6 Jewish     Female    29 0.569  56.9
 7 None       Male     339 0.548  54.8
 8 None       Female   280 0.452  45.2
 9 Other      Male      58 0.365  36.5
10 Other      Female   101 0.635  63.5
11 <NA>       Male      11 0.611  61.1
12 <NA>       Female     7 0.389  38.9
```

Notice, we used the same code as before, but we've now added another step, a `mutate()` function to create two new columns, `freq` (relative frequency) and `pct` (percentage).

We previously calculated the `total` or the number of observations for each sub-group (e.g., Protestants who are males, Protestants who are females, etc.). The `mutate()` function takes the `total` we calculated in the previous step and uses it to calculate first the relative frequency and then the percentage for each sub-group.

To calculate the relative frequency, we used `freq = total / sum(total)` or in plain English "create a new value called `freq` and then calculate this value by taking the number of observations for each sub-group (`total`) and then dividing it by the sum of the totals for all sub-groups (`sum(total)`)."

For the religious group Protestant, we have two sub-groups, male and female, and so the frequency for males Protestants is calculated as `559 / (559 + 812)`, which equals `0.408` , or exactly what you see in the first row in the frequency column in our new tibble. Similarly, the frequency for female Protestants would

be `812 / (559 + 812)` or `0.592` or what you see in the frequency column in the second row of our new tibble.

What about the percentage or `pct`? In the second argument of our `mutate()` function, we told `R` to take the `freq` we calculated in the previous step, multiply it by 100 (to make it a percentage), and then round it to the first decimal place using the `round()` function. `0.408`, the relative frequency of male Protestants, therefore becomes 40.8%.

As you can see, calculating relative frequencies and percentages using `dplyr` and the pipe function can be a bit of a beast. The good news is that the general form is always the same and so you'll be able to re-use the code often.

3.8 How R Reads Functions

In the previous examples, you may have noticed a bunch of *nested functions*, which is when a function is used as an argument inside another functions, e.g., `summarize(total = n())`. It's worth pausing for a moment to think about how `R` reads code, since you will be using these types of constructions quite often.

Functions are always read inside out, so a nested function will always evaluate the inner-most function first. Pipe operations, on the other hand, are always read from left-to-right or top-to-bottom (if you're breaking up your code using new lines, as you should be). The two commands below evaluate in the same way, but `R` reads them in a slightly different ordering.

```
# Inside-out evaluation
sum(c(1,2,3))               # A vector, {1,2,3} is created first AND THEN summed
```

```
[1] 6
```

```
# Left-to-right/top-to-bottom (sequential) evaluation
c(1,2,3) %>%                # A vector is created AND THEN
  sum()                     # it is summed
```

```
[1] 6
```

In both cases, a vector is being created first and then summed.

3.9 Filtering

Back to the data. What if we only wanted to see the Protestant results for our previous examples? We can use a `filter()` function.

```
gss %>%
  group_by(religion, sex) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1)) %>%
  filter(religion == "Protestant")
```

```
`summarise()` has grouped output by 'religion'. You can override using the
`.groups` argument.
```

```
# A tibble: 2 x 5
# Groups:   religion [1]
  religion   sex    total  freq   pct
  <fct>      <fct>  <int> <dbl> <dbl>
1 Protestant Male     559 0.408  40.8
2 Protestant Female   812 0.592  59.2
```

In a filter function, you use logical and comparison operators (see the slides from Session 3 if you'd like a refresher) to define the criteria for your new tibble. In this case, we want only the observations for which the `religion` variable is equal to "Protestant".

R is case-sensitive and so if the values in your data are "protestant", for example, you won't see those results in the tibble output here.

Usually, you will want to use the `filter()` function at the beginning of your query. Here's another example. This time, we're only interested in religious affiliation among holders of graduate degrees.

```
gss %>%
  filter(degree == 'Graduate') %>%
  group_by(religion) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
# A tibble: 6 x 4
  religion   total    freq   pct
  <fct>      <int>   <dbl> <dbl>
1 Protestant   126 0.396    39.6
2 Catholic      63 0.198    19.8
3 Jewish        15 0.0472    4.7
4 None          82 0.258    25.8
5 Other         31 0.0975    9.7
6 <NA>           1 0.00314   0.3
```

Now, we can see that 39.6% of graduate-degree holding respondents were Protestant and 25.8% had no religious affiliation. Later on, we'll learn how to turn this sort of thing into a nice graph.

```
# What happens if I use a lower-case 'g' in 'Graduate' instead?
gss %>%
  filter(degree == 'graduate') %>%
  group_by(religion) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
# A tibble: 0 x 4
# i 4 variables: religion <fct>, total <int>, freq <dbl>, pct <dbl>
```

3.10 Conditional Filtering

What if we want to filter our respondents for multiple degree types? We may want to see in our table of religious affiliation, for example, only people who have a bachelor's degree **or** a graduate degree.

For these types of queries, we can use other logical operators in our `filter()` criteria. Here, specifically, we'll use `|` which stands for '**or**'.

```
gss %>%
  filter(degree == 'Graduate' | degree == 'Bachelor') %>%
  group_by(religion) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
# A tibble: 6 x 4
  religion   total    freq   pct
  <fct>      <int>   <dbl> <dbl>
1 Protestant   367 0.430    43
2 Catholic     193 0.226    22.6
3 Jewish        27 0.0316    3.2
4 None         204 0.239    23.9
5 Other         59 0.0691    6.9
6 <NA>           4 0.00468   0.5
```

Now our results include only college graduates and graduate degree holders. If we want to see them broken out separately after we have filtered, all we need to do is change `group_by(religion)` to `group_by(religion, degree)`.

What if we want to filter our observations for all individuals with less than a bachelor's degree? We can create a vector with our specific criteria and then use it in our filter argument. Look at this:

```
filter_criteria <- c('Lt High School', 'High School', 'Junior College')

gss %>%
  filter(degree %in% filter_criteria) %>%
  group_by(religion, degree) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1))
```

```
`summarise()` has grouped output by 'religion'. You can override using the
`.groups` argument.
```

```
# A tibble: 17 x 5
# Groups:   religion [6]
   religion   degree         total   freq   pct
   <fct>      <fct>          <int>  <dbl> <dbl>
 1 Protestant Lt High School   155 0.155   15.5
 2 Protestant High School      742 0.740   74
 3 Protestant Junior College   106 0.106   10.6
 4 Catholic   Lt High School   100 0.220   22
 5 Catholic   High School      322 0.708   70.8
 6 Catholic   Junior College    33 0.0725   7.3
 7 Jewish     Lt High School     1 0.0417   4.2
 8 Jewish     High School       17 0.708   70.8
 9 Jewish     Junior College     6 0.25    25
10 None       Lt High School    62 0.150   15
11 None       High School      298 0.722   72.2
12 None       Junior College    53 0.128   12.8
13 Other      Lt High School    10 0.1     10
14 Other      High School       73 0.73    73
15 Other      Junior College    17 0.17    17
16 <NA>       High School        9 0.9     90
17 <NA>       Junior College     1 0.1     10
```

We've first created a vector, called `filter_criteria`, with all of the degree-levels we want to include in our data (we've left out 'Graduate' and 'Bachelor'). Then, we've set the filter criteria to say, "Take all respondents who have a degree listed in our vector, `filter_criteria`." In code, we write this as: `filter(degree %in% filter_criteria)`.

3.10.1 The %in% Operator

`%in%` is a special logical operator that checks to see whether the values you are specifying are contained in an object. If the value is contained in the object, your computer will return `TRUE` and if not, it will return `FALSE`. This is especially useful for `filter()` since `filter()` selects rows based on whether they meet a criteria (`TRUE`) or not (`FALSE`).

Here's a simple example of how this operator works in general:

```
1 %in% c(1,2,3,4,5)
```

```
[1] TRUE
```

```
6 %in% c(1,2,3,4,5)
```

```
[1] FALSE
```

3.11 Fancy Tables with `kable()`

If we want to make a summary table look a little bit nicer, we can add the `knitr::kable()` function to the end of our query to produce something more polished.

```
gss %>%
  filter(degree == 'Graduate') %>%
  group_by(religion) %>%
  summarize(total = n()) %>%
  mutate(freq = total / sum(total),
         pct = round((freq*100), 1)) %>%
  knitr::kable()
```

religion	total	freq	pct
Protestant	126	0.3962264	39.6
Catholic	63	0.1981132	19.8
Jewish	15	0.0471698	4.7
None	82	0.2578616	25.8
Other	31	0.0974843	9.7
NA	1	0.0031447	0.3

The `::` operator here tells `R` to pull the `kable()` function from the `knitr` package (which is located in the tidyverse). This is useful when there are multiple functions with the same name in different packages.

You can also add additional code to your `kable()` function to customize the look of your table (see here for examples).

3.12 Another Example

What if we want to do something ultra-specific like find all survey respondents who are Protestant or Catholic, voted for Obama in the 2012 U.S. Presidential election, and have children? And, we'd like to know their breakdown by relative frequency across regions of the U.S.

Here's a brief example:

```
gss %>%
  filter(religion == "Protestant" | religion == "Catholic") %>%
  filter(obama == 1) %>%
  filter(childs > 0) %>%
  group_by(region) %>%
  summarize(total = n()) %>%
  mutate(freq = round(total / sum(total),4),
         pct = round((freq*100), 1))
```

```
# A tibble: 9 x 4
  region          total   freq   pct
  <fct>           <int>  <dbl> <dbl>
1 New England        33 0.0602   6
2 Middle Atlantic    57 0.104   10.4
3 E. Nor. Central   119 0.217   21.7
4 W. Nor. Central    36 0.0657   6.6
5 South Atlantic    121 0.221   22.1
6 E. Sou. Central    35 0.0639   6.4
7 W. Sou. Central    57 0.104   10.4
8 Mountain           35 0.0639   6.4
9 Pacific            55 0.100   10
```

Now we have the skills to find the percentage of Protestants and/or Catholics with children who voted for Obama in 2012 and reside in the South Atlantic census region (29.2%).

3.13 Practice Exploring Data

You can see here that the `dplyr` functions provide an enormous amount of flexibility and power. `R`, like other programming languages, is also very sensitive to mistakes in syntax or spelling: a missing comma in a set of function arguments, a hanging pipe operator, a misspelled filter criteria, or an erroneous object name can all cause output errors. Check your code carefully, take a deep breath, and try again. You'll get the hang of it in no time.

Use the remainder of class time today to explore the `gss_sm` data. Try summarizing different variables according to different groupings. Try using other measures like `mean()` or `sd()` to summarize numeric variables (like the number of children).

If you are feeling overwhelmed at the moment - don't despair, we're going to continue practicing these skills throughout the rest of the course.

Chapter 4

Visualizing with `ggplot2`

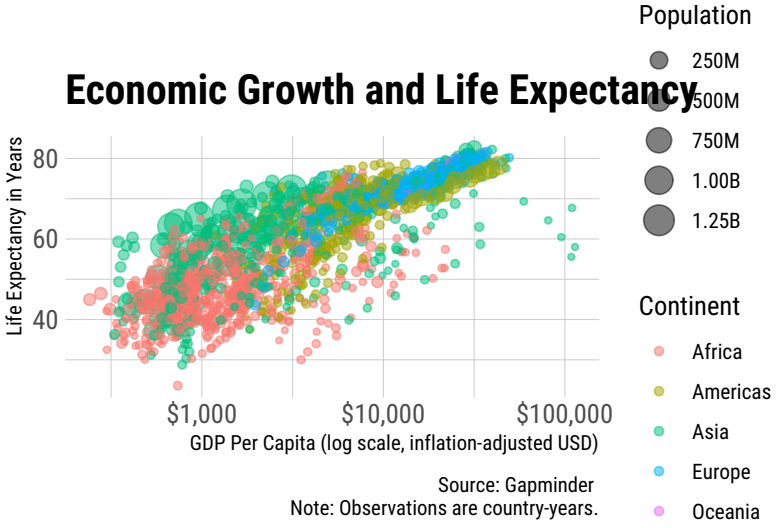
At this point, you might be wondering how I managed to win all of those trophies in my office.[1] The short answer: `ggplot2`. What is `ggplot2`, you ask?[2] It's a data visualization package from the tidyverse which allows you to build highly customizable (and sometimes beautiful) graphics. It's also the topic of this chapter.

But first, we need to step back and talk a little bit more about description, the purpose of data visualization, and where this all fits in. We'll continue from the previous chapter with a brief refresher on descriptive statistics, then move on to the principles of data visualization, and finish with some practical applications of `ggplot2`. Our end goal for today is to create something informative and rather nice-looking, like this:[3]

[1] I have neither trophies nor an office, but let's not let that spoil things.

[2] R.I.P. ggplot1 (2006-2008)

[3] Credit for this visualization and the series of examples derived from it belong to Healy (2019).



4.1 Descriptive Statistics

In the previous chapter, we learned how to use `dplyr` functions to summarize data. We started with individual-level observations (i.e., GSS respondents) and used `group_by()` , `summarize()`, and `mutate()` to distill our granular data into summary statistics, such as the proportion of GSS respondents by religious affiliation or the mean number of children by respondent's degree level. We can't say much yet about whether more Americans are Protestant or Catholic or whether U.S. college graduates tend to have more or less children than high school graduates — these questions require inference — but, we're now able to produce some of the statistics we'll need to examine these types of questions later on.

The point of producing **descriptive statistics**, like proportions or means, is that they allow us to identify characteristics of a set of observations (usually, a sample). For quantitative variables, if you recall from your statistics class, we can describe data with different types of measures. We have, for instance: **measures of central tendency**, which give us an indication of where the center of our distribution is (or what the typical observation may be); **measures of spread**, which tell us how far apart observations are from the center of the distribution; and what we might call other distributional measures, which can tell us how many values are in our sample or what the largest and smallest values may be. Categorical variables are simpler and we can generally describe them with a **frequency** (count) or **relative frequency** (the count expressed as a proportion or percentage) alone.

4.1.1 Measures for a Single Quantitative Variable

The tables below provide a brief overview of some of the measures we've already used or might use to describe a quantitative variable.

Central Tendency

Measure	*Description*	`R` *Function*
Mean	The sum of the values divided by the count. It is sensitive to outliers.	`mean()`
Median	The middle value, where half of the values are above and half are below. It is resistant to outliers.	`median()`

Spread

Measure	*Description*	`R` *Function*
Variance	The sum of squared deviations from the mean divided by the count minus one.[4] It gives us a sense of how far values typically are from the mean.	`var()`
Standard Deviation	The square root of the variance.[5] The more commonly reported measure of spread which, again, tells us how far values typically are from the mean.	`sd()`
Interquartile Range (IQR)	The distance between the 75th percentile value and the 25th percentile value. It gives us an indication of the spread for the middle-most values.	`IQR()`

Other Distributional Measures

Measure	*Description*	`R` *Function*
Minimum	The smallest value.	`min()`
Maximum	The largest value.	`max()`
Count	The number of values.	`n()`

[4]The formula for the sample variance is: $S^2 = \frac{\sum (x_i - \overline{x})^2}{n-1}$

[5]The formula for the sample standard deviation is: $S = \sqrt{\frac{\sum (x_i - \overline{x})^2}{n-1}}$

Example

Below is a table of descriptive statistics for the popular vote share received by candidates in the 2016 U.S. Presidential election by state (including the District of Columbia). Note, the unit of observation is a U.S. state and so we can read this as saying that Trump received 4.09% of the vote share in his lowest performing state and 68.17% in his highest, with a mean of 48.26% and standard deviation of 11.92% across states.

candidate	median	mean	var	sd	iqr	min	max
Trump	48.17	48.26	142.03	11.92	16.20	4.09	68.17
Clinton	46.17	44.61	148.49	12.19	15.73	21.88	90.86
Johnson	3.44	3.72	2.05	1.43	1.79	1.19	9.34
Other	2.74	3.41	12.30	3.51	1.53	0.00	24.32

As a general rule, we don't use variance in our descriptions and we report mean and standard deviation together. These latter two are especially important for certain inferential methods.

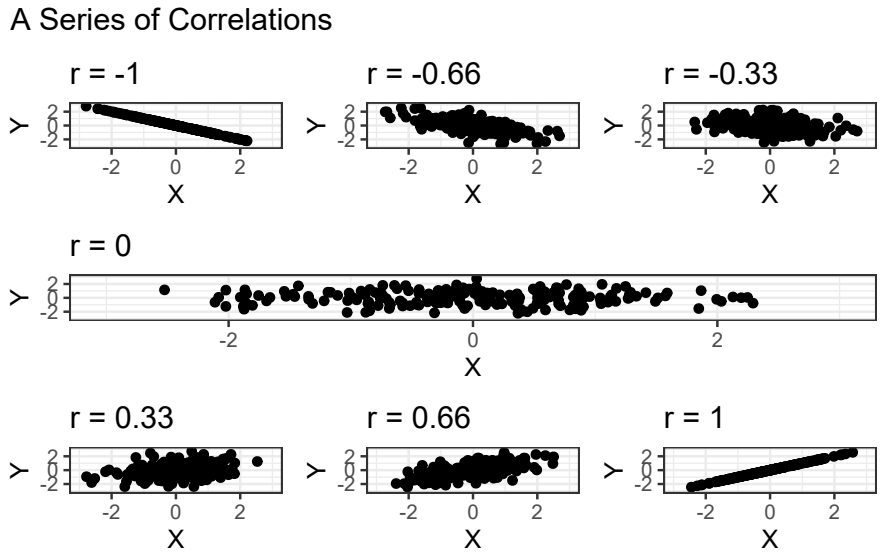
4.1.2 Measure for Two Quantitative Variable

To these univariate characteristics, we can add a measure for describing the relationship between two quantitative variables: the *correlation coefficient*.

Measure	*Description*	*R Function*
Correlation (r)	A measure of the strength and direction of the linear association between two quantitative variables.	`cor()`

In statistics, we generally make a distinction between an **association**, a relationship between two variables, and a **correlation**, or the linear association between two *quantitative* variables. Associations can refer to some relationship between variables of any type, but a correlation is a measure we calculate for two quantitative variables using a specific formula (or the `cor()` function in `R`). The distinction between association and correlation often gets lost in everyday language, but we'll try to maintain some precision here.

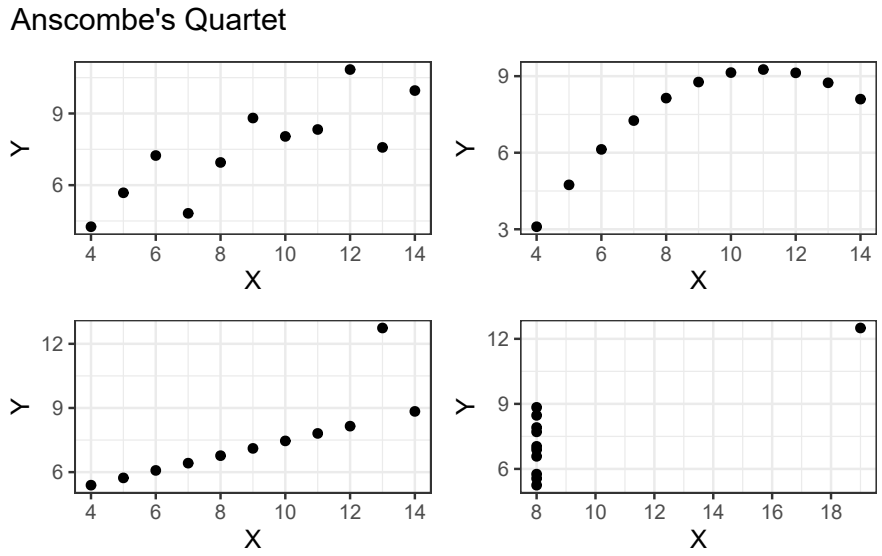
Correlations range between -1 and +1, with both extremes representing a perfect linear association of data points with some slope. The figure below shows a range of correlations for different sets of observations.



To describe a relationship between quantitative variables, it is useful to talk about:

- Strength, whether there is a strong (closer to -1 or +1) or weak correlation (closer to 0)
- Direction, whether the relationship is positive or negative
- Form, whether the association is linear or non-linear
- Outliers, whether there are observations that break the general pattern

The correlation coefficient is sensitive to outliers, which means that a stray observation can greatly influence the measure, and the general form of the relationship. You can see the effect of both in Francis Anscombe's classic example. The figure below shows four different sets of observations, each with the same correlation ($r \approx 0.82$) and other summary statistics.



4.2 Why Visualize?

This brings us to the central point of this chapter: data visualization isn't just fun, it is necessary. Correlations and other summary measures can be terribly misleading if used blindly. Checking a visual presentation of our data provides us with the opportunity to ensure that the underlying data matches our expectations. In the case of Anscombe's quartet, only one of the plots corresponds to what we might expect for a correlation of 0.82.[6]

There are other clear benefits to data visualization beyond the purely analytic. They can convey complex data in simple terms, for instance, and they can form lasting impressions.

These communicative benefits can be difficult to overstate. But it is important to remember that as much as we may want to convince others with aesthetically pleasing figures, it is the underlying veracity of our visualizations which matters most. To put it bluntly, if the visualization is eye catching, but uses poor quality data, it is not a good visualization. Similarly, if the visualization presents good data in a misleading way or fails to convey any meaning at all, it is not a good visualization. We need good data to make good visualizations and we must act as good analysts to ensure that accurate meanings are being conveyed.

[6]For an even more extreme example, see Alberto Cairo's Datasaurus Dozen, all of which have approximately the same correlation and summary statistics.

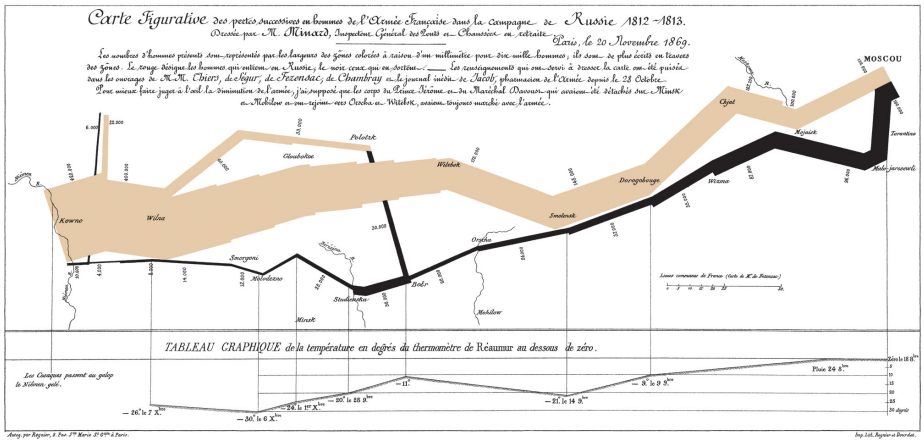


Figure 4.1: Charles Minard's famous, "*Carte figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812–1813"* (Source: Wikimedia).

4.3 Some Principles

What makes for a good visualization then? The unsatisfying answer is that it depends. But, here are at some guiding principles that may be helpful:

Avoid features which distract from the data. Better charts, as Healy (2019) argues, usually maximize the data-to-ink ratio. This means that we don't want to add extras when they provide no benefit to interpretation and we should ensure that the features of the visualization all speak to the data in some way. We should avoid, for example, making 3D charts when an extra dimension serves no purpose.

Avoid perceptual traps. You have no doubt seen graphs with truncated Y-labels, which can overemphasize volatility in trends. Contrary to what you may have heard, these types of graphs can sometimes be appropriate - especially, when a small marginal change in an otherwise stable trend is of great consequence. But this sort of example belies a bigger issue, which is the challenge of matching the perception of the reader with the actual patterns in the data. You must take care not only when deciding on the appropriate scale for an axis, but also on the type of graph, the ordering and size of various elements, the choice of color gradient, and the relative width and height (aspect ratio) of the final product. Pie charts, as an example of a type of graph, happen to be particularly unintuitive because of the difficulty we human beings have in perceiving the relative size of different segments of a circle. In a bar chart, by contrast, we only need to compare the length of different bars to understand relative size, a much simpler cognitive undertaking.

Use the right measure. When it comes to analyzing data, you will no doubt

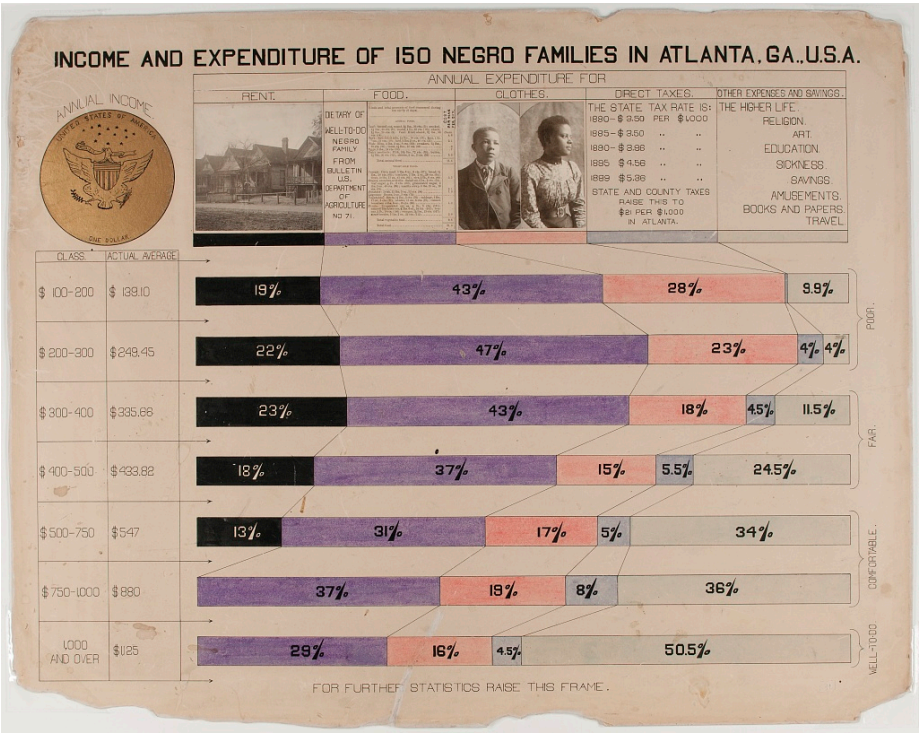


Figure 4.2: A bar chart produced by the American sociologist W.E.B. Du Bois (1868-1963) for the Paris Exposition Universelle in 1900 to show the economic progress of African Americans after emancipation (Source: U.S. Library of Congress).

have many options in terms of the measures you can use to convey your findings. But it is equally important that you choose the measure which is most appropriate for the comparisons you are making. This is not a problem specific to data visualization, per se, but it is one which crops up all too often. If you want, for example, to compare crime rates across geographic units, you will want to adjust your data to a *per capita* basis.[7] If you wish to compare typical worker salaries across countries, you will want to compare medians rather than means.

It may be apparent, in the foregoing discussion, that the most interesting visualizations generally involve two or more variables and bring the reader's attention to the relationships between them. The principles discussed here are, of course, not intended to be exhaustive and you'll sometimes find that the choices we make in visualizations come down to taste. At the very least, however, we should all endeavor to use visualizations to convey our information clearly and truthfully.

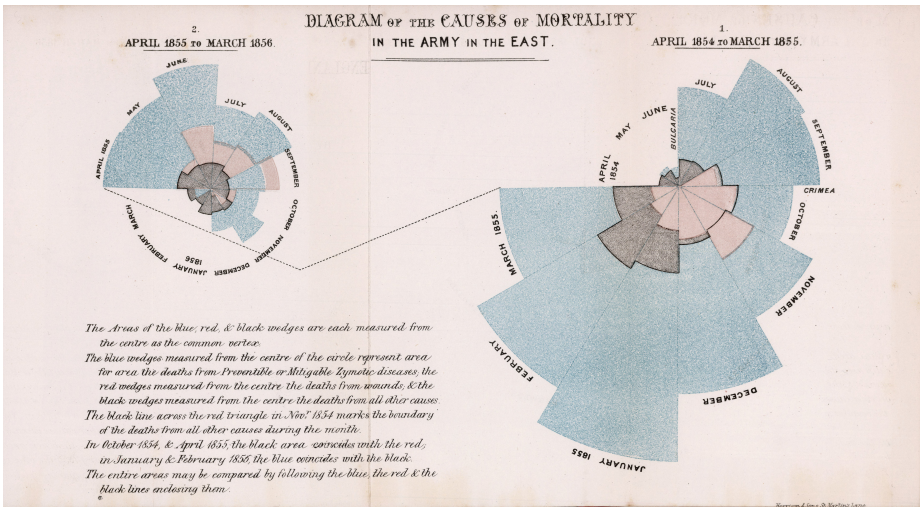


Figure 4.3: Florence Nightingale's (1858), "Diagram of the causes of mortality in the army in the East." A more effective pie chart where the perceptual relative size issue is negated by the amount of information conveyed and the potentially seasonal nature of the data.

4.4 Some Practicalities

The everyday graphs we make when conducting data analysis will usually be more functional than pretty, but that doesn't have to stop us from combining

[7] This problem is particularly common in maps, as illustrated in this blog post about density maps.

the two today. In the example below, we'll focus mainly on the mechanics of constructing a visualization using `ggplot2` rather than on how to use them analytically.

To get started, we'll load a new data set called `gapminder`, which contains data on countries. Conveniently, the `gapminder` data is located in the `gapminder` package. As usual, we want to be sure that we've installed the package before loading it for the first time.

```
library(gapminder)
data(gapminder)
```

The `data()` function used above is an alternative to `gapminder <- gapminder`. It loads the `gapminder` data from the package into a `gapminder` object in our environment. You can use either, but it's good to keep learning new functions at this stage so that you can understand what they do when you see them elsewhere.

As will become second nature to you to you soon, we can inspect this data using `glimpse()`, `view()`, or by clicking on the object in the environment pane.

```
glimpse(gapminder)
```

```
Rows: 1,704
Columns: 6
$ country   <fct> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan", ~
$ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, ~
$ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, ~
$ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854, 40.8~
$ pop       <int> 8425333, 9240934, 10267083, 11537966, 13079460, 14880372, 12~
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.1134, ~
```

Our new tibble has 1,704 rows and 6 variables (`?gapminder` provides some more information on the variables). There is something a little bit different about this data compared to the GSS data. Whereas in the GSS data each row corresponds to a separate observation (i.e., a respondent), in the Gapminder data, each row corresponds to a year for a particular country. We have, for instance, a row with data for Afghanistan in 1952 and another row for Afghanistan in 1957 in the next. The unit of observation here is called a "country-year." Consider for a moment how this might affect the answers you get when using `mean()` or `median()`.

In the social sciences, we call this format, **long data**. The differences in the way tabular data is stored has important implications for the way we analyze it. We'll discuss this more in depth in the next chapter. Luckily for us, the `gapminder` data is already in an ideal format for `ggplot2`.

Brief Exercise

As a brief exercise and to refresh your memory, let's use `dplyr` to find the minimum and maximum years in the `gapminder` data. Try it on your own first and then check your answer below.

Answer

```
gapminder %>%
  summarize(min_year = min(year),
            max_year = max(year))
```

```
# A tibble: 1 x 2
  min_year max_year
     <int>    <int>
1     1952     2007
```

Now see whether you can find the minimum and maximum year for *each country* along with the number of times each country appears in the data.

Answer

```
gapminder %>%
  group_by(country) %>%
    summarize(min_year = min(year),
              max_year = max(year),
              n = n())
```

```
# A tibble: 142 x 4
   country     min_year max_year     n
   <fct>          <int>    <int> <int>
 1 Afghanistan     1952     2007    12
 2 Albania         1952     2007    12
 3 Algeria         1952     2007    12
 4 Angola          1952     2007    12
 5 Argentina       1952     2007    12
 6 Australia       1952     2007    12
 7 Austria         1952     2007    12
 8 Bahrain         1952     2007    12
 9 Bangladesh      1952     2007    12
10 Belgium         1952     2007    12
# i 132 more rows
```

4.5 How ggplot2 Works

Back to visualizations: much like a cake, `ggplot2` involves adding layers. We start with a bare plot which has only our axes and their labels and then we work our way up to the final product, layer by layer.

4.5.1 Making the Base Plot

Just as with `dplyr`, we can use the pipe operator to work with `ggplot2`. We first take the Gapminder data *and then* add a new function, `ggplot()`.

```
gapminder %>%
  ggplot()
```


Without any arguments supplied, the `ggplot()` function produces a blank plot (shown above), a canvas we'll use to paint our visualization. If you are following along in an R Script, you should be able to see this plot in the lower right pane of your R Studio window (under the 'Plots' tab) after running it. We'd rather see a completed canvas than a blank canvas, however, so we're going to supply an argument called `mapping`. The `mapping` argument tells `ggplot` how we are going to map the data to the plot.

```
gapminder %>%
  ggplot(mapping = )
```

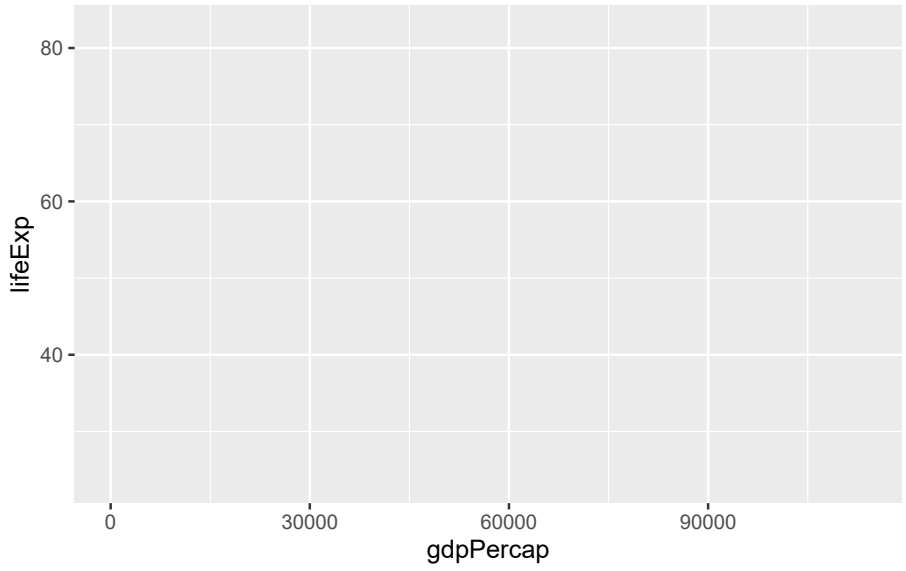
This mapping argument, in turn, requires us to specify an 'aesthetic' which will always be contained in an `aes()` function. So now we have:

```
gapminder %>%
  ggplot(mapping = aes())
```

If we were to run this, we would still get a blank plot. The `ggplot()` function knows we're using the gapminder data (since we used the pipe operator), but it doesn't yet know what we would like to see on our x- or y-axes. For this, we need to define the aesthetic characteristics of our plot.

Since our goal is to recreate the graph we saw in the beginning of this chapter, which showed the relationship between GDP per capita (`gdpPerCap`) and life expectancy (`lifeExp`), we'll supply these variables to the `x` and `y` arguments inside the `aes()` function.

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp))
```



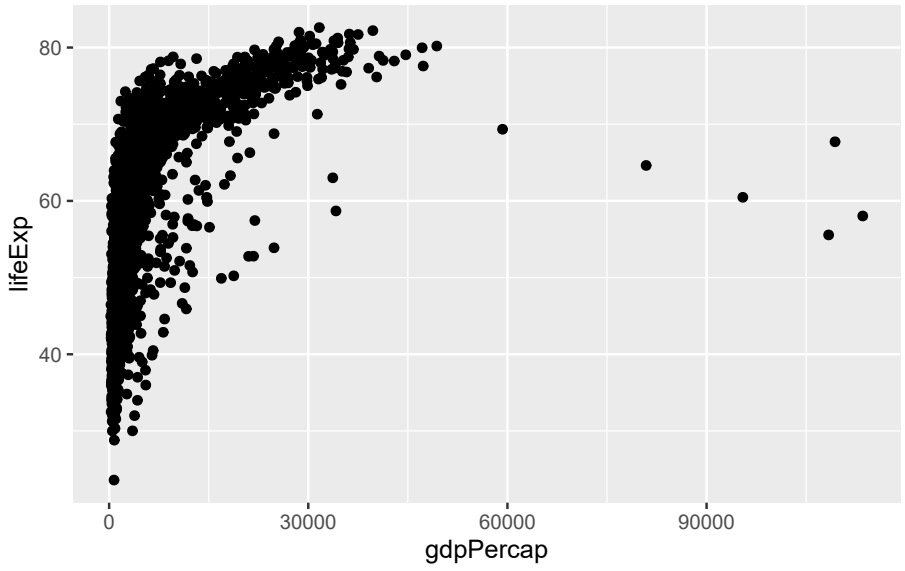
Now we have a not-so-blank plot. We can see instead an x-axis, as specified, showing `gdpPerCap`, and a y-axis, showing `lifeExp`. But where are our data?

4.5.2 Specifying the Type of Plot

In order to add data, we have to tell ggplot exactly what type of plot we'd like to create. We could produce a scatterplot, for example, which will cause the data to appear as points, or we could create a line graph, which will connect points into lines. There are other options, of course, but these seem like the most logical choices for this plot.

In `ggplot()`, the different types of plots are called *geoms* and we can add them as a layer to our base plot by using the `+` operator followed by the `geom_` function that corresponds to the type of plot we want to see. If we wanted to see a line plot, for instance, we would use `geom_line()`. We want to see a scatterplot, so we'll use the `geom_point()` function. Let's see how the scatterplot looks:

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp)) +
  geom_point()
```



We now have a plot which shows us each country-year as a point. The x-value is the GDP per capita of a country and the y-value is life expectancy at birth, a measure of typical longevity. Notice, we didn't need to supply an argument to `geom_point()` nor did we have to tell `ggplot()` anything other than the mapping of `x` and `y` (and, of course, the initial source of data, `gapminder`, via the pipe operator).

ggplot objects are unique in that we can add additional layers by using the + operator to join them to the base plot and each other. Just like with the pipe operator, however, we need to make sure that the + appears at the end of each intermediate line and not at the beginning of a line. Be on the lookout for these subtle syntax errors:

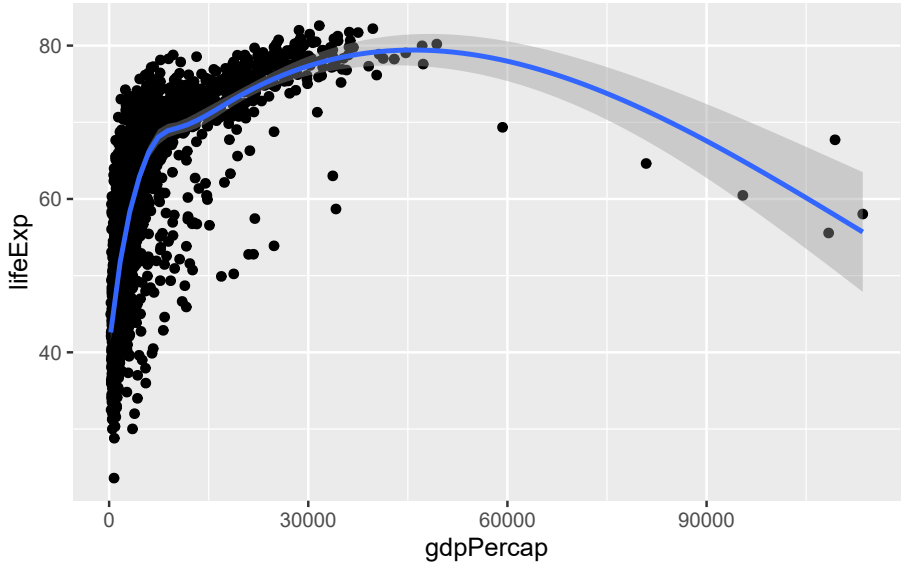
```
# This will not produce a plot with points,
# because the + operator is in the wrong spot.
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp))
  + geom_point()

# This will produce a plot with points.
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp)) +
  geom_point()
```

4.5.3 Adding a Smoother

Can we add more layers to our plot? You bet. We can, for instance, add a line of best fit on top of our points with a geom_smooth() function:

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp)) +
  geom_point() +
  geom_smooth()
```

```
`geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```



The output warning here tells us that `geom_smooth()` used a default argument and formula to calculate the line of best fit.[8]

4.5.4 Mapping More Aesthetics

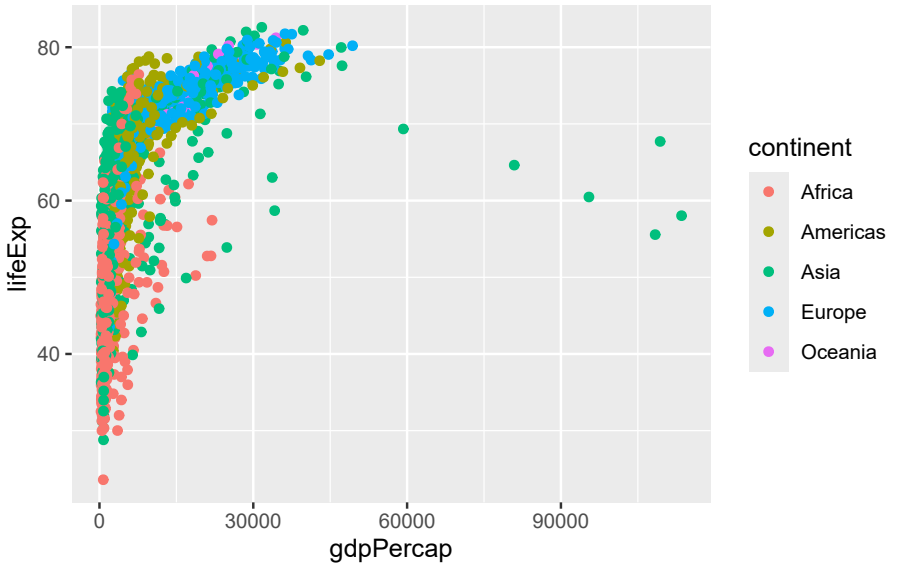
What else can we do with this visualization? Well, we might want to see if there are other elements that can be changed to reveal more patterns in the data. What if we compared the country-level relationship between `gdpPercap` and `lifeExp` by continent, for example? We could create a plot for each continent, showing only the relevant countries for each, or we could keep one plot and modify another element like the color of the data points. In this way, each color would represent the continent a country is located in and the points would be visually differentiated.

To do this, we need to modify the aesthetics of our data mapping. We'll add another argument to the `aes()` function inside of the `ggplot()` mapping argument for `color`. And, of course, since we want to color points by continent, we need to specify `color = continent`. We'll skip the line of best fit this time.

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       color = continent)) +
```

[8] 'gam' stands for general additive model and is one method of adding a line of best fit. 'lm', or linear model, is another best fit method.

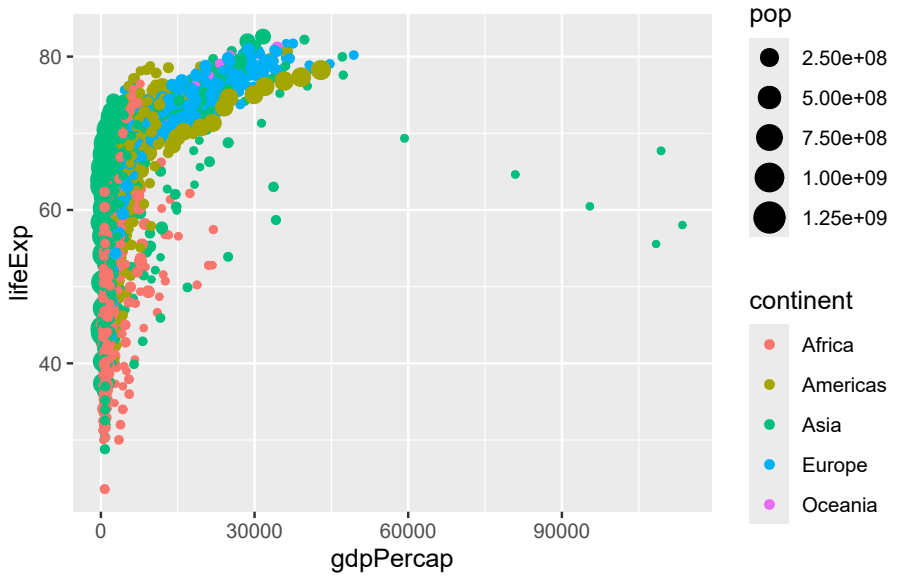
```
  geom_point()
```



Now we can see how the relationship between GDP and life expectancy plays out among countries across different continents.

It might also be interesting to see how this relationship plays out by population size. Since population size is a continuous quantitative variable, discrete colors may not be a good choice. We could add a color gradient scale (as you might see in a heat map, for example) or we could modify some other element. What about changing the size of the points according to population? Bigger countries could have larger points and smaller countries could have smaller points with a continuum in between. To do this, we need to add a size argument to the aesthetic mapping, this time according to population (`pop`).

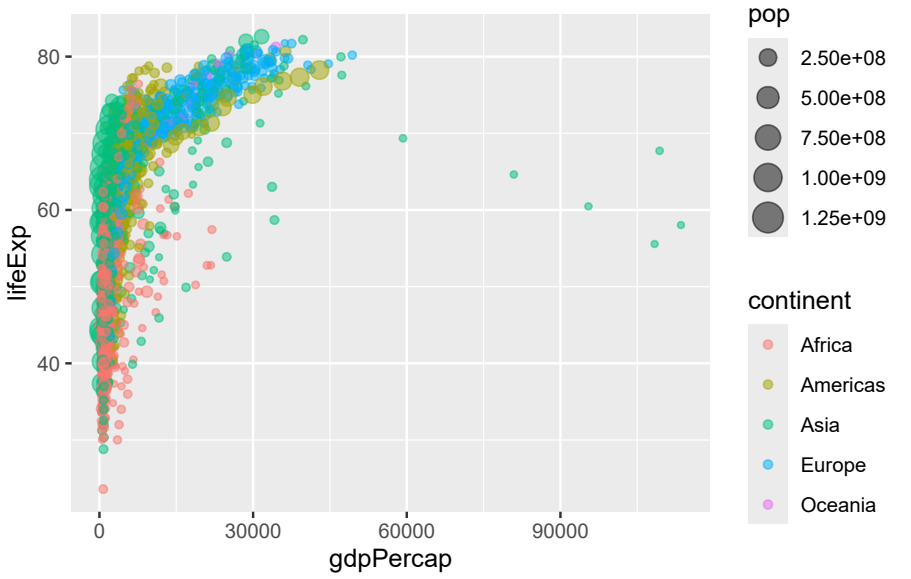
```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       color = continent,
                       size = pop)) +
  geom_point()
```



You can now see some trends for some specific countries, including a certain rich and high population country in the Americas. Each time we add an aesthetic,`ggplot()` makes the necessary change to the plot and then adds a key to interpret each element. We now have scales for population (the `size` value in our aesthetic mapping) and continent (the `color` value in our mapping). Another optional aesthetic argument you can use is `shape`, which changes the points from dots to different symbols (like x's or o's). `line` is another option which changes the type of line for `geom_line()`.

Because we have a lot of data points and they're overlapping, we're going to skip shape and make it so that the points have some transparency. We can do this by adding an `alpha` argument to the `geom_point()` component which controls transparency. `alpha` takes a value between 0 and 1, where 0 is completely translucent and 1 is not-transparent. We'll choose a halfway number, `0.5`, but you can play around with the different levels to find your preferred value.

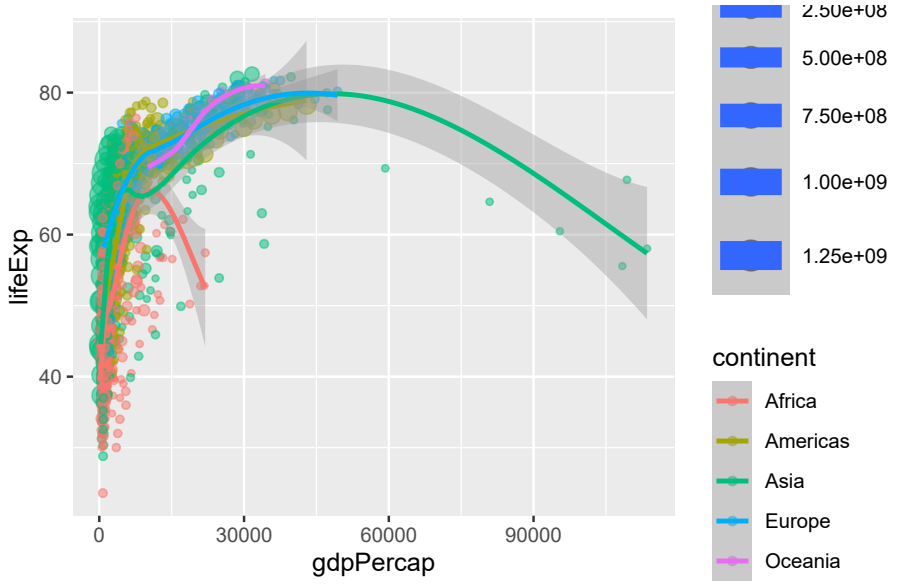
```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       color = continent,
                       size = pop)) +
  geom_point(alpha = 0.5)
```



We are getting pretty close to the graph we started the chapter with. At this point, we could add `geom_smooth()` back to our plot. Take a look at what happens when you do though.

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       color = continent,
                       size = pop)) +
  geom_point(alpha = 0.5) +
  geom_smooth()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



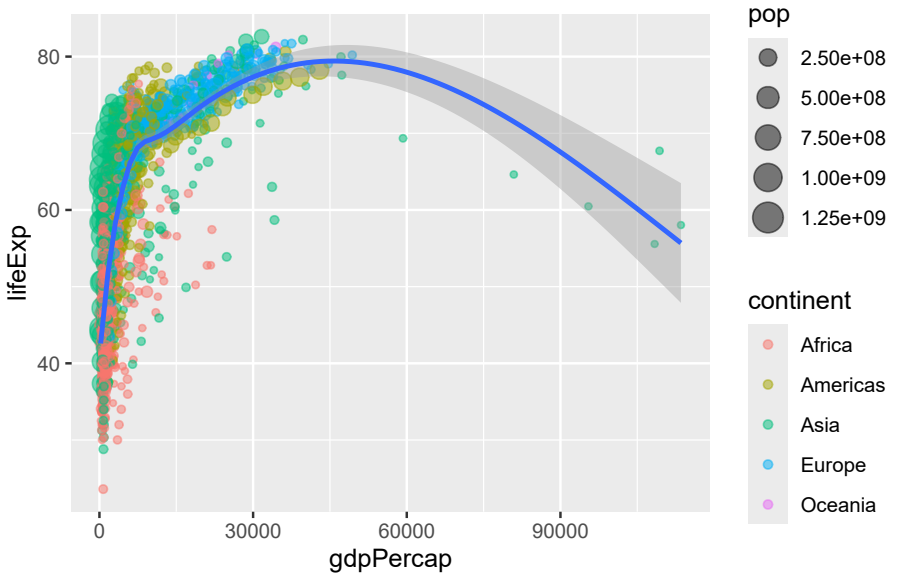
That's maybe not the result we thought it was going to be. Instead of one smooth curve, as before, we now have a different colored curve for each of the continents. We can also see in the key on the right-hand side that population size is affecting the width of the lines as well.

One thing to know about `ggplot2` is that each item added to the `ggplot()` object inherits `ggplot()`'s aesthetics. So because we defined `color` by continent and `size` by population in `ggplot()`'s mapping argument, `geom_point()` and `geom_smooth()` are also also colored by continent and sized by population.

If we want to instead ensure that the `color` and `size` arguments only affect `geom_point()`, we need to move those aesthetics to `geom_point()`'s own aesthetic mapping. See below:

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp)) +
  geom_point(mapping = aes(color = continent,
                       size = pop),
             alpha = 0.5) +
  geom_smooth()
```

```
`geom_smooth()` using method = 'gam' and formula = 'y ~ s(x, bs = "cs")'
```

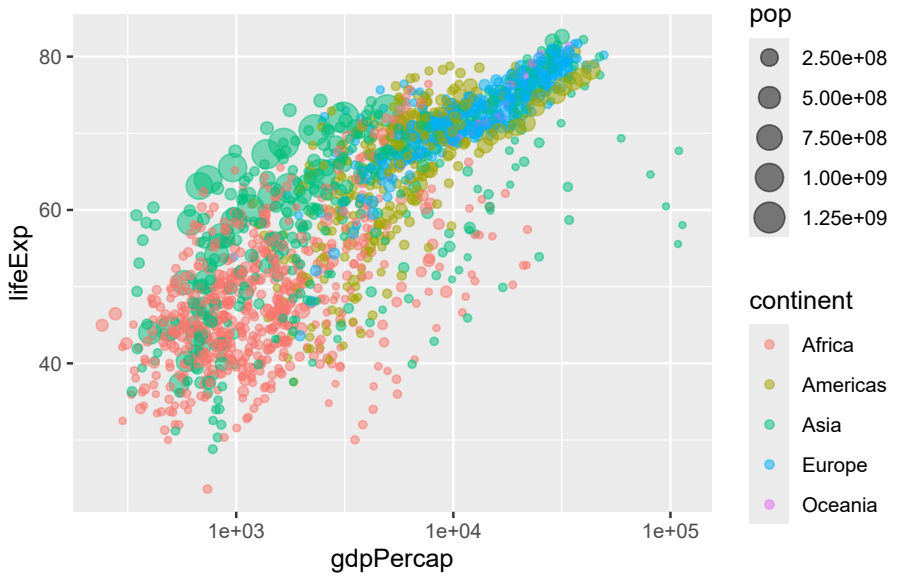


`geom_point()`'s mapping accepts the same form of argument as `ggplot()`. Let's remove `geom_smooth()` again anyways, since it doesn't seem particularly helpful and the plot looks better without it. We can return the `color` and `size` arguments to `ggplot()` or we can leave them as is. Just a few more changes left before we get to a finished product.

4.5.5 Changing Scales

Let's change the x-axis scale to a logarithmic scale, since the data appears to follow a logarithmic form. Scales can be changed by adding functions from the `scale_` family to our plot. Like `geom_`, there are a number of different options depending on the need. In this case, we want a logarithmic scale for our x-axis in base 10, so we'll use `scale_x_log10()`.

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       size = pop,
                       color = continent)) +
  geom_point(alpha = 0.5) +
  scale_x_log10()
```



Now we can see the relationship between GDP per capita and life expectancy more clearly. We don't always have to change the scale of our x- and y-axes, but in this case, the distribution of our x-values calls for it.[9] If you don't add a `scale_` function, `ggplot2` will simply use the default. We don't need a scale transformation for our y-axis here, so we'll skip adding a `scale_y_` function and let `ggplot2` use the default.

Note that there is a conceptual difference between a scale (the numeric distance between positions on some axis) and labels (how the values of those different positions are recorded). Just because the numbers on the axis are written in a strange or unhelpful format, in other words, doesn't necessarily mean we that will need to change the scale. We may just need to edit the labels. As you can see in our previous example, even after the scale change, the x-axis labels are still not recorded in the most legible format (scientific notation).

4.5.6 Changing Scale Labels

Changing the labels for axes and other scales can be a bit of a pain. Fortunately, there is a very helpful package we can use called `scales`.

You should already have a copy of `scales` installed and you can load it via `library()`.[10] Another way to access it's functions is to use the name of the

[9] GDP per capita is not normally distributed hence the need for a log-transformation here. The decision on whether to transform an axis or not is a statistical matter, which we won't go into here.

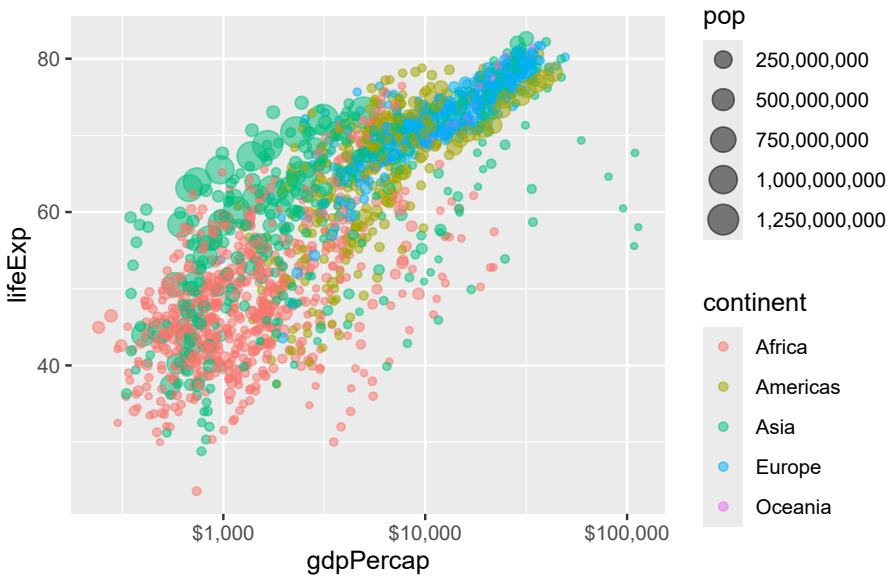
[10] `scales` is contained in the `tidverse`, but it isn't automatically loaded when you use `library(tidyverse)`.

package followed by `::` and the name of the desired function. We used this same method in the previous chapter for `knitr::kable()`.

For the scale on the x-axis, which corresponds to a variable in U.S. dollars, we'll use the function `scales::label_currency()`.[11] We'll add this helper function to the `labels =` argument of our `scale_` function in the example below. Other useful `scales` functions include `scales::comma` and `scales::percent` - neither of which require parentheses at the end, unlike `label_currency()`.

Since we also want to fix the label for the `size` function, we'll also add a `scale_` function for our size aesthetic (in a separate object) and then set the `labels` argument to use commas. See below for both steps put together:

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       size = pop,
                       color = continent)) +
  geom_point(alpha = 0.5) +
  scale_x_log10(labels = scales::label_currency()) +
  scale_size(labels = scales::comma)
```



Our scale labels have been fixed and are much easier to read.

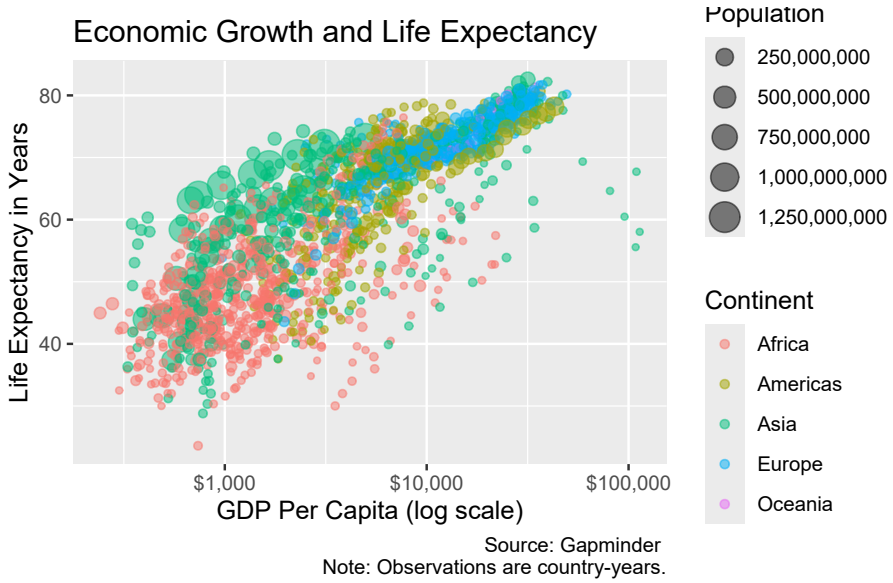
[11] The default currency for `label_currency()` is U.S. dollars.

4.5.7 Adding Titles

To change the titles of different elements, we can add a `labs()` function to the end of our object. The `labs()` function will set titles for each part of the plot according to the values given to a set of corresponding arguments.

In the code below, you can see that we've changed the title for the `x` and `y` axes, the `size` key ("Population"), the `color` key ("Continent"), the overall `title` of the graph ("Economic Growth and Life Expectancy"), and then added a `caption` at the bottom.

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       size = pop,
                       color = continent)) +
  geom_point(alpha = 0.5) +
  scale_x_log10(labels = scales::label_currency()) +
  scale_size(labels = scales::comma) +
  labs(x = "GDP Per Capita (log scale)",
       y = "Life Expectancy in Years",
       size = "Population",
       color = "Continent",
       title = "Economic Growth and Life Expectancy",
       caption = " Source: Gapminder \n Note: Observations are country-years.")
```



At this point, we have a good looking graph and could call it a day. As you will discover though, there are endless opportunities for customization with `ggplot2`. It's the reason why `ggplot2` graphics can be made to look so good.

4.5.8 Adding a Theme

Themes are customizable sets of aesthetic characteristics that change things like font types and sizes, the alignment of different elements, and the presence of gridlines. You can adjust many of these things by adding a `theme()` function to the end of your plot and playing around with the different available arguments.

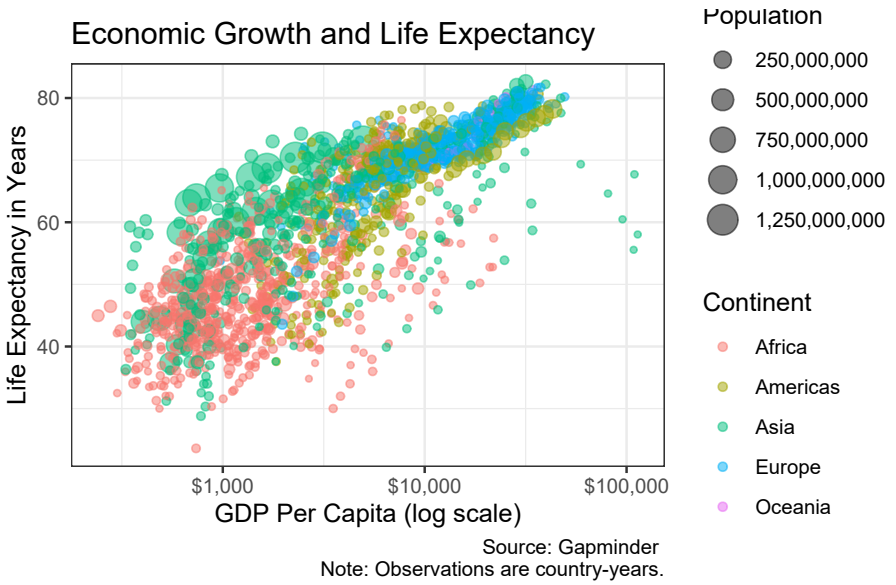
Alternatively, you can use a theme that someone else has created by installing their package and using the related function. This is often ideal, because you can then find a theme that matches your general preferences and tweak minor elements as needed by overlaying another `theme()` layer. Playing around with theme settings on your own can be a time consuming affair and in general, isn't recommend for the personal graphs you use for analytic purposes.

Among pre-packaged themes, `ggthemes`, for example, is a popular package with themes that mimic the styles used in *The Economist* (`theme_economist()`), for example, and the *Wall Street Journal* (`theme_wsj()`). You can see some more of the styles available in `ggthemes` here. The theme I used for the graph at the start of this chapter is called `theme_ipsum_rc()` which comes from the `hrbrthemes` package. Remember, if you use a theme from a package, you need to first download the package and then load the library (or access the specific function using `::`). Some custom themes, like ipsum, also require you to install and register new fonts in `R`, which can be a pain.

If you'd like to avoid the trouble of installing extra packages, you can also use some of the default themes provided in `ggplot2`, many of which are also quite nice. Adding `theme_bw()` from `ggplot2` to the plot from the previous example, for instance, does this:

```
gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       size = pop,
                       color = continent)) +
  geom_point(alpha = 0.5) +
  scale_x_log10(labels = scales::label_currency()) +
  scale_size(labels = scales::comma) +
  labs(x = "GDP Per Capita (log scale)",
       y = "Life Expectancy in Years",
       size = "Population",
       color = "Continent",
       title = "Economic Growth and Life Expectancy",
```

```
      caption = " Source: Gapminder \n Note: Observations are country-years.") +
  theme_bw()
```



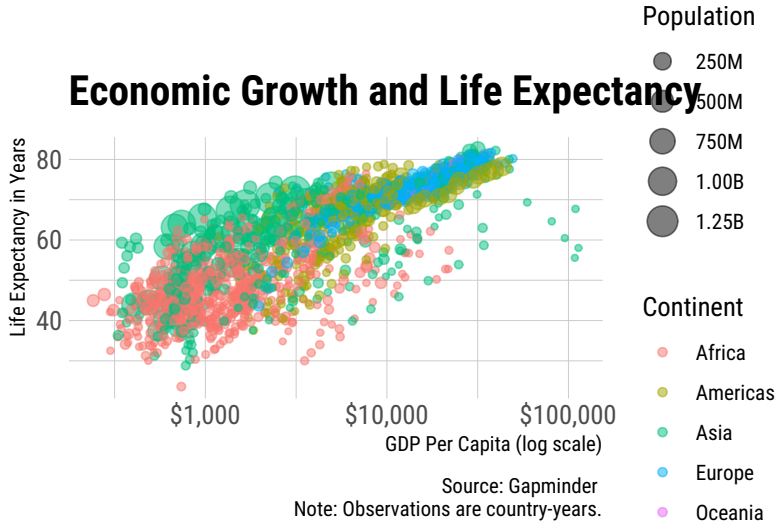
You can see that it has added a border to the plot and removed the gray background from both the plot and the scales. Try using `theme_minimal()`, `theme_classic()`, and `theme_void()` to see how they change the aesthetics instead.

4.6 The Final Product

To return to the final product, I'll use `theme_ipsum_rc()` from `hrbrthemes`. I'll also replace the size_scale argument with a more complicated set of functions that makes it even easier to read.

```
library(hrbrthemes) # A theme used for graphs

gapminder %>%
  ggplot(mapping = aes(x = gdpPercap,
                       y = lifeExp,
                       size = pop,
                       color = continent)) +
  geom_point(alpha = 0.5) +
```

```
scale_x_log10(labels = scales::label_currency()) +
scale_size(labels = scales::label_number(scale_cut = scales::cut_short_scale())) +
labs(x = "GDP Per Capita (log scale)",
     y = "Life Expectancy in Years",
     size = "Population",
     color = "Continent",
     title = "Economic Growth and Life Expectancy",
     caption = " Source: Gapminder \n Note: Observations are country-years.") +
theme_ipsum_rc()
```



Whichever plot you choose to use as your final plot, you can save it by clicking on the "Plots" tab in the lower right-hand corner of your R Studio window followed by export. We'll discuss better ways of doing this later on.

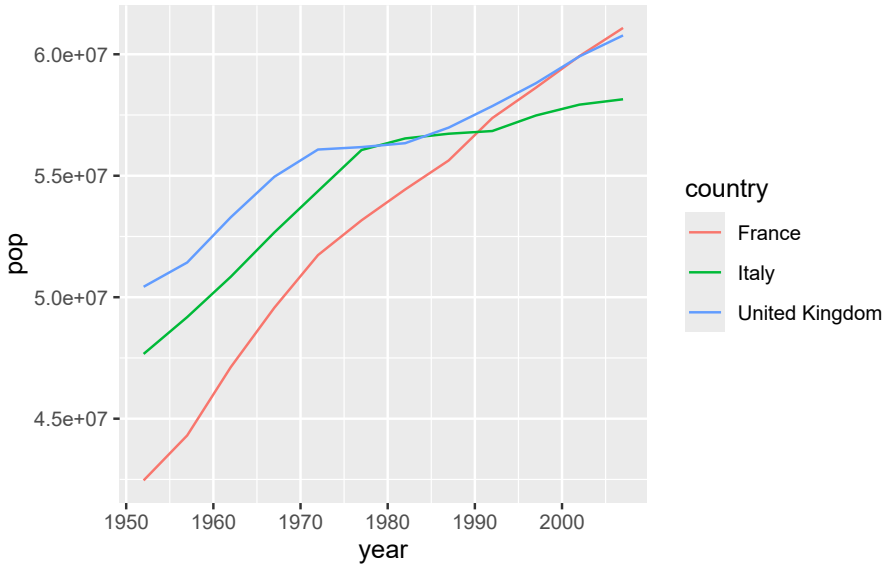
4.7 Other Plots

We've so far only covered one type of plot, a scatterplot. It won't surprise you to learn that there are many other types of plots that we can create using `ggplot2`. The good news is that the structure and components of plots are generally consistent across types and that once you start creating plots, you can always re-use the code.

A quick example of a line chart using the `gapminder` data is shown below. Note, we've made a few changes. We first filtered the data for a small subset of

countries so that our graph won't be swamped with too many countries. Then we used `geom_line()` as our `geom_` instead of `geom_point()`. Perhaps most importantly, we've set `color` to represent each country in order to ensure that our data is mapped to the appropriate unit of observation.

```
my_countries = c('France', 'United Kingdom', 'Italy')

gapminder %>%
  filter(country %in% my_countries) %>%
  ggplot(aes(x = year,
             y = pop,
             color = country)) +
  geom_line()
```

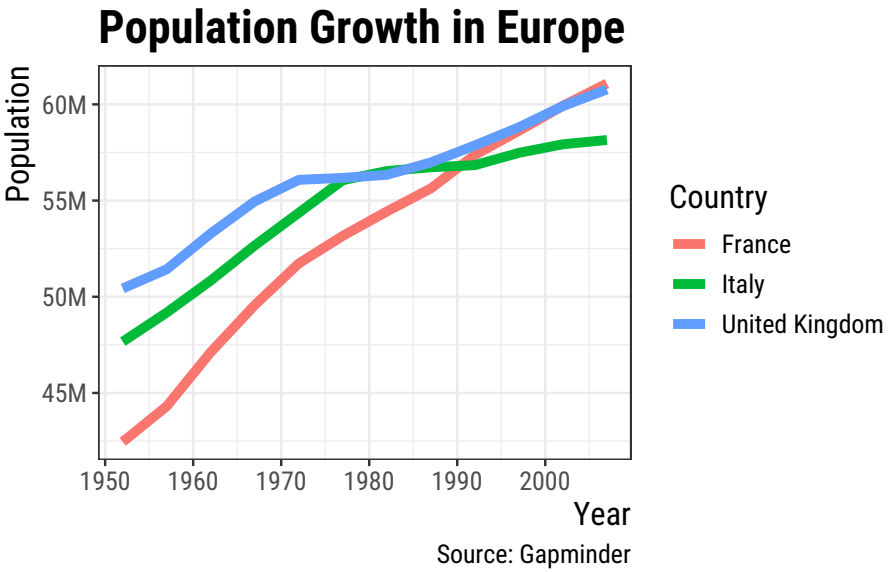


Try setting `color = continent` in the example above and see what happens. You end up with a bit of a mess. `ggplot2` doesn't naturally understand the correct unit of observation, so specifying continent as `color` leads it to believe that it needs connect the data points according to the continent for each year and across years. Since France, Italy, and the U.K. are in the same continent, it draws a line connecting each of the three data points inside each year and then connects them across years, leading to a jagged, meaningless graph.

Here's a slightly more polished looking version of the initial line graph with some aesthetic and label changes. We can easily change the countries used and other features like the size of `geom_line()`. Perhaps confusingly, `geom_line()`

can also take a `size` argument outside of the mapping argument. Note also that you may wish to change the font family in the graphic below to something like 'Arial', since you may not have 'Roboto Condensed' installed on your computer.

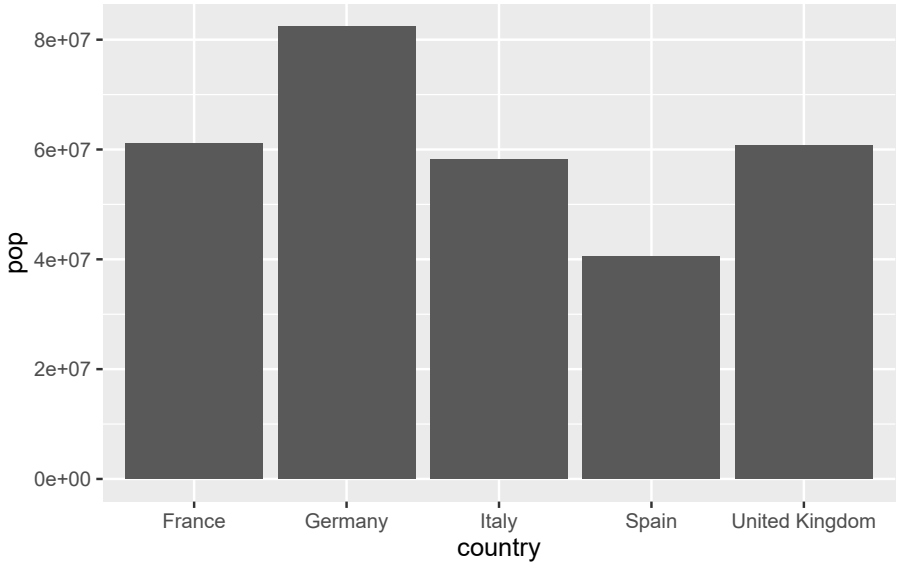
```
my_countries = c('France', 'United Kingdom', 'Italy')

gapminder %>%
  filter(country %in% my_countries) %>%
  ggplot(aes(x = year,
             y = pop,
             color = country)) +
  geom_line(size = 2) +
  scale_y_continuous(labels = scales::label_number(scale_cut = scales::cut_short_scale())) +
  labs(x="Year",
       y = "Population",
       color = "Country",
       title = "Population Growth in Europe",
       caption = "Source: Gapminder") +
  theme_bw() +
  theme(text = element_text(size = 14, family = "Roboto Condensed"),
                    plot.title = element_text(size = 20, face = "bold"),
                    axis.title.x = element_text(hjust=1),
                    axis.title.y = element_text(hjust=1))
```



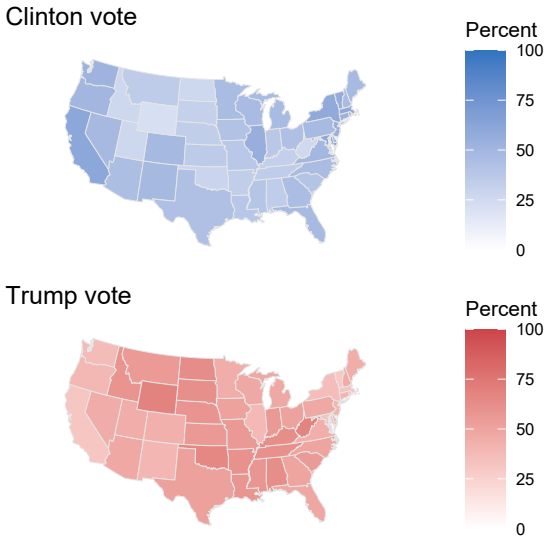
Last, but not least, we have a bar chart. Here again we've filtered for some

countries. We've then filtered for a specific year, removed color (which works slightly differently for bar charts), and have added the `geom_col()` geom to specify that it is a bar chart we are making.

```
my_countries = c('France', 'United Kingdom', 'Italy', "Germany", "Spain")

gapminder %>%
  filter(country %in% my_countries) %>%
  filter(year == 2007) %>%
  ggplot(mapping = aes(x = country,
                       y = pop)) +
  geom_col()
```



We will work through other examples of visualizations, such as the map below, and the quirks of how they work in future chapters.



4.8 Summary

In this chapter, we've reviewed some of the measures we might use to describe data, discussed some general principles for producing good data visualizations, and learned how to create and modify some basic plots in `ggplot2`.

You will likely need to read this chapter and reference the code more than once. `ggplot2`'s structure is not very intuitive to new users. The more you use it, however, and get a feel for how the different plot elements map to the various objects and arguments, the more control you will have over the visualizations you produce.

So keep practicing and save your work, adding comments so that you can remember what you were doing. Go back to some of the examples used here and play around with the different arguments. Try to change the plot types. Use different scales. See if you can make something interesting. As you inevitably get errors, try to make a mental note of what works and what doesn't. Eventually, you'll have generated a stockpile of code that you can re-use and adjust to create the right visualizations whenever you need them.

4.9 Exercises

Here are a few exercises to complete either in-class or on your own for homework.

1. Use the Gapminder data to produce a line graph showing growth in GDP per capita for the United Kingdom, France and Italy.
2. Do the same for life expectancy using three other countries of your choice.
3. Produce a scatterplot which shows the relationship between GDP per capita and life expectancy for all countries in 2007. Produce another which shows the relationship for 1952 and compare the two.
4. Produce a bar chart for life expectancy among countries in Oceania in 2007.

Chapter 5

Workflows and Wrangling

At this point in the course, we have primarily been working with data from packages, which is generally convenient and straight forward. We install the package with the data we want, we load the package from the library, and then we access the data and begin summarizing it. The bad news is that the data we might want to use for a particular project oftentimes isn't contained in a package. The even worse news is that we usually need to clean and transform the data in order to get it into a format that works for analysis.

This chapter covers some of the processes and functions in `R` made for dealing with these unfortunate eventualities. We'll start with the general problem of working with files in `R`, then learn how to get data in, and then we'll learn more about how to make data analyzable.

This chapter will not, of course, provide you with the answers to all of the data wrangling problems you will eventually encounter. It will barely scratch the surface. But, once you manage to get data in, data wrangling becomes a matter of learning new functions and practicing the tidy data skills you've already started to learn.

5.1 What's Happening Under the Hood?

Up until now, we've been writing our code in plain-text files saved with a `.R` file extension (or what we've been calling R script files) and we haven't needed to load data from other files or to save anything.

That last part isn't entirely true though. We have actually been loading data from files saved on our computers. It just so happens that the packages we installed earlier and the `library()` function have gone to great lengths to simplify and conceal the back-end interactions that led to data showing up in our

RStudio environment. All of this data is saved somewhere on our computers, we just might not know where it is.

As with any type of computer program, `R` itself operates from somewhere on your hard drive. It uses data, functions, and other compiled code which have been saved across a number of files (and file types) in different locations to run the program and allow us to interact with it. This is the case for everything software-related, from the apps on your smartphone to the operating system on your laptop. It's all running from code saved somewhere in the device's memory.

For this chapter, we don't have to go deep into the code that composes `R` or `RStudio`. All we need to know, instead, is how to get `R` to interact with files saved in different locations on our computer as well as how to organize them in a way that makes them easy to work with.

5.2 File Structures, File Paths, and the Working Directory

Your computer's operating system (Windows, MacOS, or Linux) organizes the files on your computer into folders. You might have created some folders on your own, renamed them, or stored files like photos, documents, and other items across them. Pictures might go in a "Pictures" folder, for example, and documents in a "Documents" folder. You might have a "Sciences Po" folder and then a sub-folder for "SPSSUR."[1]

The way files are organized across folders and sub-folders on a computer's hard drive is usually referred to as your computer's **file structure**. Within a file structure, each file has a **file path**, which is an address that identifies where the file is located. In Windows, they usually look something like this:

```
"C:\Users\wcs26\Documents\Sciences Po\SPSSUR\my_file.R"
```

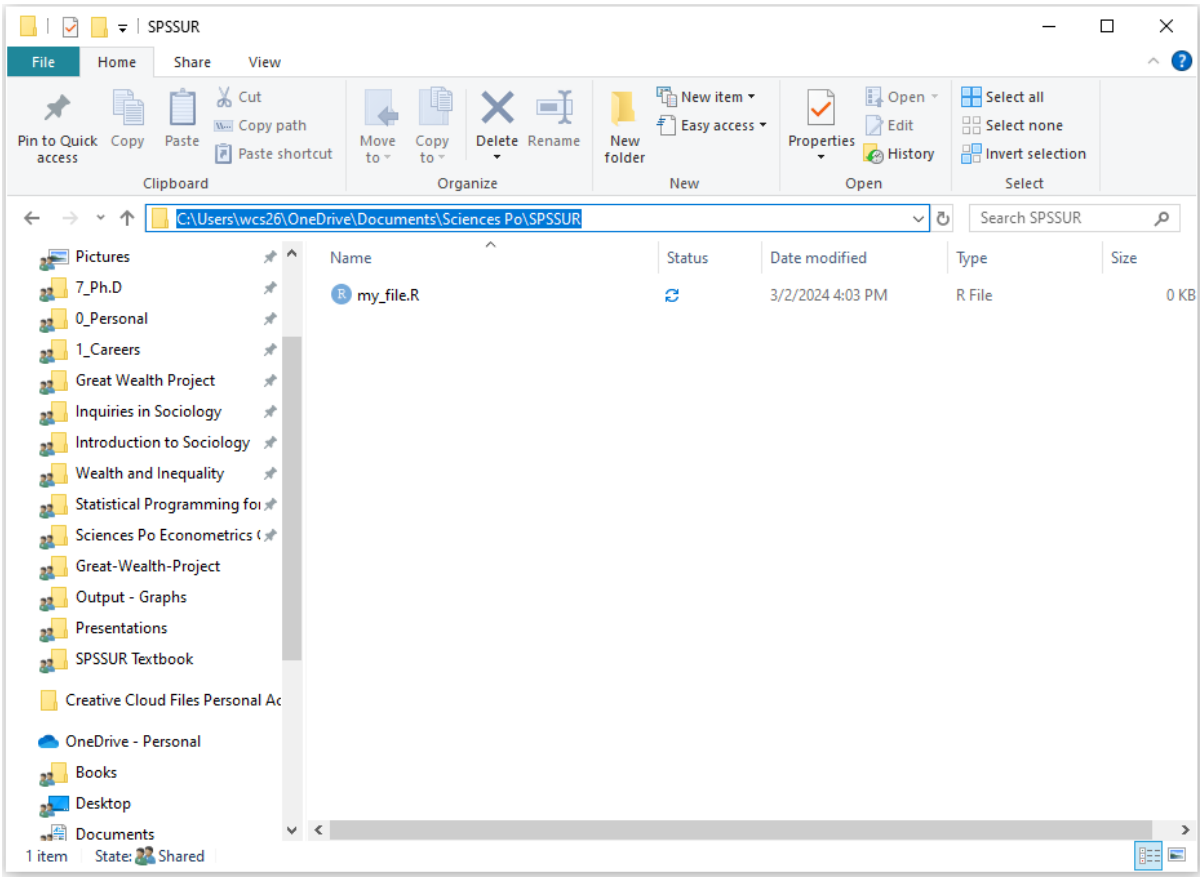
In MacOS, they might look more like this:

```
"/Users/wcs26/Documents/Sciences Po/SPSSUR/my_file.R"
```

As an important aside, when you write file paths in `R`, you will generally want to write them in the MacOS format you see above (i.e., using forwards slashes, `/`) even if you are using a PC. This is because the backwards slash, `\`, is a special character in `R`.

[1] You may also have all of your files saved on your "Desktop" folder, in which case, you should consider applying the Marie Kondo principles of tidying up to your digital spaces. A minor caveat, though, which is that sparking joy may not be a good criteria for computer system files.

Within the operating systems themselves, file paths are slightly easier to find in Windows than in MacOS since you can get most of the way there by clicking in the address bar at the top of a Windows Explorer window (see below for an example).[2] This gives the folder path, which when followed by another slash, the file name, and the file extension, gives the file path. In MacOS, finding a file path requires a little more effort (see here for some guidance).[3]



When working with multiple files in R, as you will in more involved analysis projects, you will need to pay some attention to the file structure and file paths. When you load data from a file, for instance, R will need to know exactly where the file is located. If you are saving a graph, similarly, R will need to know where you want to save it.

[2]Note that if you copy and paste a path from a Windows Explorer into R, you'll have to change the direction of the \ to / or deal with the \ issue in a different way (there are other ways, but hopefully you won't need to copy and paste paths anyways).

[3]If you are using a Linux-based OS, you probably won't need an explanation of file structures and file paths, given the greater transparency of file paths and structures you regularly encounter.

R makes an educated guess as to where in your file structure you are working from based on how you open your R session. This is called the **working directory** and you can identify where R has located it using the command below:

```
getwd()
```

Sometimes the working directory doesn't quite match where you think it should be and you might need to change it manually as a result. We will endeavor to avoid this when we can, but if you must, you can always manually set it using `setwd()`.

5.3 The Problem with File Paths

Functions that load data generally require you to provide a file path that leads to the data file. The problem with file structures and file paths, however, is that everyone's is different. So, if we have a file saved in a specific location on our computer and then some code that reads it, how do we make sure that other people can use our code when their file structure is going to be different?

To make this more concrete, let's say that I send you an R Script along with a data file called `important_data.csv`, so that you can replicate some analysis and visualizations I did. In that R Script, I might have a line that looks like this:

```
read_csv("C:/Users/wcs26/Documents/Sciences Po/SPSSUR/important_data.csv")
```

This command will try to read in a data file, `important_data.csv`, located within `C:/`, the `Users/` folder, the `wcs26/` folder, and so on and so forth. As soon as you run it, R will attempt to read the file located at the end of this path by working its way through the file structure. As soon as it hits a folder it can't find on your computer, however, it will stop and produce an error that looks something like this:

```
Error: 'C:/Users/wcs26/Documents/Sciences Po/SPSSUR/important_data.csv' does not exist
```

This type of file path construction, in which every folder and sub-folder on the way to the destination is provided, is called an **absolute** file path. If any part of the address is incorrect, R won't be able to find the file, the file won't be read, and the code fails to run. So, what's the alternative?

Well, one solution is to use what are called **relative** file paths. A relative file path might look something like this:

```
read_csv("important_data.csv")
```

In this case, because we haven't provided the full address, R fills in the rest by guessing and, naturally, it guesses that the missing part of the address is the working directory (i.e., C:/Users/wcs26/Documents/Sciences Po/SPSSUR/). If R guessed correctly and the file is located in your working directory, then this works perfectly fine and the data will be read correctly. If the file isn't located in the working directory, however, it will fail.

The trouble is that sometimes you might start your R session in one place and then open files from another place. Since R can only keep track of one working directory at a time per session, it (or rather you) will get confused quickly and your relative file paths will also fail.

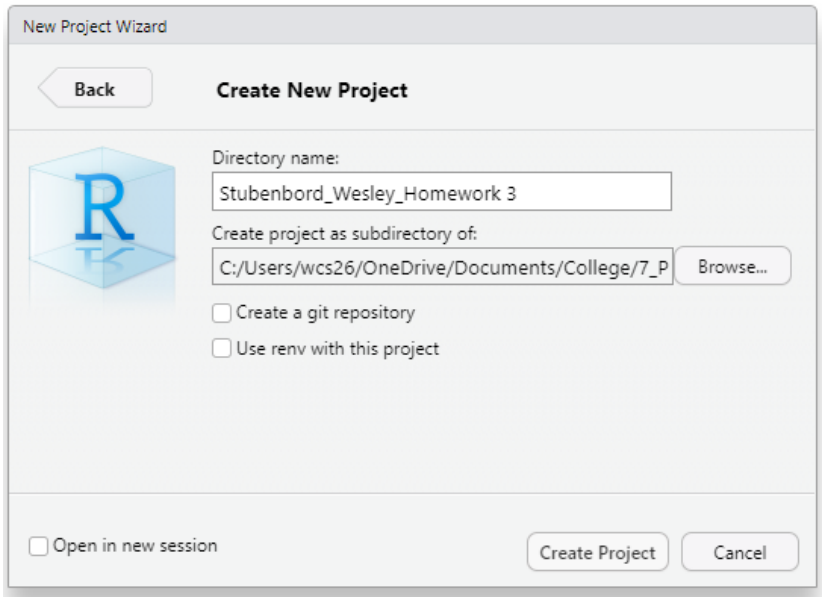
5.4 R Projects and here

Fortunately, there is a better way. To avoid these working directory problems and hard-coding absolute file paths in scripts, we're going to use two solutions that will help keep them straight.

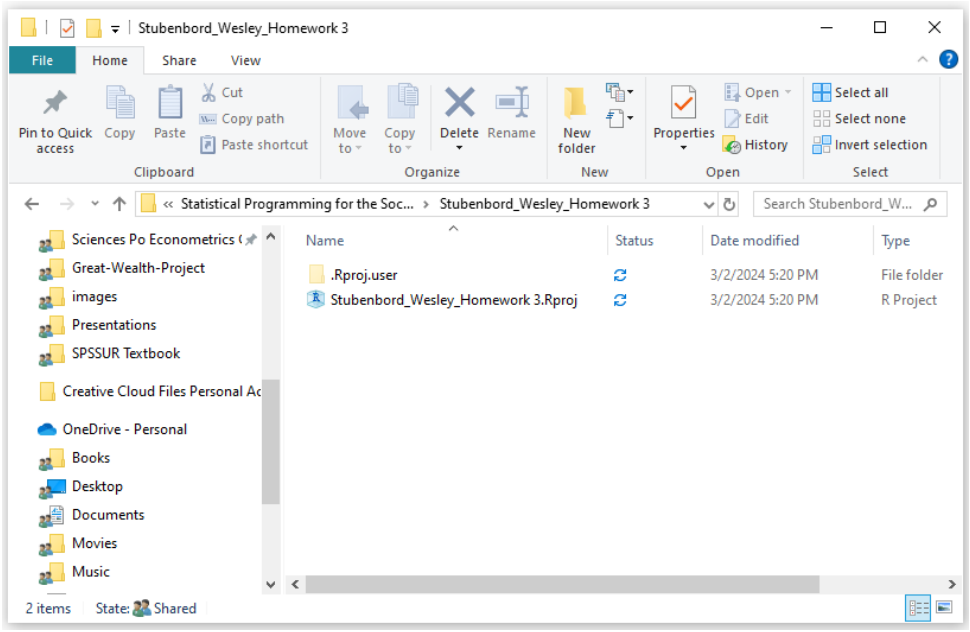
5.4.1 R Projects

The first is R Projects, a tool to organize your files in R Studio. When you create an R Project file (.Rproj), R Studio creates a new folder on your computer which is made to store all of the associated files you may have for a project (e.g., your data, your code, and any outputs). Every time you open that R Project file, R Studio opens a new working session with a working directory correctly set to the location of your project file. All of your files will be right where you need them.

Let's create one for today's classwork. In R Studio, use the navigation bar at the top of your screen to go to File > New Project > New Directory > New Project. Then, in the provided prompt, type in a project name that matches our course naming standards (e.g., "Stubenbord_Wesley_Session 5 Classwork"). Create this project as a sub-directory of your SPSSUR course folder (wherever this may be and whatever it may be called on your computer). Once this has been created, this is where your projects files will live, a permanent home just for them. You can verify that the new project folder has been created by navigating to it in your computer's file browser (e.g., Windows Explorer in Windows).

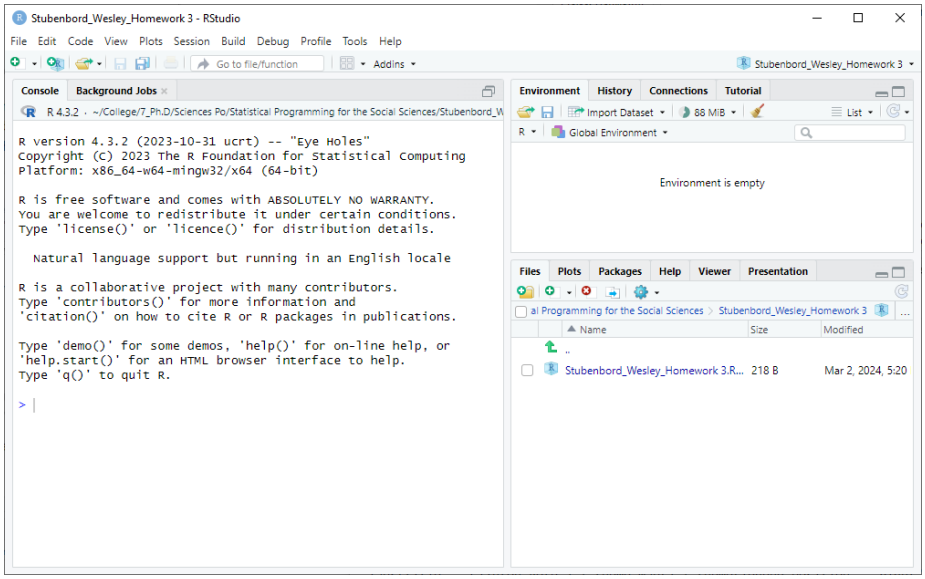


After you create a new project, you'll find yourself in a clean R Studio session with only a console window open. The folder in your computer system will contain just the new `.Rproj` file and a sub-folder with some saved settings. This `.Rproj` file is how you will access your project from now on. Instead of opening a .R script to access your code, you'll always want to open your `.Rproj` file first and then the .R script second. Don't put anything in this new folder other than files directly associated with the project you are working on. In this case, that means that this folder is only intended to store files associated with today's classwork.

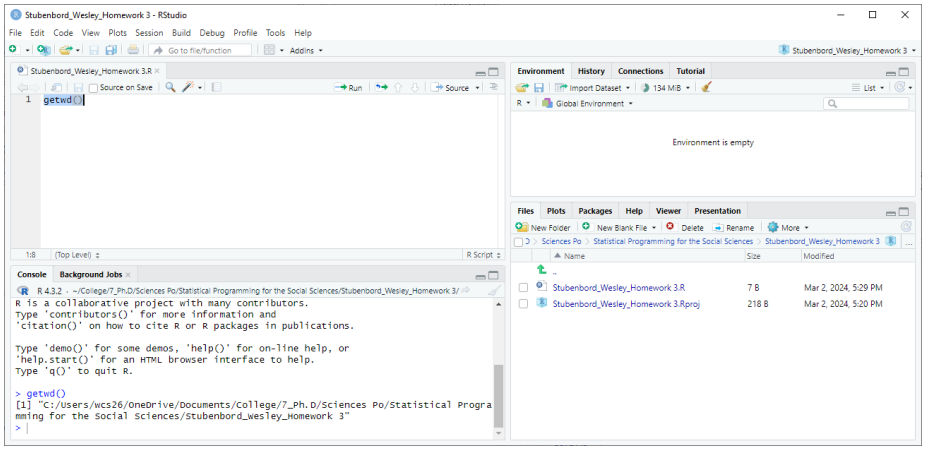


Back in R Studio, in the lower-right hand pane with the "Files" tab, you'll see all of the files currently associated with the project. At the moment, there shouldn't be anything apart from the `.Rproj` file itself).

In this same "Files" tab, you can add a new R Script file to this project by clicking on the "New Blank File" button > "R Script". Go ahead and create one and then save it with an appropriate file name.



Running `getwd()` from your new script file or directly in the console will show you that your working directory does indeed match your R Project location. Huzzah.



5.4.2 `here`

The second tool is a package called `here`. `here` contains a useful function, called `here()`, which will help ensure that your code is always oriented to the correct working directory: the location of the associated R Project file. There are some occasions where, despite our best efforts, R Projects won't save us from erroneous working directories. `here` helps cover those cases.

We'll come back to the application of this function in a moment, but for now, install `here`, load it, and test the `here()` function.

```
#install.packages('here')
library(here)
```

```
here() starts at C:/Users/wcs26/OneDrive/Documents/College/7_Ph.D/Sciences Po/Statistic
```

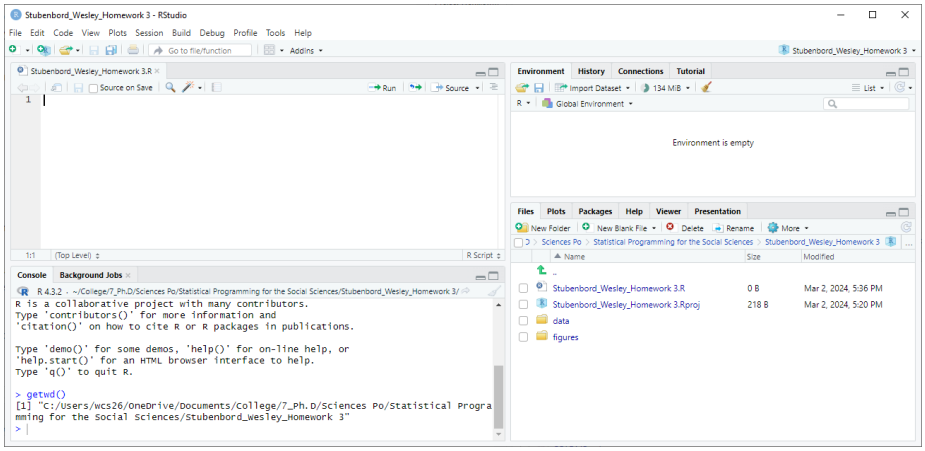
```
here()
```

```
[1] "C:/Users/wcs26/OneDrive/Documents/College/7_Ph.D/Sciences Po/Statistical Programmi
```

You should see, again, the correct working directory for your R Project.

5.4.3 Organizing Your Project

RProjects and `here`, while both extremely useful, also won't spare you entirely from the task of organizing your project files. A clean and organized work space will make your life significantly easier in the long-run. To facilitate this, I always recommend creating a sub-folder within your project directory to store data sets (called `data` for example), an additional sub-folder to store figure or graphs (called `figures`, for example), and another to store documentation (called `docs` for example). These sub-folder names are commonly used among programmers and also help to ensure consistency across projects with many collaborators. You can create these folders directly in R Studio using the "New Folder" button near the top of the 'Files' tab in the lower-right hand pane.



When finished, you will have your code in the main project folder along with the `.RProj` file and several affiliated sub-folders for storing things later.

5.5 Getting Data into R

Let us get back to the primordial problem now, which is getting data into R. In the social sciences, the data we use can come from a variety of different sources: we might take our data from long-running surveys, for instance, such as the General Social Survey, the European Social Survey, or the World Values Survey. We can also use administrative data produced by government agencies, like the Federal Bureau of Investigation's Uniform Crime Reports or data from New York City's Open Data initiative.

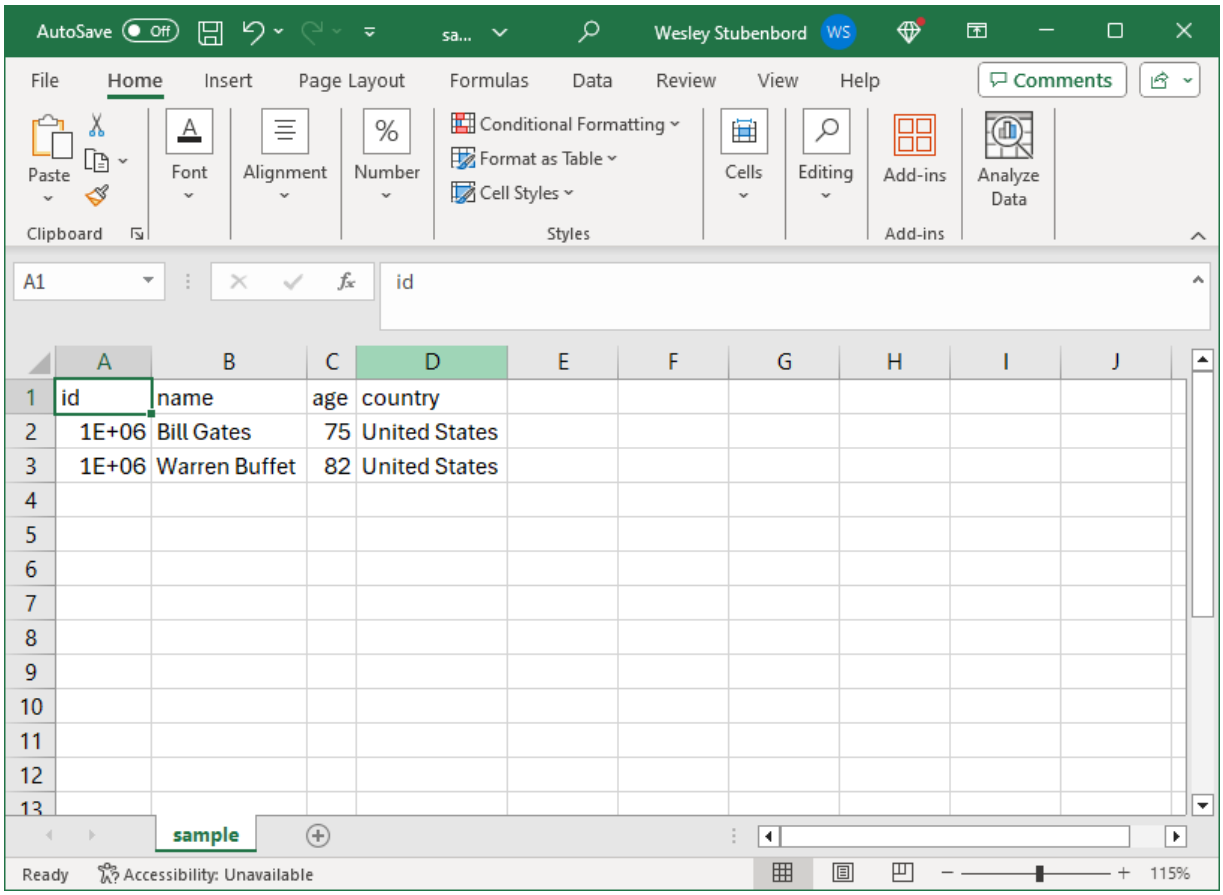
Alternatively, we can use data from other sources, which may not have been produced with research or administrative purposes in mind. For example, we can scrape data from social media to see how political sentiment changes in times of crises or we can use data from online dating apps to see how social norms around courtship have changed.

In any case, the most common format you are likely to find data stored in is a CSV file, which stands for comma separated values. CSVs are particularly appealing for data storage because they are lightweight and they simple. They don't contain any extra formatting - all they consist of is plain-text data in rows (separated by new lines) and columns (separated by commas). A CSV file could consist of the following, for example:

```
"id","name","age","country"
1011232,"Bill Gates",75,"United States"
1022234,"Warren Buffet",82,"United States"
```

When you open a CSV file in software like Microsoft Excel, it is usually automatically parsed into different columns, based on the position of the commas, and into different rows, based on the line breaks.[4] Excel will automatically recognize, for example, that name is a column containing two values, "Bill Gates" in one row and "Warren Buffett" in another. CSVs don't always use commas to separate their values - sometimes (especially in Europe), they use semi-colons. The character (e.g., `,` or `;`) that separates values in a data file is called a **delimiter**.

[4] **Parsing** refers to how software reads the values.



5.6 Loading files from CSVs

Thankfully, loading CSV files is not very difficult in `R`. `R` has a base function for loading CSVs, `read.csv()`, but there is a better version called `read_csv()` which comes from the `readr` package. `readr` is also located in the `tidyverse`, but must also be loaded separately, just like the `scales` package in the previous chapter.[5]

On the course Moodle site, you'll find a CSV titled `billionaires_2020-2023.csv`, which contains some limited data on the world's billionaires from my own research. Go ahead and download this file and then save it in the `data` sub-folder you created in your `Session 5 Classwork` project folder. Open the `Session 5 Classwork` project and a script file inside of it, if you haven't already. We'll load a few libraries first and then the data set.

[5] As before, rather than load the entire `readr` package via `library()`, you can also use the `::` syntax, as in `readr::read_csv()`.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.0     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to
```

```
library(readr)
library(here)
```

As with any function that reads data, `read_csv()` requires a file path to locate the data nad read it into our `R` session. Because the data set is located in a sub-folder and because we want to use a relative path (rather than an absolute path), we'll use `here()`.

Remember, `here()` provides the file path to your R Project folder. Any additional folder or file names used as arguments inside of the `here()` function (separated by a comma) will be added to the project folder path. If your R Project folder is located at `C:\Users\Documents\My Projects`, for example, `here("data")` will output `C:\Users\Documents\My Projects\data`. You must enclose the name of your sub-folders in quotation marks.

Entering the following should return the file path of the data set if you have saved it within your `data` subfolder:

```
here("data","billionaires_2020-2023.csv")
```

```
[1] "C:/Users/wcs26/OneDrive/Documents/College/7_Ph.D/Sciences Po/Statistical Programmi
```

To read the actual data into `R`, now all we need to do is put this `here` function inside of `read_csv()` in the `file =` argument.

```
read_csv(file = here("data","billionaires_2020-2023.csv"))
```

```
Rows: 2640 Columns: 13
-- Column specification --------------------------------------------------------
```

```
Delimiter: ","
chr (8): name, gender, country, countrycode, region, marital, residence_coun...
dbl (5): id, 2020, 2021, 2022, 2023

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# A tibble: 2,640 x 13
      id name        gender country countrycode region marital residence_country
   <dbl> <chr>       <chr>  <chr>   <chr>       <chr>  <chr>   <chr>
 1     1 A. Jayson ~ male   United~ USA         North~ Married United States
 2     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 3    10 Abdulla bi~ male   United~ ARE         Middl~ <NA>    United Arab Emir~
 4    12 Abdulsamad~ male   Nigeria NGA         Sub-S~ Married Nigeria
 5    13 Abhay Firo~ male   India   IND         South~ Married India
 6    14 Abhay Soi   male   India   IND         South~ <NA>    <NA>
 7    16 Abigail Be~ female United~ USA         North~ <NA>    <NA>
 8    17 Abigail Jo~ female United~ USA         North~ Married United States
 9    18 Abilio dos~ male   Brazil  BRA         Latin~ Married Brazil
10    20 Acharya Ba~ male   India   IND         South~ Single  India
# i 2,630 more rows
# i 5 more variables: selfmade <chr>, `2020` <dbl>, `2021` <dbl>, `2022` <dbl>,
#   `2023` <dbl>
```

As you can see from the output above, `read_csv()` worked! If our data had used a semi-colon as a delimiter instead of a comma, we would have just used `read_csv2()` instead.

We can see from the resulting output that this data set has 2,640 rows and 13 columns. Let's save this into a new object in our environment.

```
billionaires <- read_csv(file = here("data","billionaires_2020-2023.csv"))
```

```
Rows: 2640 Columns: 13
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (8): name, gender, country, countrycode, region, marital, residence_coun...
dbl (5): id, 2020, 2021, 2022, 2023

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You'll notice that `read_csv()` says a lot each time you run it. Don't confuse the red text you see here with errors, however. It's just trying to be helpful by giving you some more information about the data we're reading in. We can see

from this output, for example, that there are 8 character columns and 5 `dbl` columns. `dbl` stands for `double` and is a type of numeric value. We can, of course, use `glimpse()` to see the variables again along with some of the first few values.

```
glimpse(billionaires)
```

```
Rows: 2,640
Columns: 13
$ id                <dbl> 1, 9, 10, 12, 13, 14, 16, 17, 18, 20, 25, 26, 27, 29~
$ name              <chr> "A. Jayson Adair", "Abdulla Al Futtaim", "Abdulla bi~
$ gender            <chr> "male", "male", "male", "male", "male", "male", "fem~
$ country           <chr> "United States", "United Arab Emirates", "United Ara~
$ countrycode       <chr> "USA", "ARE", "ARE", "NGA", "IND", "IND", "USA", "US~
$ region            <chr> "North America", "Middle East & North Africa", "Midd~
$ marital           <chr> "Married", "Married", NA, "Married", "Married", NA, ~
$ residence_country <chr> "United States", "United Arab Emirates", "United Ara~
$ selfmade          <chr> "self-made", "self-made", "inherited", "self-made", ~
$ `2020`            <dbl> NA, 2.437, 4.293, 3.365, 1.740, NA, NA, 12.531, 2.66~
$ `2021`            <dbl> 1.110, 2.441, 3.107, 5.437, 2.663, NA, NA, 23.189, 2~
$ `2022`            <dbl> 1.140, 2.591, 2.695, 7.151, 2.902, NA, NA, 21.971, 2~
$ `2023`            <dbl> 1.3, 2.4, 3.0, 8.2, 2.7, 1.2, 1.1, 21.6, 2.4, 3.4, 2~
```

There are other ways you can take a peek at your data. We can take a random slice of the data using `slice_sample()` from `dplyr`, for example. We can also use `slice_head()` to see the first few rows, `slice_tail()` to see the last few rows, or `slice_min()` and `slice_max()` to see the rows with the largest or smallest values for some variable. Give them each a try and look at their documentation to see some helpful optional arguments you can use as well, like `n =`.

```
billionaires %>%
  slice_sample(n = 5)
```

```
# A tibble: 5 x 13
     id name         gender country countrycode region marital residence_country
  <dbl> <chr>        <chr>  <chr>   <chr>       <chr>  <chr>   <chr>
1  4589 Xie Juhua    female China   CHN         East ~ Widowed China
2  3931 Sol Daurella female Spain   ESP         Europ~ Married Spain
3  2472 Leonard Ste~ male   United~ USA         North~ Married United States
4   153 Alfredo Har~ male   Mexico  MEX         Latin~ Married Mexico
5  3784 Scott Cook   male   United~ USA         North~ Married United States
# i 5 more variables: selfmade <chr>, `2020` <dbl>, `2021` <dbl>, `2022` <dbl>,
#   `2023` <dbl>
```

No matter how we choose to do it, though, our purpose at this point of reading data in should always be to get a sense for any potential issues lurking beneath the surface of our data file.

You may have noticed from `glimpse` that there is something a little bit weird about this data. Four columns at the end have numbers as titles. What are these columns? Since this data doesn't come from a package and there's no documentation, I will tell you: they're the estimated net worth of each of the listed individuals in billions of 2023-inflation adjusted dollars for each of the named years (i.e., 2020, 2021, 2022, 2023). The values are in billions of US dollars.[6]

Now that we know what the data is, we can start to make sense of it. We can use `slice_max()` , for instance, to see who was the richest person in 2023. In this case, note that when we reference a variable which begins with a number, we have to use the ` character to enclose the variable name. This is because `R` does not allow object names to start with a number.

```
billionaires %>%
  slice_max(`2023`, n = 1)
```

```
# A tibble: 1 x 13
     id name         gender country countrycode region marital residence_country
  <dbl> <chr>        <chr>  <chr>   <chr>       <chr>  <chr>   <chr>
1   425 Bernard Arn~ male   France  FRA         Europ~ Married France
# i 5 more variables: selfmade <chr>, `2020` <dbl>, `2021` <dbl>, `2022` <dbl>,
#   `2023` <dbl>
```

Now we can see that Bernard Arnault was the richest person in 2023 with an estimated net worth of $211 billion. What if we want to see what the cumulative net worth of all French billionaires was for the years covered in this data? Maybe we could use `dplyr` to summarize:

```
billionaires %>%
  group_by(country) %>%
  filter(country == "France") %>%
  summarize(total_nw = sum())
```

```
# A tibble: 1 x 2
  country total_nw
  <chr>      <int>
1 France         0
```

[6]The net worth estimates here are derived from the Forbes' annual *World's Billionaires* list and inflation-adjusted using an implict GDP price deflator from the U.S. Federal Reserve.

Here we run into a problem. We can't `group_by(country, year)`, because there is no variable called `year`. There's also no `net_worth` variable that we can put into our `summarize()` function. So, we're stuck with bad options. We could try something like this, for example:

```
billionaires %>%
  group_by(country) %>%
  filter(country == "France") %>%
  summarize(nw_2023 = sum(`2023`),
            nw_2022 = sum(`2022`),
            nw_2021 = sum(`2021`),
            nw_2020 = sum(`2020`))
```

```
# A tibble: 1 x 5
  country nw_2023 nw_2022 nw_2021 nw_2020
  <chr>     <dbl>   <dbl>   <dbl>   <dbl>
1 France     585.      NA      NA      NA
```

That got us the total net worth of French billionaires in 2023, at least, but it didn't calculate a result for the other years. The reason why we're having a hard time here is that our data is not in the right format for `tidyverse` functions.

5.7 Tidy Data

In the previous chapter, we learned that the `gapminder` data was in just the right format for `ggplot`. We called this **long** format data, which is usally how we describe this format in the social sciences. In a long format, rows are repeated observations for some unit of analysis (like a country) across some dimension (like time). In the case of the `gapminder` data, each of these observations (country-years) had a value for a set of variables of interest (e.g., life expectancy and GDP per capita) as shown below.

country	year	lifeExp	gdpPercap
France	1952	67.41	7029.809
France	1957	68.93	8662.835
France	1962	70.51	10560.486
France	1967	71.55	12999.918
France	1972	72.38	16107.192

In the `billionaires` data, we instead have one row per unit of analysis (billionaires) and multiple columns for a single variable of interest (net worth).

id	name	2020	2021	2022	2023
1	A. Jayson Adair	NA	1.110	1.140	1.3
9	Abdulla Al Futtaim	2.437	2.441	2.591	2.4
10	Abdulla bin Ahmad Al Ghurair	4.293	3.107	2.695	3.0

The column titles (e.g., 2020, 2021), to be clear, are not actually different variables, they are simply values of a single variable (e.g., year). When data is split in this way, we say it is in a **wide** format. Wide data does have it's uses. It is very convenient, for instance, for storing a large amount of data in a small amount of space - a particular concern when you are printing data sets. Long data, on the other hand, is much better for data analysis.

The `tidyverse` functions require data to be in what Hadley Wickham and collaborators call a **tidy** format. In tidy data, each variable is in a column, each observation is in a row, and each cell contains a single value (Wickham et al. 2019). It is, in other words, in a consistent *long*-format. If we want to be able to use all of the great features of `dplyr` and `ggplot`, we need to transform our data into a tidy format.

5.8 Pivoting from Wide to Long

Fortunately, `dplyr` has some handy functions for this. Since we have data in a wide format and wish to change it to a long format, we'll need to use a function called `pivot_longer()`. There are three arguments we need to provide to `pivot_longer()`.

First, in the `cols =` argument, we need to provide the columns we are trying to combine into a single variable. In our case, our net worth values are distributed across the `` `2020` ``, `` `2021` ``, `` `2022` ``, and `` `2023` `` columns, so we'll put those in a vector and use them for this argument. Next, in the `names_to =` argument, we need to identify where we want to put the names for each of those values (in other words, the titles for each of our former columns). Since each of those column names corresponds to a year, we'll tell it to put them in a `"year"` column. Last, we need to specify a name for the variable holding all of the values that were stored across those columns. Since the values were net worth, we'll call this new variable `net_worth`. We might also want to drop the rows which contain NA values for net worth and so we'll add an optional fourth argument, `values_drop_na =`.

```
billionaires %>%
  pivot_longer(cols = c(`2020`, `2021`, `2022`, `2023`),
               names_to = "year",
               values_to = "net_worth",
```

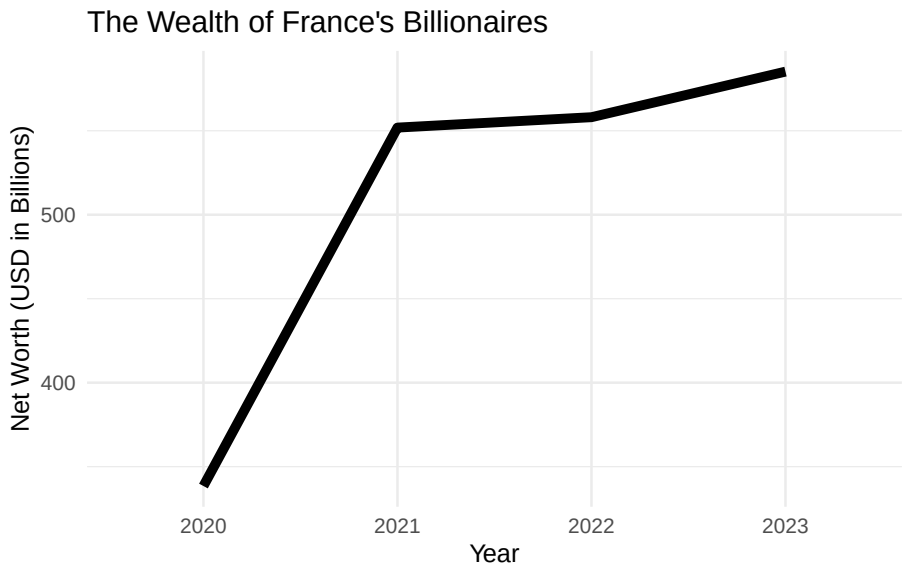
```
                values_drop_na = TRUE)
```

```
# A tibble: 9,142 x 11
      id name        gender country countrycode region marital residence_country
   <dbl> <chr>       <chr>  <chr>   <chr>       <chr>  <chr>   <chr>
 1     1 A. Jayson ~ male   United~ USA         North~ Married United States
 2     1 A. Jayson ~ male   United~ USA         North~ Married United States
 3     1 A. Jayson ~ male   United~ USA         North~ Married United States
 4     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 5     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 6     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 7     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 8    10 Abdulla bi~ male   United~ ARE         Middl~ <NA>    United Arab Emir~
 9    10 Abdulla bi~ male   United~ ARE         Middl~ <NA>    United Arab Emir~
10    10 Abdulla bi~ male   United~ ARE         Middl~ <NA>    United Arab Emir~
# i 9,132 more rows
# i 3 more variables: selfmade <chr>, year <chr>, net_worth <dbl>
```

If we take a look at the data now, we can see that we've increased our number of rows substantially (to 9,142) and each net worth value is now in a separate row according to the corresponding year and individual. It looks tidy. Now we can do much of the same type of data analysis and visualization we have been doing over the past couple of chapters.

```
# A quick and dirty plot.
tidy_bil <- billionaires %>%
  pivot_longer(cols = c(`2020`, `2021`, `2022`, `2023`),
               names_to = "year",
               values_to = "net_worth",
               values_drop_na = TRUE)

tidy_bil %>%
  group_by(country, year) %>%
  filter(country == "France") %>%
  summarize(agg_nw = sum(net_worth)) %>%
  ggplot(mapping = aes(x = year, y = agg_nw, group = country)) +
  geom_line(linewidth = 2) +
  labs(title = "The Wealth of France's Billionaires",
       x = "Year",
       y = "Net Worth (USD in Billions)") +
  theme_minimal()
```

```
`summarise()` has grouped output by 'country'. You can override using the
`.groups` argument.
```



5.9 Merging Data

Let's say that we want to do some further analysis involving additional data not contained in the data set we are currently using. Can we add it to our existing data set? The answer is yes.

On the course Moodle site, you'll find another data set called `age.xlsx`. This is an Excel file. Fortunately, the `readxl` package (also contained in the `tidverse` and requiring separate loading) has just the function. Download the file from the Moodle page, add it to your project's data folder, and then use the command below:

```
library(readxl)

bil_age <- read_xlsx(path = here("data", "age.xlsx"), sheet = "Sheet1")

glimpse(bil_age)
```

```
Rows: 2,724
Columns: 3
$ id   <dbl> 1, 9, 12, 13, 14, 16, 17, 18, 20, 25, 26, 27, 29, 32, 33, 34, 36,~
$ year <dbl> 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023, 2023,~
$ age  <dbl> 53, 83, 62, 78, 49, 42, 61, 86, 50, 40, 44, 81, 39, 54, 52, 39, 6~
```

As we can see from the output, there isn't too much in here, just an ID, a year, and an age. If we want to use this data in our next analysis, we now need to merge it with our previously tidied data. We might just want to check the `year` column to see how many years this age data covers.

```
  bil_age %>%
    distinct(year)
```

```
# A tibble: 2 x 1
   year
  <dbl>
1  2023
2    NA
```

```
  #An alternative way:
  #unique(bil_age$year)
```

There's only one year, unfortunately, and quite a few missing values. Before we go ahead and merge, we should also check to make sure that there is only one age value for each ID. Merging data with multiple values for each unit of analysis will cause rather large problems. Keep a careful eye on your data as you do these types of things. The code below will count the number of rows:

```
  bil_age %>%
    count(id) %>%
    filter(n > 1)
```

```
# A tibble: 59 x 2
      id     n
   <dbl> <int>
 1     1     4
 2     9     2
 3    12     5
 4    13     5
 5    14     2
 6    15     2
 7    16     2
 8    17     5
 9    18     5
10    19     2
# i 49 more rows
```

It looks like there are a few duplicates in here. We can investigate further, but

let's see if dropping the NAs fixes our problem. They won't be useful in our analysis later anyways.

```
bil_age %>%
  drop_na(age) %>%
  count(id) %>%
  filter(n > 1)
```

```
# A tibble: 0 x 2
# i 2 variables: id <dbl>, n <int>
```

Good, no results, which means that dropping the NAs solved our problem with duplicates. Let's save the tibble without the missing values and carry on with our merge.

```
bil_age %>%
  drop_na(age) -> bil_age
```

Note that we've use our assignment operator in a somewhat unorthodox way here. Instead of using it at the beginning of the piped function, we've added it to the end. This works and is also acceptable.

Now, let's merge. Here, we'll use a function from `dplyr` called `left_join()`. There are other types of `_join()` functions depending on the use case. In our case, we have an existing data set and simply want to add data from another tibble to it. We don't care much about what happens to the rows in the `bil_age` data set that don't match up. We'll use `left_join()` here as a result. Other types of joins include a `right_join()`, an `inner_join()`, a `left_join()` and a `full_join()`. Take a look at the supporting documentation to learn more about them.

```
left_join(x = tidy_bil,
          y = bil_age,
          join_by(id, year))
```

This didn't quite work and, if we look at our error, we can see why. The `year` column in our `tidy_bil` data is a character and the the `year` column in our age data is a double. We'll have to convert the column in `tidy_bil` to continue. We should probably convert `year` to a date format, but we're going to cheat here and just convert it to a numeric variable for convenience. Hopefully this won't come back to bite us later.

```
tidy_bil %>%
  mutate(year = as.numeric(year)) -> tidy_bil
```

Let's try the merge again:

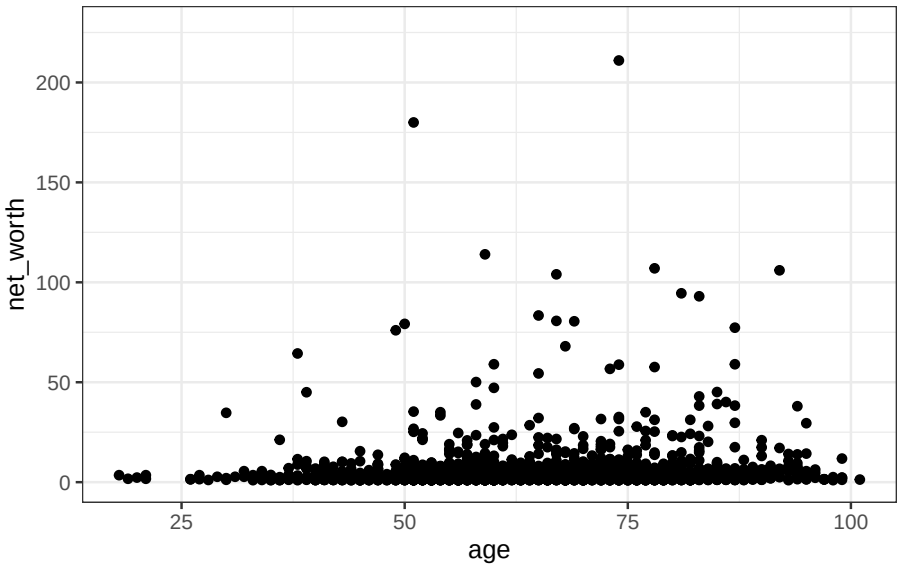
```
tidy_bil <- left_join(x = tidy_bil,
                      y = bil_age,
                      join_by(id, year))
# An alternative way to do this is:
#tidy_bil <- tidy_bil %>%
#  left_join(bil_age,
#            join_by(id, year))
```

Success! Whenever you do these sorts of things, try running the code without re-assigning it back to the original object first and then assign it after you are sure it works. Otherwise, you may start running into unexpected errors as you change your data. You can always re-load it if you've made a mistake somewhere.

If we take a look at `tidy_bil`, we can now see that we have an age column. Now we can do something like this:

```
p <- ggplot(data = tidy_bil,
            mapping = aes(x = age,
                          y = net_worth))

p + geom_point() + theme_bw()
```

```
Warning: Removed 6569 rows containing missing values or values outside the scale range
(`geom_point()`).
```




```
# An alternative way to do this is:
#tidy_bil %>%
#  ggplot(mapping = aes(x = age,
#                       y = net_worth)) +
#  geom_point() + theme_bw()
```

Notice here that we are also using `ggplot()` in a different way from the previous chapter. In this case, we first save our base `ggplot()` layer to an object (arbitrarily named `p`) and then we add layers to the object in a separate line of code. You'll see this method used often in references elsewhere. Either style of writing your `ggplot()` code is fine, but I tend to prefer the piped form from the previous chapter for own use, since it allows you to add filters and do other data manipulations in the same piped command generating the plot. This is a stylistic and workflow preference - you will find many forking paths that will produce the same result as you continue to learn `R`.

A more general understanding to take away here, however, is the fact that since we aren't using a piped function, we do need to specify a `data =` argument in the `ggplot()` function. The use of the pipe operator ordinarily absolves us of having to specify a data argument in all `dplyr` functions, because the pipe operator does it for us (i.e., "take this *AND THEN*...").

For example, to use a filter on a tibble you could simply enter `filter(data = tidy_bil, year == 2020)` as a stand-alone function and it would give you the same answer as `tidy_bil %>% filter(year == 2020)`. The pipe operator, however, allows us to string together multiple of these such functions, which makes it efficient for complex data analysis.

5.10 Pivoting from Long to Wide

There is one more thing we should learn here: recoding. Recoding is useful when the categories within a data set aren't quite appropriate for how we want to analyze them.

Let's say we want to compare the net worth of billionaires in our data set by marital status, for example. For this, we could use the wide data. Let's imagine that we don't have our wide data in the first place though. Could we get our long data into a wide format? Yes, using `pivot_wider()`:

```
tidy_bil %>%
  pivot_wider(
    names_from = year,
    values_from = net_worth
  )
```

```
# A tibble: 5,040 x 14
      id name        gender country countrycode region marital residence_country
   <dbl> <chr>       <chr>  <chr>   <chr>       <chr>  <chr>   <chr>
 1     1 A. Jayson ~ male   United~ USA         North~ Married United States
 2     1 A. Jayson ~ male   United~ USA         North~ Married United States
 3     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 4     9 Abdulla Al~ male   United~ ARE         Middl~ Married United Arab Emir~
 5    10 Abdulla bi~ male   United~ ARE         Middl~ <NA>    United Arab Emir~
 6    12 Abdulsamad~ male   Nigeria NGA         Sub-S~ Married Nigeria
 7    12 Abdulsamad~ male   Nigeria NGA         Sub-S~ Married Nigeria
 8    13 Abhay Firo~ male   India   IND         South~ Married India
 9    13 Abhay Firo~ male   India   IND         South~ Married India
10    14 Abhay Soi   male   India   IND         South~ <NA>    <NA>
# i 5,030 more rows
# i 6 more variables: selfmade <chr>, age <dbl>, `2021` <dbl>, `2022` <dbl>,
#   `2023` <dbl>, `2020` <dbl>
```

We'll just use the original data anyways though. To see what values are actually in our `marital` variable, we'll use this:

```
billionaires %>%
  distinct(marital)
```

```
# A tibble: 9 x 1
  marital
  <chr>
1 Married
```

```
2 <NA>
3 Single
4 Widowed
5 Separated
6 Divorced
7 Widowed, Remarried
8 In Relationship
9 Engaged
```

There are 9 distinct categories. Nine is perhaps too many. What if we decide we want to use just three categories: "Married/Widowed", "Single", and "Other" instead?

5.11 case_match() for Recoding

To recode our data according to the new groupings, we can use the `case_match()` function inside of a `dplyr mutate()` function. Essentially, we are modifying a column so that the values of the new column take on the recoded values of the old column. We could mutate the original variable directly (in this case, `marital`), but it's generally best practice to put our recoded values inside of their own, new column to make sure we don't make an error. Otherwise, we'd have to reload our data and we'd need to re-run all of the other code in our analysis. We'll call this recoded variable `marital_rc`.

The first argument in `case_match()` is the column that needs to be recoded. The following arguments, which contain the recoding scheme, follow the format `old_values ~ new_values`. The original values go on the left-hand side and the values to be recoded go on the right. To separate the left and right-hand sides, we use a `~` character and then a comma to separate each set of old and new values. Since we are re-coding multiple old values at a time, we'll put those sets of old values inside of vectors. Remember, test this without re-assigning the result to your original object to make sure that you've done it correctly and then re-assign it once you are confident you've done it right.

```
billionaires %>%
  mutate(
    marital_rc = case_match(marital,
                            c('Married', 'Widowed, Remarried', 'Widowed') ~ 'Married/Widowed',
                            c('Single') ~ 'Single',
                            c('Engaged', 'In Relationship', 'Divorced', 'Separated') ~ 'Other'
```

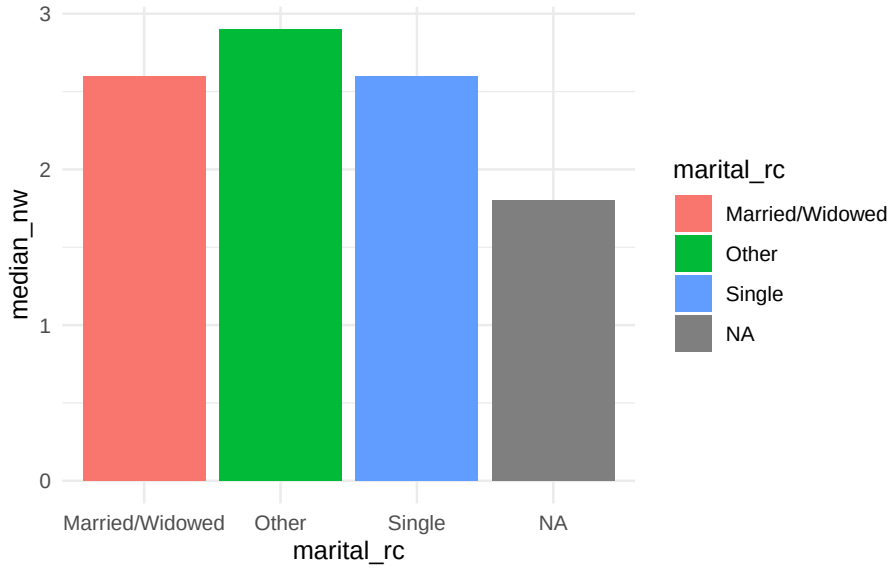
Did it work correctly?

```
billionaires %>%
  distinct(marital_rc)
```

```
# A tibble: 4 x 1
  marital_rc
  <chr>
1 Married/Widowed
2 <NA>
3 Single
4 Other
```

Yes, we've recoded successfully. There seem to be some missing values in our data, but we don't necessarily want to drop them. Let's make a quick graph showing median net worth with our recoded marital status variable for 2023.

```
billionaires %>%
  group_by(marital_rc) %>%
  summarize(median_nw = median(`2023`)) %>%
  ggplot(mapping = aes(x = marital_rc,
                       y = median_nw,
                       fill = marital_rc)) +
  geom_col() + theme_minimal()
```



Note, of course, that we can't make any claims about associations here. There

are a lot of missing values (`NA`) and we don't know whether they would be biased towards a particular category. Our `Other` category is also quite expansive and maybe not analytically appropriate.

Last, but not least, we don't know whether any of the visual differences we are seeing in median net worth across category are due to random chance or due to some relationship between the variables. We'll return to this latter point in the next chapter when we begin talking about inferential methods.

5.12 Exercises

For the remainder of class or for homework, keep playing around with this data set to practice the new functions you've learned and the `dplyr` functions you've learned in previous chapters.

Part I

Assignments

Homework 1

Due Date: Tuesday, 13 February by 23:59:59

Submission Instructions: Submit your completed R script file to Moodle.

This homework will be relatively short and straight-forward. The goal is to ease you into `R` now so that you are ready to complete some of the more complex data analysis that will take place later.

Question #1:

Create an `R` script and save it with an appropriate name. Add a header to your R script file in the format below.

```
# Name: [first_name] [last_name]
# Date: [date]
# Description: [brief description of the file]

# Question 2:
```

Question #2:

In your `R` script file, load the `tidyverse` package. Show the code used.

Question #3:

Create a vector with the following set of numbers: $30, 60, 90, 120, 150$. Perform the following operations, showing the code used for each.

Part A: Multiply the vector by 2. In a brief comment, tell me what the result was.

Part B: Take the vector and divide it by 3. Tell me what the result was in a brief comment.

Part C: Multiply the vector by itself. Tell me what the result was in a brief comment.

Part D: Return the third element of the vector.

Part E: Replace the second element of the vector with a missing value (`NA`).

Part F: Sum the vector, excluding the missing value. In a comment, write the answer.

Question #4:

Using the `socviz` package (see section 2.3 of the course textbook), load the `election` dataset into a new object called `elec`. Complete the following tasks, showing the code used for each.

Part A: Find the total popular vote received by Gary Johnson using the `johnson_vote` variable.

Part B: Find the total popular vote received by 'Other' candidates using the `other_vote` variable.

Part C: In a comment answer the following question: who received more votes, Gary Johnson or "other" candidates? By how much?

Part D: Use the `sum()` function on the `state` variable. In a brief comment, explain why this didn't work and what the error message is telling you.

Homework 2

Due Date: Wednesday, 28 February by 23:59:59

Submission Instructions: Submit your completed R script to the corresponding Moodle assignment. Your file should contain a header and the file name should be formatted according to the guidelines discussed in class and posted in the course slides.

Be sure to show your work. This means that your answer to each question should show the code you used to obtain the answer. You do not need to show other steps that may have been taken to get the answer (e.g., trial and error), only the code that provides the answer.

Question #1:

Part A:

Using the `socviz` package, load the GSS data into an object called `gss`. Use the `table()` function and subset operator to find the number of respondents by `agegrp`.

In a comment, identify the number of respondents between the ages of 35-45.

Part B:

Next, use the `dplyr` functions to output a tibble which summarizes the number of respondents by `agegrp`.

In a comment, identify the number of respondents between the ages of 45-55.

Part C:

Output another tibble which summarizes respondents by `agegrp` and whether they voted for `obama` in the 2012 U.S. presidential election. In this tibble, include both the count (call it `total`), relative frequency (`freq`), and percentage (`pct`). You do not have to round the percentages but can if you wish.

In a comment, identify the number of respondents between the ages of 18-35 who voted for Obama. In a separate comment, identify the number of respondents between the ages of 18-35 that have missing values for the `obama` variable.

Question #2:

Again using the GSS data from the `socviz` package, create a tibble called `marit_happy` which summarizes the happiness of GSS respondents by marital status. Filter your data so that it shows only respondents who are "Married" or "Never Married".

In a comment consisting of a few brief sentences, compare the reported happiness of respondents who are married to those who have never been married.[7]

Question #3:

Also using the same GSS data, output a tibble with: (1) the number of respondents by `degree` (i.e., the respondent's highest degree level) and (2) the mean number of children by `degree`. Hint: you may need to use an `na.rm` argument somewhere.

In a brief comment, identify the relationship between degree-level and number of children among respondents.

[7] *A brief technical note*: although the GSS is a nationally representative survey of U.S. adults, we are using the term "respondents" throughout this assignment rather than "U.S. adults."

The reason for this is that estimating population-level statistics (like the proportion of U.S. adults who are married) requires using *survey weights*, which is a slightly more complicated procedure that takes into account the survey design. Survey weights are used to help ensure that the statistics being reported from survey data accurately reflect the population.

In practice, the numbers you will obtain in this assignment are very close to the best estimates possible for U.S. adults, but since they're not exactly precise (i.e., they don't take into account the survey weights), it's more accurate to refer to your results as relating to the respondents of the GSS rather than all U.S. adults. The differences between the properly weighted results and the results you obtain in this assignment are within 1%, however.

Homework 3

Due Date: Wednesday, March 13 by 23:59:59

Instructions: For this homework, you will need to submit a zipped R Project folder containing an R Script or Quarto document with your code.[8] Your R Project folder should follow the conventions discussed in class. Your R Script (or Quarto file) should contain both an appropriately formatted header and file name.

Your code must also compile. This means that any R user should be able to use your R Project to replicate the answers you get in this assignment without receiving a terminal error while running it. Packages used in this assignment should be loaded at the beginning of the code after the header. You can assume that this R user has the necessary packages installed.

Data files needed for this assignment can be found on the course Moodle page.

Question 1

For this question, use the "billionaires_2020-2023.csv" file introduced in class, performing any necessary transformations to answer the questions.

Part A:

For each country in the data set, identify the number of billionaires and the mean, median, standard deviation, minimum and maximum of net worth for 2023. Sort the resulting tibble by descending order in terms of number of billionaires. Output the results to the console.

Part B:

Find the top 5 individuals by net worth for each year. Output the resulting tibble to the console.

Part C:

Identify the individual(s) who appeared among the annual top 5 richest individuals most frequently. Output the results to the console.

[8] If you are using Quarto, whenever the instructions say to output the results to console, your results should output within the document itself after running the code chunk.

Question 2

For this question, use the "billionaires_2020-2023.csv" and "age.xlsx" files used in class, again, performing any necessary transformations to answer the questions.

Part A:

Create a tibble which shows the median age of billionaires for the United States, China, France, Germany, and Italy in 2023. In a brief comment, identify the country with the oldest median age.

Part B:

Plot the results of *Part A* in a bar chart.

Question #3

For this question, use the file wdi.csv, which contains an extract from the World Bank Development Indicators, and eu.csv, which contains a list of countries in the European Union.

Part A:

Merge the European Union data with the World Bank data.

Part B:

Create a re-coded `EU` variable such that countries in the European Union are labeled as "EU" and countries not in the European Union are labeled as "Non-EU" (Hint: check the help page for `case_match()` for an example on how to code missing values).

Part C:

Filter for countries in the "Europe & Central Asia" region. Now, create a scatterplot showing the relationship between GDP per capita and life expectancy. In your scatter plot, make a visual distinction between E.U. and non-E.U. countries.

Part D:

In 2-3 sentences, briefly compare the relationship between GDP and life expectancy across E.U. and non-E.U. countries for the data used in Part C.

References

Healy, Kieran. 2019. *Data Visualization: A Practical Introduction*. Princeton: Princeton University Press. socviz.co.

Part II

Appendix