R Training

Contents

Chapter 1

Welcome

Insert something overarching about the course here..

Part I

Pre-Training Material

Chapter 2

Setting up your computer

It's time to set up your system. Ready? Let's go.

Install R

First, let's get the right programs installed on your computer. Then we will explain what they are and why you need them.

First, download and install `R`:

Go to the following website, click the *Download* button, and follow the website's instructions from there. https://mirrors.nics.utk.edu/cran/

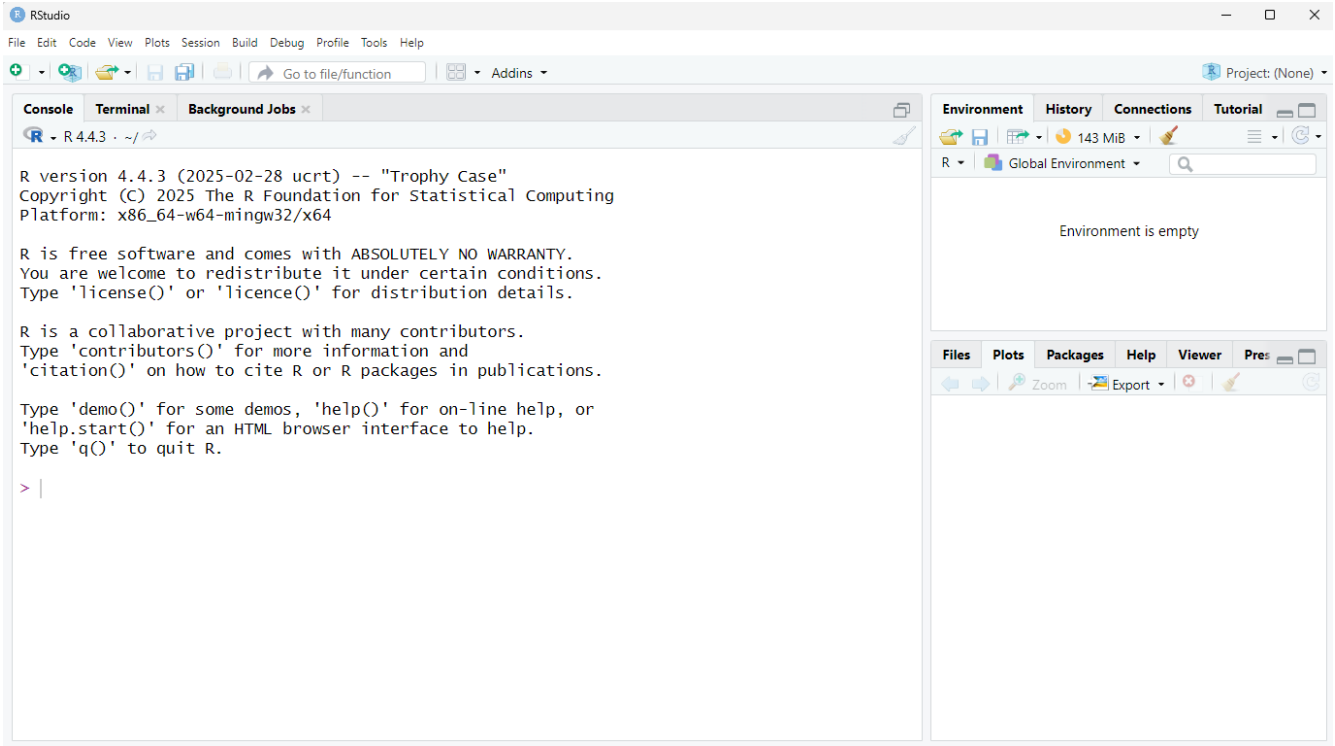
Install RStudio

Second, download and install `RStudio`:

Go to the following website and choose the free Desktop version: https://rstudio.com/products/rstudio/download/

Third, make sure `RStudio` opens successfully:

Open the `RStudio` app. A window should appear that looks like this:

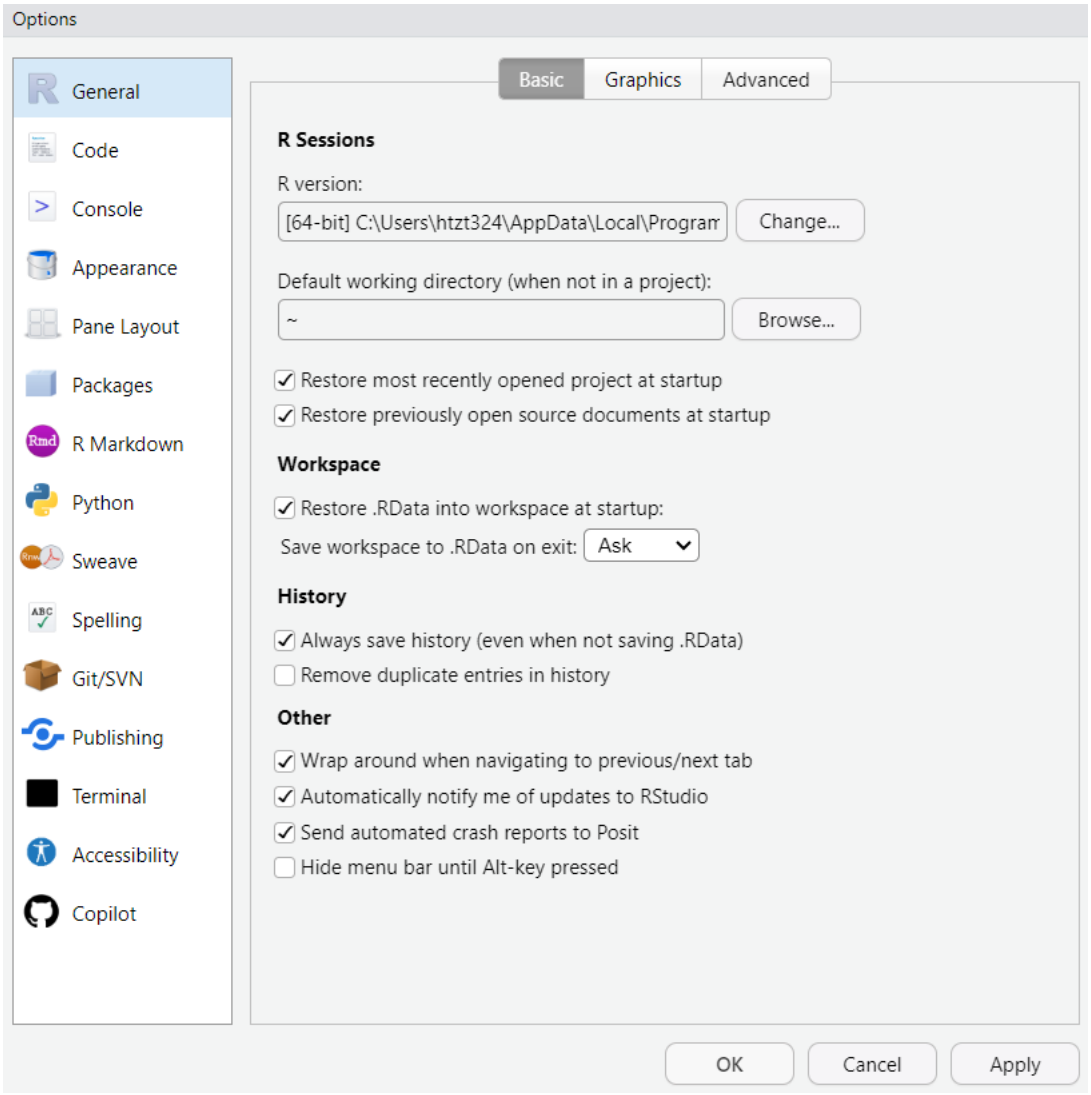


Fourth, make sure `R` is running correctly in the background:

In `RStudio`, in the pane on the left (the "Console"), type 2+2 and hit Enter.
If `R` is working properly, the number "4" will be printed in the next line down.

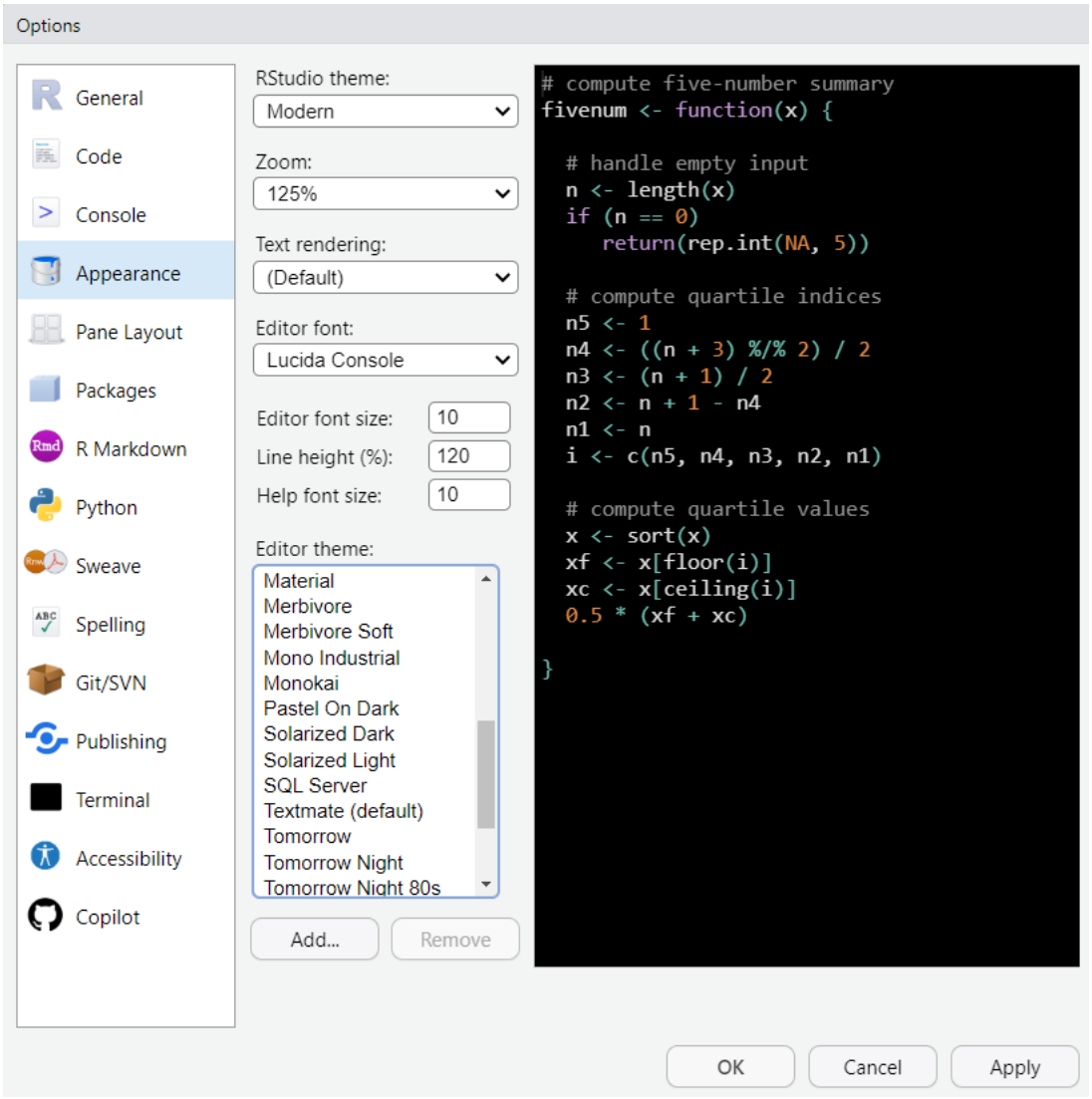
Finally, some minor adjustments to make `RStudio` run smoother (and look cooler):

Go to `Tools > Global Options` and make sure your `General` settings match these exactly:



Specifically, **uncheck** the option under *Workspace* to 'Restore .RData into workspace at startup.'

Now go to the `Appearance` settings and choose a cool theme!



Boom! You've got R up and running. Now it's time to install some packages.

Install packages

Packages are what makes R so great. To install a package, open Rstudio, make sure you're connected to Wi-Fi, and type the following code into the "Console" in the bottom left

Did that work? Probably not, because "packagename" is not a real R package. The packages we'll be using are the following:

```
tidyverse
RColorBrewer
leaflet
ggthemes
rmarkdown
sp
```

To install all of these, you can run `install.packages` for each package:

```
install.packages('tidyverse')
install.packages('RColorBrewer')
# etc., etc.
```

So, now you've got R, RStudio, and some cool packages. Good job. You're ready for the next module.

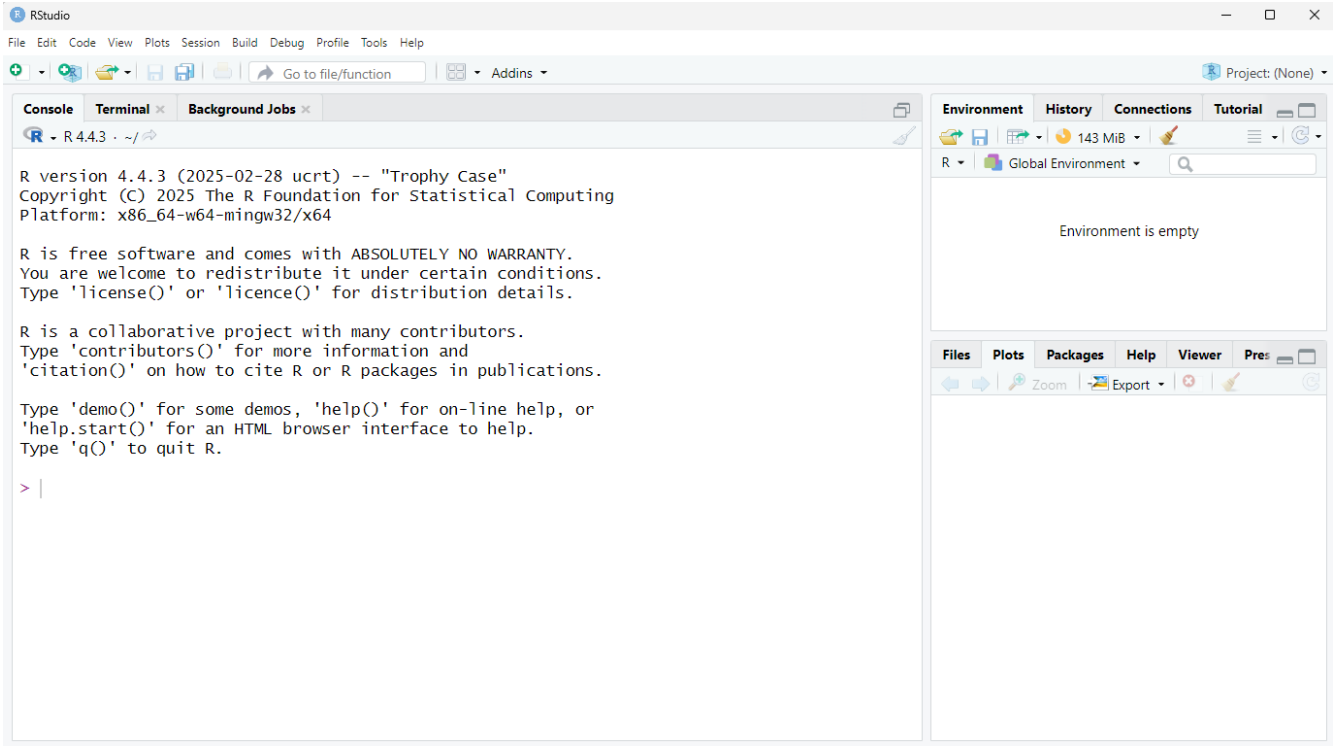
Chapter 3

Preparing for the R Training

Learning goals

- Learn how to run code in R
- Learn how to use R as a calculator
- Learn how to use mathematical and logical operators in R

When you open `RStudio`, you see several different panes within the program's window. You will get a tour of `RStudio` in the next module. For now, look at the left half of the screen. You should see a large pane entitled the *Console*.



`RStudio`'s *Console* is your window into `R`, the engine under the hood. The *Console* is where you type commands for `R` to run, and where `R` prints back the results of what you have told it to do. Think of the *Console* as a chatroom, where you and `R` talk back and forth.

3.1 Running R code in the *Console*

Type your first command into the *Console*, then press `Enter`:

```
1 + 1
[1] 2
```

When you press `Enter`, you send your line of code to `R`; you post it for `R` to see. Then `R` takes it, does some processing, and posts a result (`2`) just below your command.

Note that spaces don't matter. Both of the following two commands are legible to `R` and return the same thing:

```
4+4
[1] 8
```

```
4        +        4
[1] 8
```

However, it is better to make your code as easy to read as possible, which usually means using a single space between numbers:

```
4 + 4
[1] 8
```

Try typing in other basic calculations:

R as a calculator

As you can tell from those commands you just ran, `R` is, at heart, a fancy calculator.

Some calculations are straightforward, like addition and subtraction:

```
490 + 1000
[1] 1490
```

```
490 - 1000
[1] -510
```

Division is pretty straightfoward too:

```
24 / 2
[1] 12
```

For multiplication, use an asterisk (`*`):

```
24 * 2
[1] 48
```

You denote exponents like this:

```
2 ^2
[1] 4
```

```
2 ^3
[1] 8
```

```
2 ^4
[1] 16
```

Finally, note that R is fine with negative numbers:

```
9 + -100
[1] -91
```

Re-running code

If you want to re-run the code you just ran, or if you want to recall the code so that you can adjust it slightly, click anywhere in the *Console* then press your keyboard's Up arrow.

If you keep pressing your Up arrow, R will present you with sequentially older commands. R keeps a history of everything you have said to it since you opened this window.

If you accidentally recalled an old command without meaning to, you can reset the *Console*'s command line by pressing Escape.

Incomplete commands

Similar to talking to a friend, if you start a sentence and don't finish it, your friend will get confused. R gets confused when you enter an incomplete command, and will wait for you to write the remainder of your command on the next line in the *Console* before doing anything.

For example, try running this code in your *Console*:

```
45 -
```

You will find that R gives you a little + sign on the line under your command, which means it is waiting for you to complete your command.

If you want to complete your command, add a number (e.g., 3) and hit Enter. You should now be given an answer (e.g., 48).

Or, if instead you want R to stop waiting and stop running, just press the Escape key.

Getting errors

R only understands your commands if they follow the rules of the R language (often referred to as its *syntax*). If R does not understand your code, it will throw an error and give up on trying to execute that line of code.

For example, try running this code in your *Console*:

```
4 + y
```

You probably received a message in red font stating: Error: object 'y' not found. That is because R did know how to interpret the symbol y in this case, so it just gave up.

Get used to errors! They happen all the time, even (especially?) to professionals, and it is essential that you get used to reading your own code to find and fix its errors.

Here's another piece of code that will produce an error:

```
dfjkltr9fitwt985ut9e3
```

Using parentheses

R is usually great about following classic rules for Order of Operations, and you can use parentheses to exert control over that order. For example, these two commands produce different results:

```
2*7 - 2*5 / 2
[1] 9
```

```
(2*7 - 2*5) / 2
[1] 2
```

Note that parentheses need to come in pairs: whenever you type an open parenthesis, `(`, eventually you need to provide a corresponding closed parenthesis, `)`.

The following line of code will return a plus sign, `+`, since R is waiting for you to close the parenthetical before it processes your command:

```
4 + (5
```

Remember: **parentheses come in pairs!** The same goes for other types of brackets: `{...}` and `[...]`.

You can ask R basic questions using *operators*.

For example, you can ask whether two values are equal to each other.

```
96 == 95
[1] FALSE
```

```
95 + 2 == 95 + 2
[1] TRUE
```

R is telling you that the first statement is `FALSE` (`96` is not, in fact, equal to `95`) and that the second statement is `TRUE` (`95 + 2` is, in fact, equal to itself).

Note the use of *double* equal signs here. You must use two of them in order for R to understand that you are asking for this logical test.

You can also ask if two values are *NOT* equal to each other:

```
96 != 95
[1] TRUE
```

```
95 + 2 != 95 + 2
[1] FALSE
```

This test is a bit more difficult to understand: In the first statement, R is telling you that it is `TRUE` that `96` is different from `95`. In the second statement, R is saying that it is `FALSE` that `95 + 2` is not the same as itself.

Note that R lets you write these tests another, even more confusing way:

```
! 96 == 95
[1] TRUE
```

```
! 95 + 2 == 95 + 2
[1] FALSE
```

The first line of code is asking `R` whether it is not true that `96` and `95` are equal to each other, which is `TRUE`. The second line of code is asking `R` whether it is not true that `95 + 2` is the same as itself, which is of course `FALSE`.

Other commonly used operators in `R` include greater than / less than symbols (`>` and `<`, also known as the *left-facing alligator* and *right-facing alligator*), and greater/less than or equal to (`>=` and `<=`).

```
100 > 100
[1] FALSE
```

```
100 >= 100
[1] TRUE
```

```
(100 != 100) == FALSE
[1] TRUE
```

Use built-in R functions

R has some built-in "functions" for common calculations, such as finding square roots and logarithms. Functions are packages of code that take a given value, transform it according to some internal code instructions, and provide an output. You will learn more about functions in a few modules.

To find the square-root of a number, use the 'squirt' command, `sqrt()`:

```
sqrt(16)
```

Note the use of parentheses here. When you are calling a function, when you see parentheses, think of the word 'of'. This line translates to 'the square root of 16'. You are taking the sqrt of the number inside the parenthetical.

To get the log of a value:

```
log(4)
```

Note that the function `log()` is the natural log function (i.e., the value that e must be raised to in order to equal 4). To calculate a base-10 logarithm, use `log10( )`.

```
log(10)

log10(10)
```

Another handy function is round(), for rounding numbers to a specific number of decimal places.

```
100/3

round(100/3)

round(100/3,digits=1)

round(100/3,digits=2)

round(100/3,digits=3)
```

Finally, R also comes with some built-in values, such as pi:

```
pi
```

Congratulations! You are all prepped for the R Training in Brazzaville. We will see you there.

Other Resources

Hobbes Primer, Table 1 (Math Operators, pg. 18) and Table 2 (Logical operators, pg. 22)

Part II

Getting Started

Chapter 4

Welcome!

Learning goals

- Learn what this course is about
- Learn about the why
- Start coding

What this is, and what it isn't

This is not a textbook or an encyclopedia. This is not a reference manual. It is not exhaustive or comprehensive.

So what is this? This guide is an *accelerator,* an *incubator* designed to guide you along the most direct path from your first line of code to becoming a capable researcher. Our goal is to help you through the most dangerous period in your data science education: your very first steps. The first three weeks. That is when 99% percent of people give up on learning to code.

But it doesn't need to be this way.

This approach is based on three core premises:

Premise 1: We learn best by doing. Our goal is to get you *doing* data science. We will keep theory and detail to a minimum. We will give you the absolute basics, then offer you exercises and puzzles that motivate you to learn the rest. Then, once you've been *doing* data science for a bit, you soon begin *thinking* like a data scientist. By that, we mean tackling ambiguous problems with persistence, independence, and creative problem solving.

Premise 2: We learn best with purpose. Once you gain comfort with the basic skills, you will be able to start working on real data, for real projects, with real impact. You will start to *care about what you are coding.* And that is when the learning curve *skyrockets* – because you are motivated, and because you are learning *reactively*, instead of preemptively. Our goal is to get you to the point of take-off as quickly as possible.

Premise 3: A simple toolbox is all you need to build a house. Once you become comfortable with a few basic coding tools, you can build pretty much anything. The toolbox doesn't need to be that big; if you know how to use your tools well, and if you have enough building supplies (i.e., data), the possibilities are limitless.

One more thing that this is not: This is not a fancy interactive tutorial with bells or whistles. We purposefully designed this to be simple and "analog". You will not be typing your code into this website and getting feedback from a robot, or setting up an account to track your progress, or getting pretty merit badges or points when you complete each module.

Instead, you will be doing your work on your *own machine*, working with *real folders and files*, downloading data and moving it around, etc. – all the things you will be doing as a real data scientist in the real world.

What you will learn

- As you are learning to code, we will sprinkle in conceptual foundations and motivations for this work: what data science is, why it matters, and ethical issues surrounding it: the good, the bad, and the ugly. The most important thing, at first, is to start writing code.
- The entire point of this course is to learn how to use `R` (in `RStudio`). Here you will add the first and most important tools to your toolbox: working with variables, vectors, dataframes, scripts, and file directories.
- You will gain the skills to bring in your own data and work with it in `R`. You will learn how to produce beautiful plots and how to reorganize, filter, and summarize your datasets. You will also learn how to conduct basic statistics, from exploratory data analyses (e.g., producing and comparing distributions) to significance testing.
- If and when you conquer all of the above, we will provide various puzzles that allow you to apply the basic `R` skills from the previous unit to fun questions and scenarios. In each of these exercises, questions are arranged in increasing order of difficulty, so that beginners will not feel stuck right out of the gate, nor will experienced coders become bored. This is where you really begin to cut your teeth on real-world data puzzles: figuring out how to use the `R` tools in your toolbag to tackle an ambiguous problem and deliver an excellent data product.

Where did this come from

This curriculum was modified based on a curriculum originally developed for the **DataLab** at Sewanee: The University of the South, TN, USA. Thanks Databrew, Eric Keen, Joe Brew, Ben Brew, and Matthew Rudd.

Chapter 5

Using `RStudio` & `R` scripts

Learning goals

- Understand the difference between `R` and `RStudio`
- Understand the `RStudio` working environment and window panes

- Understand what `R` scripts are, and how to create and save them
- Understand how to add comments to your code, and why doing so is important
- Understand what a *working directory* is, and how to use it
- Learn basic project work flow

`R` and `RStudio`: what's the difference?

These two entities are similar, but it is important to understand how they are different.

In short, `R` is a open-source (i.e., free) coding language: a powerful programming engine that can be used to do really cool things with data.

`R Studio`, in contrast, is a free *user interface* that helps you interact with `R`. If you think of `R` as an engine, then it helps to think of `RStudio` as the car that contains it. Like a car,`RStudio` makes it easier and more comfortable to use the engine to get where you want to go.

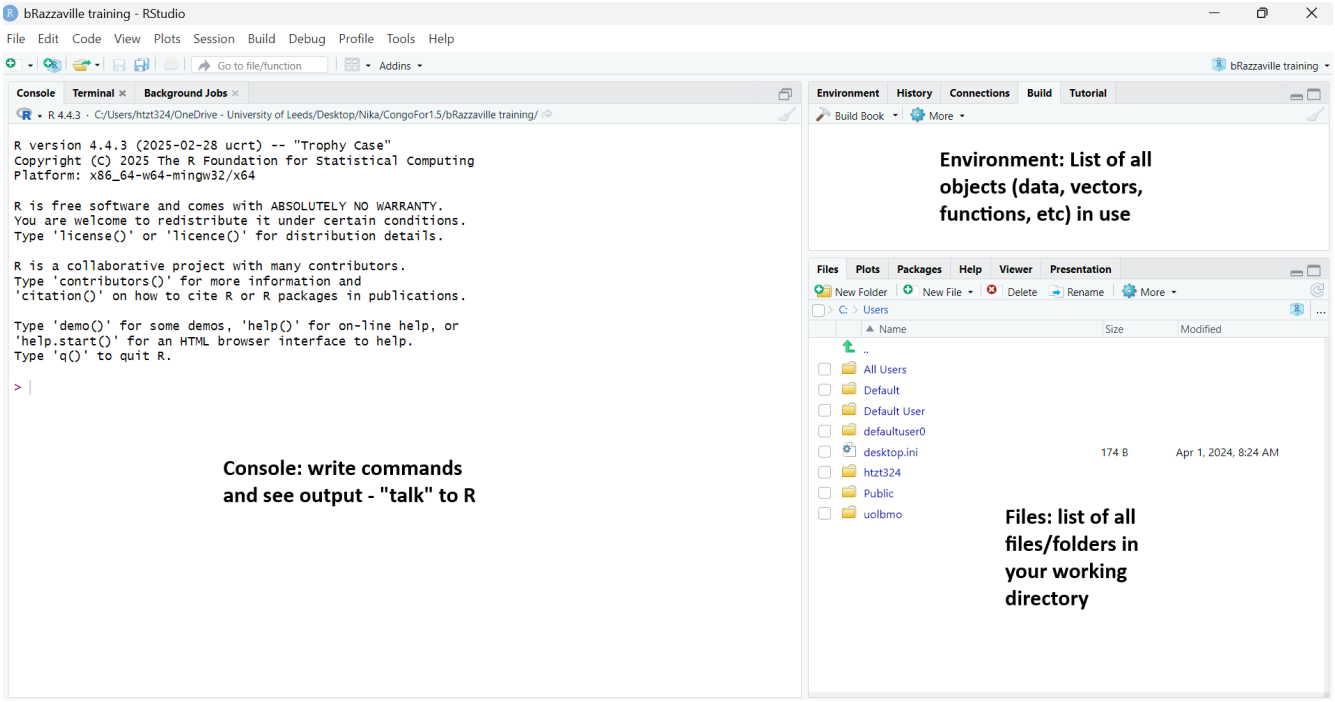
`R Studio` needs `R` in order to function, but `R` can technically be used on its own outside of `RStudio` if you want. However, just as a good car mechanic can get an engine to run without being installed within a car, using `R` on its own is a bit clunky and requires some expertise. For beginners (and everyone else, really), `R` is just so much more pleasant to use when you are operating it from within `RStudio`.

`RStudio` also has increasingly powerful *extensions* that make `R` even more useful and versatile in data science. These extensions allow you to use `R` to make interactive data dashboards, beautiful and reproducible data reports, presentations, websites, and even books. And new features like these are regularly being added to `RStudio` by its all-star team of data scientists.

That is why this book *always* uses `RStudio` when working with `R`.

Two-minute tour of `RStudio`

When you open `RStudio` for the first time, you will see a window that looks like the screenshot below.



Console

You are already acquainted with `RStudio`'s *Console*, the window pane on the left that you use to "talk" to `R`. (See Preparing for R Training Module.)

Environment

In the top right pane, the *Environment*, `RStudio` will maintain a list of all the datasets, variables, and functions that you are using as you work. The next modules will explain what variables and functions are.

Files, Plots, Packages, & Help

You will use the bottom right pane very often.

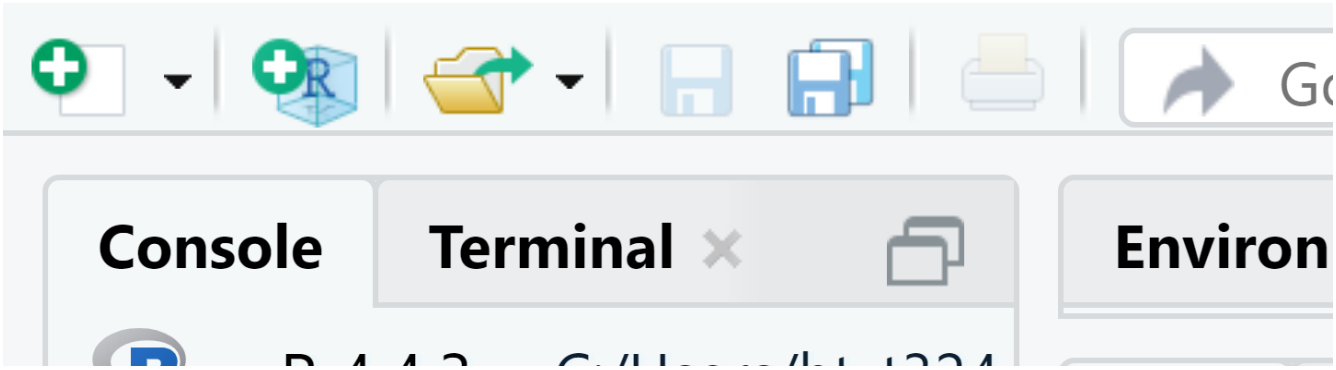
- The **Files** tab lets you see all the files within your **working directory**, which will be explained in the section below.
- The **Plots** tab lets you see the plots you are producing with your code.
- The **Packages** tab lets you see the *packages* you currently have installed on your computer. Packages are bundles of `R` functions downloaded from the internet; they will be explained in detail a few modules down the road.
- The **Help** tab is very important! It lets you see *documentation* (i.e., user's guides) for the functions you use in your code. Functions will also be explained in detail a few modules down the road.

These three panes are useful, but the most useful window pane of all is actually *missing* when you first open `RStudio`. This important pane is where you work with **scripts**.

Scripts

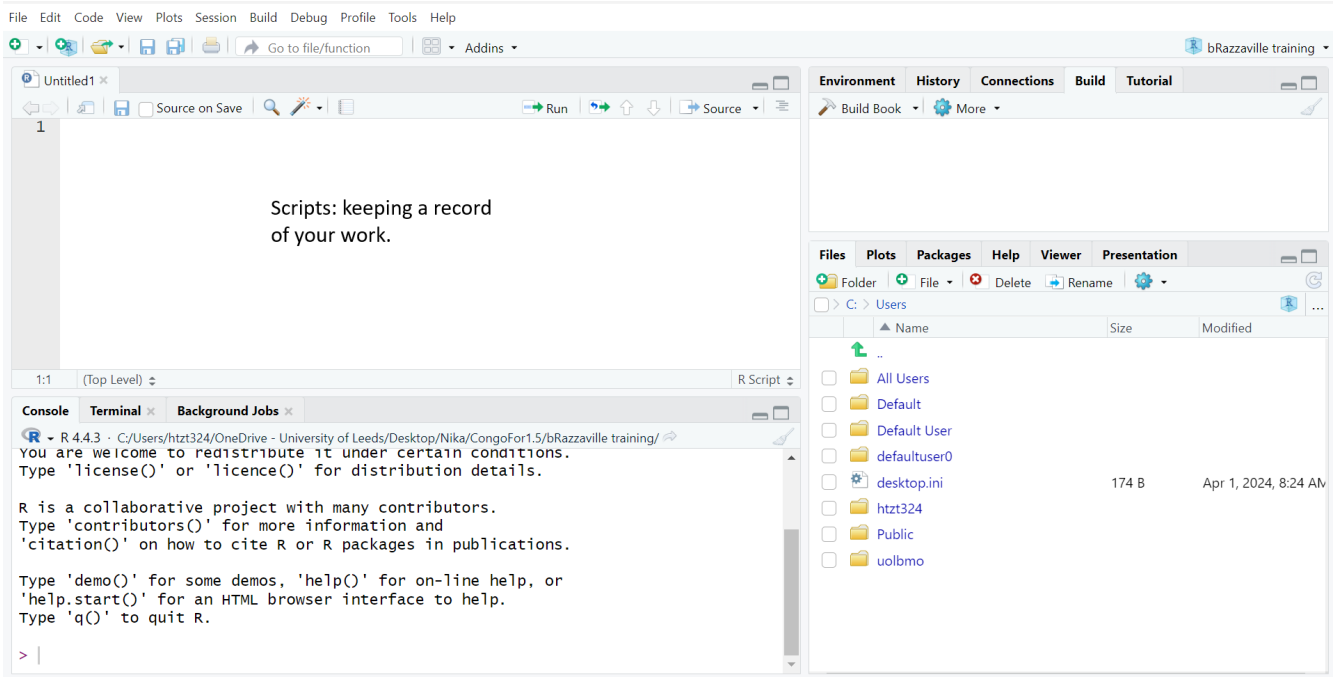
Before explaining what scripts are and why they are awesome, let's start a new script.

To start a new script, go to the top left icon in the `RStudio` window, and click on the green plus sign with a blank page behind it:



A dropdown window will appear. Select "R Script".

A new window pane will then appear in the top left quadrant of your `RStudio` window:



You now have a blank script to work in!

Now type some simple commands into your script:

```
2  + 10
16 * 32
```

Notice that when you press `Enter` after each line of code, nothing happens in the *Console*. In order to send this code to the Console, press `Enter + Command` at the same time (or `Enter + Control`, if you are on Windows) for each line of code.

To send both lines of code to the *Console* at once, select both lines of code and hit `Enter + Command`.

(To select multiple lines of code, you can (1) click and drag with your mouse or (2) hold down your `Shift` key while clicking your down arrow key. To select *all* lines of code, press `Command + A`.)

So let's build up this script. Add a few more lines to your script, such that your script now looks like this.

```
2  + 10
16 * 32
1080 / 360
500 - 600
```

Run all of these lines of code at once.

Now add `10` to the first number in each row, and re-run all of the code.

Think about how much more efficient part (B) was thanks to your script! If you had typed all of that directly into your *Console*, you would have to recall or retype each line individually. That difference builds up when your number of commands grows into the hundreds.

What is an R script, and why are scripts so awesome?

An R script is a file where you can keep a record of your code. Just as a script tells actors exactly what to say and when to say it, an R script tells R exactly what code to run, and in what order to run it.

When working with R, you will almost always type your code into a script first, *then* send it to the *Console*. You can run your code immediately using `Enter + Command`, but you also have a script of what you have done so that you can run the exact same code at a later time

To understand why R scripts are so awesome, consider a typical workflow in *Excel* of *GoogleSheets*. You open a big complicated spreadsheet, spend hours making changes, and save your changes frequently throughout your work session.

The main disadvantages of this workflow are that:

1. There is no detailed record of the changes you have made. You cannot prove that you have made changes correctly. You cannot pass the original dataset to someone else and ask them to revise it in the same way you have. (Nor would you want to, since making all those changes was so time-consuming!) Nor could you take a different dataset and guarantee that you are able to apply the exact same changes that you applied to the first. In other words, your work is not reproducible.
2. Making those changes is labor-intensive! Rather than spend time manually making changes to a single spreadsheet, it would be better to devote that energy to writing R code that makes those changes for you. That code could be run in this one case, but it could also be run at any later time, or easily modified to make similar changes to other spreadsheets.
3. Unless you are an advanced *Excel* programmer, you are probably modifying your original dataset, which is always dangerous and a big No-No in data science. Each time you save your work in *Excel* or *GoogleSheets* (which automatically saves each change you make), the original spreadsheet file gets replaced by the updated version. But if you brought your dataset into R instead, and modified it using an R script, then you leave the raw data alone and keep it safe. (Sure, you can always save different versions of your Excel file, but then you run the risk of mixing up versions and getting confused.)

Working with R scripts allows you to avoid all of these pitfalls. When you write an R script, you are making your work ….

- **Efficient.** Once you get comfortable writing R code, you will be able to write scripts in a few minutes. Those scripts can modify datasets within seconds (or less) in ways that would take hours (or years) to carry out manually in *Excel* or *GoogleSheets*.
- **Reproducible.** Once you have written anR script, you can reproduce your own work whenever you want to. You can send your script to a colleague so that they can reproduct your work as well. Reproducible work is defensible work.

- **Low-risk.** Since your R script does not make any changes to the original data, you are keeping your data safe. It is *essential* to preserve the sanctity of raw data!

Note that there is nothing fancy or special about an `R` script. An `R` script is a simple text file; that is, it only accepts basic text; you can't add images or change font style or font size in an `R` script; just letters, numbers, and your other keyboard keys. The file's extension, `.R` tells your computer to interpret that text as `R` code.

Commenting your code

Another advantage of scripts is that you can include *comments* throughout your code to explain what you are doing and why. A *comment* is just a part of your script that is useful to you but that is ignored by `R`.

To add comments to your code, use the hashtag symbol (`#`). Any text following a `#` will be ignored by `R`.

Here is the script above, now with comments added:

```
# Define variable x
x <- 2
x

# Make a new variable, y, based on x
y <- x*56

z <- y / 23 # Make a third variable, z, based on y

x + y + z # Now get the sum of all three variables
```

Adding comments can be more work, but in the end it saves you time and makes your code more effective. Comments might not seem necessary in the moment, but it is amazing how helpful they are when you come back to your code the next day. Frequent and helpful comments make the difference between good and great code. Comment early, comment often!

You can also use lines of hashtags to visually organize your code. For example:

```
##############################################
# Setup
##############################################

# Define variable x
x <- 2
x

# Make a new variable, y, based on x
y <- x*56

z <- y / 23 # Make a third variable, z, based on y


##############################################
# Get result
##############################################

x + y + z # Now get the sum of all three variables
```

This might not seem necessary with a 5-line script, but adding visual breaks to your code becomes immensely helpful when your code grows to be hundreds of lines long.

Saving your work

R scripts are only useful if you save them! Unlike working with *GoogleDocs* or *GoogleSheets*, R will not automatically save your changes; you have to do that yourself. (This is inconvenient, but it is also safer; most of coding is trial and error, and sometimes you want to be careful about what is saved.)

Step 1: Decide where to save your work. The folder in which you save your R script will be referred to as your *working directory* (see the next section). For the sake of these tutorials, it will be most convenient to save all of your scripts in a single folder that is in an easily accessed location.

Step 2: In that location, make a new folder named bRazzaville: We suggest making a new folder on your Desktop and naming it bRazzaville, but you can name it whatever you want and place it wherever you want.

Step 3: Save your script in that folder To save the script you have opened and typed a few lines of code into, press Command + S (or Control + S). Alternatively, go to File > Save. Navigate to the folder you just created and type in a file name that is simple but descriptive. We suggest making a new R script for each module, and naming those scripts according to each module's name. In this case, we recommend naming your script intro_to_scripts.

(It is good practice to avoid spaces in your file names; it will be essential later on, so good to begin the correct habit now. Start using an underscore, _, instead of a space.)

Your working directory

When you work with data in R, R will need to know where in your computer to look for that data. The folder it looks in is known as your **working directory**.

To find out which folder R is currently using as your working directory, use the function getwd():

```
getwd()
[1] "C:/Users/htzt324/OneDrive - University of Leeds/Desktop/Nika/CongoFor1.5/bRazzaville_training"
```

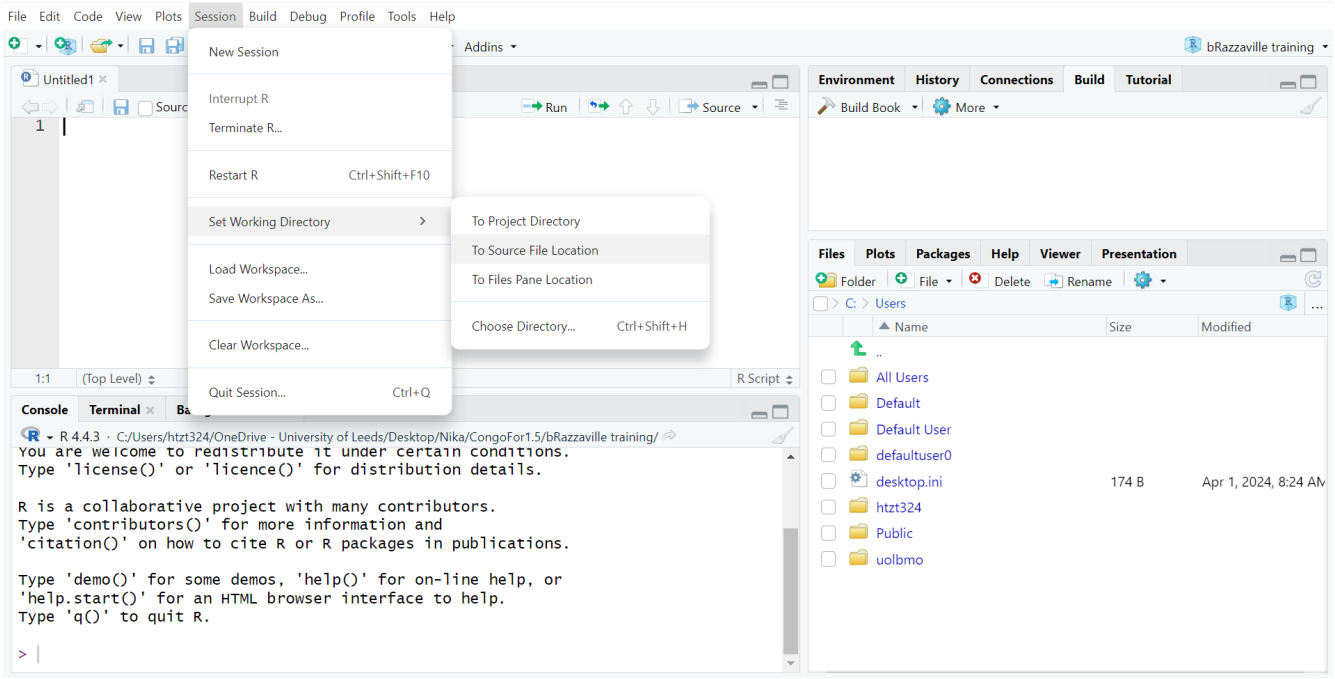
Almost always, you want to set your working directory to the folder your R script is in.

How to set your working directory

Whenever you begin a new R script, setting your working directory will be one of the first things you do.

There are three ways to set your working directory:

1. **Manually without code.** On the top banner in RStudio, go to *Session > Set Working Directory > To Source File Location*:



This action sets your working directory to the same folder that your R script is in. When you do this, you will see that a command has been entered into your *Console*:

```
setwd("~/Desktop/bRazzaville")
```

(Note that the filepath may be different on your machine.) This code is using the function `setwd()`, which is also used in the next option. Go ahead and copy this `setwd(...)` code and paste it into your script, so it will be easy to use next time.

2. **Manually with code, using `setwd()`**: You can manually provide the filepath you want to set as your working directory. This option allows you to set your `wd` to whatever folder you want. The character string within the `setwd()` command is the path to a folder. The formatting of this string must be exact, otherwise R will throw an error. Use option 1 at first to get a sense of how your computer formats its folder paths. Copy, paste, and modify the output from option 1 in order to type your path correctly.

3. **Automatically with code**: There is a command you can run that automatically sets your working directory to the folder that your R script is in. This is the most efficient and useful method, in our experience.

To use this command, you must first install a new package. Run this code:

```
install.packages("rstudioapi")
library(rstudioapi)
```

For now, you do not need to understand what this code is doing. We will explain packages and the `library()` function in a later module.

You can now copy, paste, and run this code to set your working directory automatically:

```
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
```

This is a complicated line of code that you need not understand. As long as it works, it works! Confirm that R is using the correct working directory with the command `getwd()`.

Typical workflows

Now that you know how to create a script and set your working directory, you are prepared to work on data projects in RStudio.

The workflow for beginning a new data project typically goes like this:

In your file explorer…

1. **Create a folder for your project** somewhere on your computer. This will become your working directory.
2. **Create subfolders** within your working directory, if you want. We recommend creating a `data` subfolder, for keeping data, and a `z` subfolder, for keeping miscellaneous documents. The goal is to keep your working directory visually simple and organized; ideally, the only files not within subfolders are your R scripts.
3. **Add data** to your working directory, if you have any.

In RStudio …

4. **Create a new R script.**
5. **Save it** inside your intended working directory.
6. At the top of your script, use comments to **add a title, author info, and brief description.**
7. Add the code to **set your working directory.**
8. **Begin coding!**

Template R script

Here is a template you can use to copy and paste into each new script you create:

```
################################################################################
# < Add title here >
################################################################################
#
# < Add brief description here >
#
# < Author >
# Created on <add date here >
#
################################################################################
# Set working directory
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))

################################################################################
################################################################################
# Code goes here




################################################################################
# (end of file)
```

Exercises

1 *(if not already complete).* Create a working directory for this course. Call it whatever you like, but `bRazzaville` could work great. Place it somewhere convenient on your computer, such as your Desktop.

2. Within this working directory, create three new folders: (1) a `data` folder, which is where you will store the data files we will be using in subsequent modules and (2) a `modules` folder, which is where you will keep the code you use to work on the material in these modules.

3. Now follow the *Typical Workflow* instructions above to create a script. Save it within your `modules` folder. Name it `template.R`. Copy and paste the template `R` code provided above into this file, and save it. This is now a template that you can use to easily create new scripts for this course.

4. Now make a copy of `template.R` to stage a script that you can use in the next module. To do so, in `RStudio` go to the top banner and click `File > Save As`. Save this new script as `variables.R` (because the next module is called *Variables in `R`*).

5. Modify the code in `variables.R` so that you are prepared to begin the next module. Change the title, and look ahead to the next module to fill in a brief description. Don't forget to add your name as the author and specify today's date.

Boom!

Other Resources

A Gentle Introduction to `R` from the `RStudio` team

Chapter 6

Review your R Training Prep

Prior to arriving in Brazzaville, we asked that you download R and RStudio and do some preparatory work. Let's work through some of that information together.

Exercises

Use `R` like a calculator

1. Type a command in the *Console* to determine the sum of 596 and 198.

2. Re-run the sum of 596 and 198 without re-typing it.

3. Recall the command again, but this time adjust the code to find the sum of 596 and 298.

4. Practice escaping an accidentally called command: recall your most recent command, then press the right key to clear the *Console*'s command line.

Recalling commands

5. Find the sum of the ages of everyone in your immediate family.

6. Now recall that command and adjust it to determine the *average* age of the members of your family.

7. Find the square root of *pi* and round the answer to the 2 decimal places.

Finding errors

8. This line of code won't run; instead, `R` will wait for more with a `+` symbol. Find the problem and re-write the code so that it works.

```
5 * 6 +
```

9. The same goes for this line of code. Fix it, too.

```
sqrt(16
```

10. This line of code will trigger an error. Find the problem and re-write the code so that it works.

```
round(100/3,digits+3)
```

11. Type a command of your own into R that throws an error, then recall the command and revise so that `R` can understand it.

Show that the following statements are TRUE:

12. `pi` is greater than the square root of 9

13. It is FALSE that the square root of 9 is greater than `pi`

14. `pi` rounded to the nearest whole number equals the square root of 9

Asking `TRUE` / `FALSE` questions

15. Write and run a line of code that asks whether these two calculations return the same result:

```
2*7 - 2*5 / 2
```

```
(2*7 - 2*5) / 2
[1] 2
```

16. Now write and run a line of code that asks whether the first calculation is larger than the second:

Chapter 7

Variables

Learning goals

- How to define variables and work with them in R
- Learn the various possible classes of data in R

Introducing variables

So far we have strictly been using R as a calculator, with commands such as:

```
3 + 5
[1] 8
```

Of course, R can do much, much more than these basic computations. Your first step in uncovering the potential of R is learning how to use **variables**.

In R, a variable is a convenient way of referring to an underlying value. That value can be as simple as a single number (e.g., 6), or as complex as a spreadsheet that is many Gigabytes in size. It may be useful to think of a variable as a cup; just as cups make it easy to hold your coffee and carry it from the kitchen to the couch, variables make it easy to contain and work with data.

Declaring variables

To assign numbers or other types of data to a variable, you use the < and - characters to make the arrow symbol <-.

```
x <- 3+5
```

As the direction of the <- arrow suggests, this command stores the result of 3 + 5 into the variable x.

Unlike before, you did not see 8 printed to the *Console*. That is because the result was stored into x.

Calling variables

If you wanted R to tell you what x is, just type the variable name into the *Console* and run that command:

```
x
[1] 8
```

Want to create a variable but also see its value at the same time? Here's a handy trick: put your command in parentheses:

```
(x <- 3*12)
[1] 36
```

When you do that, `x` gets assigned a value, then that value is printed to the console.

You can also update variables.

```
(x <- x * 3)
[1] 108
```

```
(x <- x * 3)
[1] 324
```

You can also add variables together.

```
x <- 8
y <- 4.5
x + y
[1] 12.5
```

Naming variables

Here are a few rules:

1. A variable name has to have at least one letter in it. These examples work:

2. A variable name has to be connected. No spaces! It is usually best to represent a space using a period (`.`) or an underscore (`_`). Note that periods and underscores can be used in variable names:

```
my.variable <- 5 # periods can be used
my_variable <- 5 # underscores can be used
```

However, hyphens *cannot* be used, since that symbol is used for subtraction.

3. Variables are case-sensitive. If you misspell a variable name, you will confuse `R` and get an error. For example, ask `R` to tell you the value of capital `X`. The error message will be `Error: object 'X' not found`, which means `R` looked in its memory for an object (i.e., a variable) named `X` and could not find one.

4. Variable names can be as complicated or as simple as you want.

5. Some names need to be avoided, since `R` uses them for special purposes. For example, `data` should be avoided, as should `mean`, since both are functions built-in to `R` and `R` is liable to interpret them as such instead of as a variable containing your data.

```
supercalifragilistic.expialidocious <- 5
supercalifragilistic.expialidocious  # still works
[1] 5
```

So those are the basic rules, but naming variables well is a bit of an art. The trick is using names that are clear but are not so complicated that typing them is tedious or prone to errors.

Note that `R` uses a feature called 'Tab complete' to help you type variable names. Begin typing a variable name, such as `supercalifragilistic.expialidocious` from the example above, but after the first few letters press the `Tab` key. `R` will then give you options for auto-completing your word. Press `Tab` again, or `Enter`, to accept the auto-complete. This is a handy way to avoid typos.

Types of data in R

So far we have been working exclusively with numeric data. But there are many different data types in R. We call these "types" of data **classes**:

- Decimal values like 4.5 are called **numeric** data.
- Natural numbers like 4 are called **integers**. Integers are also numerics.
- Boolean values (TRUE or FALSE) are called **logical** data.
- Text (or string) values are called **character** data.

In order to be combined, data have to be the same class.

R is able to compute the following commands …

```
x <- 6
y <- 4
x + y
[1] 10
```

… but not these:

```
x <- 6
y <- "4"
x + y
```

That's because the quotation marks used in naming `y` causes `R` to interpret `y` as a `character` class.

To see how `R` is interpreting variables, you can use the `class()` function:

```
x <- 100
class(x)
[1] "numeric"
```

```
x <- "100"
class(x)
[1] "character"
```

```
x <- 100 == 101
class(x)
[1] "logical"
```

Another data type to be aware of is **factors**, but we will deal with them later.

Exercises

Finding the errors

1. This code will produce an error. Can you find the problem and fix it so that this code will work?

```
# Assign 5 to a variable
my_var < 5
```

2. Same for this one:

```
# Assign 5 to a variable
my_var == 5
```

3. Same for this one:

```
x <- 5
y <- 1
X + y
```

Your Bananas-to-ICS ratio

4. Estimate how many bananas you've eaten in your lifetime and store that value in a variable (choose whatever name you wish). (By the way, what is a good method for estimating this as accurately as you can?)

5. Now estimate how many ice cream sandwiches you've eaten in your lifetime and store that in a different variable.

6. Now use these variables to calculate your Banana-to-ICS ratio. Store your result in a third variable, then call that variable in the Console to see your ratio.

7. Who in the class has the highest ratio? Who has the lowest?

Creating boolean variables

8. Assign a `FALSE` statement of your choosing to a variable of whatever name you wish.

9. Confirm that the class of this variable is "logical."

10. Confirm that the variable equals `FALSE`.

Converting Fahrenheit to Celsius:

11. Assign a variable `fahrenheit` the numerical value of 32.

12. Assign a variable `celsius` to equal the conversion from Fahrenheit to Celsius. Unless you're a meteorology nerd, you may need to Google the equation for this conversion.

13. Print the value of `celsius` to the *Console*.

14. Now use this code to determine the *Celsius* equivalent of 212 degrees *Fahrenheit*.

Wrapping up

15. Now ensure that your entire script is properly commented, and make sure your script is saved in your `bRazzaville` working directory before closing.

Chapter 8

Vectors

Learning goals

- Learn the various structures of data in R
- How to work with vectors in R

Data belong to different *classes*, as explained in the previous module, and they can be arranged into various **structures**.

So far we have been dealing only with variables that contain a single value, but the real value of R comes from assigning *entire sets* of data to a variable.

The simplest data structure in R is a **vector**. A vector is simply a set of values. A vector can contain only a single value, as we have been working with thus far, or it can contain many millions of values.

Declaring and using vectors

To build up a vector in R, use the function `c()`, which is short for "concatenate".

```
x <- c(5,6,7,8)
x
[1] 5 6 7 8
```

Whenever you use the `c()` function, you are telling R: 'Hey, get ready. I'm about to give you more than one value at once.'. Cough Cough.

You can use the `c()` function to concatenate two vectors together:

```
x <- c(5,6,7,8)
y <- c(9,10,11,12)
z <- c(x,y)
z
[1]  5  6  7  8  9 10 11 12
```

You can also use `c()` to add values to a vector:

```
x <- c(5,6,7,8)
x <- c(x,9)
x
[1] 5 6 7 8 9
```

You can also put vectors through logical tests:

```
x <- c(1,2,3,4,5)
4 == x
[1] FALSE FALSE FALSE  TRUE FALSE
```

This command is asking `R` to tell you whether each element in `x` is equal to `4`.

You can create vectors of any data class (i.e., data type).

```
x <- c("Greta","Nika","Crispin")
x
[1] "Greta"   "Nika"    "Crispin"
```

```
y <- c(TRUE,TRUE,FALSE)
y
[1]  TRUE  TRUE FALSE
```

Note that all values within a vector *must* be of the same class. You can't combine numerics and characters into the same vector. If you did, `R` would try to convert the numbers to characters. For example:

```
x <- 4
y <- "6"
z <- c(x,y)
z
[1] "4" "6"
```

Math with two vectors

When two vectors are of the same length, you can do arithmetic with them:

```
x <- c(5,6,7,8)
y <- c(9,10,11,12)
x + y
[1] 14 16 18 20
```

```
x - y
[1] -4 -4 -4 -4
```

```
x * y
[1] 45 60 77 96
```

```
x / y
[1] 0.5555556 0.6000000 0.6363636 0.6666667
```

What happens when two vectors are *not* the same length?

Well, it depends. If one vector is length 1 (i.e., a single number), then things usually work out well.

```
x <- 5
y <- c(1,2,3,4,5,6,7,8,10)
x + y
[1]  6  7  8  9 10 11 12 13 15
```

In this command, the single element of `x` gets added to each element of `y`.

Another example, which you already saw above:

```
a <- c(1,2,3,4,5)
b <- 4
a == b
[1] FALSE FALSE FALSE  TRUE FALSE
```

In this command, the single element of `b` gets compared to each element of `a`.

However, when both vectors contain multiple values but are not the same length, **be warned**: wonky things can happen. This is because `R` will start recycling the shorter vector:

```
a <- c(1,2,3,4,5)
b <- c(3,4)
a + b
[1] 4 6 6 8 8
```

As this warning implies, this doesn't make much sense. The command will still run, but do not trust the result.

Functions for handling vectors

We are about to list a bunch of core functions for working with vectors. Think of this like a toolbag. Each tool has a specific purpose and limited value: you can't quite build a house with just a hammer. But when you learn how to use all of the tools in your tool bag *together*, you can build almost anything. But you have to know how to use each tool individually first.

`length()` tells you the number of elements in a vector:

```
x <- c(5,6)
length(x)
[1] 2
```

```
y <- c(9,10,11,12)
length(y)
[1] 4
```

The **colon symbol `:`** creates a vector with every integer occurring between a min and max:

```
x <- 1:10
x
 [1]  1  2  3  4  5  6  7  8  9 10
```

`seq()` allows you to build a vector using evenly spaced *sequence* of values between a min and max:

```
seq(0,100,length=11)
 [1]   0  10  20  30  40  50  60  70  80  90 100
```

In this command, you are telling R to give you a sequence of values from 0 to 100, and you want the length of that vector to be 11. R then figures out the spacing required between each value in order to make that happen.

Alternatively, you can prescribe the interval between values instead of the length:

```
seq(0,100,by=7)
 [1]  0  7 14 21 28 35 42 49 56 63 70 77 84 91 98
```

rep() allows you to repeat a single value a specified number of times:

```
rep("Hey!",times=5)
[1] "Hey!" "Hey!" "Hey!" "Hey!" "Hey!"
```

You can also use **rep()** to repeat each element of a vector a set number of times:

```
rep(c("Hey!","Wohoo!"),each=3)
[1] "Hey!"   "Hey!"   "Hey!"   "Wohoo!" "Wohoo!" "Wohoo!"
```

head() and **tail()** can be used to retrieve the first 6 or last 6 elements in a vector, respectively.

```
x <- 1:1000
head(x)
[1] 1 2 3 4 5 6
tail(x)
[1]  995  996  997  998  999 1000
```

You can also adjust how many elements to return:

```
head(x,2)
[1] 1 2
tail(x,10)
 [1]  991  992  993  994  995  996  997  998  999 1000
```

sort() allows you to order a vector from its smallest value to its largest:

```
x <- c(4,8,1,6,9,2,7,5,3)
sort(x)
[1] 1 2 3 4 5 6 7 8 9
```

rev() lets you reverse the order of elements within a vector:

```
x <- c(4,8,1,6,9,2,7,5,3)
rev(x)
[1] 3 5 7 2 9 6 1 8 4
```

```
rev(sort(x))
[1] 9 8 7 6 5 4 3 2 1
```

min() and **max()** lets you find the smallest and largest value in a vector.

```
min(x)
[1] 1
```

```
max(x)
[1] 9
```

which() allows you to ask, "For which elements of a vector is the following statement true?"

```
x <- 1:10
which(x==4)
[1] 4
```

If no values within the vector meet the condition, a vector of length zero will be returned:

```
x <- 1:10
which(x == 11)
integer(0)
```

which.min() and **which.max()** tells you which element is the smallest and largest in the vector, respectively:

```
which.min(x)
[1] 1
```

```
which.max(x)
[1] 10
```

%in% is a handy operator that allows you to ask whether a value occurs *within* a vector:

```
x <- 1:10
4 %in% x
[1] TRUE
```

```
11 %in% x
[1] FALSE
```

is.na() is a way of asking whether a vector contains missing, broken, or erroneous values. In R, such values are referred to using the phrase NA. When you see NA, think of R telling you, *'Nah ah! Nope! Not Available!'*

```
x <- c(3,5,7,NA,9,4)
is.na(x)
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

This function is stepping through each element in the vector x and telling you whether that element is NA.

Subsetting vectors

Since you will eventually be working with vectors that contain thousands of data points, it will be useful to have some tools for *subsetting* them – that is, looking at only a few select elements at a time.

You can subset a vector using square brackets [ ]. Whenever you use you use brackets, you are telling R: 'Hey, I want some numbers, but *not everything*: just certain ones.'

```
x <- 50:100
x[10]
[1] 59
```

This command is asking R to return the 10th element in the vector x.

```
x[10:20]
 [1] 59 60 61 62 63 64 65 66 67 68 69
```

This command is asking `R` to return elements `10:20` in the vector `x`.

```
Error in file(filename, "r", encoding = encoding): cannot open the connection
```

```
Error in teacher_tip(tip): could not find function "teacher_tip"
```

Exercises

Creating sequences of numbers

1. Use the colon symbol to create a vector of length 5 between a minimum and a maximum value of your choosing.
2. Create a second vector of length 5 using the `seq()` function. Use code to confirm that the length of this vector is 5.
3. Create a third vector of length 5 using the `rep()` function. Use code to confirm that the length of this vector is 5.
4. Finally, concatenate the three vectors and check that the length equals 15.

Basic vector math

5. Create a variable `x` that is a list of numbers of any size. Create a variable `y` of the same length.
6. Check to see if each values of `x` is greater than each value of y.
7. Check to see if the smallest value of `x` is greater than or equal to the average value of `y`.

Vectors and object classes

8. Create a vector with at least one number, then a second vector with at least one character string, then a third vector with at least one logical value. Identify the class of all three vectors.
9. Now concatenate these three vectors into a fourth vector. Identify the class of this fourth vector.

Heads & tails

10. Create a vector with at least 15 values.
11. Show the first six values of that vector using the `head()` function.
12. Figure out how to show the same result without a function, but instead with your new vector subsetting skills. Now replicate the `tail()` function, using those same skills. You may need to call the `length()` function as well.

Shoe sizes

13. Create a vector called `shoes`, which contains the shoe sizes of five people sitting near you. Use comments to keep track of which size is whose.
14. Arrange this set of shoe sizes in ascending order.
15. Arrange this set of shoe sizes in descending order.
16. Use code to find the the two largest shoe sizes in your vector. Don't use subsetting; instead, write a line of code that would work even if more shoes were added to your vector.
17. What is the shoe size is closest to the mean of these shoe sizes?
18. Use the `which()` function to figure out which of your five neighbors this shoe size belongs to.

Swimming timelines

19. Now create a new vector called `swim_days`, which contains the number of days since those same five people last went swimming (in any body of water; estimating the days since is fine).
20. Use code to ask whether anyone went swimming less than five days ago.
21. Which of your neighbors, if any, went swimming in the last month?
22. Which of your neighbors, if any, have not been swimming the last month?
23. On average, how long has it been since these people have gone swimming?

Dealing with `NA`s

24. Create a vector named `x` with these values: `c(4,7,1,NA,9,2,8)`.
25. Use a function to decide whether or not each element of `x` is `NA`.
26. Use another function to find out which element in `x` is `NA`.
27. Write code that will subset `x` only to those values that are `NA`.
28. Write code that will subset `x` only to those values that are *not* `NA`.

Sleep deficits

29. Now create a vector called `sleep_time` with the number of hours you slept for each day in the last week.
30. Check if you slept more on day 3 than day 7.
31. Get the total number of hours slept in the last week.
32. Get the average number of hours slept in the last week.
33. Check if the total number of hours in the first 3 days is less than the total number of hours in the last 4 days.
34. Now create an object named `over_under`. This should be the difference between how much you slept each night and 8 hours (ie, 1.5 means you slept 9.5 hours and -2 means you slept 8 hours).
35. Write code to use `over_under` to calculate your sleep deficit / surplus this week (ie, the total hours over/under the amount of sleep you would have gotten had you slept 8 hours every night).
36. Write code to get the minimum number of hours you slept this week.

37. Write code to calculate how many hours of sleep you would have gotten had you sleep the minimum number of hours every night.

38. Write code to calculate the average of the hours of sleep you got on the 3rd through 6th days of the week.

39. Write code to calculate how many hours of sleep you would get in a year if you were to sleep the same amount every night as the average amount you slept from the 3rd to 6th days of the week.

40. Write code to calculate how many hours of sleep per year someone who sleeps 8 hours a night gets.

41. How many hours more/less than the 8 hours per night sleeper do you get in a year, assuming you sleep every night the average of the amount you slept on the first and last day of this week?

42. What is your total sleep deficit for the last week?

43. How many more hours per night, on average, do you need to sleep for the rest of the month so that, by the end of the month, you have a sleep deficit of zero?

Chapter 9

Calling functions

Learning goals

- Understand what functions are, and why they are awesome
- Understand how functions work
- Understand how to read function documentation

You have already worked with many R functions; commands like `getwd()`, `length()`, and `unique()` are all functions. You know a command is a function because it has parentheses, `()`, attached at its end.

Just as **variables** are convenient names used for calling *objects* such as vectors or dataframes, **functions** are convenient names for calling *processes* or *actions*. An R function is just a batch of code that performs a certain action.

Variables represent data, while functions represent code.

Most functions have three key components: (1) one or more inputs, (2) a process that is applied to those inputs, and (3) an output of the result. When you call a function in R, you are saying, "Hey R, take this information, do something to it, and return the result to me." You supply the function with the inputs, and the function takes care of the rest.

Take the function `mean()`, for example. `mean()` finds the arithmetic mean (i.e., the average) of a set of values.

```
x <- c(4,6,3,2,6,8,5,3) # create a vector of numbers
mean(x) # find their mean
[1] 4.625
```

In this command, you are feeding the function `mean()` with the input `x`.

Perhaps this analogy will help: When you think of functions, think of vending machines: you give a vending machine two inputs – your money and your snack selection – then it checks to see if your selection of choice is in stock, and if the money you provided is enough to pay for the snack you want. If so, the machine returns one output (the snack).

Base functions in R

There are hundreds of functions already built-in to R. These functions are called *"base functions"*. Throughout these modules, we have been – and will continue – introducing you to the most commonly used base functions.

You can access other functions through bundles of external code known as *packages*, which we explain in an upcoming module.

You can also write your *own* functions (and you will!). We provide an entire module on how to do this.

Note that not all functions require an input. The function `getwd()`, for example, does not need anything in its parentheses to find and return current your working directory.

Saving function output

You will almost always want to save the result of a function in a new variable. Otherwise the function just prints its result to the *Console* and `R` forgets about it.

You can store a function result the same way you store any value:

```
x <- c(4,6,3,2,6,8,5,3)
x_mean <- mean(x)
x_mean
[1] 4.625
```

Functions with multiple inputs

Note that `mean()` accepts a second input that is called `na.rm`. This is short for `NA.remove`. When this is set to `TRUE`, `R` will remove broken or missing values from the vector before calculating the mean.

```
x <- c(4,6,3,2,NA,8,5,3)  # note the NA
mean(x,na.rm=TRUE)
[1] 4.428571
```

If you tried to run these commands with `na.rm` set to `FALSE`, `R` would throw an error and give up.

Note that you provided the function `mean()` with two inputs, `x` and `na.rm`, and that you separated each input with a comma. This is how you pass multiple inputs to a function.

Function defaults

Note that many functions have default values for their inputs. If you do not specify the input's value yourself, `R` will assume you just want to use the default. In the case of `mean()`, the default value for `na.rm` is `FALSE`. This means that the following code would throw an error …

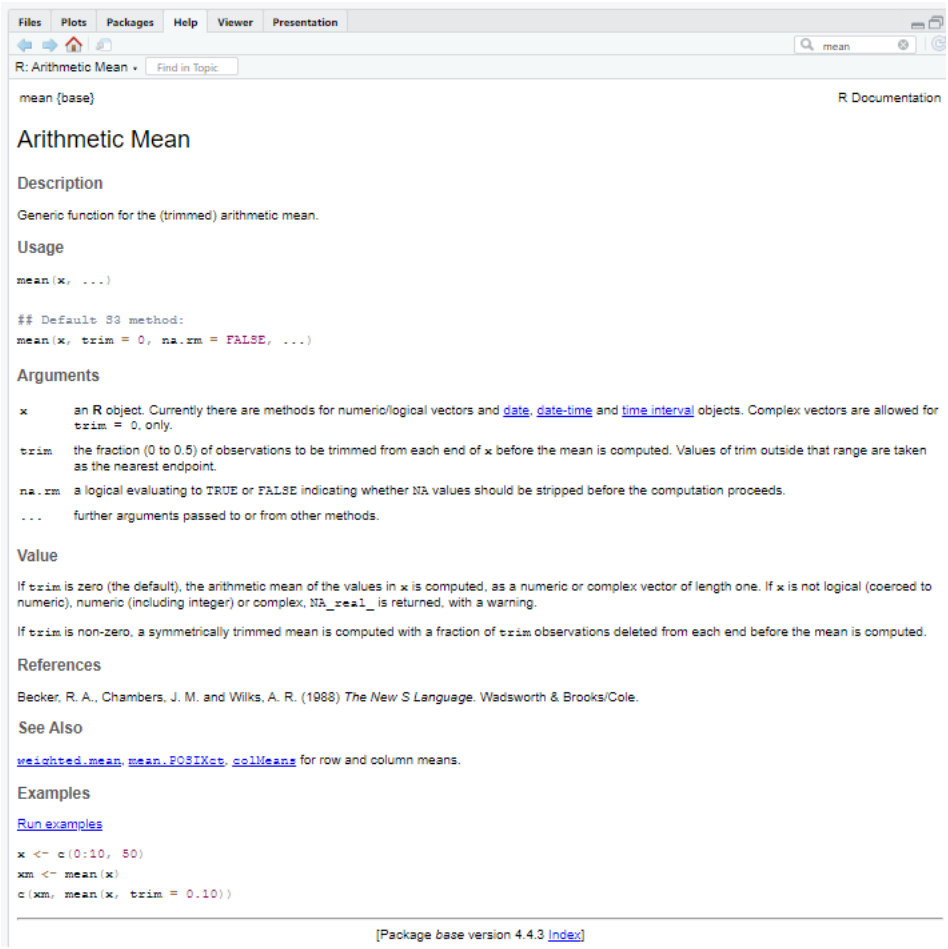
```
x <- c(4,6,3,2,NA,8,5,3)  # note the NA
mean(x)
[1] NA
```

Because `R` will assume you are using the default value for `na.rm`, which is `FALSE`, which means you do not want to remove missing values before trying to calculate the mean.

Function documentation (i.e., getting help)

Functions are designed to accept only a certain number of inputs with only certain names. To figure out what a function expects in terms of inputs, and what you can expect in terms of output, you can call up the function's help page:

When you enter this command, the help documentation for `mean()` will appear in the bottom right pane of your `RStudio` window:



Learning how to read this documentation is essential to becoming competent in using `R`.

Be warned: not all documentation is easy to understand! You will come to really resent poorly written documentation and really appreciate well-written documentation; the few extra minutes taken by the function's author to write good documentation saves users around the world hours of frustration and confusion.

- The `Title` and `Description` help you understand what this function does.
- The `Usage` section shows you how type out the function.
- The `Arguments` section lists out each possible argument (which in `R` lingo is another word for *input* or *parameter*), explains what that input is asking for, and details any formatting requirements.
- The `Value` section describes what the function returns as output.
- At the bottom of the help page, example code is provided to show you how the function works. You can copy and paste this code into your own script of *Console* and check out the results.

Note that more complex functions may also include a `Details` section in their documentation, which gives more explanation about what the function does, what kinds of inputs it requires, and what it returns.

Function examples

`R` comes with a set of base functions for **descriptive statistics**, which provide good examples of how functions work and why they are valuable.

We can use the same vector as the input for all of these functions:

```
x <- c(4,6,3,2,NA,8,9,5,6,1,9,2,6,3,0,3,2,5,3,3)  # note the NA
```

mean() has been explained above.

```
result <- mean(x,na.rm=TRUE)
result
[1] 4.210526
```

median() returns the median value in the supplied vector:

```
result <- median(x,na.rm=TRUE)
result
[1] 3
```

sd() returns the standard deviation of the supplied vector:

```
result <- sd(x,na.rm=TRUE)
result
[1] 2.594416
```

summary() returns a vector that describes several aspects of the vector's distribution:

```
result <- summary(x,na.rm=TRUE)
result
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  0.000   2.500   3.000   4.211   6.000   9.000       1
```

Exercises

Sock survey

You conducted a survey of your peers in this class, asking each of them how many pairs of socks they own. Most of your peers responded, but one person refused to tell you. Instead of giving you a number, they told you, "Nah!".

Your results look like this:

```
# People you asked
peers <- c('Keri','Eric','Joe','Ben','Matthew','Jim')

# Their responses
socks <- c(6,8,14,1,NA,4)
```

1. Calculate the average pairs of socks owned by your peers.

2. Calculate the total pairs of socks owned by cooperative students in this class.

3. Use code to find the name of the person who refused to tell you about their socks.

4. Use code to find the names of the people who were willing to cooperate with your survey.

5. Use code to find the name of the person with the most socks.

6. Use code to find the name of the person with the fewest socks.

Age survey

7. Create a vector named `years` with the years of birth of everyone in the room.

8. What is the average year of birth?

9. What is the median year of birth?

10. Create a vector called `ages` which is the (approximate) age of each person.

11. What is the minimum age?

12. What is the maximum age?

13. What is the median age?

14. What is the average age?

15. "Summarize" `ages`.

16. What is the range of ages?

17. What is the standard deviation of ages?

18. Look up help on the function `sort()`.

19. Created a vector called `sorted_ages`. It should be, well, sorted ages.

20. Look up the `length()` function.

21. How many people are the group?

22. Create an object called `old`. Assign to this object an age (such as 36) at which someone should be considered "old".

23. Create an object called `old_people`. This should be a boolean/logical vector indicating if each person is old or not.

24. Is the seventh person in `ages` old?

25. How many years from being old or young is person 12?

Rolling the dice

26. Look up the help page for the function `sample()`.

Here's an example of how this function works. This line of code will sample a single random number from the vector `1:10`.

```
sample(1:10,size=1)
[1] 10
```

This command will draw three random samples:

```
sample(1:10,size=3)
[1] 3 1 5
```

27. Use this function to simulate the rolling of a single die.

28. Now use this to simulate the rolling of a die 10 times. (*Note:* look at the `replace` input for `sample()`.)

29. Now use this to roll the die 10,000 times, and assign the result to a new variable. (*Note:* look at the `replace` input for `sample()`.)

30. Look up the help page for the function `table()`.

31. Use the `table()` function to ask whether the diel you rolled in question 29 is fair, or if it is weighted or biased toward a certain side. Can you describe what the `table()` function is doing?

30. Now use the `sample()` function to solve a different problem: your friends want to order take out from the Tavern, but no one in your group of 4 wants to be the one to go pick it up. Write code that will randomly select who has to go.

Chapter 10

Subsetting & filtering

Learning goals

- Understand how to subset / filter data

You have been introduce to subsetting and filtering briefly in previous modules, but it is such an important concept that we want to devote an entire module to practicing it.

Subsetting with indices

You have already learned that certain elements of a vector can be called by specifying an index:

```
x <- 55:65

# Call x without subsetting
x
 [1] 55 56 57 58 59 60 61 62 63 64 65
```

```
# Now call only the third element of x
x[3]
[1] 57
```

Remember: brackets indicate that you don't want everything from a vector; you just want certain elements. 'I want `x`, *but not all of it.*'

You can also subset an object by calling multiple indices:

```
# Now call the third, fourth, and fifth element of x
x[c(3,4,5)]
[1] 57 58 59
```

```
# Another way of doing the same thing:
x[3:5]
[1] 57 58 59
```

Subsetting with booleans

You can also subset objects with 'booleans'. This will eventually be your most common way of filtering data, by far.

Recall that boolean / logical data have two possible values: `TRUE` or `FALSE`. For example:

```
# Store Joe's age
joes_age <- 35

# Set the cutoff for old age
old_age <- 36

# Ask whether Joe is old
joes_age >= old_age
[1] FALSE
```

Recall also that you can calculate whether a condition is `TRUE` or `FALSE` on multiple elements of a vector. For example:

```
# Build a vector of multiple ages
ages <- c(10, 20, 30, 40, 50, 60)

# Set the cutoff for old age
old_age <- 36

# Ask which ages are considered old
ages >= old_age
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Boolean vectors are super useful for subsetting. Think of 'subsetting' as keeping only those elements of a vector for which a condition is `TRUE`.

```
x <- 55:59

# Call x without subsetting
x
[1] 55 56 57 58 59
```

```
# Now subset to the second, third, and fourth element
x[c(FALSE, TRUE, TRUE, TRUE, FALSE)]
[1] 56 57 58
```

That command returned elements for which the subetting vector was `TRUE`.

This is equivalent to...

```
x[2:4]
[1] 56 57 58
```

You can also get the same result using a logical test, since logical tests return boolean values:

```
# Develop your logical test: ask which values of x are in the vector 56:58
x %in% c(56,57,58)
[1] FALSE  TRUE  TRUE  TRUE FALSE
```

```
# Now plug taht test it into the subsetting brackets
x[ x %in% c(56,57,58) ]
[1] 56 57 58
```

This methods gets really useful when you are working with bigger datasets, such as this one:

```
# Make a large dataset of random numbers
y <- sample(1:1000,size=100)
length(y)
[1] 100
```

```
range(y)
[1]   1 992
```

With a dataset like this, you can use a boolean filter to figure out how many values are greater than, say, 90.

First, develop your logical test, which will tell you whether each value in the vector is greater than 90:

```
# Develop your logical test,
y > 90
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [25] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
 [37]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
 [49] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
 [61]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
 [73]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
 [85]  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [97] FALSE  TRUE  TRUE  TRUE
```

Now, to get the values corresponding to each `TRUE` in this list, plug your logical test into your subsetting brackets.

```
y[y > 90]
 [1] 819 710 357 361 655 914 546 238 400 169 682 607 482 452 691 894 101 950 715
[20] 468 106 927 187 620 370 260 737 792 807 644 706 383 665 868 480 111 766 884
[39] 356 871 214 915 637 103 590 292 159 611 207 310 247 270 555 190 812 848 827
[58] 532 788 952 472 410 605 237 501 362 720 177 397 757  98 803 992 139 477 862
[77] 557 398 337 243 639 733 929 938 442
```

Here's another way you can do the same thing:

```
# Save the result of your logical test in a new vector
verdicts <- y > 90

# Use that vector to subset y
y[verdicts]
 [1] 819 710 357 361 655 914 546 238 400 169 682 607 482 452 691 894 101 950 715
[20] 468 106 927 187 620 370 260 737 792 807 644 706 383 665 868 480 111 766 884
[39] 356 871 214 915 637 103 590 292 159 611 207 310 247 270 555 190 812 848 827
[58] 532 788 952 472 410 605 237 501 362 720 177 397 757  98 803 992 139 477 862
[77] 557 398 337 243 639 733 929 938 442
```

You can use double logical tests too. For example, what if you want all elements between the values 70 and 90?

```
verdicts <- y > 70 & y < 90
y[verdicts]
integer(0)
```

Review assignment

1. Create a vector named `nummies` of all numbers from 1 to 100

2. Create another vector named `little_nummies` which consists of all those numbers which are less than or equal to 30

3. Create a boolean vector named `these_are_big` which indicates whether each element of `nummies` is greater than or equal to 70

4. Use `these_are_big` to subset `nummies` into a vector named `big_nummies`

5. Create a new vector named `these_are_not_that_big` which indicates whether each element of `nummies` is greater than 30 and less than 70. You'll need to use the `&` symbol.

6. Create a new vector named `meh_nummies` which consists of all `nummies` which are greater than 30 and less than 70.

7. How many numbers are greater than 30 and less than 70?

8. What is the sum of all those numbers in `meh_nummies`

Chapter 11

Dataframes

Learning goal

- Practice exploring, summarizing, and filtering dataframes

A vector is the most basic data structure in R, and the other structures are built out of vectors. But, as a data scientist, the most common data structure you will be working with – by far – is a **dataframe**.

A dataframe, essentially, is a spreadsheet: a dataset with rows and columns, in which each column represents is a vector of the same class of data.

Here is what a dataframe looks like:

```
# Using one of R's built-in datasets
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

In this dataframe, each row pertains to a unique iris plant. The columns contain related information about each individual plant.

Here's another data.frame, built from scratch, which shows that dataframes are just a group of vectors:

```
x <- 25:29
y <- 55:59
df <- data.frame(x,y)
df
   x  y
1 25 55
2 26 56
3 27 57
4 28 58
5 29 59
```

In this command, we used the `data.frame()` function to combine two vectors into a dataframe with two columns named `x` and `y`. R then saved this result in a new variable named `df`. When we call `df`, R shows us the dataframe.

The great thing about dataframes is that they allow you to relate different data types to each other.

```
df <- data.frame(name=c("Ben","Joe","Eric"),
                 height=c(75,73,80))
df
  name height
1  Ben     75
2  Joe     73
3 Eric     80
```

This dataframe has one column of class `character` and another of class `numeric`.

Subsetting & exploring dataframes

To explore dataframes, let's use a dataset on fuel mileage for all cars sold from 1985 to 2014.

```
# need to install first install.packages('fueleconomy')
library(fueleconomy)
data(vehicles)
head(vehicles)
     id  make       model year           class           trans
1 13309 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
2 13310 Acura 2.2CL/3.0CL 1997 Subcompact Cars    Manual 5-spd
3 13311 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
4 14038 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
5 14039 Acura 2.3CL/3.0CL 1998 Subcompact Cars    Manual 5-spd
6 14040 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
              drive cyl displ    fuel hwy cty
1 Front-Wheel Drive   4   2.2 Regular  26  20
2 Front-Wheel Drive   4   2.2 Regular  28  22
3 Front-Wheel Drive   6   3.0 Regular  26  18
4 Front-Wheel Drive   4   2.3 Regular  27  19
5 Front-Wheel Drive   4   2.3 Regular  29  21
6 Front-Wheel Drive   6   3.0 Regular  26  17
```

To look at this dataframe in full, you call display it in a separate tab within `RStudio` using the `View()` function:

```
View(vehicles)
```

A dataframe has rows of data organized into columns. In this dataframe, each row pertains to a single vehicle make/model – i.e., a single *observation*. Each column pertains to a single *type* of data. Columns are named in the *header* of the dataframe.

All the same useful exploration and subsetting functions that applied to vectors now apply to dataframes. In addition to those functions you already know, let's add some new functions to your inventory of useful functions.

Exploration

`head()` and **`tail()`** summarize the beginning and end of the object:

```
head(vehicles)
     id  make       model year           class           trans
1 13309 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
```

```
2 13310 Acura 2.2CL/3.0CL 1997 Subcompact Cars    Manual 5-spd
3 13311 Acura 2.2CL/3.0CL 1997 Subcompact Cars Automatic 4-spd
4 14038 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
5 14039 Acura 2.3CL/3.0CL 1998 Subcompact Cars    Manual 5-spd
6 14040 Acura 2.3CL/3.0CL 1998 Subcompact Cars Automatic 4-spd
              drive cyl displ    fuel hwy cty
1 Front-Wheel Drive   4   2.2 Regular  26  20
2 Front-Wheel Drive   4   2.2 Regular  28  22
3 Front-Wheel Drive   6   3.0 Regular  26  18
4 Front-Wheel Drive   4   2.3 Regular  27  19
5 Front-Wheel Drive   4   2.3 Regular  29  21
6 Front-Wheel Drive   6   3.0 Regular  26  17

tail(vehicles)
         id make              model year            class        trans
33437 28868 Yugo GV Plus/GV/Cabrio 1990 Minicompact Cars Manual 4-spd
33438  6635 Yugo GV Plus/GV/Cabrio 1990  Subcompact Cars Manual 5-spd
33439  3157 Yugo             GV/GVX 1987  Subcompact Cars Manual 4-spd
33440  5497 Yugo             GV/GVX 1989  Subcompact Cars Manual 4-spd
33441  5498 Yugo             GV/GVX 1989  Subcompact Cars Manual 5-spd
33442  1745 Yugo       Gy/yugo GVX 1986 Minicompact Cars Manual 4-spd
                  drive cyl displ    fuel hwy cty
33437 Front-Wheel Drive   4   1.3 Regular  27  21
33438 Front-Wheel Drive   4   1.3 Regular  28  23
33439 Front-Wheel Drive   4   1.1 Regular  29  24
33440 Front-Wheel Drive   4   1.1 Regular  29  24
33441 Front-Wheel Drive   4   1.3 Regular  28  23
33442 Front-Wheel Drive   4   1.1 Regular  29  22
```

names() tells you the column names:

```
names(vehicles)
 [1] "id"    "make"  "model" "year"  "class" "trans" "drive" "cyl"   "displ"
[10] "fuel"  "hwy"   "cty"
```

nrow(), **ncol()**, and **dim()** tell you about the dimensions of your dataframe:

```
nrow(vehicles)
[1] 33442
```

```
ncol(vehicles)
[1] 12
```

```
dim(vehicles)
[1] 33442    12
```

Note that **length()** does not work the same on dataframes as it does with vectors. In dataframes, **length()** is the equivalent of **ncol()**; it will *not* give you the number of rows in a dataset.

Importantly, you can use **is.na()** to ask whether columns or rows contain NAs:

```
# Check for NAs

# Which rows in the `hwy` column have NA's?
which(is.na(vehicles$hwy))
```

```
integer(0)

# (No NAs in that column!)

# What about rows in the `cyl` column?
which(is.na(vehicles$cyl))
 [1]  1232  1233  2347  3246  3247  3248  6115  6116  6533  7783  7784  8472
[13] 10613 10614 11696 11697 12411 12412 12413 12928 12929 12934 12935 12944
[25] 16429 16430 21070 23472 23473 23474 24485 24486 24487 24488 24489 26150
[37] 28628 28704 28705 28706 28707 28708 28709 29314 29315 30023 30024 30025
[49] 30026 30027 30028 31063 31064 31065 31066 31067 31068 31069

# (lots of NAs in that column!)
```

Subsetting

Recall that dataframes are filtered by row and/or column using this format: **`dataframe[rows,columns]`**. To get the third element of the second column, for example, you type `dataframe[3,2]`.

```
vehicles[3,2]
[1] "Acura"
```

Note that the comma is necessary even if you do not want to specify columns. If you try to type this …

```
vehicles[3]
```

…R will assume you are asking for the third column, not the third row.

To filter a dataframe to multiple values, you can specify vectors for the `row` and `column`

```
vehicles[1:3,11:12] # can use colons
  hwy cty
1  26  20
2  28  22
3  26  18
vehicles[1:3,c(1,11:12)]  # can use c()
     id hwy cty
1 13309  26  20
2 13310  28  22
3 13311  26  18
```

Columns can also be called according to their names. Use the **`$`** sign to specify a column.

```
vehicles$hwy[1:5]
[1] 26 28 26 27 29
```

Note that when you use a `$`, you will not need to use a comma within your brackets. If you try to run this …

```
vehicles$hwy[1:5,]
```

…R will throw a fit.

Also recall that you can use logical tests, which return boolean values `TRUE` or `FALSE`, to filter dataframes to rows that meet certain conditions. For example, to filter to only the rows for cars with better than 100 mpg, you can use this syntax:

```
# Build your logical test
verdicts <- vehicles$hwy > 100

# Subset with booleans
vehicles[verdicts,2:3]
           make    model
6533  Chevrolet Spark EV
10613      Fiat     500e
10614      Fiat     500e
16429     Honda   Fit EV
16430     Honda   Fit EV
24487    Nissan     Leaf
24488    Nissan     Leaf
24489    Nissan     Leaf
28628     Scion    iQ EV
```

Or you can write all this in a single line, to be more efficient:

```
vehicles[ vehicles$hwy > 100  , 2:3]
           make    model
6533  Chevrolet Spark EV
10613      Fiat     500e
10614      Fiat     500e
16429     Honda   Fit EV
16430     Honda   Fit EV
24487    Nissan     Leaf
24488    Nissan     Leaf
24489    Nissan     Leaf
28628     Scion    iQ EV
```

Recall that the logical test is returning a bunch of `TRUE`'s and `FALSE`'s, one for each row of `vehicles`. Only the `TRUE` rows will be returned.

Summarizing

The same summary functions that you have used for vectors work for the columns in dataframes, since each column is also a vector. Check it out:

```
min(vehicles$hwy)
[1] 9
```

```
max(vehicles$hwy)
[1] 109
```

```
mean(vehicles$cty)
[1] 17.491
```

```
sd(vehicles$cty)
[1] 5.582174
```

```
str(vehicles$make)
 chr [1:33442] "Acura" "Acura" "Acura" "Acura" "Acura" "Acura" "Acura" ...
```

```
class(vehicles$hwy)
[1] "numeric"
```

You can also use the **summary()** function, which provides summary statistics for each column in your dataframe:

```
summary(vehicles)
       id             make              model                year     
 Min.   :    1   Length:33442       Length:33442       Min.   :1984  
 1st Qu.: 8361   Class :character   Class :character   1st Qu.:1991  
 Median :16724   Mode  :character   Mode  :character   Median :1999  
 Mean   :17038                                         Mean   :1999  
 3rd Qu.:25265                                         3rd Qu.:2008  
 Max.   :34932                                         Max.   :2015  

    class              trans              drive                cyl        
 Length:33442       Length:33442       Length:33442       Min.   : 2.000  
 Class :character   Class :character   Class :character   1st Qu.: 4.000  
 Mode  :character   Mode  :character   Mode  :character   Median : 6.000  
                                                          Mean   : 5.772  
                                                          3rd Qu.: 6.000  
                                                          Max.   :16.000  
                                                          NA's   :58      
     displ            fuel                hwy              cty        
 Min.   :0.000   Length:33442       Min.   :  9.00   Min.   :  6.00  
 1st Qu.:2.300   Class :character   1st Qu.: 19.00   1st Qu.: 15.00  
 Median :3.000   Mode  :character   Median : 23.00   Median : 17.00  
 Mean   :3.353                      Mean   : 23.55   Mean   : 17.49  
 3rd Qu.:4.300                      3rd Qu.: 27.00   3rd Qu.: 20.00  
 Max.   :8.400                      Max.   :109.00   Max.   :138.00  
 NA's   :57                                                          
```

The function **unique()** returns unique values within a column:

```
unique(vehicles$fuel)
 [1] "Regular"                     "Premium"                    
 [3] "Diesel"                      "Premium or E85"             
 [5] "Electricity"                 "Gasoline or E85"            
 [7] "Premium Gas or Electricity"  "Gasoline or natural gas"    
 [9] "CNG"                         "Midgrade"                   
[11] "Regular Gas and Electricity" "Gasoline or propane"        
[13] "Premium and Electricity"    
```

Finally, the **order()** function helps you sort a dataframe according to the values in one of its columns.

```
# Sort dataframe by highway mileage
# Only keep certain columns
vehicles_sorted <- vehicles[order(vehicles$hwy),
                            c(2,3,4,10:12)]
head(vehicles_sorted)
              make                  model year    fuel hwy cty
397   Aston Martin                Lagonda 1985 Regular   9   7
398   Aston Martin                Lagonda 1985 Regular   9   7
406   Aston Martin Saloon/Vantage/Volante 1985 Regular   9   7
408   Aston Martin Saloon/Vantage/Volante 1985 Regular   9   7
27725  Rolls-Royce               Camargue 1987 Regular   9   7
27726  Rolls-Royce             Continental 1987 Regular   9   7
```

Reverse the order by wrapping **rev()** around the order() call:

```
vehicles_sorted <- vehicles[rev(order(vehicles$hwy)),
                            c(2,3,4,10:12)]
head(vehicles_sorted)
           make    model year        fuel hwy cty
6533  Chevrolet Spark EV 2014 Electricity 109 128
10614      Fiat     500e 2014 Electricity 108 122
10613      Fiat     500e 2013 Electricity 108 122
28628     Scion    iQ EV 2013 Electricity 105 138
16430     Honda   Fit EV 2014 Electricity 105 132
16429     Honda   Fit EV 2013 Electricity 105 132
```

Creating dataframes

As shown above, to create a new dataframe, use the data.frame() function.

```
             car mgp_hwy mpg_city
100 Acura Legend      23       15
101 Acura Legend      22       17
102 Acura Legend      23       16
103 Acura Legend      21       16
104 Acura Legend      22       17
105 Acura Legend      23       16
106 Acura Legend      24       16
```

Note how the columns were named in the data.frame() call, and that each column is separated by a comma.

You can also stage an empty dataframe, which sounds useless but will become very useful as you start working with for loops and other higher-order R tools.

```
df <- data.frame()
df
data frame with 0 columns and 0 rows
```

To coerce an object into a format that R interprets as a dataframe, use **as.dataframe()**:

```
df <- as.data.frame(vehicles)
df[1:4,1:4]
     id  make       model year
1 13309 Acura 2.2CL/3.0CL 1997
2 13310 Acura 2.2CL/3.0CL 1997
3 13311 Acura 2.2CL/3.0CL 1997
4 14038 Acura 2.3CL/3.0CL 1998
```

Modifying dataframes

Combining dataframes

To bind multiple dataframes together by row, use **rbind()**:

```
# Build up a dataframe
df1 <- data.frame(name=c("Ben","Joe","Eric","Isabelle"),
                  instrument=c("Nose harp","Concertina","Ukelele","Drums"))
df1
      name instrument
1      Ben  Nose harp
2      Joe Concertina
3     Eric    Ukelele
4 Isabelle      Drums
```

```
# Build up a second dataframe
df2 <- data.frame(name=c("Matthew"),
                  instrument=c("Washboard"))
```

```
# Combine those dataframes together
rbind(df1,df2)
      name instrument
1      Ben  Nose harp
2      Joe Concertina
3     Eric    Ukelele
4 Isabelle      Drums
5  Matthew  Washboard
```

Note that to be combined, two dataframes have to have the exact same number of columns and the exact same column names.

The only exception to this is adding a dataframe with content an empty dataframe. That can work, and that will be helpful in the **Deep R** modules ahead.

```
df <- data.frame() # stage empty dataframe

df1 <- data.frame(name=c("Ben","Joe","Eric","Isabelle"),
                  instrument=c("Nose harp","Concertina","Ukelele","Drums"))

df <- rbind(df,df1)

df
      name instrument
1      Ben  Nose harp
2      Joe Concertina
3     Eric    Ukelele
4 Isabelle      Drums
```

You can also bind multiple dataframes together by column, using **cbind()**:

```
df1 <- data.frame(name=c("Ben","Joe","Eric","Isabelle"),
                  instrument=c("Nose harp","Concertina","Ukelele","Drums"))

df <- data.frame(age=c(33,35,35,20), home=c("Canada","Spain","USA","USA"))

df <- cbind(df,df1)

df
  age   home     name instrument
1  33 Canada      Ben  Nose harp
2  35  Spain      Joe Concertina
```

```
3  35    USA     Eric    Ukelele
4  20    USA Isabelle      Drums
```

Note that to be combined, two dataframes have to have the exact same number of rows and the exact same column names.

Adding columns

To create a new column for a dataframe, use the $ symbol and provide the name of the new column:

```
df$x_factor <- c(3,20,60,40)

df
  age   home     name instrument x_factor
1  33 Canada      Ben  Nose harp        3
2  35  Spain      Joe Concertina       20
3  35    USA     Eric    Ukelele       60
4  20    USA Isabelle      Drums       40
```

Altering values

To alter certain values in the dataframe, you can assign new values to a subset of your dataframe.

Here are four ways to do the same thing: upating Isabelle's X-factor:

Option 1: Subsetting a single column

```
df$x_factor[4] <- 70
```

Option 2: Subsetting both rows and columns

```
df[4,5] <- 70
```

Option 3: Subsetting a column based on a logical test

```
df$x_factor[df$name == 'Isabelle'] <- 70
```

Option 3: Subsetting row and columns using logical tests

```
df[df$name == 'Isabelle', names(df) == 'x_factor'] <- 70
```

```
df
  age   home     name instrument x_factor
1  33 Canada      Ben  Nose harp        3
2  35  Spain      Joe Concertina       20
3  35    USA     Eric    Ukelele       60
4  20    USA Isabelle      Drums       70
```

Exercises

Reading for errors

What is wrong with these commands? Why will each of them throw an error if you run them, and how can you fix them?

1. `vehicles[1,15,]`

2. `vecihles[1:5,]`

3. `vehicles$hwy[15,]`

4. `vehicles[1:5,1:13]`

Subsetting and filtering

5. Subset one field according to a logical test: With no more than two lines of code, get the number of Honda cars in the `vehicles` dataset.

6. Subset one field according to a logical test for a different field: In a single line of code, show the mileages of all the Toyotas in the dataset.

7. Subset a dataframe to a single subgroup: In a single line of code, determine how many differet car makes/models were produced in 1995.

8. Get the mean value for a subgroup of data: What is the average city mileage for Subaru cars in the dataset?

9. Subset a dataframe to only data from between two values: According to this dataset, how many different car makes/models have been produced with highway mileages between 30 and 40 mpg?

10. Subset by removing `NA`s: Create a new version of the `vehicles` dataframe that does not have any `NA`s in the `trans` column.

Creating dataframes

11. Create a vector called `people` of 5 peoples names from the class.

12. Show with code how many people are in your vector

13. Create another vector called `height` which is the number of centimeters tall each of those 5 people are.

14. Combine these two vectors into a data frame.

Now let's create a new object named `animals`. This is going to be a dataframe with 4 different columns: `species`, `weight` (in kg), `color`, `veg` (whether or not the animal is a vegetarian / herbivore).

15. Come up with five species to add to your dataframe and list them in a vector named `species`.

16. Make the other vectors with details about those species in the correct order.

17. Combine these vectors into a dataframe named `animals`.

Altering dataframes

18. Add a column to your `animals` dataframe named `rank`, which ranks each animal from your least favorite (0) to your most favorite (5).

19. Now write code to manually switch the ranking for your top two favorite animals.

20. What is the mean weight of the herbivorous animals that you listed, if any?

21. What is the mean weight of the omnivorous/carnivorous animals that you listed?

Chapter 12

Packages

Learning goals

- Learn what `R` packages are and why they are awesome
- Learn how to find and read about the packages installed on your machine
- Learn how to install `R` packages from `CRAN`
- Learn how to install `R` packages from `GitHub`

As established in the **Calling functions** module, `R` comes with hundreds of built-in base functions and datasets ready for use. You can also write your *own* functions, which we will cover in an upcoming module.

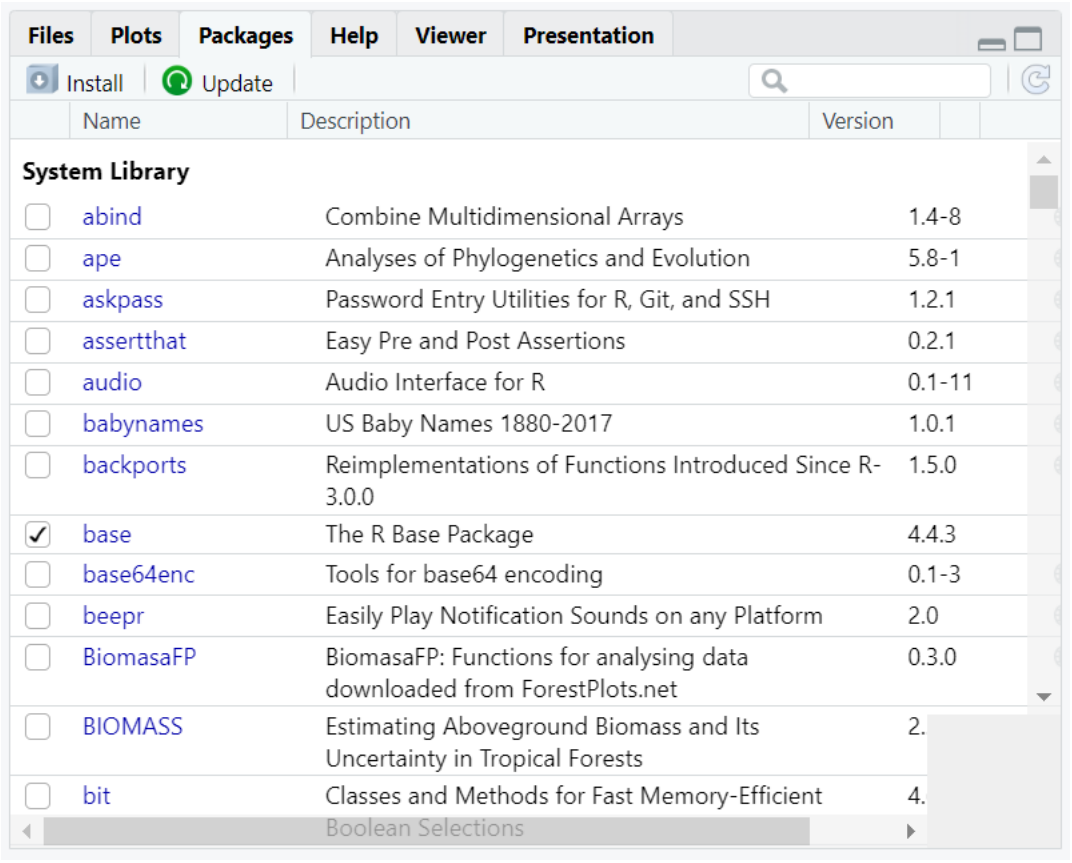
You can also access thousands of other functions and datasets through bundles of external code known as **packages**. Packages are developed and shared by `R` users around the world – a global community working together to increase `R`'s versatility and impact.

Some packages are designed to be broadly useful for almost any application, such as the packages you will be learning in this course (`ggplot`, `dplyr`, `stringr`, etc.). Such packages make it easier and more efficient to do your work with `R`.

Others are designed for niche problems that can be made much more doable with specialized functions or datasets. For example, the package `PBSmapping` contains shoreline, seafloor, and oceanographic datasets and custom mapping functions that make it easier for marine scientists at the Pacific Biological Station (PBS) in British Columbia, Canada, to carry out their work.

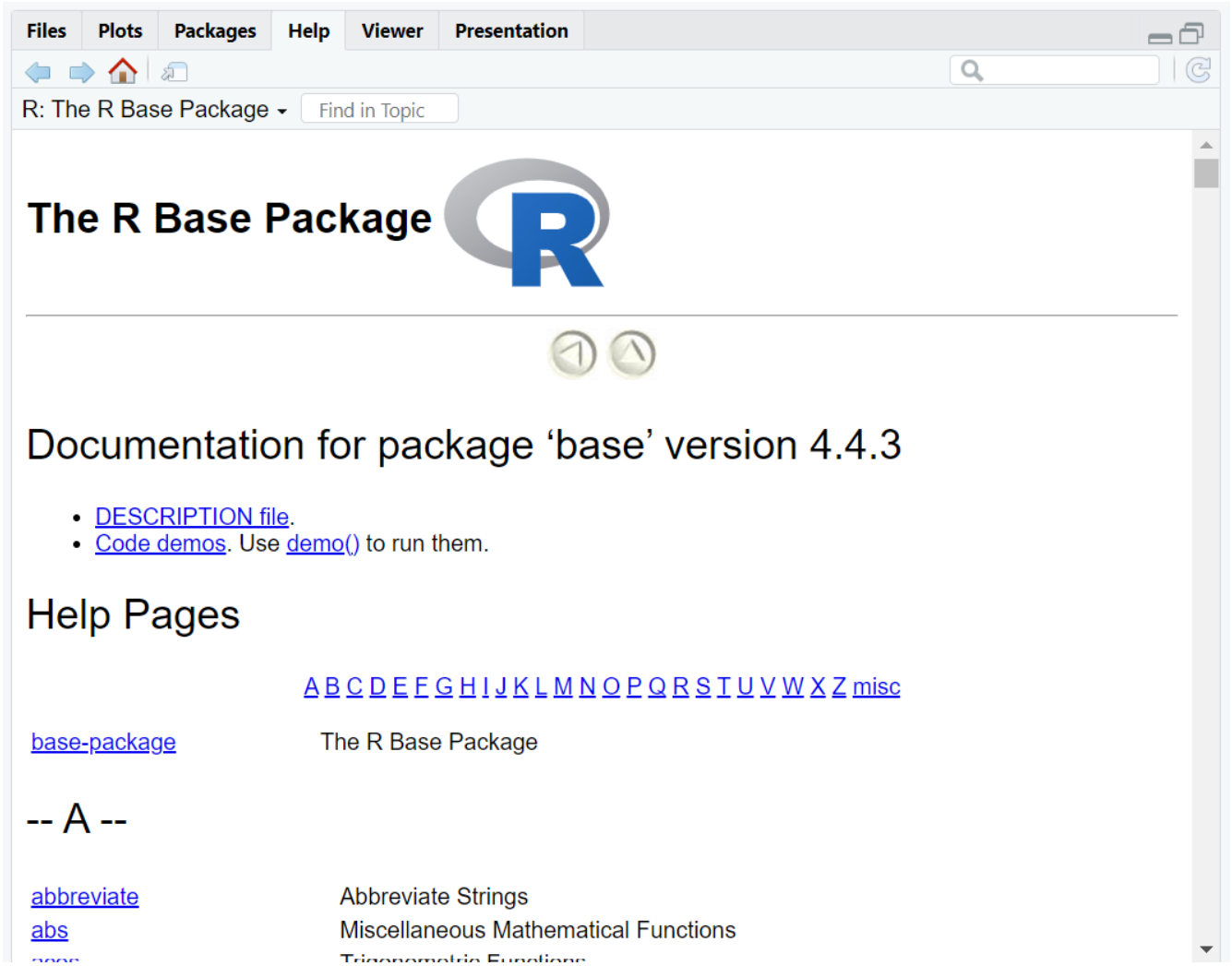
Packages you already have

In `RStudio`, look to the pane in the bottom right and click on the *Packages* tab. You should see something like this:



This is displaying all the packages already installed in your system.

If you click on one of these packages (try the `base` package, for example), you will be taken to a list of all the functions and datasets contained within it.



When you click on one of these functions, you will be taken to the help page for that function. This is the equivalent of typing `? <function_name>` into the *Console*.

Installing a new package

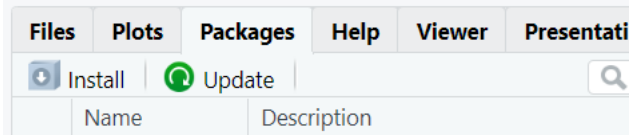
There are a couple ways to download and install a new `R` package on your computer. Most packages are available from an open-source repository known as `CRAN` (which stands for Comprehensive `R` Archive Network). However, an increasingly common practice is to release packages on a public repository such as `GitHub`.

Installing from `CRAN`

You can install `CRAN` packages one of two ways:

Through clicks:

In `RStudio`, in the bottom-right pane, return to the *Packages* tab. Click on the "Install" button.



You can then search for the package you wish to install then click **Install**.

Through code:

You can download packages from the *Console* using the `install.packages()` function.

```
install.packages('fun')
```

Note that the package name must be in quotation marks.

Installing from GitHub

To install packages from `GitHub`, you must first download a `CRAN` package that makes it easy to do so:

```
install.packages("devtools")
```

Most packages on `GitHub` include instructions for downloading it on its `GitHub` page.

For example, visit this `GitHub` page to see the documentation for the package **wesanderson**, which provides color palette themes based upon Wes Anderson's films. On this site, scroll down and you will find instructions for downloading the package. These instructions show you how to install this package from your `R` *Console*:

```
devtools::install_github("karthik/wesanderson")
```

Now go to your *Packages* tab in the bottom-right pane of `RStudio`, scroll down to find the **wesanderson** package, and click on it to check out its functions.

Loading an installed package

There is a difference between *installed* and *loaded* packages. Go back to your *Packages* tab. Notice that some of the packages have a checked box next to their names, while others don't.

These checked boxes indicate which packages are currently *loaded*. All packages in the list are *installed* on your computer, but only the checked packages are *loaded*, i.e., ready for use.

To load a package, use the `library()` function.

```
library(fun)
library(wesanderson)
```

Now that your new packages are loaded, you can actually use their functions.

To emphasize: a package is installed *only once*, but you `library()` the package in *each and every* script that uses it. Think of a package as camping gear. Like an `R` package, camping gear helps you do cool things that you can't really do with the regular stuff in your closet. And, like an `R` package, you only need to install (i.e., purchase) your gear once; but it is useless unless you pack it in your car (i.e., `library()` it) *every time* you go on a trip.

Calling functions from a package

Most functions from external packages can be used by simply typing the name of the function. For example, the package `fun` contains a function for generating a random password:

```
random_password(length=24)
[1] "eEhly5.NFm,u&Lx\\qc$'n(%g"
```

Sometimes, however, R can get confused if a new package contains a function that has the same name of some function from a different package. If R seems confused about a function you are calling, it can help to specify which package the function can be found in. This is done using the syntax <package_name>::<function_name>. For example, the following command is a fine alternative to the command above:

```
fun::random_password(length=24)
[1] "?t]vhC!inFZP\\Yu.|r8H=xJT"
```

Note that this was done in the example above using the devtools package.

Side notes

Package dependencies

Most packages contain functions that are built using functions built from other packages. Those new functions depend on the functions from those other packages, and that's why those other packages are known as *dependencies*. When you install one function, you will notice that R typically has to install several other packages at the same time; these are the dependencies that allow the package of interest to function.

Package versions

Packages are updated regularly, and sometimes new versions can break the functions that use it as a dependency. Sometimes you may have to install a new version (*or sometimes an older version!*) of a dependency in order to get your package of interest to work as desired.

Review: the workflow for using a package

To review how to use functions from a non-base package in R, follow these steps (examples provided:)

1. Install the package *once.*

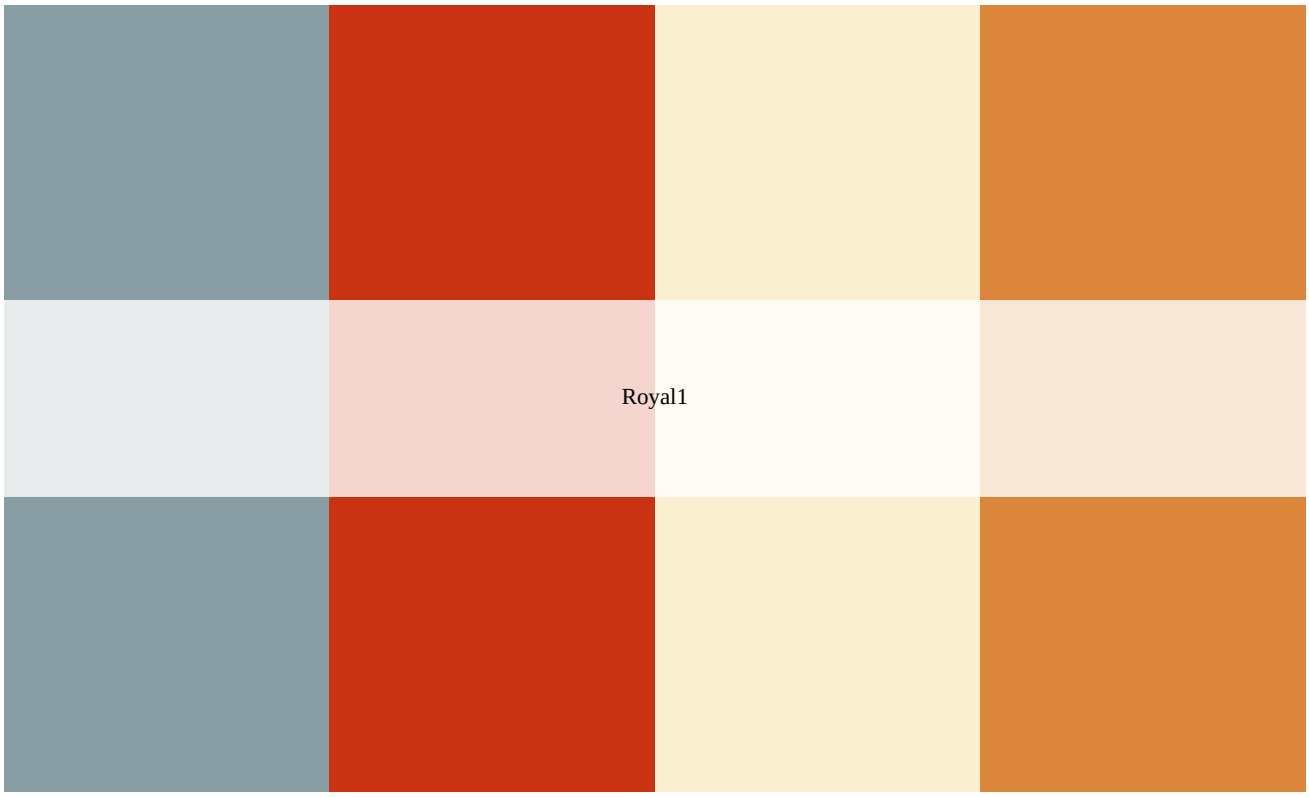
```
# Example from CRAN
install.packages("wesanderson")

# Example from GitHub
devtools::install_github("karthik/wesanderson")
```

2. Load the package *in each script.*

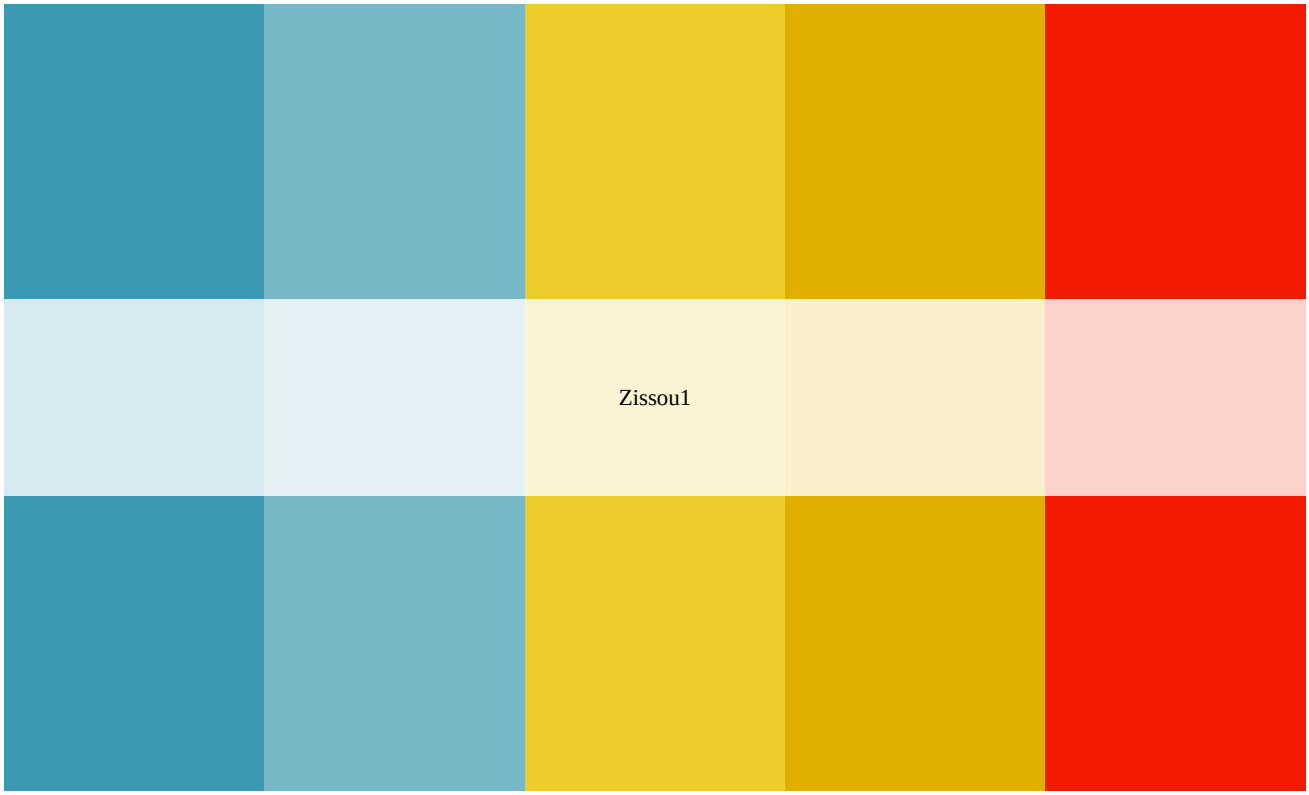
```
# Example
library(wesanderson)
```

3. Call the function.

```
wes_palette("Royal1")
```




```
wesanderson::wes_palette("Zissou1")
```



(This function creates a plot displaying the different colors contained within the specified palette.)

4. Get help with the question mark: `?`

```
?wes_palette
```

Exercises

Let's install some packages:

1. Install the `babynames` package.

2. Install `ggplot2`.

3. Install `dplyr`.

4. Install `RColorBrewer`.

5. Install `tidyr`.

6. Install `gapminder`.

7. Install `readr`.

8. Install `gsheet`.

9. Install `readxl`.

10. Write 7 lines of code which *load* the above packages.

(PART) Basic `R` workflow

Chapter 13

Importing data

Learning goals

- How to load, or "read", your data into `R`
- How to format your data for easily importing data in `R`
- Understand what a `.csv` file is, and why they are important in data science
- How to set up your project directory and read data from other folders

To work with your own data in `R`, you need to load your data in `R`'s memory. This is called **reading in** your data.

Reading in data

The general workflow for reading in data is as follows:

1. In `RStudio`, set your working directory.
2. Place your data file in your working directory. (See the section below if you want to keep your data somewhere else.)
3. In your `R` script, read in your data file with one of the core functions below.

You can use this simple data file, , to practice.

Core functions for reading data

To become agile in reading various types of data into `R`, here are five key functions you should know:

`readr::read_csv()`

Reading in data is simple and easy if your data are saved as a `.csv`, a comma-separated file. You can find functions for reading all sorts of file types into `R`, but the quickest way to begin working with your own data in `R` is to maintain that data in `.csv`'s.

The function `read_csv()`, from a package named `readr`, becomes useful when you begin working with (1) data from the internet, (2) data within the `tidyverse`, which you will be introduced to in the next module, and/or (2) very large dataset, since it reads data much more quickly and provides progress updates along the way.

Here's an example of reading a file directly from the internet …

```
library(readr)
df <- read_csv('https://raw.githubusercontent.com/databrew/intro-to-data-science/main/data/deaths.csv')
df
# A tibble: 891 x 12
   PassengerId Survived Pclass Name   Sex     Age SibSp Parch Ticket  Fare Cabin
         <dbl>    <dbl>  <dbl> <chr>  <chr> <dbl> <dbl> <dbl> <chr>  <dbl> <chr>
 1           1        0      3 Braun~ male     22     1     0 A/5 2~  7.25 <NA>
 2           2        1      1 Cumin~ fema~    38     1     0 PC 17~ 71.3  C85
 3           3        1      3 Heikk~ fema~    26     0     0 STON/~  7.92 <NA>
 4           4        1      1 Futre~ fema~    35     1     0 113803 53.1  C123
 5           5        0      3 Allen~ male     35     0     0 373450  8.05 <NA>
 6           6        0      3 Moran~ male     NA     0     0 330877  8.46 <NA>
 7           7        0      1 McCar~ male     54     0     0 17463  51.9  E46
 8           8        0      3 Palss~ male      2     3     1 349909 21.1  <NA>
 9           9        1      3 Johns~ fema~    27     0     2 347742 11.1  <NA>
10          10        1      2 Nasse~ fema~    14     1     0 237736 30.1  <NA>
# i 881 more rows
# i 1 more variable: Embarked <chr>
```

Note that when you use `read_csv()` instead of `read.csv()`, your data are read in as a `tibble` instead of a dataframe. You will be introduced to `tibbles` in the next modules on dataframes; for the time being, think of a `tibble` as a fancy version of a dataframe that can be treated exactly as a regular dataframe.

read.csv()

This function, `read.csv()`, is the base function for reading in a `.csv`. It is strictly used for reading in local files (not from the internet).

This function reads in your data file as a dataframe. Save your dataset into `R`'s memory using a variable (in this case, `df`).

```
df <- read.csv("data/super_data.csv")
df
```

```
  patient_id height_in weight_lb                  comment
1          1        74       135            not very nice
2          2        56       112               kinda cute
3          3        59       156 kinda cute but had a ring
4          4        43       102                so small!
```

The `read.csv()` function has plenty of other inputs in the event that your data file is unable to follow the formatting rules outlined above (see `?read.csv()`). The three most common inputs you may want to use are **`header`**, **`skip`**, and **`stringsAsFactors`**.

- Use the **`header`** input when your data does not contain column names. *For example*, `header=FALSE` indicates that your datafile does not have any column names.
- Use the **`skip`** input when you want to skip some lines of metadata at the top of your file. This is handy if you really don't want to get rid of your metadata in your header. *For example,* `skip=2` skips the first two rows of the datafile before `R` begins reading data.
- Use the **`stringsAsFactors`** input when you want to make absolutely sure that `R` interprets any non-numeric fields as characters rather than factors. We have not focused on factors yet, but it can be frustrating when `R` mistakes a column of character strings as a column factors. To avoid any possible confusion, use `stringsAsFactors=TRUE` as an input.

For example, here is how to read in this data without column names:

```
df <- read.csv("data/super_data.csv",skip=1,header=FALSE)
df
  V1 V2  V3                          V4
1  1 74 135              not very nice
2  2 56 112                 kinda cute
3  3 59 156 kinda cute but had a ring
4  4 43 102                  so small!
```

If you do this without setting header to FALSE, your first row of data gets used as column names and it becomes a big ole mess:

```
df <- read.csv("data/super_data.csv",skip=1)
df
  X1 X74 X135             not.very.nice
1  2  56  112                kinda cute
2  3  59  156 kinda cute but had a ring
3  4  43  102                 so small!
```

`readxl()`

To read in an *Excel* spreadsheet, use the `read_xlsx()` function from the package `readxl`.

If `super_data` were an `.xlsx` file, the command would look like this:

```
library(readxl)
df <- read_xlsx("data/super_data.xlsx",sheet=1)
```

`gsheet()`

To read in an *GoogleSheets* spreadsheet, use the `gsheet2tbl()` function from the package `gsheet`.

Make sure link sharing is turned on for the *GoogleSheet* you are trying to access:

```
library(gsheet)
df <- gsheet2tbl("https://docs.google.com/spreadsheets/d/1uQ5PfGITnjHngK41FWHeedioca30wpyEsBOpC_B6T3E/edit
df
# A tibble: 4 x 4
  patient_id height_in weight_lb comment
       <dbl>     <dbl>     <dbl> <chr>
1          1        74       135 not very nice
2          2        56       112 kinda cute
3          3        59       156 kinda cute but had a ring
4          4        43       102 so small
```

`readRDS()`

Another niche function for reading data is `readRDS()`. This function allows you to read in *R data objects*, which have the file extension *.rds*. These data objects need not be in the same format as a `.csv` or even a dataframe, and that is what makes them so handy. A colleague could send you an `.rds` object of a vector, a list, a plotting function, or any other kind of `R` object, and you can read it in with `readRDS()`.

For example, contains a `tibble` version of the dataframe above. When you read in that `.rds` file, it is already formatted as a `tibble`:

```
df <- readRDS("data/super_data.rds")
df
# A tibble: 4 x 4
  patient_id height_in weight_lb comment
       <dbl>     <dbl>     <dbl> <chr>
1          1        74       135 not very nice
2          2        56       112 kinda cute
3          3        59       156 kinda cute but had a ring
4          4        43       102 so small!
```

`.csv` files

The `.csv` is such an important file type when you work with data that it is worth introducing it with more detail.

When you preview a `.csv`, it looks something like this:

station	year	doy	sit	x	y	n	bhvr	id
Boat	2004	159	2004060701	-129.1875	53.101	NA	TR	BCY0470
Boat	2004	189	2004070701	-129.1875	53.101	NA	BNF	BCX0361
Boat	2004	196	2004071401	-129.1875	53.101	NA	BNF	BCX0361
Boat	2004	196	2004071401	NA	NA	NA	BNF	BCX0711
Boat	2004	200	2004071801	-129.1875	53.101	NA	BNF	BCX0275
Boat	2004	209	2004072701	-129.194	53.1018	NA	MI-FE	BCY0321
Boat	2004	216	2004080301	-129.1875	53.101	NA	BNF	BCY0049
Boat	2004	216	2004080301	NA	NA	NA	BNF	BCX0255
Boat	2004	217	2004080401	-129.1916	53.0696	NA	MI-FE	BCY0117
Boat	2004	217	2004080401	NA	NA	NA	MI-FE	BCY0189
Boat	2004	218	2004080501	-129.18067	53.0899	NA	BNF	BCX0083
Boat	2004	218	2004080501	NA	NA	NA	BNF	BCX0231
Boat	2004	219	2004080601	-129.1875	53.101	NA	BNF	BCY0013
Boat	2004	221	2004080801	-129.1875	53.101	NA	TR	BCX0231
Boat	2004	221	2004080801	NA	NA	NA	TR	BCX0121
Boat	2004	223	2004081001	-129.1875	53.101	NA	TR	BCX0121
Boat	2004	227	2004081401	-129.1875	53.101	NA	TR	BCX0171
Boat	2004	228	2004081501	NA	NA	NA	TR	BCX0022
Boat	2004	228	2004081502	NA	NA	NA	TR	BCX0121
Boat	2004	229	2004081601	-129.1875	53.101	NA	TR	BCX0022
Boat	2004	230	2004081701	-129.1875	53.101	NA	RE-TR	BCY0013
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCX0171
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCY0013
Boat	2004	230	2004081701	NA	NA	NA	BNF	BCX0694
Boat	2004	230	2004081703	NA	NA	NA	RE-TR	BCX0239
Boat	2004	230	2004081704	NA	NA	NA	TR	BCY0117
Boat	2004	231	2004081801	-129.194	53.1018	NA	MI-FE	BCX0567
Boat	2004	231	2004081802	-129.237	53.068	NA	BNF	BCX0083
Boat	2004	233	2004082001	NA	NA	NA	TR	BCX0174
Boat	2004	235	2004082201	-129.1875	53.101	NA	TR	BCX0380
Boat	2004	235	2004082201	NA	NA	NA	TR	BCX0380

A neat spreadsheet of rows and columns.

When you open up this same dataset in a simple text editor, it looks like this:

hw-master.csv

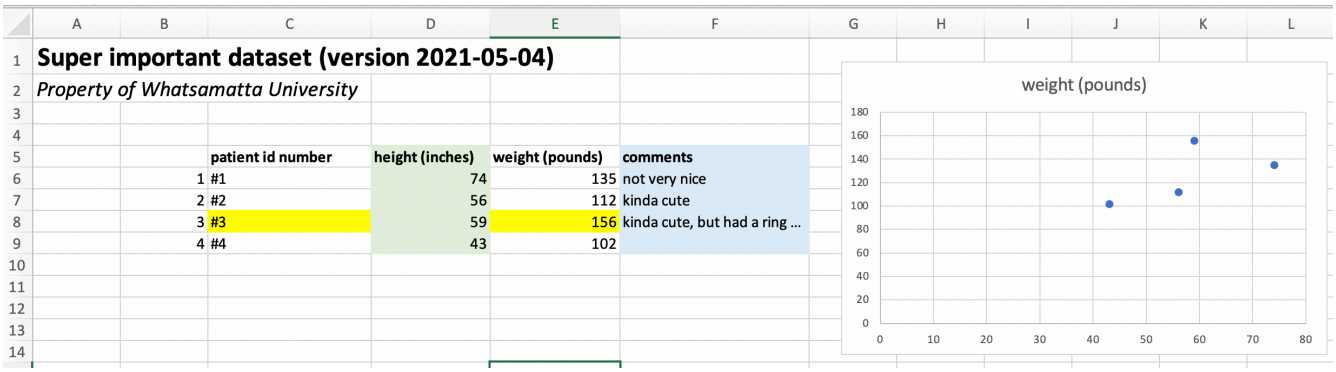
```
station,year,doy,sit,x,y,n,bhvr,id,name,stage,bhvr2,bhvr3,loc.conf,type,conf,month,dom,date,time
Boat,2004,159,2004060701,-129.1875,53.101,NA,TR,BCY0470,,A,TR,TR,0,1,6,7,6/7/04,16:35
Boat,2004,189,2004070701,-129.1875,53.101,NA,BNF,BCX0361,,F,BNF,BNF,0,1,7,7,7/7/04,13:33
Boat,2004,196,2004071401,-129.1875,53.101,NA,BNF,BCX0361,,F,BNF,BNF,0,1,7,14,7/14/04,14:55
Boat,2004,196,2004071401,NA,NA,NA,BNF,BCX0711,,A,BNF,BNF,0,1,7,14,7/14/04,14:55
Boat,2004,200,2004071801,-129.1875,53.101,NA,BNF,BCX0275,,A,BNF,BNF,0,1,7,18,7/18/04,16:52
Boat,2004,209,2004072701,-129.194,53.1018,NA,MI-FE,BCY0321,,A,MI-FE,MI-FE,0,1,7,27,7/27/04,7:35
Boat,2004,216,2004080301,-129.1875,53.101,NA,BNF,BCY0049,,F,BNF,BNF,0,1,8,3,8/3/04,13:41
Boat,2004,216,2004080301,NA,NA,NA,BNF,BCX0255,,A,BNF,BNF,0,1,8,3,8/3/04,13:41
Boat,2004,217,2004080401,-129.1916,53.0696,NA,MI-FE,BCY0117,,A,MI-FE,MI-FE,0,1,8,4,8/4/04,12:50
Boat,2004,217,2004080401,NA,NA,NA,MI-FE,BCY0189,,F,MI-FE,MI-FE,0,1,8,4,8/4/04,12:50
Boat,2004,218,2004080501,-129.18067,53.0899,NA,BNF,BCX0083,,A,BNF,BNF,0,1,8,5,8/5/04,14:23
Boat,2004,218,2004080501,NA,NA,NA,BNF,BCX0231,,A,BNF,BNF,0,1,8,5,8/5/04,14:23
Boat,2004,219,2004080601,-129.1875,53.101,NA,BNF,BCY0013,,A,BNF,BNF,0,1,8,6,8/6/04,8:19
Boat,2004,221,2004080801,-129.1875,53.101,NA,TR,BCX0231,,A,TR,TR,0,1,8,8,8/8/04,12:28
Boat,2004,221,2004080801,NA,NA,NA,TR,BCX0121,,F,TR,TR,0,1,8,8,8/8/04,8:19
Boat,2004,223,2004081001,-129.1875,53.101,NA,TR,BCX0121,,F,TR,TR,0,1,8,10,8/10/04,13:13
Boat,2004,227,2004081401,-129.1875,53.101,NA,TR,BCX0171,,F,TR,TR,0,1,8,14,8/14/04,11:44
Boat,2004,228,2004081501,NA,NA,NA,TR,BCX0022,,A,TR,TR,0,1,8,15,8/15/04,11:44
Boat,2004,228,2004081502,NA,NA,NA,TR,BCX0121,,F,TR,TR,0,1,8,15,8/15/04,18:30
Boat,2004,229,2004081601,-129.1875,53.101,NA,TR,BCX0022,,A,TR,TR,0,1,8,16,8/16/04,9:00
Boat,2004,230,2004081701,-129.1875,53.101,NA,RE-TR,BCY0013,,A,RE-TR,RE-TR,0,1,8,17,8/17/04,7:10
Boat,2004,230,2004081701,NA,NA,NA,BNF,BCX0171,,F,BNF,BNF,0,1,8,17,8/17/04,11:50
Boat,2004,230,2004081701,NA,NA,NA,BNF,BCY0013,,A,BNF,BNF,0,1,8,17,8/17/04,11:50
Boat,2004,230,2004081701,NA,NA,NA,BNF,BCX0694,,A,BNF,BNF,0,1,8,17,8/17/04,11:50
Boat,2004,230,2004081703,NA,NA,NA,RE-TR,BCX0239,,M,RE-TR,RE-TR,0,1,8,17,8/17/04,14:00
Boat,2004,230,2004081704,NA,NA,NA,TR,BCY0117,,A,TR,TR,0,1,8,17,8/17/04,15:30
Boat,2004,231,2004081801,-129.194,53.1018,NA,MI-FE,BCX0567,,A,MI-FE,MI-FE,0,1,8,18,8/18/04,14:36
Boat,2004,231,2004081802,-129.237,53.068,NA,BNF,BCX0083,,A,BNF,BNF,0,1,8,18,8/18/04,14:06
Boat,2004,233,2004082001,NA,NA,NA,TR,BCX0174,,F,TR,TR,0,1,8,20,8/20/04,11:15
Boat,2004,235,2004082201,-129.1875,53.101,NA,TR,BCX0380,,A,TR,TR,0,1,8,22,8/22/04,16:09
Boat,2004,235,2004082201,NA,NA,NA,TR,BCX0380,,A,TR,TR,0,1,8,22,8/22/04,16:09
Boat,2004,236,2004082301,-129.1933,53.0527,NA,RE-TR,BCX0049,,F,RE-TR,RE-TR,0,1,8,23,8/23/04,10:30
Boat,2004,236,2004082303,-129.18067,53.0899,NA,MI-FE,BCX0380,,A,MI-FE,MI-FE,0,1,8,23,8/23/04,13:41
Boat,2004,239,2004082603,-129.237,53.068,NA,RE-TR,BCY0189,,F,RE-TR,RE-TR,0,1,8,26,8/26/04,8:05
Boat,2004,239,2004082605,-129.2113,53.077,NA,TR,BCZ0053,,F,TR,TR,0,1,8,26,8/26/04,9:00
Boat,2004,239,2004082605,-129.1875,53.101,NA,RE-TR,BCZ0053,,F,RE-TR,RE-TR,0,1,8,26,8/26/04,19:12
Boat,2004,239,2004082605,NA,NA,NA,RE-TR,BCY0135,,A,RE-TR,RE-TR,0,1,8,26,8/26/04,19:12
Boat,2004,246,2004090201,-129.1933,53.0527,NA,TR,BCY0013,,A,TR,TR,0,1,9,2,9/2/04,10:40
Boat,2004,246,2004090202,-129.1933,53.0527,NA,RE-TR,BCX0386,,A,RE-TR,RE-TR,0,1,9,2,9/2/04,10:45
Boat,2004,246,2004090203,NA,NA,NA,RE-TR,BCX0144,,A,RE-TR,RE-TR,0,1,9,2,9/2/04,11:00
Boat,2004,246,2004090203,NA,NA,NA,RE-TR,BCX0171,,F,RE-TR,RE-TR,0,1,9,2,9/2/04,11:00
Boat,2004,249,2004090501,-129.1875,53.101,NA,TR,BCY0347,,A,TR,TR,0,1,9,5,9/5/04,9:30
Boat,2004,251,2004090701,NA,NA,NA,TR,BCX0750,,A,TR,TR,0,1,9,7,9/7/04,11:00
Boat,2004,251,2004090701,NA,NA,NA,TR,BCY0383,,A,TR,TR,0,1,9,7,9/7/04,11:00
Boat,2004,253,2004090902,-129.3198,53.0977,NA,BNF,BCY0013,,A,BNF,BNF,0,1,9,9,9/9/04,13:00
Boat,2004,253,2004090902,NA,NA,NA,BNF,BCX0239,,M,BNF,BNF,0,1,9,9,9/9/04,13:00
Boat,2004,253,2004090902,NA,NA,NA,MI-FE,BCX0239,,C,MI,MI,0,1,9,9,9/9/04,13:00
Boat,2004,253,2004090903,-129.3032,53.0711,NA,TR,BCY0013,,A,TR,TR,0,1,9,9,9/9/04,14:20
Boat,2004,253,2004090903,NA,NA,NA,TR,BCX0171,,F,TR,TR,0,1,9,9,9/9/04,14:20
Boat,2004,253,2004090903,NA,NA,NA,TR,BCX0144,,F,TR,TR,0,1,9,9,9/9/04,14:20
Boat,2004,253,2004090904,-129.2002,53.0472,NA,BNF,BCX0711,,A,BNF,BNF,0,1,9,9,9/9/04,15:30
Boat,2004,254,2004091005,NA,NA,NA,TR,BCX0022,,A,TR,TR,0,1,9,10,9/10/04,15:30
Boat,2004,258,2004091401,-129.2002,53.0472,NA,TR,BCX0567,,A,TR,TR,0,1,9,14,9/14/04,12:12
Boat,2004,258,2004091402,-129.2002,53.0472,NA,BNF,BCY0277,,A,BNF,BNF,0,1,9,14,9/14/04,13:30
Boat,2004,258,2004091402,NA,NA,NA,BNF,BCY0013,,A,BNF,BNF,0,1,9,14,9/14/04,13:30
```

This looks scary, but it is actually really simple. A `.csv` is a simple text file in which each row is on a new line and columns of data are separated by commas. As a simple text file, there is no fancy formatting. There are no "Sheets" or "Tabs", as you would find in GoogleSheets or Excel; it is a simple 'flat' file.

One of the major advantages of working with `.csv`'s is that the format is cross-platform and non-proprietary. That is, they work on Windows, Mac, Linux, and any other common type of computer, and they do not require special software to open.

Prepping your data for R

For those of us used to working in *Excel*, *GoogleSheets*, or *Numbers*, it will take some adjustment to get into the habit of formatting your data for R. We are used to seeing spreadsheets that look something like this:



	A	B	C	D	E	F
1	**Super important dataset (version 2021-05-04)**					
2	*Property of Whatsamatta University*					
3						
4						
5			patient id number	height (inches)	weight (pounds)	comments
6		1	#1	74	135	not very nice
7		2	#2	56	112	kinda cute
8		3	#3	59	156	kinda cute, but had a ring ...
9		4	#4	43	102	
10						
11						
12						
13						
14						

To read a `.csv` into R without issues or fancy code, this spreadsheet will need to be simplified to look like this:

	A	B	C	D
1	patient_id	height_in	weight_lb	comments
2	1	74	135	not very nice
3	2	56	112	kinda cute
4	3	59	156	kinda cute but had a ring
5	4	43	102	

Workflow for formatting your data

Below is the general worfklow for preparing your data for R is the following:

1. Get your data into `.csv` format. In *Excel* and *Numbers*, you can use 'Save As' to change the file format. In *GoogleSheets*, you can 'Download As' a `.csv`. This will remove any colors, thick lines, special fonts, bold or italicized font styles, and any other special formatting. All that will be left is your data, and that's the way R likes it.

2. Remove blank columns before and in the middle of your data.

3. Remove fancy elements such as graphs.

4. Simplify your 'header'. The space above your data is your spreadsheet's header. It includes column names and metadata like title, author, measurement units, etc. It is possible to read data with complex headers into R, but again we are going for simplicity here, so we suggest (1) simplifying your header to contain column names only, and (2) moving metadata to a `README.txt` file that lives in the same folder as your data.

5. Simplify column names. Remove spaces, or replace them with `.`, `-` or `_`. Make your column names as simple and brief as possible while still being informative. Include units in the column names, as in the screenshot above. Be sure that each column has a name.

6. Remove all commas and hashtags from your dataset. You can do this with the 'Find & Replace' feature built-into in most spreadsheet editors.

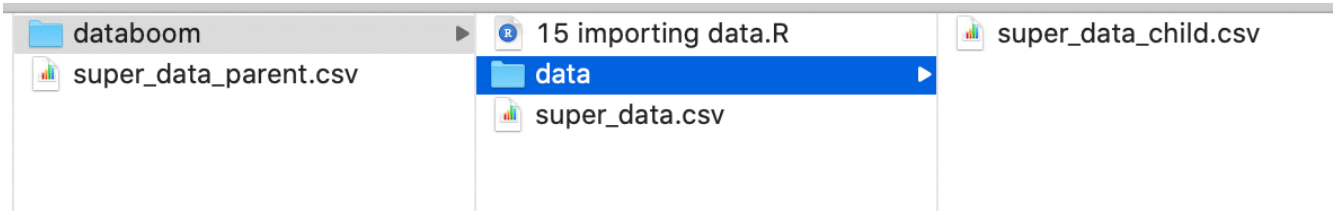
Managing files & folders

Reading data from other folders

The data-reading functions above require only a single input: the *path* to your data file. This *path* is relative to the location of your working directory. When your data file is *inside* your working directory, the path simplifies to be the same as the filename of your data:

```
df <- read.csv("data/super_data.csv")
```

Sometimes, though, you will want to keep your data somewhere nearby but not necessarily *within* your working directory. Consider the following scenario, in which three versions of the "super_data.csv" dataset occur near a working directory being used for this module:



We have a version within the same directory as our `R` file (i.e., our working directory), another version within a *child* folder within the directory (i.e., a subfolder), and another version in the *parent* folder of the working directory.

To read a file from a *child* folder, add the prefix, `./<child name>/`, to your command:

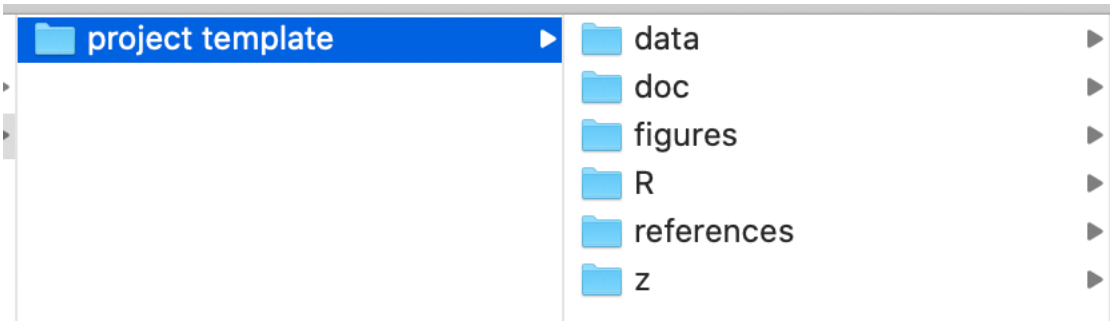
```
df <- read.csv("./data/super_data.csv")
```

To read a file from a *parent* folder, add the prefix, `../`, to your command:

```
df <- read.csv("./data/super_data.csv")
```

Managing files

Now consider the following scenario, in which your project folder structure looks like this:



This structure can be a effective and simple way of organizing your files for a project, and we recommend using it.

Here's what these child folders should contain.

- `./data/` contains data, of course.
- `./doc/` contains documents, such as manuscript drafts.
- `./figures/` contains files for graphs and figures.
- `./R/` contains `R` scripts, of course.
- `./references/` contains journal articles and other resources you are using in your research.

Since your `R` code is going into the `R` child folder, that is what you should set your working directory for those `R` scripts to. In that case, *how to read data from the* ***`data`*** *folder*, which is a separate child folder of your parent folder?

Here's how:

```
df <- read.csv("../data/super_data.csv")
```

Chapter 14

Base plots

Learning goals

- Make basic plots in R
- Basic adjustments to plot formatting

To learn how to plot, let's first create a dataset to work with:

```
country <- c("USA","Tanzania","Japan","Ctr. Africa Rep.","China","Norway","India")
lifespan <- c(79,65,84,53,77,82,69)
gdp <- c(55335,2875,38674,623,13102,84500,6807)
```

These data come from this publicly available database that compares health and economic indices across countries in 2011.

The `lifespan` column presents the average life expectancy for each country.

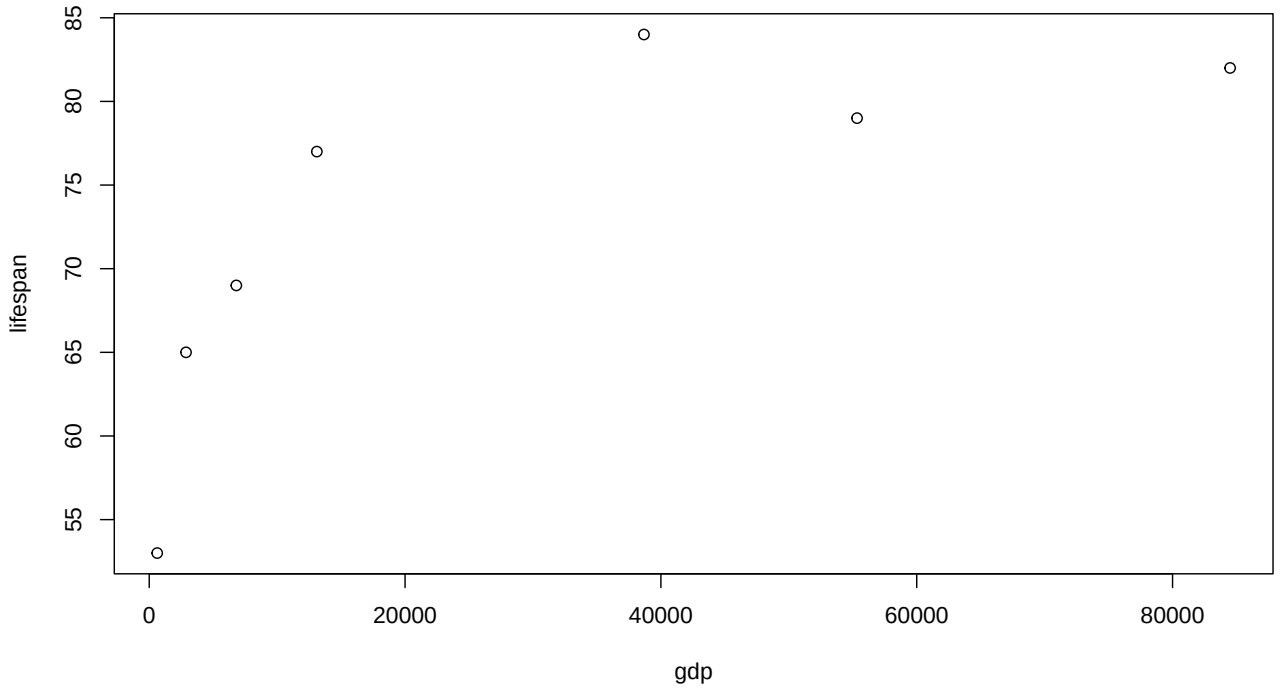
The `gdp` column presents the average GDP per capita within that country, which is a common index for the income and wealth of average citizens.

Let's see if there is a relationship between life expectancy and income.

Create a basic plot

The simplest way to make a basic plot in R is to use its built-in `plot()` function:

```
plot(lifespan ~ gdp)
```



This syntax is saying this: plot column `lifespan` as a function of `gdp`. The symbol `~` denotes *"as a function of"*. This frames `lifespan` as a dependent variable (y axis) that is affected by the independent variable (x axis), which in this case is `gdp`.

Note that `R` uses the variable names you provided as the x- and y-axes. You can adjust these labels however you wish (see formatting section below).

You can also produce this exact same plot using the following syntax:

```
plot(y=lifespan, x=gdp)
```

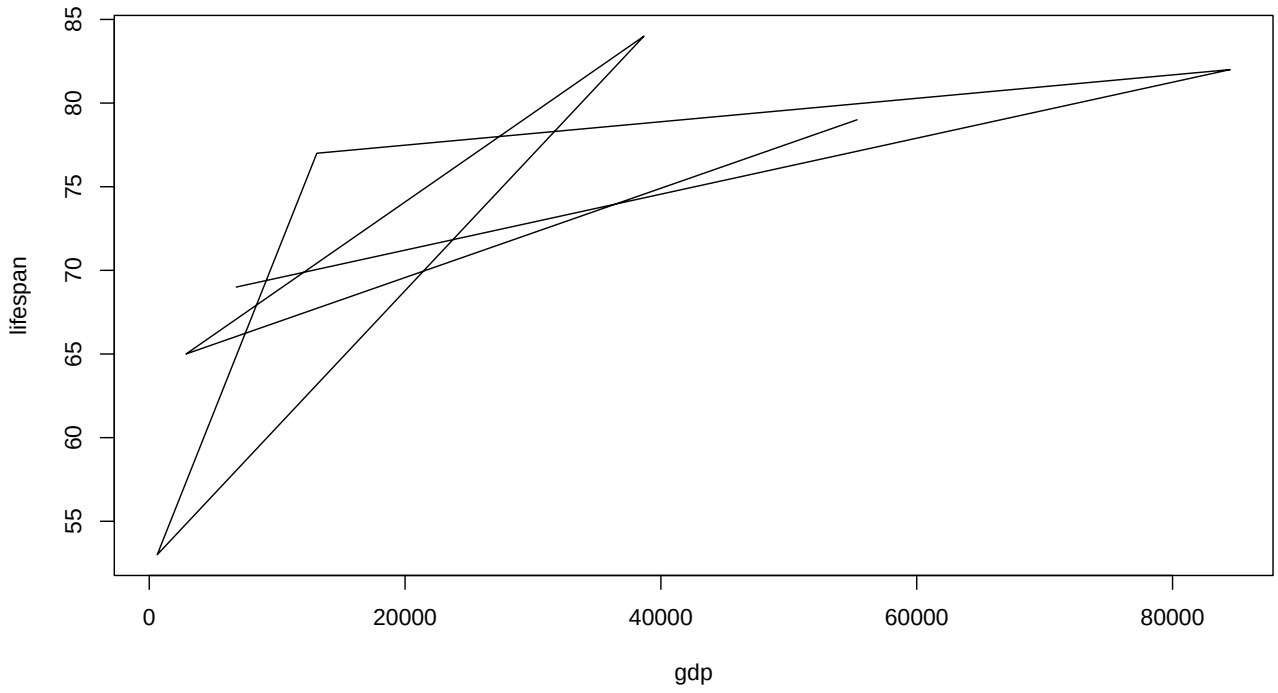
Choose whichever one is most intuitive to you.

Most common types of plots

The plot above is a **scatter plot**, and is one of the most common types of plots in data science.

You can turn this into a **line plot** by adding a parameter to the `plot()` function:

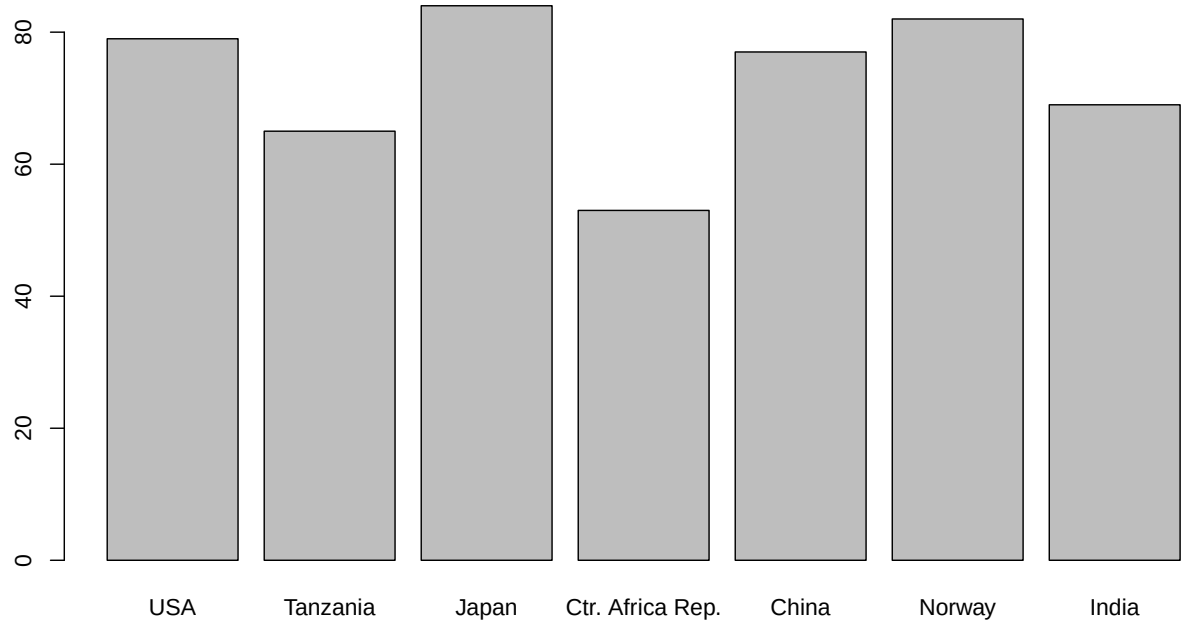
```
plot(lifespan ~ gdp, type="l")
```



What a mess! Rather than connecting these values in the order you might expect, `R` connects them in the order that they are listed in their source vectors. This is why line plots tend to be more useful in scenarios such as time series, which are inherently ordered.

Another common plot is the **bar plot**, which uses a different `R` function:

```
barplot(height=lifespan,names.arg=country)
```



In this command, the parameter `height` determines the height of the bars, and `names.arg` provides the labels to place beneath each bar.

There are many more plot types out there, but let's stop here for now.

Basic plot formatting

You can adjust the default formatting of plots by adding other inputs to your `plot()` command. To understand all the parameters you can adjust, bring up the help page for this function:

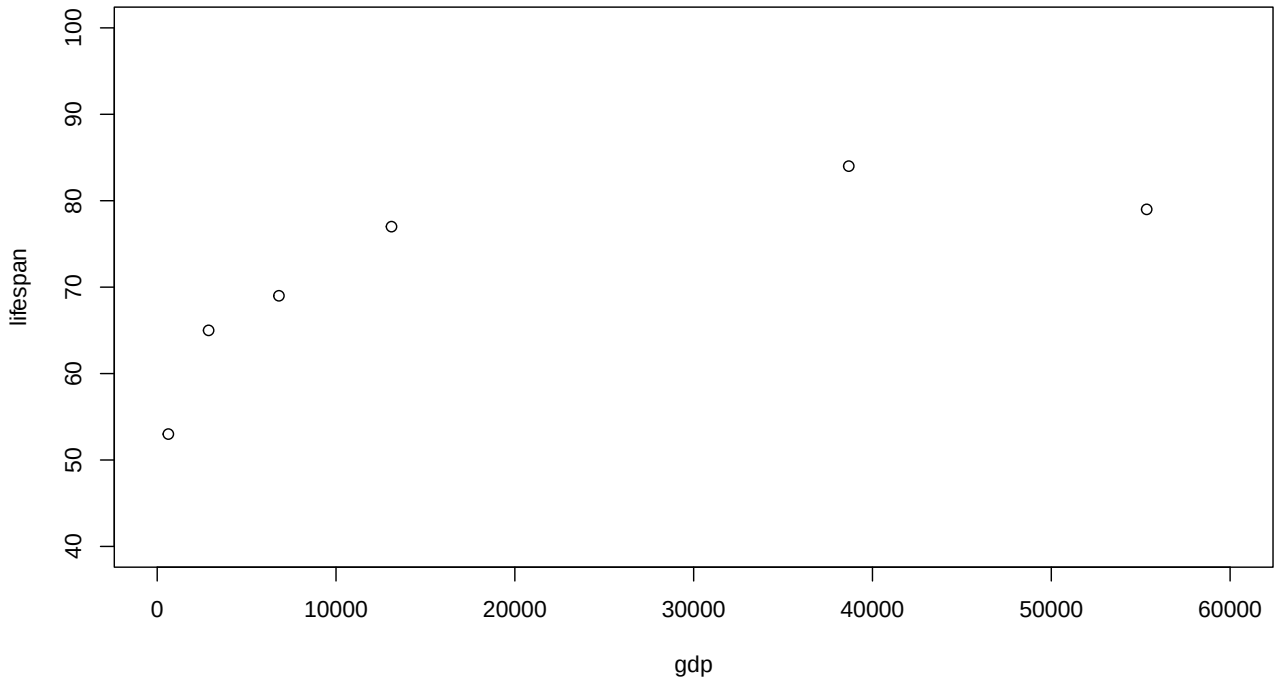
```
?plot
```

If multiple help page options are returned, select the *Generiz X-Y Plotting* page from the `base` package. This is the plot function that comes built-in to `R`.

Here we demonstrate just a few of the most common formatting adjustments you are likely to use:

Set plot range using `xlim` (for the x axis) and `ylim` (for the y axis):

```
plot(lifespan ~ gdp,xlim=c(0,60000),ylim=c(40,100))
```



In this command, you are defining axis limits using a 2-element vector (i.e., `c(min,max)`).

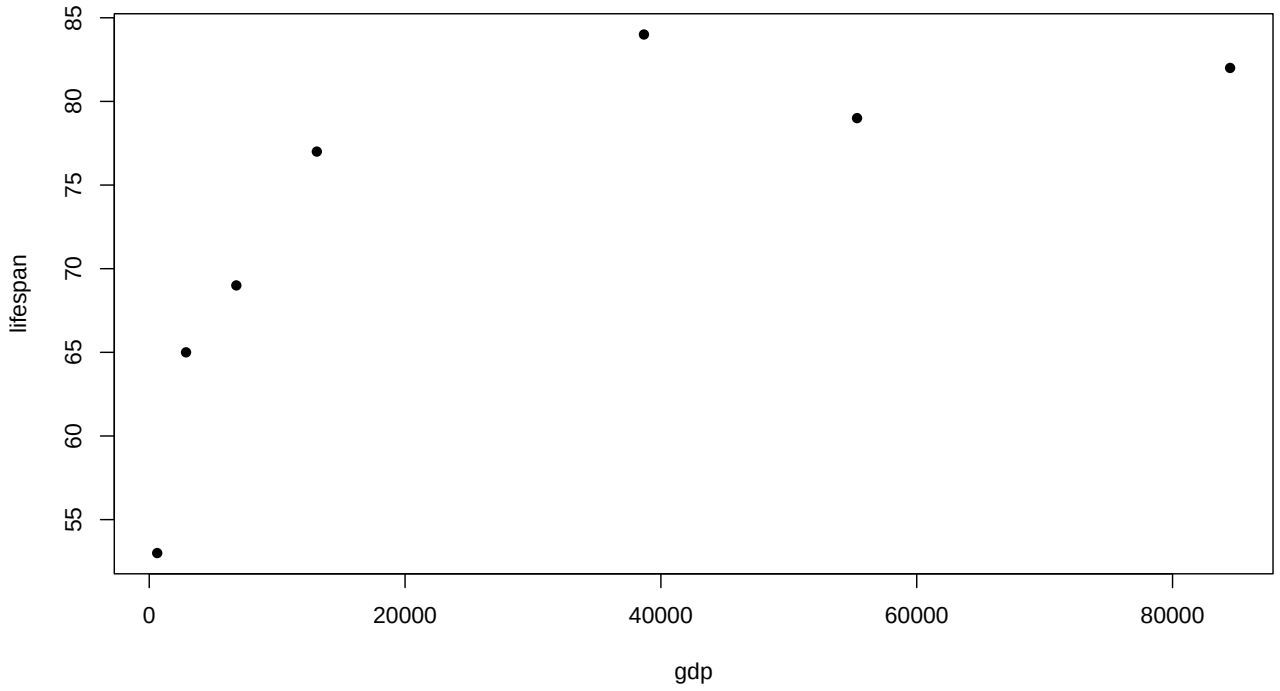
Note that it can be easier to read your code if you put each input on a new line, like this:

```
plot(lifespan ~ gdp,
     xlim=c(0,60000),
     ylim=c(40,100))
```

Make sure each input line within the function ends with a comma, otherwise you `R` will get confused and throw an error.

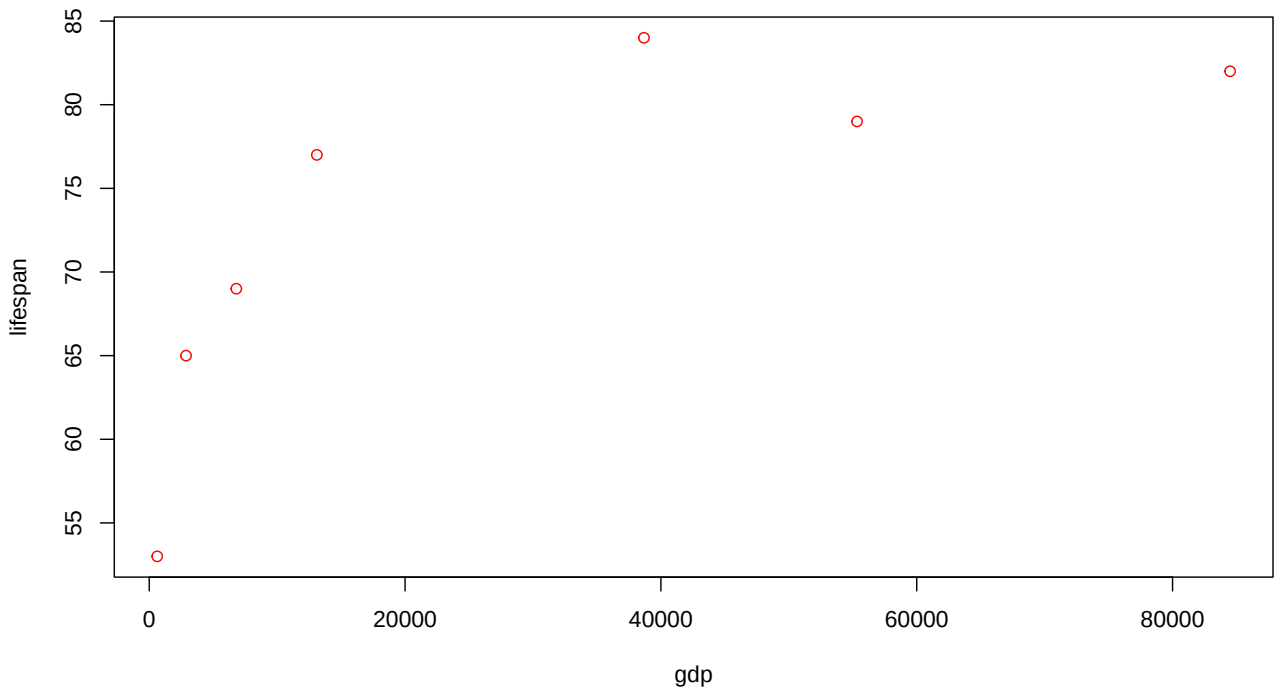
Set dot type using the input `pch`:

```
plot(lifespan ~ gdp,pch=16)
```



Set dot color using the input `col` (the default is `col="black"`)

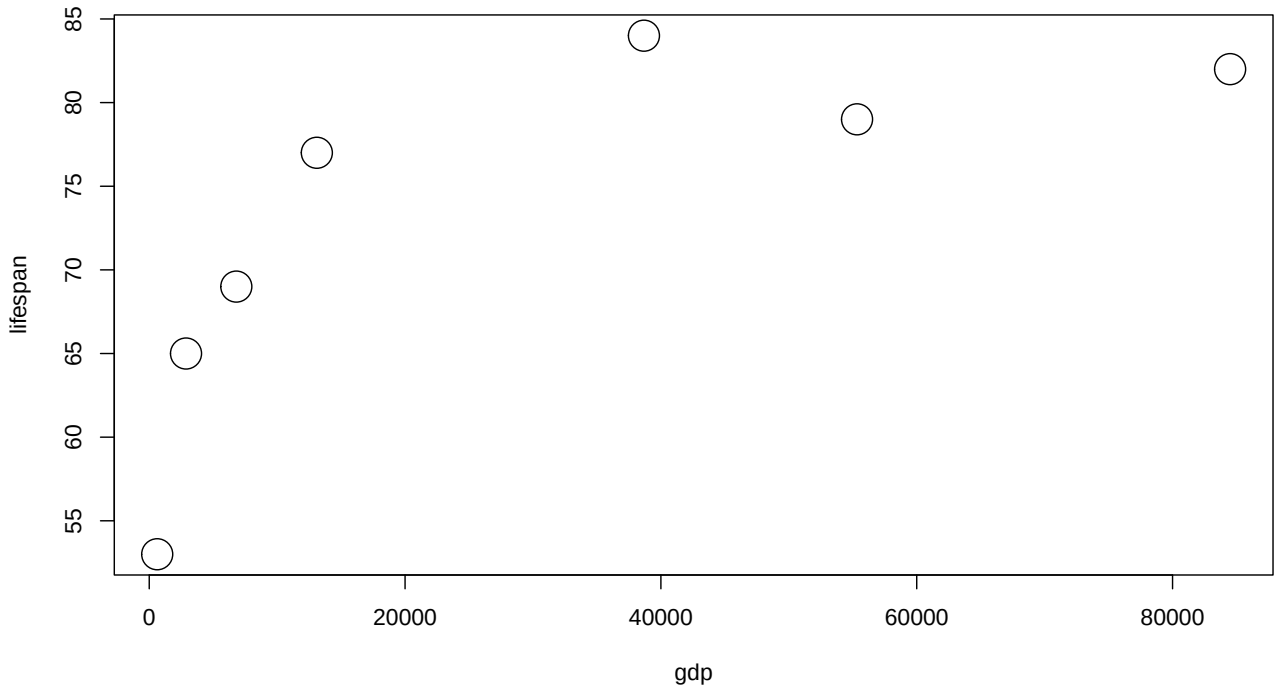
```
plot(lifespan ~ gdp,col="red")
```



Here is a great resource for color names in R.

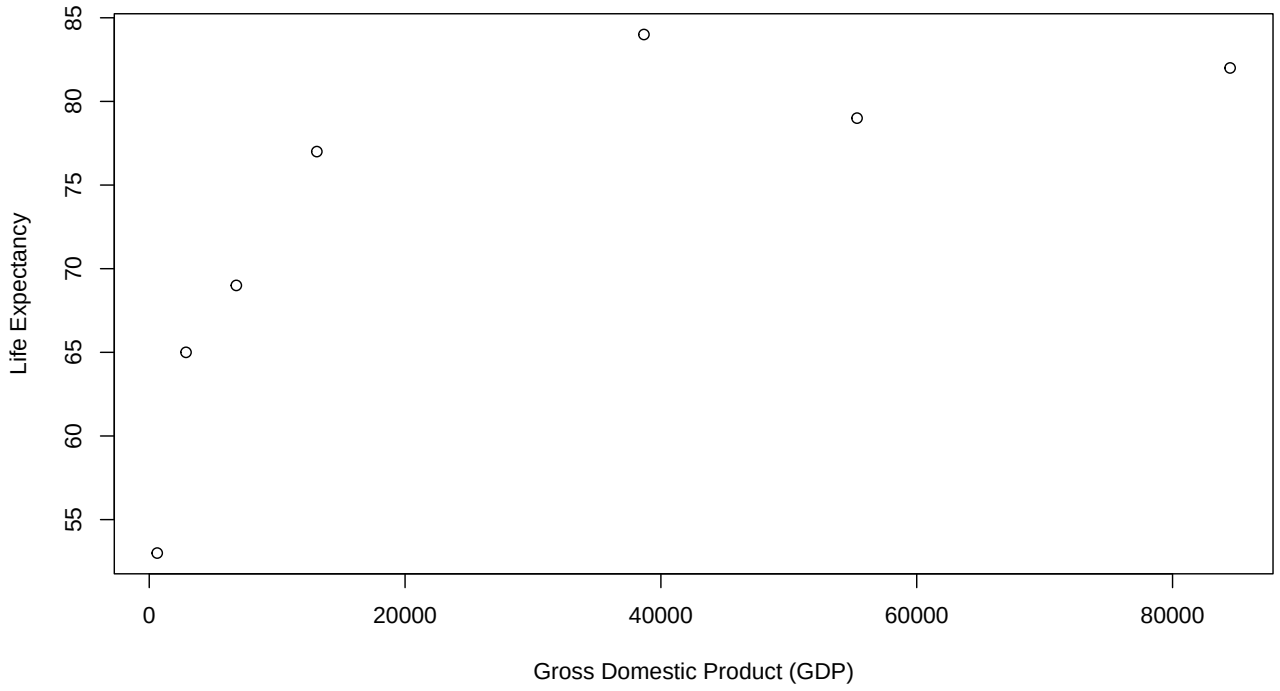
Set dot size using the input `cex` (the default is `cex=1`):

```
plot(lifespan ~ gdp,cex=3)
```



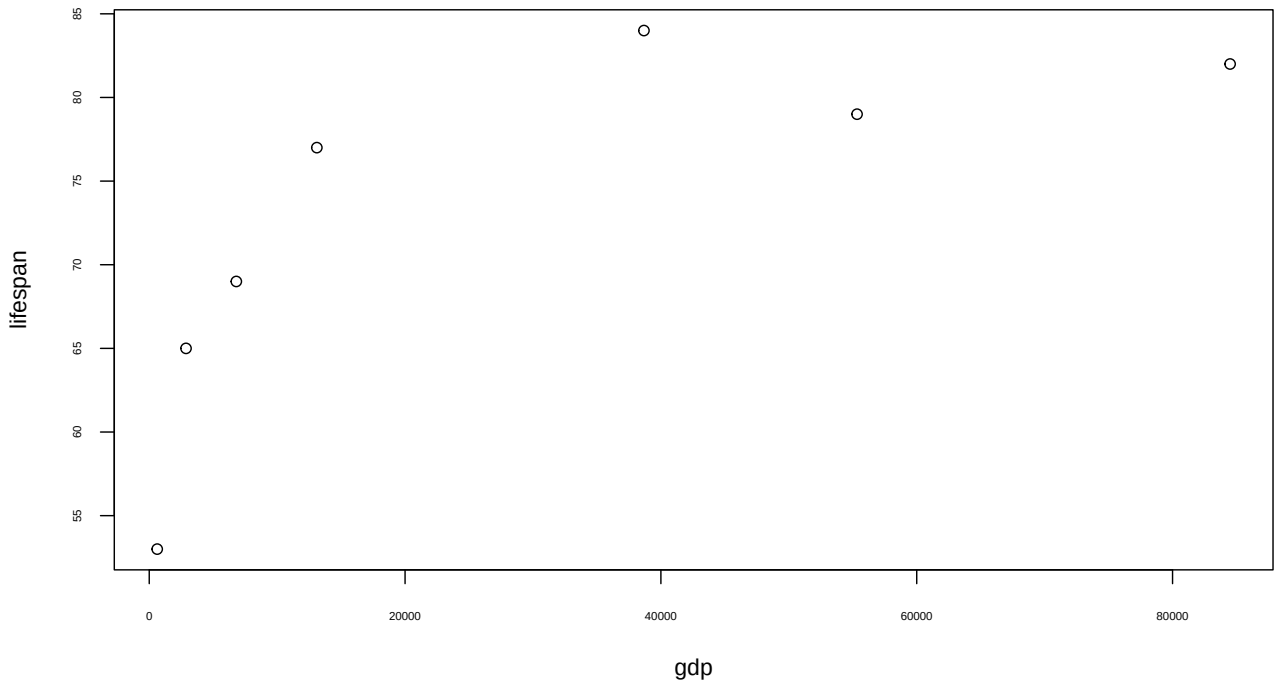
Set axis labels using the inputs `xlab` and `ylab`:

```
plot(lifespan ~ gdp, xlab="Gross Domestic Product (GDP)",ylab="Life Expectancy")
```



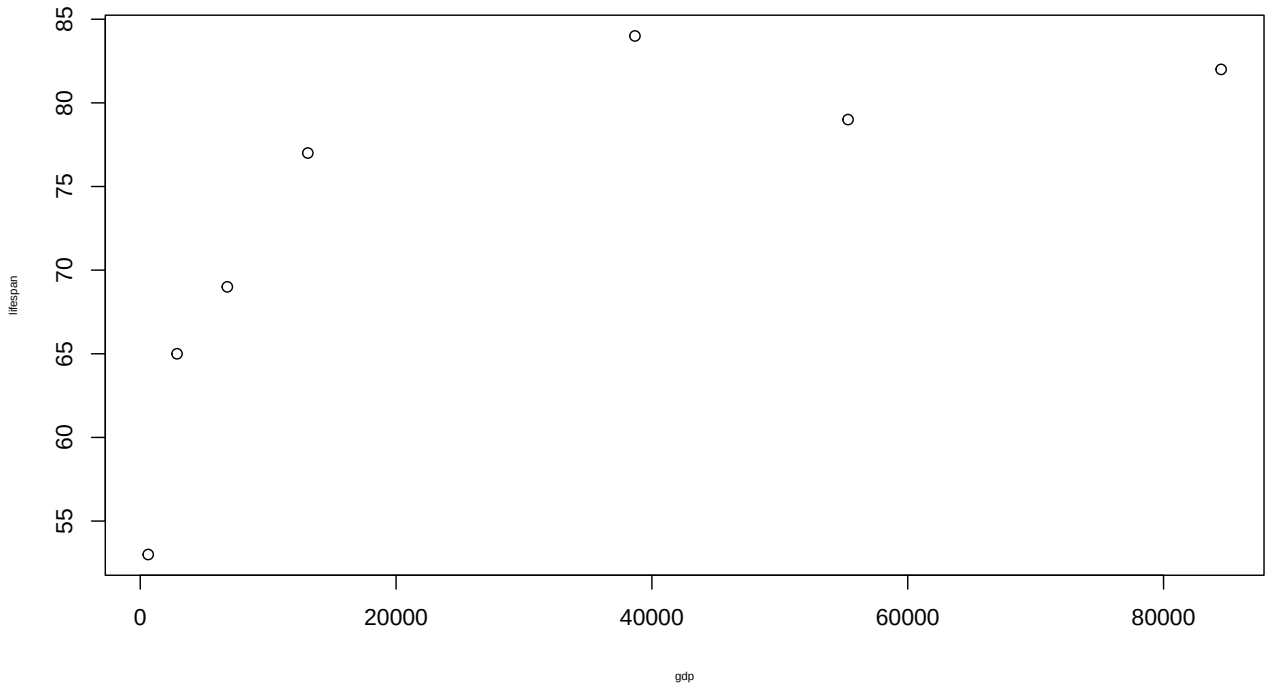
Set axis number size using the input `cex.axis` (the default is `cex.axis=1`):

```
plot(lifespan ~ gdp,cex.axis=.5)
```



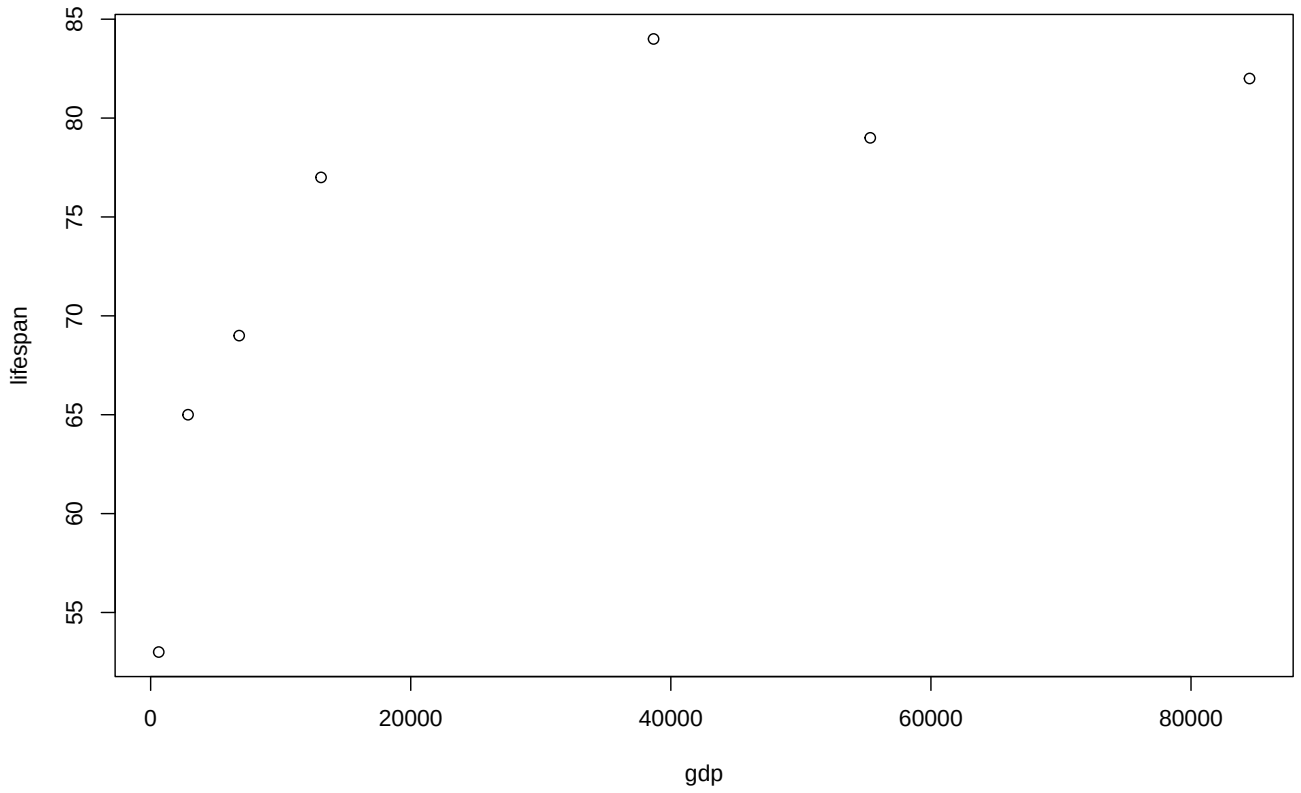
Set axis label size using the input `cex.label` (the default is `cex.lab=1`):

```
plot(lifespan ~ gdp,cex.lab=.5)
```



Set plot margins using the function `par(mar=c())` before you call `plot()`:

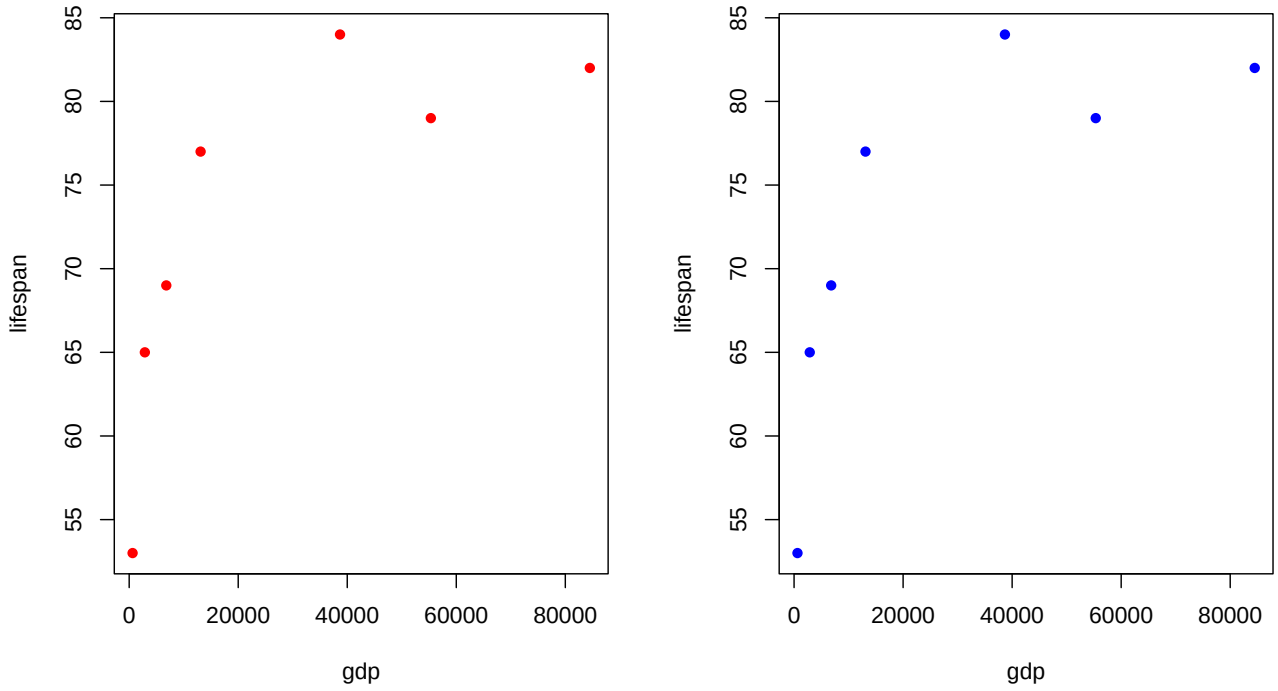
```
par(mar=c(5,5,0.5,0.5))
plot(lifespan ~ gdp)
```



In this command, the four numbers in the vector used to define `mar` correspond to the margin for the bottom, left, top, and right sides of the plot, respectively.

Create a multi-pane plot using the function `par(mfrow=c())` before you call `plot()`:

```
par(mfrow=c(1,2))
plot(lifespan ~ gdp,col="red",pch=16)
plot(lifespan ~ gdp,col="blue",pch=16)
```



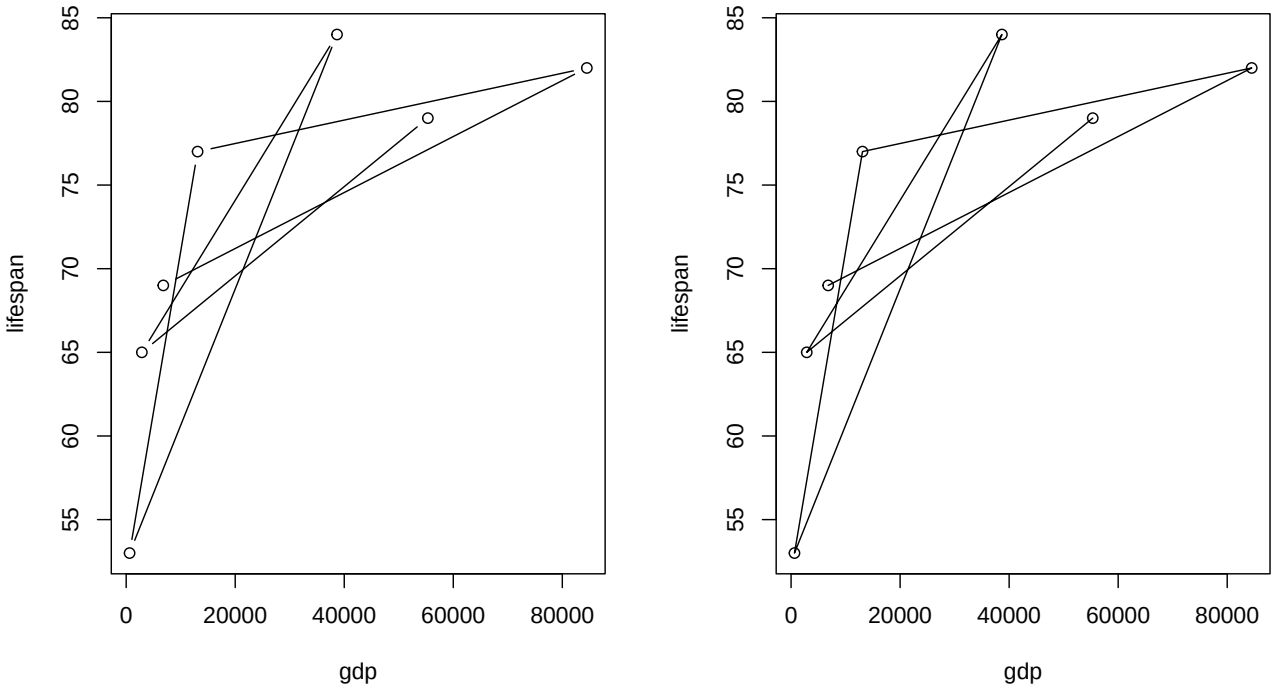
In this command, the two numbers in the vector used to define `mfrow` correspond to the number of rows and columns, respectively, on the entire plot. In this case, you have 1 row of plots with two columns.

Note that you will need to reset the number of panes when you are done with your multi-pane plot!

```
par(mfrow=c(1,1))
```

Plot dots and lines at once using the input `type`:

```
par(mfrow=c(1,2))
plot(lifespan ~ gdp,type="b")
plot(lifespan ~ gdp,type="o")
```

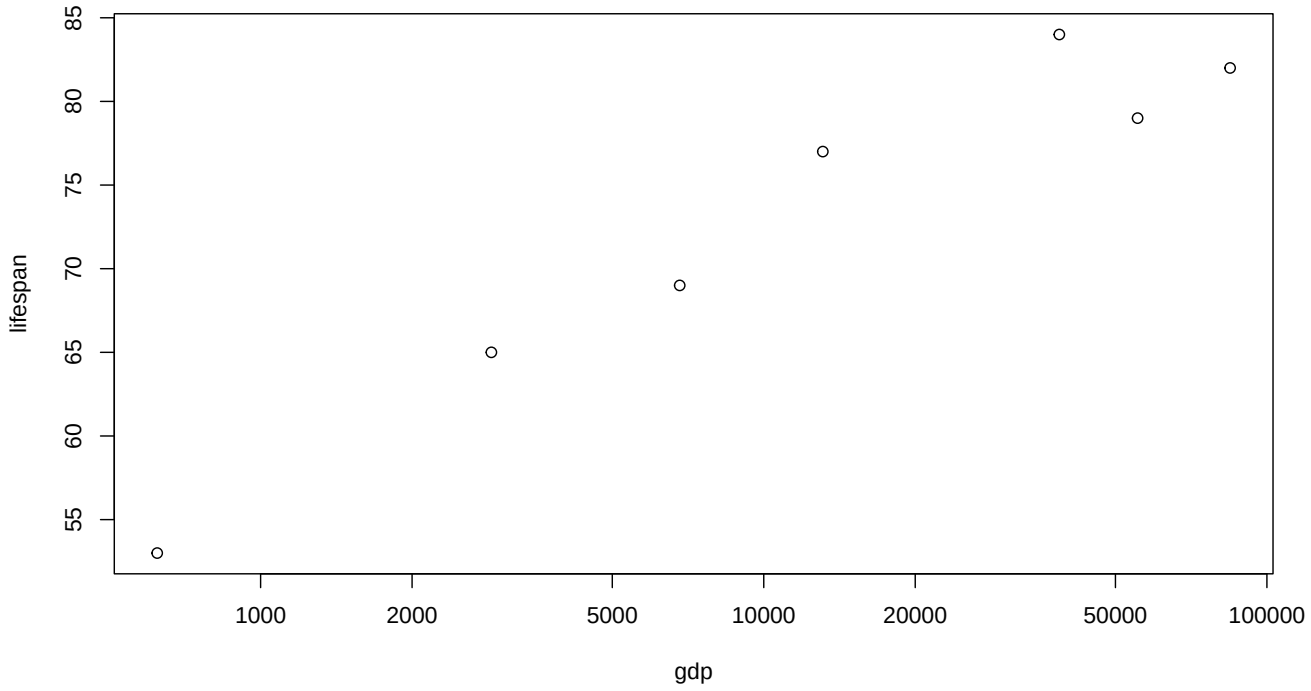



```
par(mfrow=c(1,1))
```

Note the two slightly different formats here.

Use a logarithmic scale for one or of your axes using the input `log`

```
plot(lifespan ~ gdp,log="x")
```



Plotting with data frames

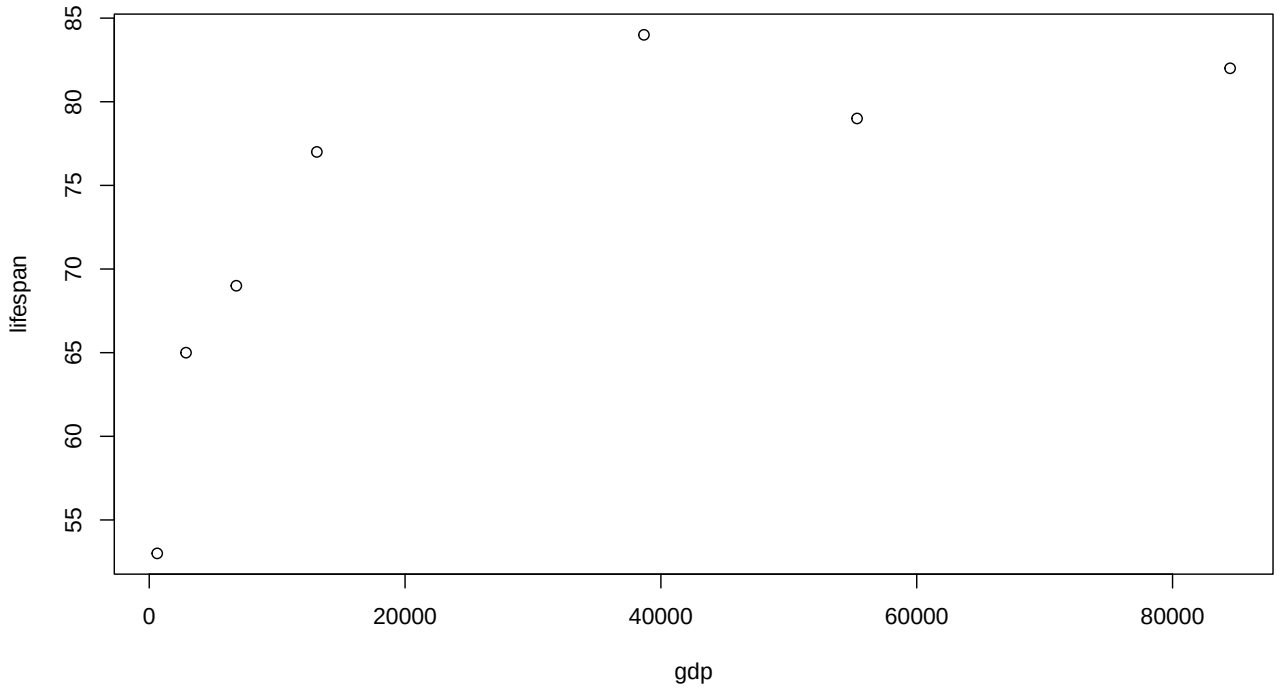
So far in this tutorial we have been using vectors to produce plots. This is nice for learning, but does not represent the real world very well. You will almost always be producing plots using dataframes.

Let's turn these vectors into a dataframe:

```
df <- data.frame(country,lifespan,gdp)
df
           country lifespan   gdp
1              USA       79 55335
2         Tanzania       65  2875
3            Japan       84 38674
4 Ctr. Africa Rep.       53   623
5            China       77 13102
6           Norway       82 84500
7            India       69  6807
```

To plot data within a dataframe, your `plot()` syntax changes slightly:

```
plot(lifespan ~ gdp, data=df)
```



This syntax is saying this: using the dataframe named `df` as a source, plot column `lifespan` as a function of column `gdp`. The symbol `~` denotes *"as a function of"*. This frames `lifespan` as a dependent variable (y axis) that is affected by the independent variable (x axis), which in this case is `gdp`.

Another way to write this command is as follows:

```
plot(df$lifespan ~ df$gdp)
```

In this command, as you learned in the dataframes module, the `$` symbol is saying, "give me the column in `df` named `lifespan`". It is a handy way of referring to a column within a dataframe by name.

Exercises

The economics of health

1. Using the data above, produce a bar plot that shows the GDP for each country.

2. Use the `df` dataframe you built above to produce a bar plot that shows life expectancy for each country.

3. Use the `df` dataframe to produce a jumbled line plot of life expectancy as a function of GDP. Reference the `plot()` documentation to figure out how to change the thickness of the line.

Shoe ~ height correlation

4. Create a vector of the names of 5 people who are sitting near you. Call it `people`.

5. Create a vector of those same 5 people's shoe sizes (in the same order!). Call it `shoe_size`.

6. Create another vector of those same 5 people's height. Call it `height`.

7. Create another vector of those same 5 people's sex. Call it `sex`.

8. Make a scatterplot of height and shoe size. Is there a correlation?

9. Look up help for `boxplot`. Make a `boxplot`.

10. Make a dataframe named `df`. It should contain columns of all the vectors created so far.

11. Make a side by side `boxplot` with shoe sizes of males vs females.

12. Try to find documentation for making a "histogram" (hint: use text autocomplete).

13. Make a histogram of people's height.

Make your plot beautiful

14. Produce a *beautifully* formatted plot that incorporates **all** of the customization inputs explained above into a multi-paned plot. Use any dataset from this module that you wish.

Other Resources

- R color palette
- The R Graph Gallery

Chapter 15

ggplot

Learning goals

- Understand what `ggplot2` is and why it's used
- Be able to think conceptually in the framework of the "grammar of graphics"
- Learn the basic syntax for creating different plots using using `ggplot2`

What is `ggplot2`?

`ggplot2` is an `R` package. It's one of the most downloaded packages in the `R` universe, and has become the gold standard for data visualization. It's extremely powerful and flexible, and allows for creating lots of visualizations of different types, ranging from maps to bare-bones academic publications, to complex, paneled charts with labeling, etc.

Because the syntax is so different from "base" `R`, it can give the impression of having a somewhat steep learning curve. But in reality, because the principles are so conceptually simple, learning is fairly fast. Generally those who choose to learn it stick with it; that is, once you go `gg`, you don't go back.

Note: we will refer heavily to this **online guide about `ggplot`**

The name & concept

```
Error in file(filename, "r", encoding = encoding): cannot open the connection
```

```
Error in teacher_tip(tip): could not find function "teacher_tip"
```

The "`gg`" in `ggplot` stands for "grammar of graphics", with "grammar" meaning "the fundamental principles or rules of an art or science" [**?**]. Just as all languages share common principles of grammar and syntax, so too do the many forms of data visualization. The basic idea is that all graphs can be described using a *layered* grammar: all graphs represent a dataset using the same layers of visual order.

Plots are made of layers. Think of how you draw a plot from scratch:

First, you get a piece of paper – a **canvas**.

Second, you draw the x axis and y axis: each direction on your canvas represents the range of a set of data. This establishes a **landscape of coordinates**.

Third, the data need to be placed somewhere in that landscape. You **map the data** to the coordinates.

Fourth, when you actually draw the data at their prescribed locations on the plot, you have to decide how to do so. You use **geometric objects** – like points, lines, and bars – and other **aesthetic attributes** – like colors, line thicknesses, and dot size.

Fifth, you add **labels** – such as axis titles, an overhead title, or a legend – to help the viewer understand the plot.

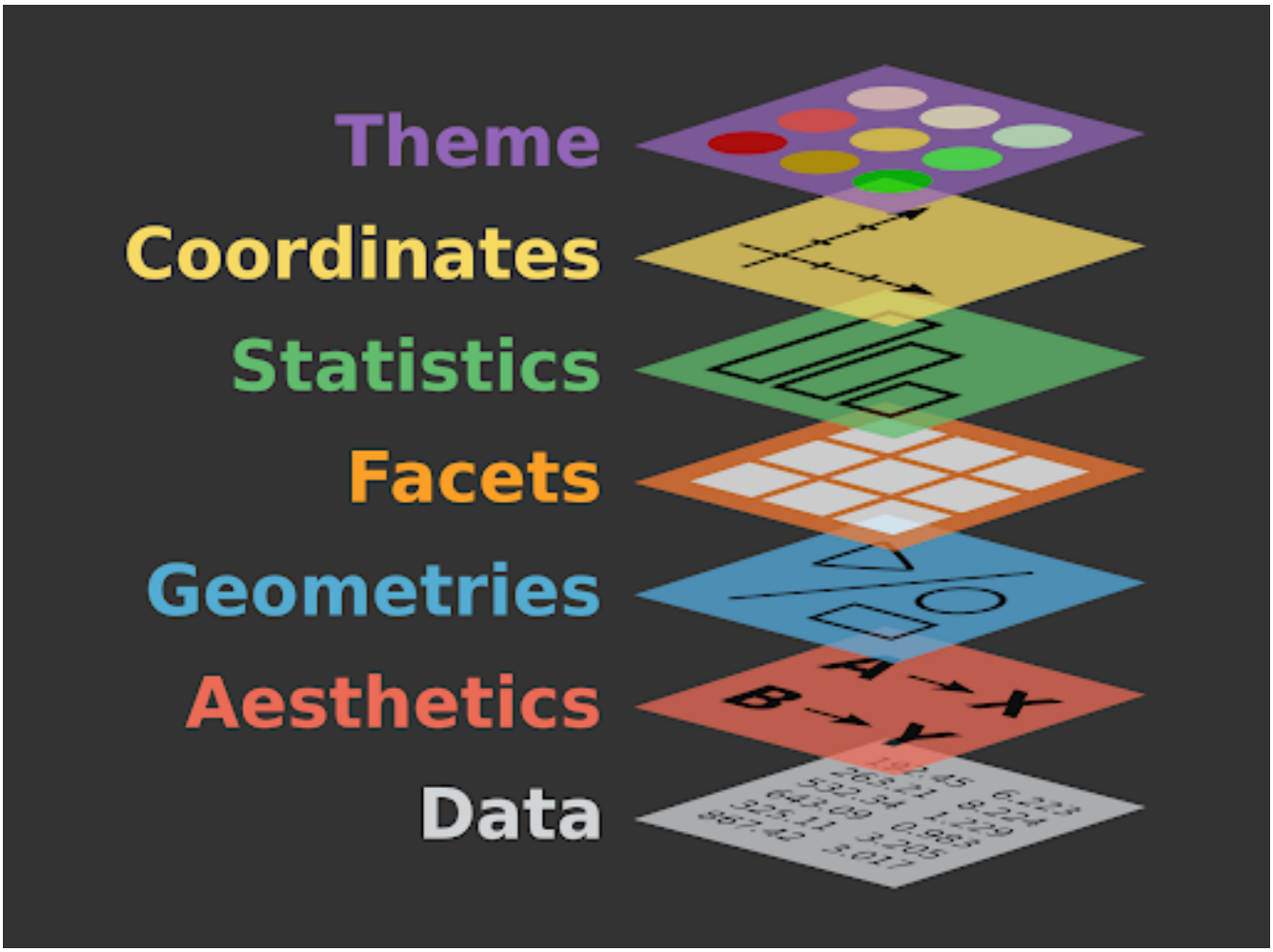
You now have a basic plot. But sometimes you will add additional layers:

Sixth, you may add **statistical summaries** – such as regression lines or standard error bars.

Seventh, you may decide to do an overhaul and split your plot into several **facets**, in which subgroups of the data are plotted separately to produce a multi-panel plot.

Finally, in the final layer, you may decide to stylize the entire plot to fit a **visual theme**, such as the trademark styles of vendors like the *The Economist* or *The New York Times*.

When you produce a plot with `ggplot`, you will mirror this same process step-by-step. This is why you will often see the process underlying `ggplot` described using a graphic like this:



Note: If you want to learn more about the theory, the most well-known "grammar of graphics" was written in 2005 and laid out some abstract principles for describing statistical graphics [**?**].

Setting up `ggplot`

Let's learn by doing. First, install and load `ggplot2` and associated packages.

```
library(ggplot2)
library(readr)
library(ggthemes)
library(dplyr)
```

Download the `titanic` dataset, the manifest of *Titanic* passengers with details such as as age, passenger class, fare paid, and whether or not they survived.

```
titanic<- read_csv("https://raw.githubusercontent.com/databrew/intro-to-data-science/main/data/deaths.csv"
```

```
Error in file(filename, "r", encoding = encoding): cannot open the connection
```

```
Error in teacher_tip(tip): could not find function "teacher_tip"
```

Scatter plot

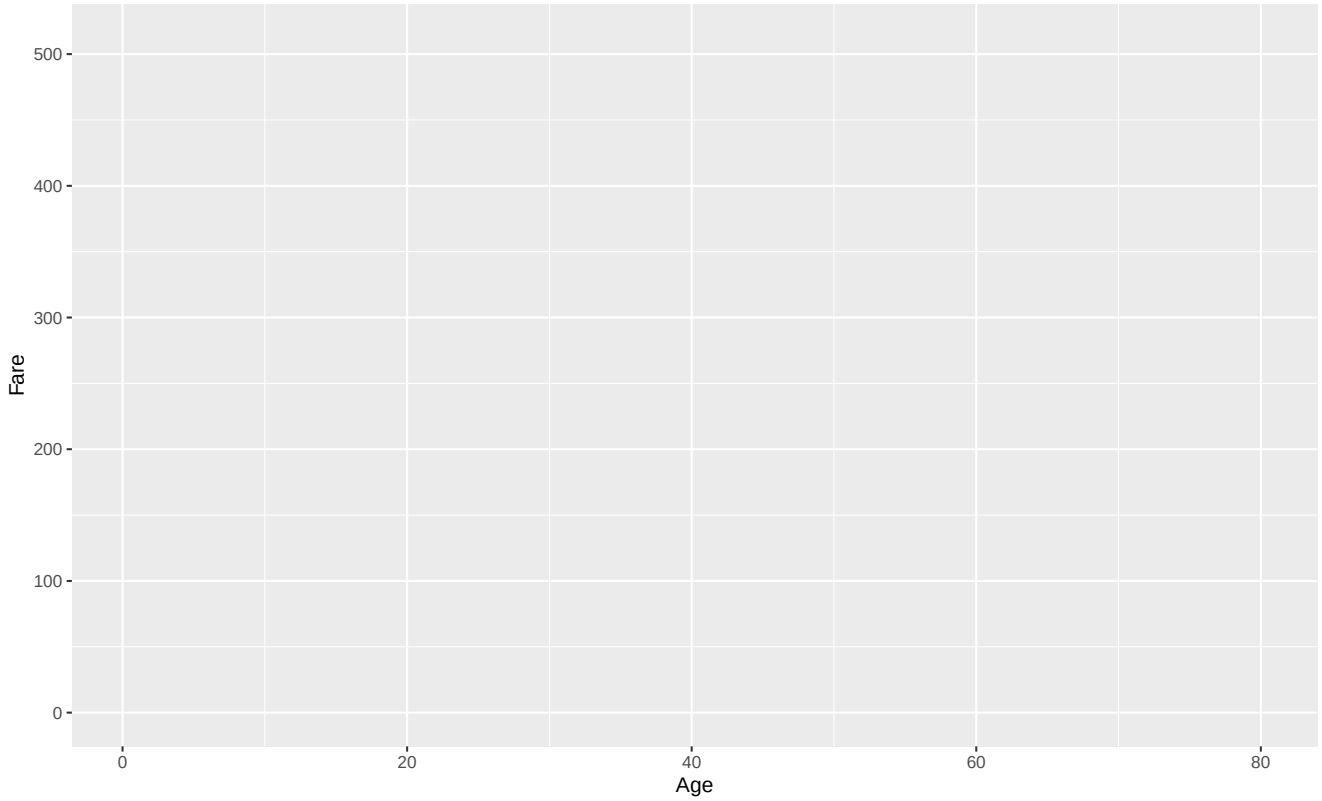
Say you want to explore the relationship between passengers' age and the fare they paid to travel aboard the *Titanic*.

(1) **Set up our canvas.** If we just type `ggplot()` without anything in the parentheses, the function will just return a blank piece of paper.

```
ggplot()
```


(2) **Draw the axes** and, 3., setup our **landscape of coordinates**. To do so, we need to feed `ggplot()` some data and tell it which columns should be mapped onto the axes.

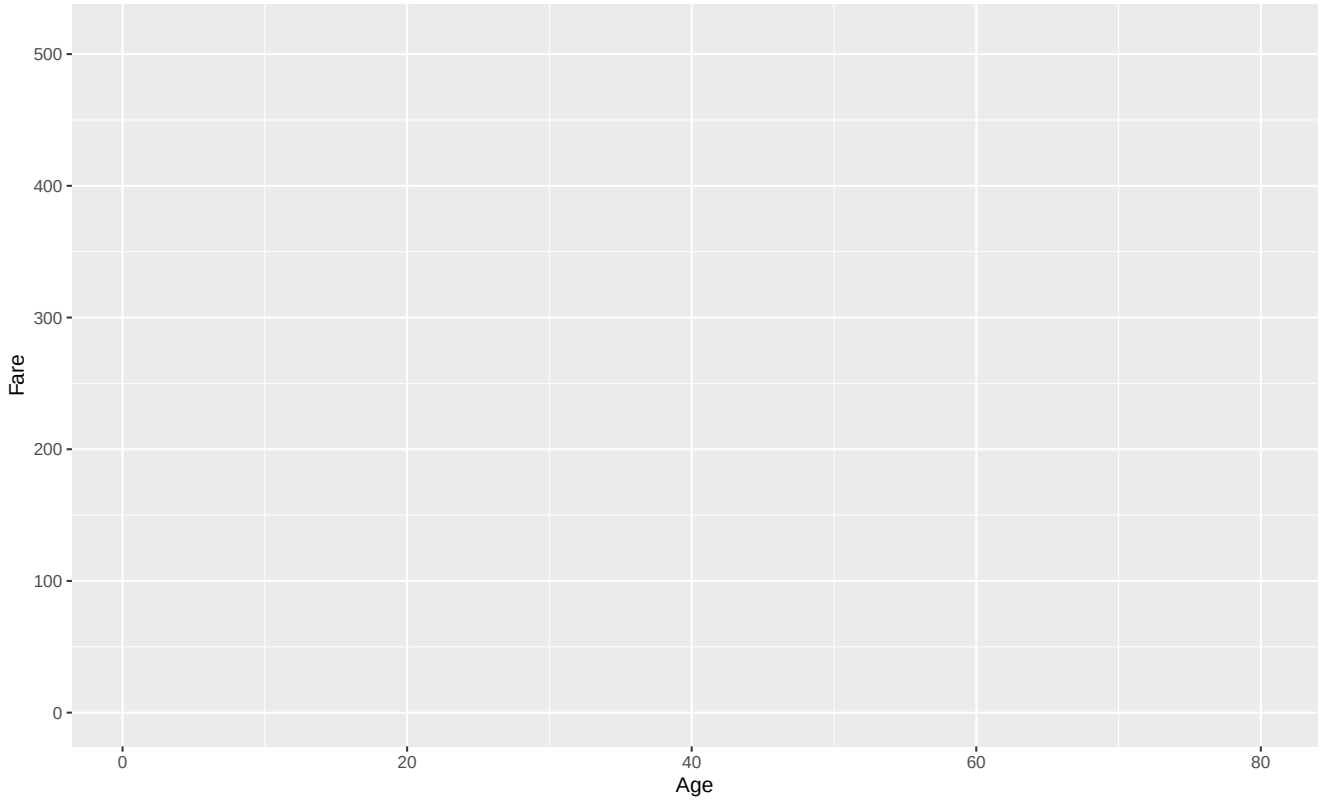
```
ggplot(data = titanic,
       aes(x = Age, y = Fare))
```



That code looks a bit clunky, we know. The `aes()` input, which is short for `aesthetics`, is actually a function. Everything included in its parentheses will be used to *map your data to the plot's aesthetic attributes*. So far we have simply said that `Age` should be mapped to the x axis and that `Fare` should be mapped to y.

But let's say we also want to color-code the points on our plot according to male/female. To do so, we will add specifications to this `aes()` function.

```
ggplot(data = titanic,
       aes(x = Age, y = Fare, color=Sex))
```



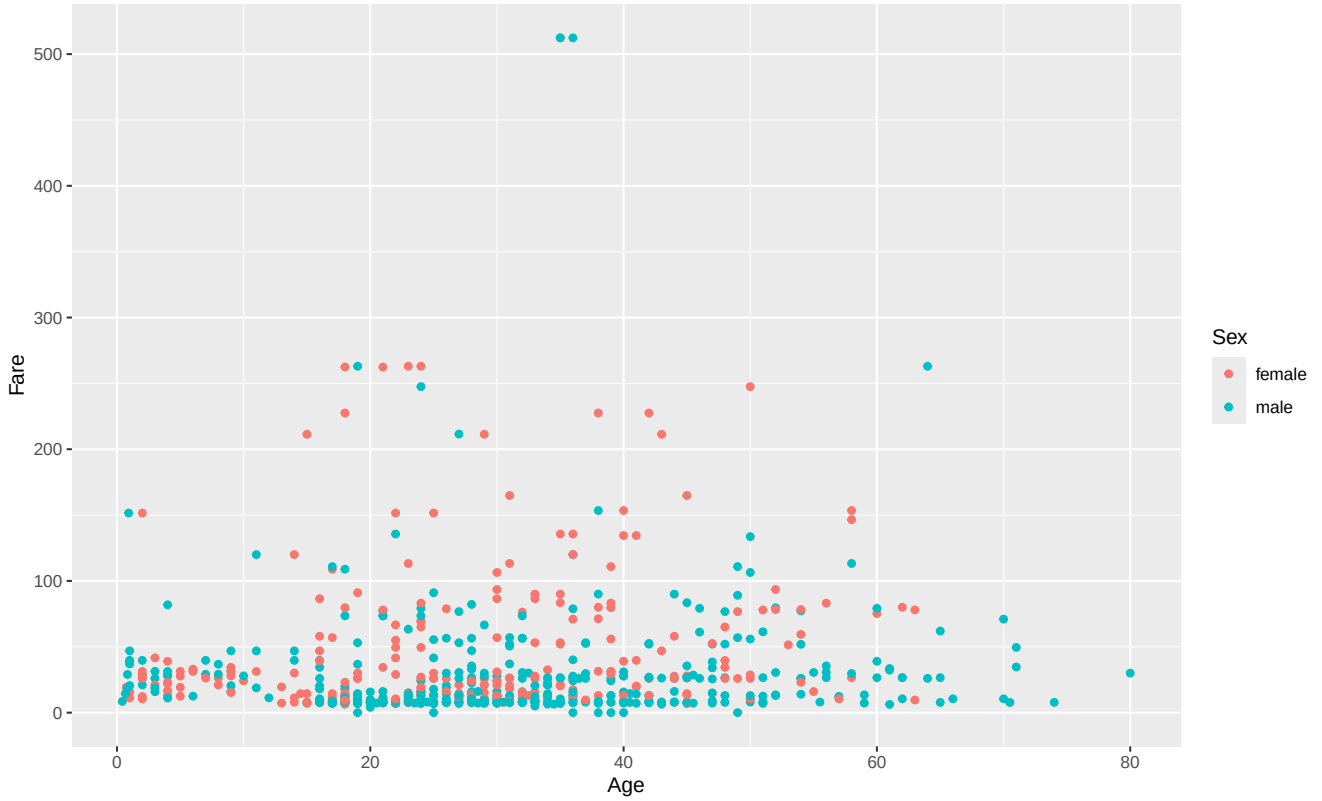
Your plot is still blank, but in the background `ggplot()` is all setup to make your plot. Since this `ggplot()` call is the basis of everything that will happen next – it contains the data and the way you want to map it to attributes of your plot – let's save it to a variable for easy recall. We'll use `p` for "plot".

```
p <- ggplot(data = titanic,
       aes(x = Age, y = Fare, color=Sex))
```

Note that you don't need to write out `titanic$Age` or `titanic$Fare`. You've told `ggplot` that your `data` is `titanic`, so it knows to look inside that dataframe for those columns.

(4) **Map our data to geometric shapes**. In this case, a scatter plot of points:

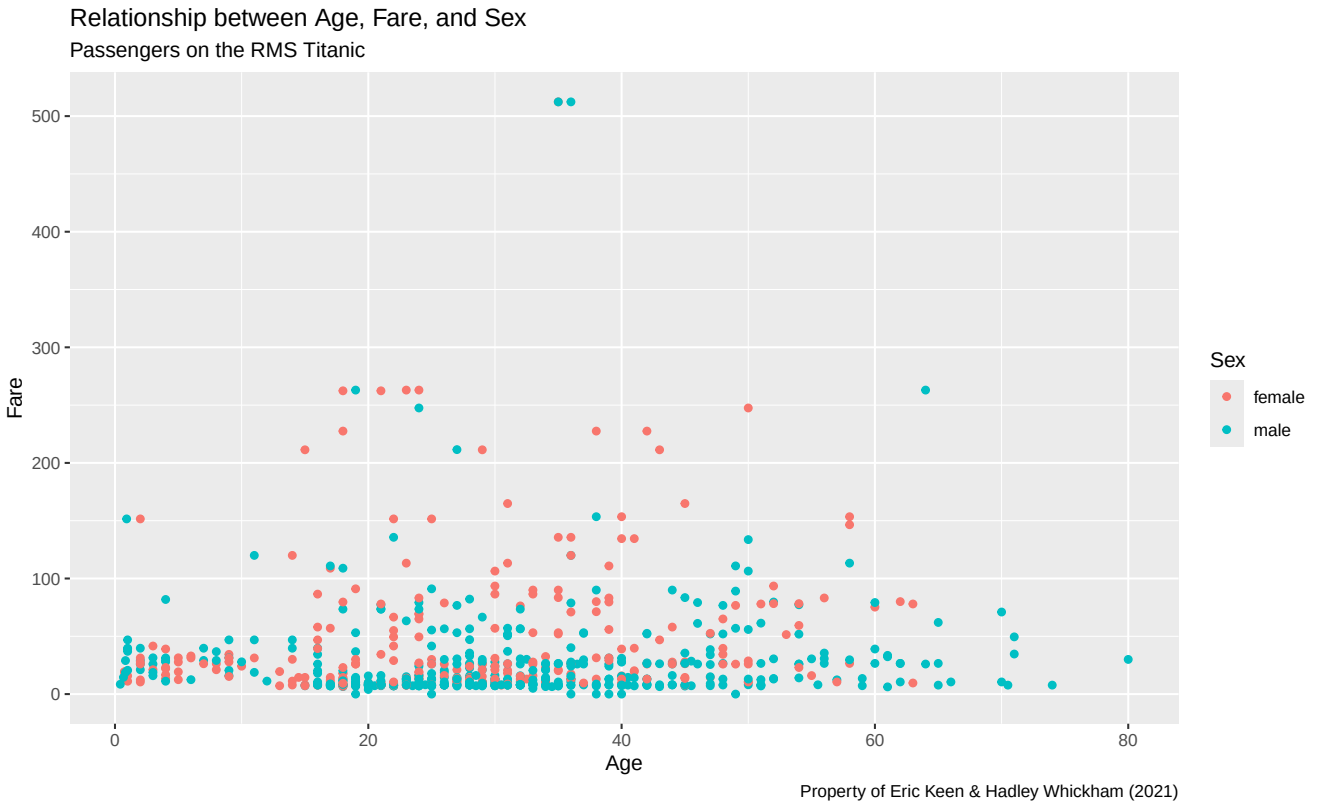
```
p <- p + geom_point()
p
```



Note the use of a plus sign, `+`. You are *adding* layers to your plot.

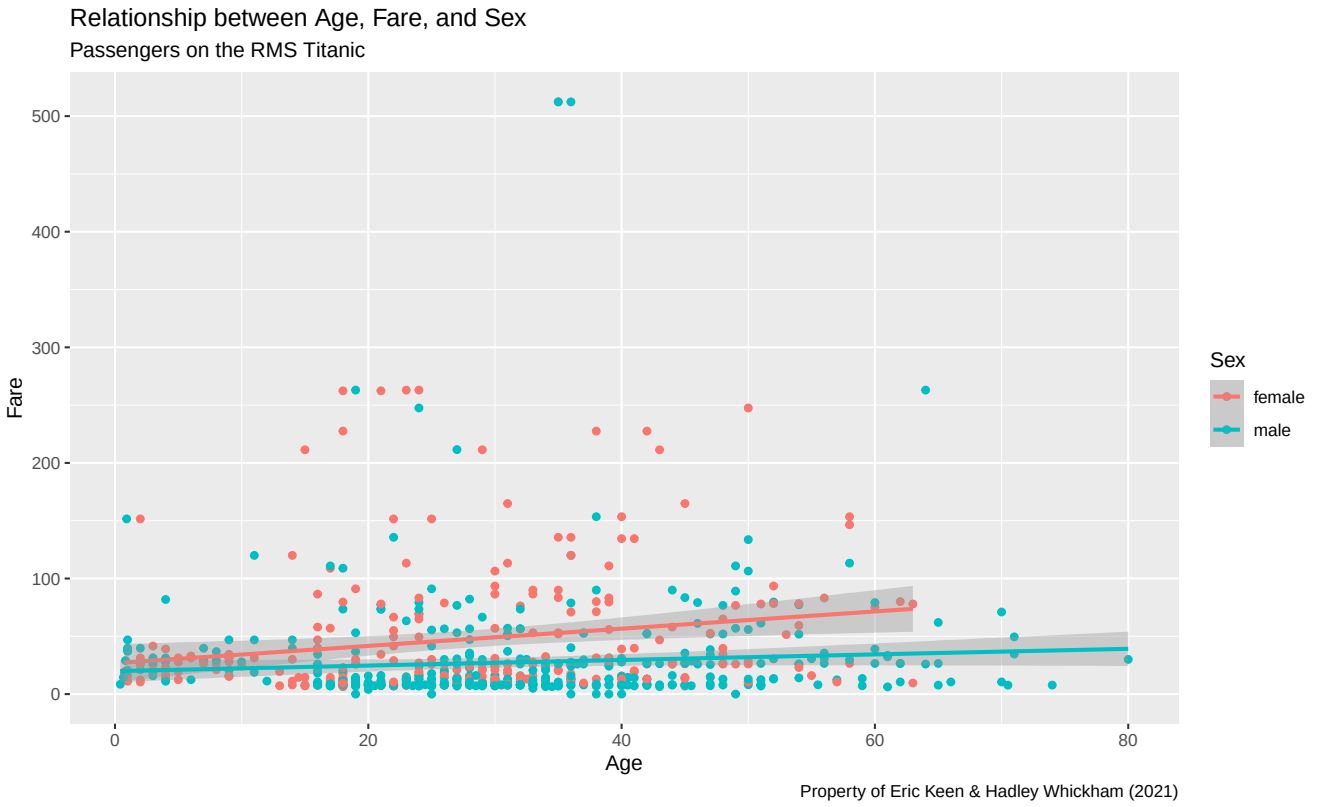
(5) **Add some more labels**. You see that `ggplot()` has automatically added axis titles and a legend, but we can add some more using the `labs()` function. Let's add an overhead title, a sub-title, and a caption.

```
p <- p + labs(title = 'Relationship between Age, Fare, and Sex',
              subtitle = 'Passengers on the RMS Titanic',
              caption = 'Property of Eric Keen & Hadley Whickham (2021)')
p
```



(6) **Add a statistical summary**, like a smoothed regression line.

```
p <- p + geom_smooth(method = 'lm')
p
```



Note that `ggplot()` automatically produced a different regression line for each sex. That's nice, but now our plot is getting pretty cluttered.

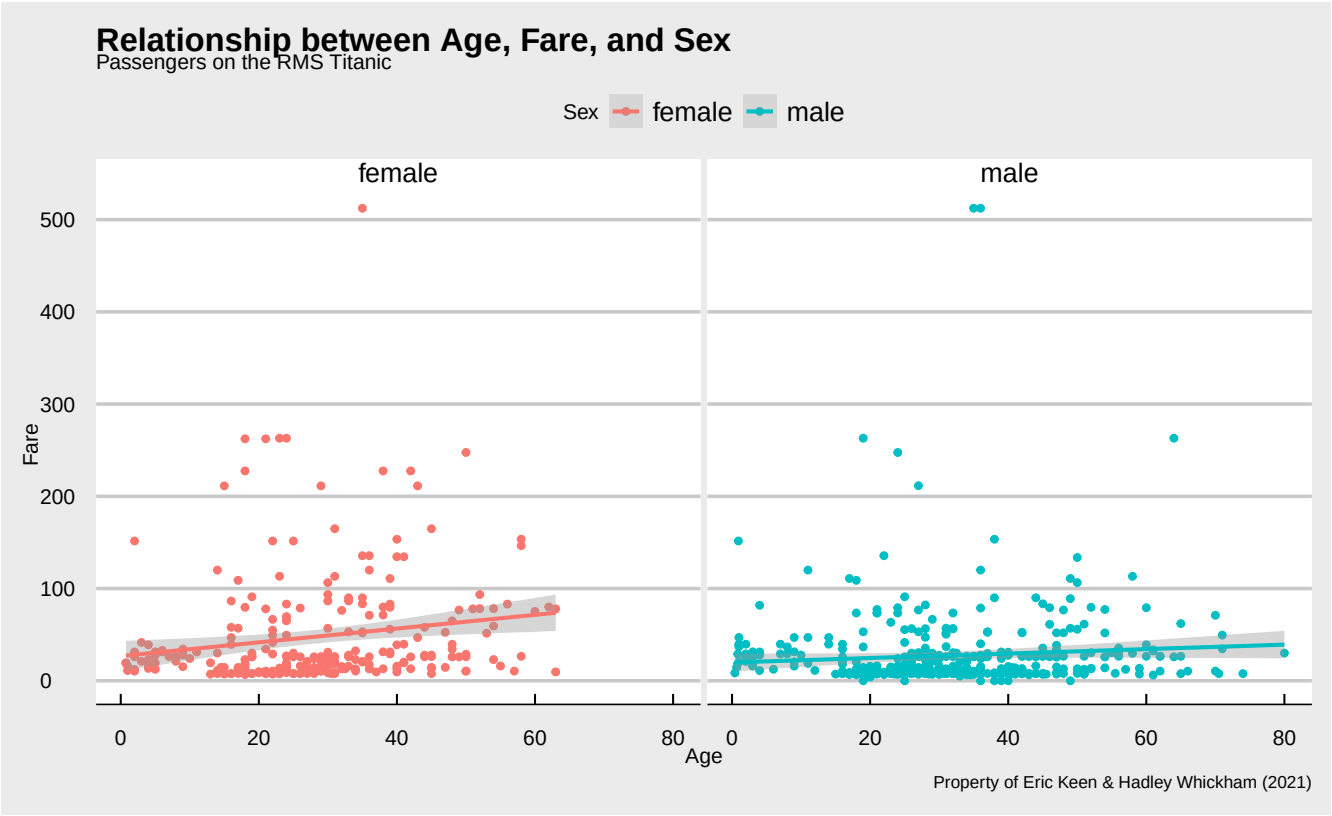
(7) Clean up the look by using **facets**: a separate plot for each sex.

```
p <- p + facet_wrap(~Sex)
p
```

Beautiful!

(8) Finally, let's **stylize** the entire plot with a different **theme**. You can find theme options in the '`ggthemes` package.

```
p <- p + theme_economist_white()
p
```

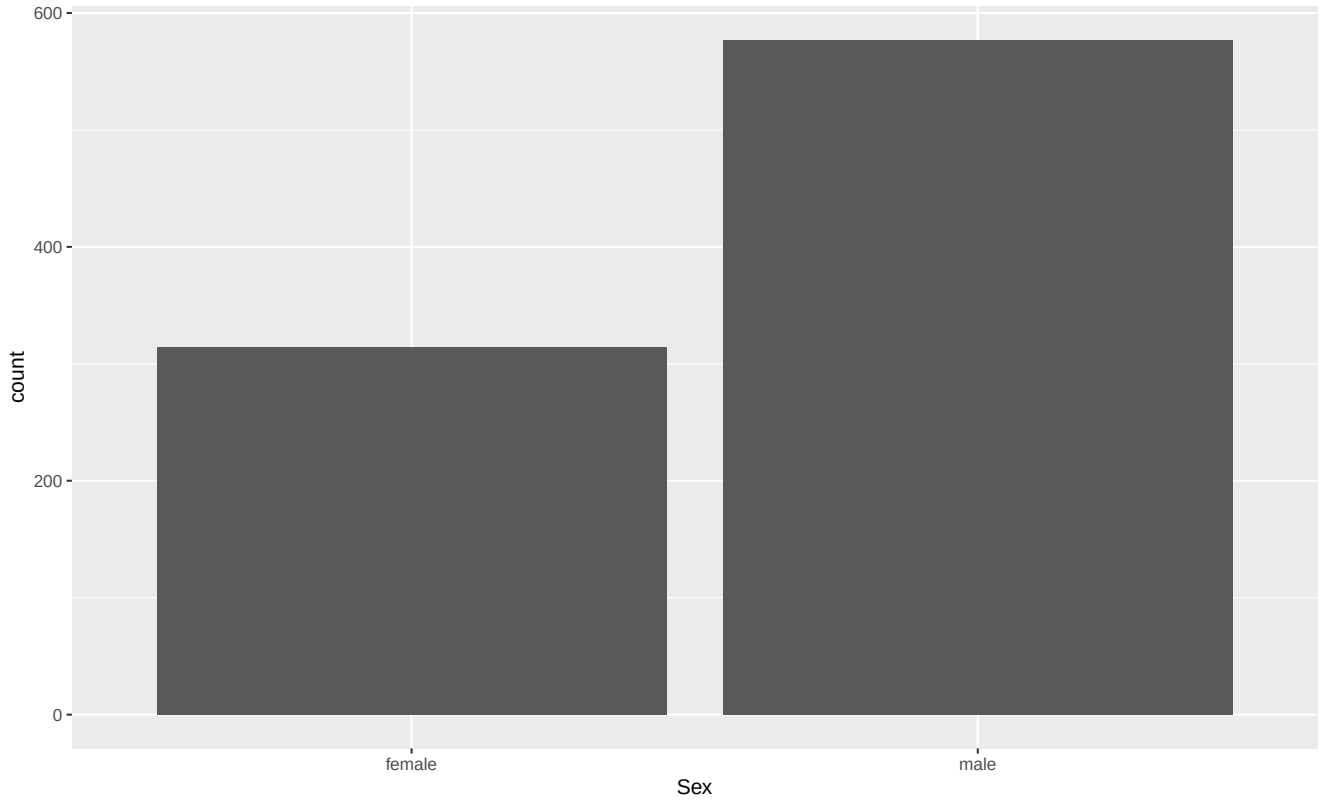


Bar plot

In a bar plot, your data are mapped to bars instead of points. And, instead of showing every data point, you are summarizing the data in some way – i.e., displaying a *statistic*. That statistic is usually just a count of the number of data points in each subgroup.

Let's make a bar plot that compares the number of men and women on the *Titanic*:

```
ggplot(data = titanic,
       aes(x = Sex)) +
  geom_bar(stat='count')
```

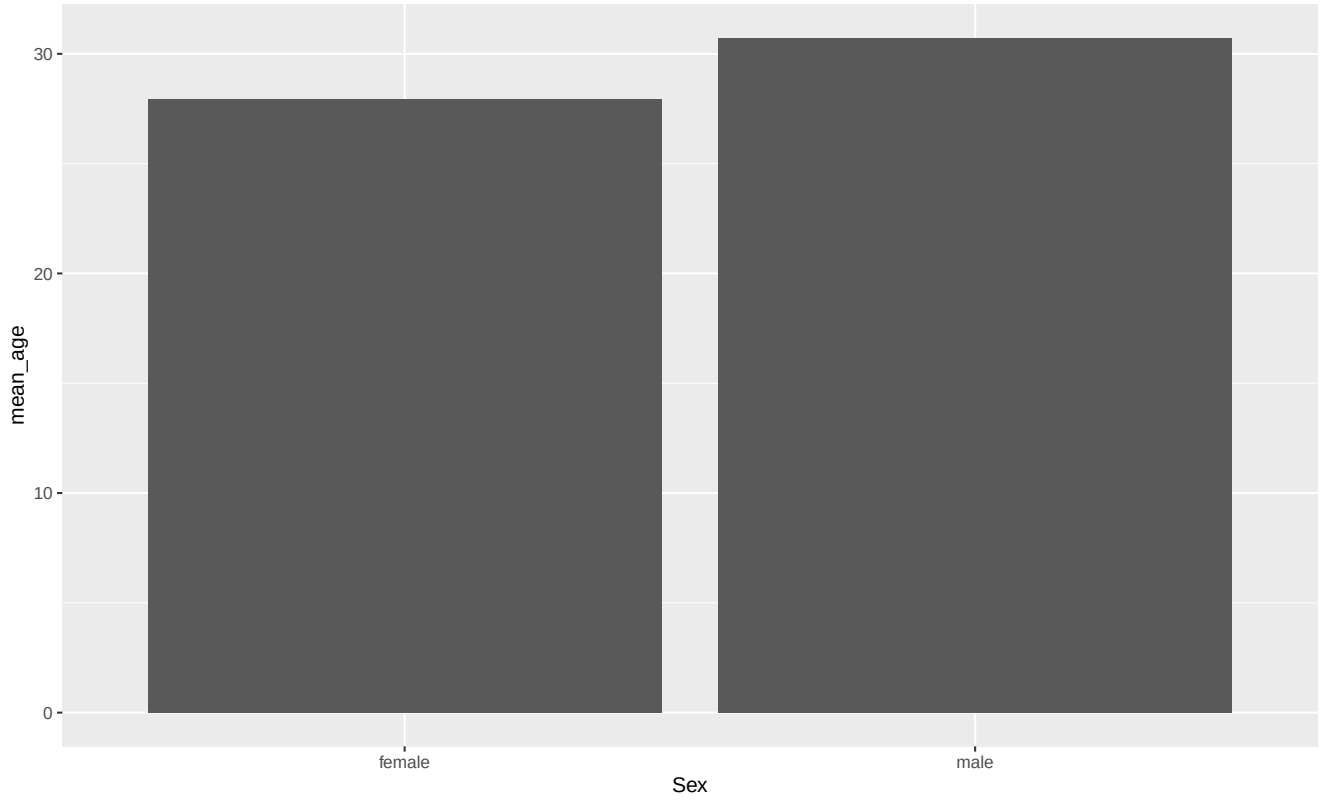


Note that, for the `aes()` call, we only provided the x axis attribute: `Sex`.

Then, in the `geom_bar()` call, we told `R` what `statistic` should be represented by that bars: `'count'`.

But you are allowed to explicitly set the bars' heights (i.e., the `y` dimension) to represent a different statistic. Let's say we wanted each bar to represent the mean age of men and women:

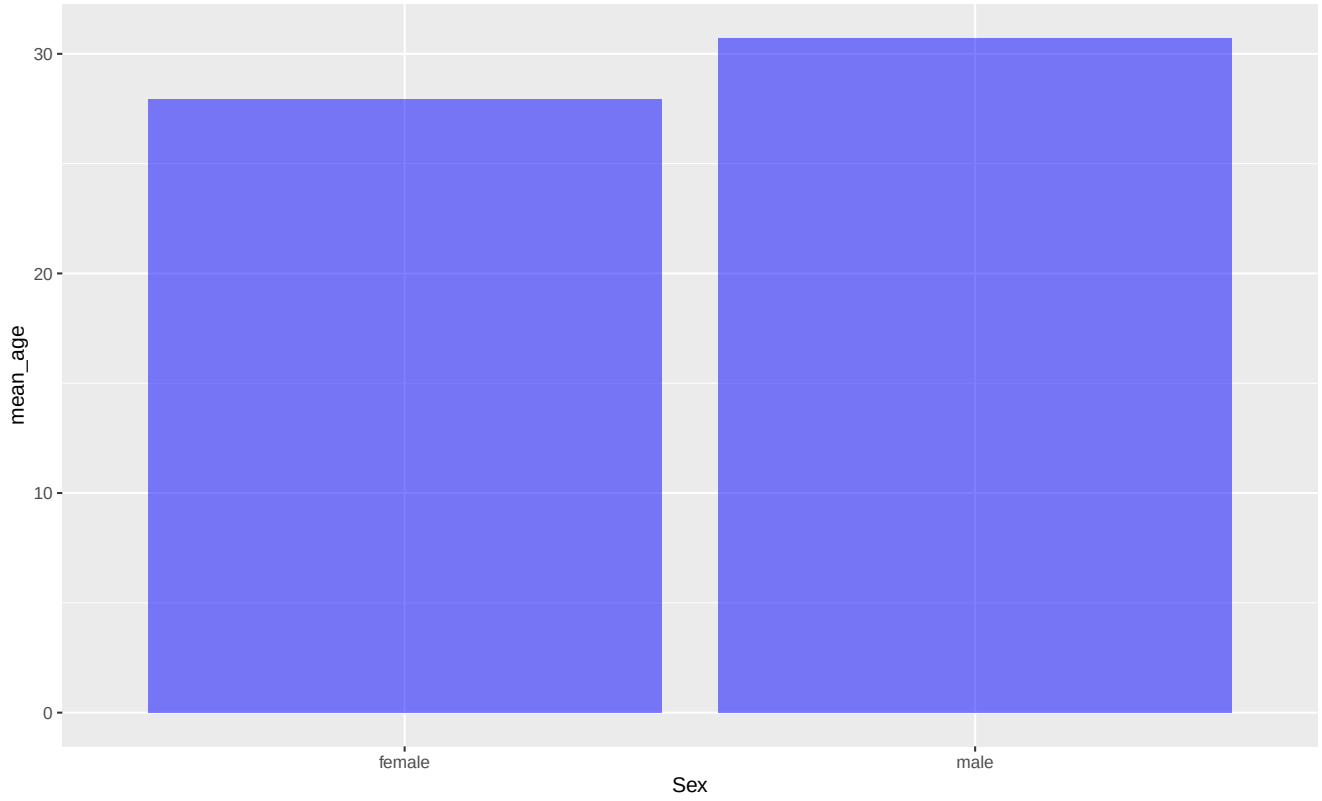
```
# First, determine the mean age of each sex
mean_age_males <- mean(titanic$Age[titanic$Sex == 'male'], na.rm = TRUE)
mean_age_females <- mean(titanic$Age[titanic$Sex == 'female'], na.rm = TRUE)

# Make a new dataframe with this summary data
titanic_age <- data.frame(Sex = c('male','female'),
                          mean_age = c(mean_age_males, mean_age_females))
# Plot it
ggplot(data = titanic_age,
       aes(x = Sex,y= mean_age)) +
  geom_bar(stat = 'identity')
```



In this case, we are explicitly defining the `y` axis in the `aes()` call, and telling `geom_bar()` to just use the values we specified in `aes()` (that's what `'identity'` means; you are telling `ggplot()` to just use what you already gave it.)

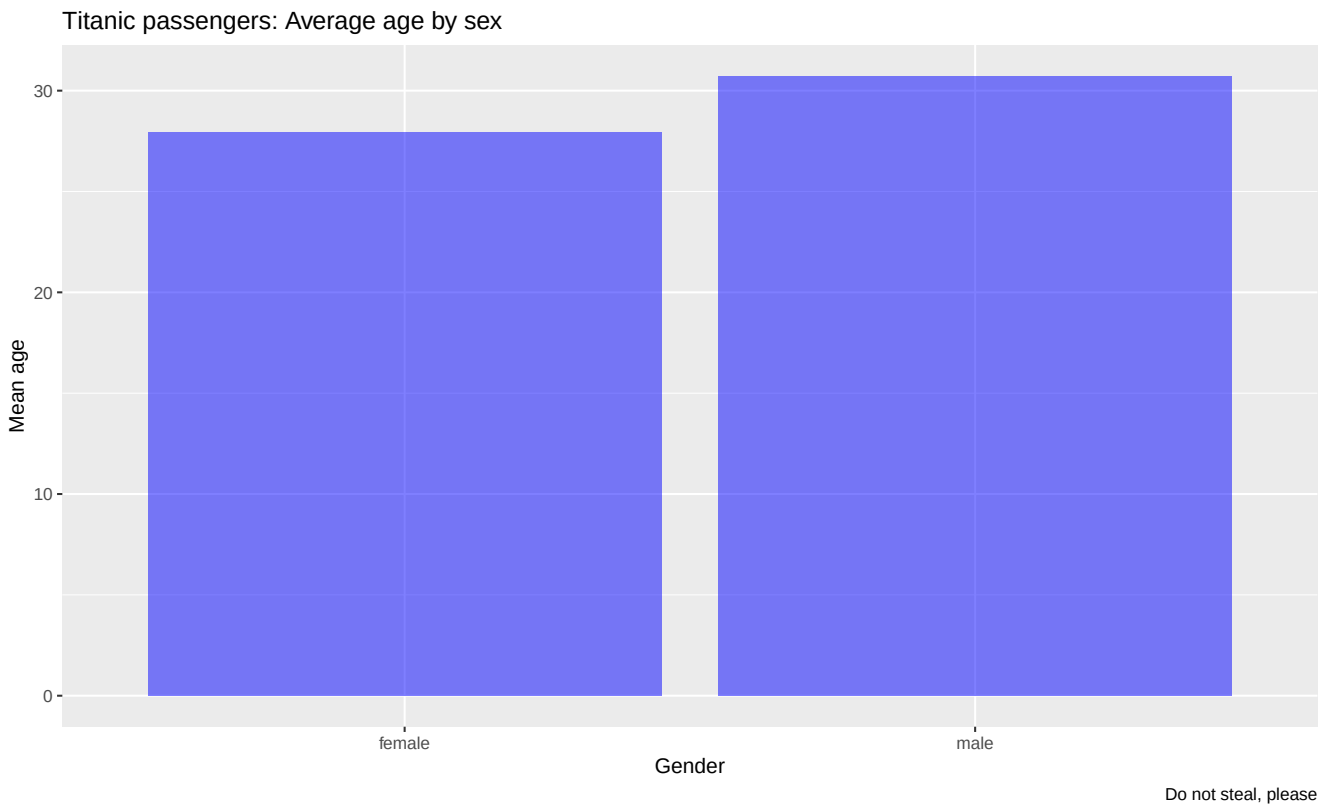
You can specify other aesthetic attributes, *unrelated to the data*, within the `geom_bar()` call:

```
ggplot(data = titanic_age,
       aes(x = Sex,y= mean_age)) +
  geom_bar(stat = 'identity', fill = 'blue', alpha = 0.5)
```



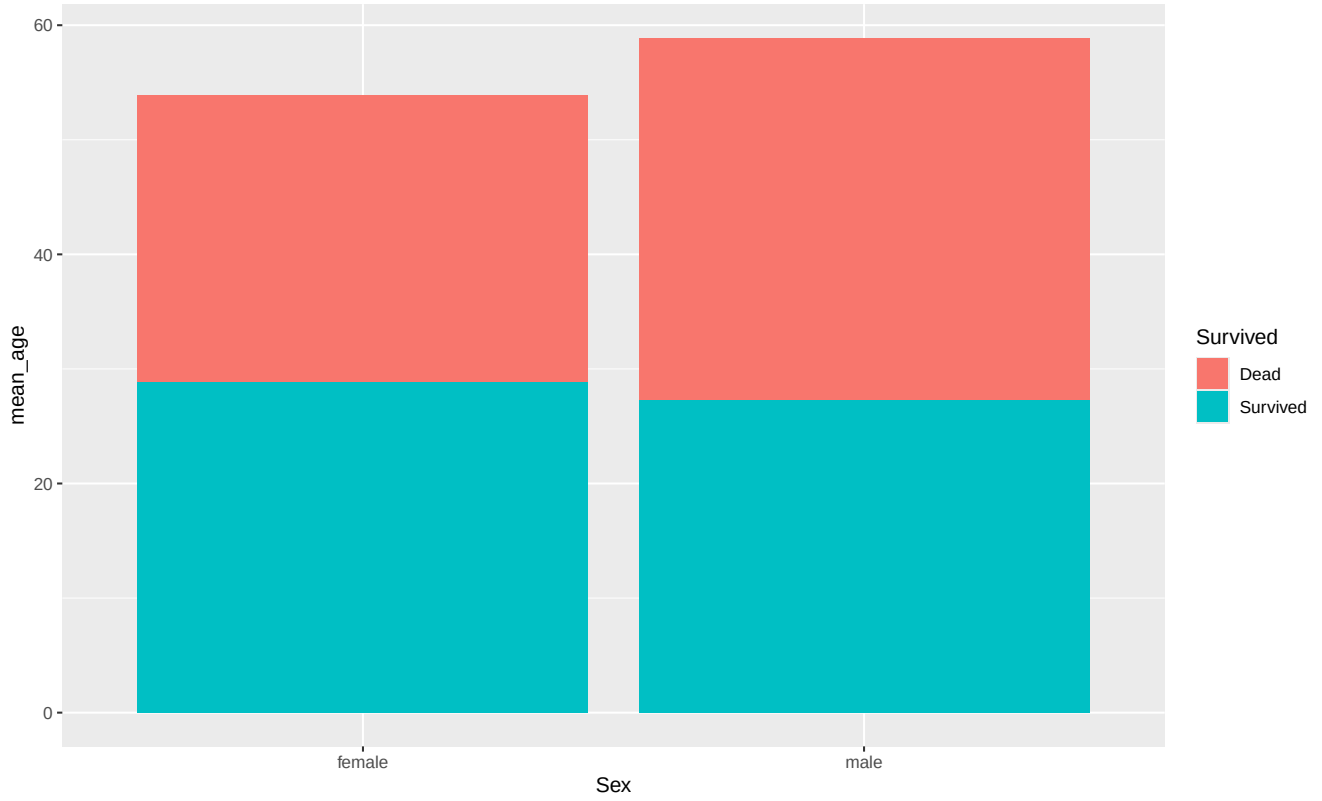
Now add better labels:

```
ggplot(data = titanic_age,
       aes(x = Sex,y= mean_age)) +
  geom_bar(stat = 'identity', fill = 'blue', alpha = 0.5) +
  labs(y = 'Mean age',
       x = 'Gender',
       title = 'Titanic passengers: Average age by sex',
       caption = 'Do not steal, please')
```



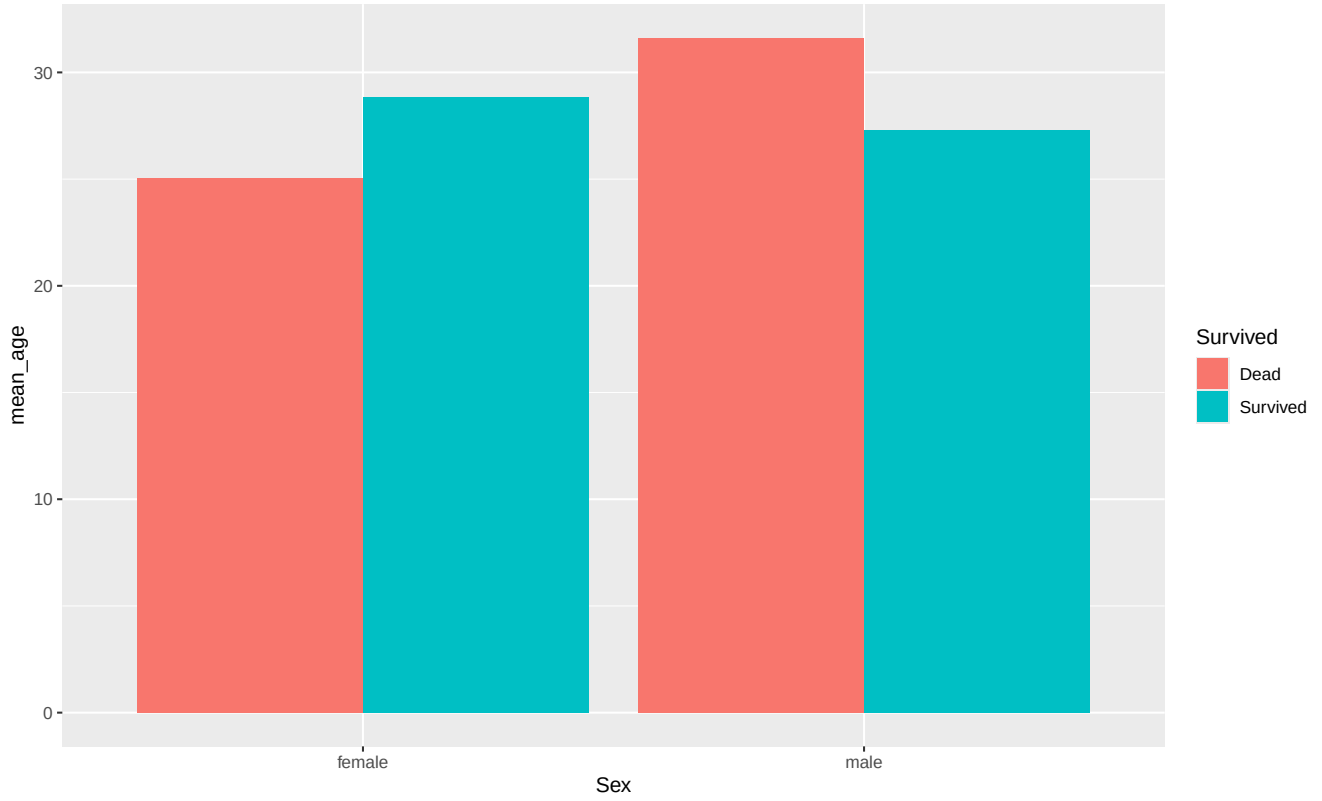
You can add another variable to your bar plot as follows. Let's say you want to see the average age in each sex, grouped by who survived and who didn't:

```
# First, produce your summary dataframe using some dplyr magic:
titanic_em <- titanic %>%
  group_by(Sex, Survived) %>%
  summarise(mean_age= mean(Age, na.rm = TRUE))

titanic_em <- titanic_em %>%
  mutate(Survived = ifelse(Survived == 1, 'Survived','Dead' ))

# Check it out
titanic_em
# A tibble: 4 x 3
# Groups:   Sex [2]
  Sex    Survived mean_age
  <chr>  <chr>       <dbl>
1 female Dead         25.0
2 female Survived     28.8
3 male   Dead         31.6
4 male   Survived     27.3

# Now plot it
ggplot(data = titanic_em,
       aes(x=Sex, y=mean_age, fill = Survived)) +
  geom_bar(stat='identity')
```



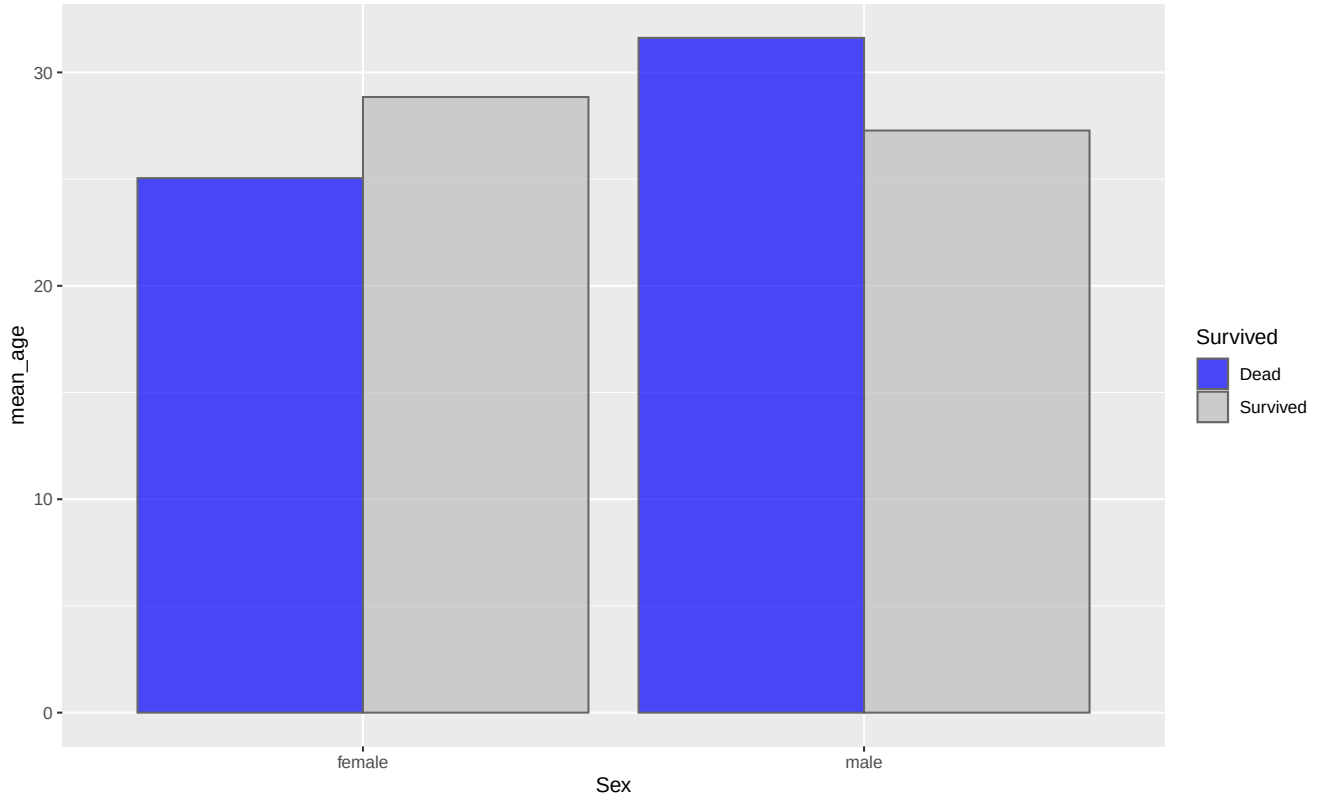
Rather than stack the bars, you can place them side by side:

```
ggplot(data = titanic_em,
       aes(x=Sex, y=mean_age, fill = Survived)) +
  geom_bar(stat='identity', position = 'dodge')
```



If you don't love these default colors (even if they are colorblind-friendly), you can manually define the colors for each group of bars:

```
ggplot(data = titanic_em,
       aes(x=Sex, y=mean_age, fill = Survived)) +
  geom_bar(stat='identity',
           position = 'dodge',
           alpha = 0.7,
           color='grey40') + # bar edge
  scale_fill_manual(values = c('blue', 'grey')) # bar fill
```



Exercises

More *Titanic* plots

1. Make a scatterplot similar to what you did above, but this time color-code by class instead of sex.

2. Notice that `ggplot()` automatically uses a continuous color scale for `Pclass`, since it has numeric values. To force `ggplot()` to consider `Pclass` as categories (1st class, 2nd class, 3rd class), replace `Pclass` with `factor(Pclass)`. Did the style of your color scale change?

3. Modify the title, subtitle, and caption to be more descriptive.

4. Produce a bar plot that compares the number of passengers in each class.

5. Make your bar plot as *ugly* as possible!

6. Now make it as *beautiful* as possible, including a concise but informative title, subtitle, and caption.

Baby names

Download the dataset on baby names given to newborns in the USA:

```
library(babynames)
bn <- babynames
```

7. Create a line chart showing the number of girls named Mary over time.

8. Change the color of the line to blue.

9. Add a fitting title to the plot.

10. Create a bar chart showing the number of girls named Emma, Olivia, Ava, Sophia, and Emily in 2010.

11. Change the X label to "Names" and the y label to "Total". (*Hint:* check out the `labs()` help page.)

12. Change the color of the bar to grey and make it more transclucent.

13. Create a bar chart showing the number of people named Emma, Olivia, Ava, Sophia, and Emily in 2010, colored by sex.

14. Create a beautiful chart showing your name over time.

Chapter 16

Dataframe wrangling

Learning goals

- Understand the importance of *tidy* dataframes
- Understand what the `tidyverse` is and why it is awesome
- Feel comfortable working with dataframes using `dplyr` functions.

The `dplyr` package

Data scientists largely work in data frames and *do things* to data. This is what the package `dplyr` is optimized for. It consists of a series of "verbs" which cover 95% of what you need to do for most basic data processing tasks.

```
install.packages('dplyr') # if you haven't yet
```

```
library(dplyr)
```

The `dplyr` package contains a set of `verbs`: things you do to dataframes. Those verbs are:

- `filter()`
- `arrange()`
- `select()`
- `rename()`
- `distinct()`
- `mutate()`
- `summarise()`

The %>% pipe

`%>%` is a "pipe". It is a way to write code without so many parentheses. For example, what if I want to find the square root of the sum of the first six elements of a sequence of 10 to 20 by 2?

Here's what that command would look like in base `R`:

```
sqrt(sum(head(seq(10, 20, 2))))
[1] 9.486833
```

Pretty overwhelming, and pretty easy to make errors in writing it out.

But the above could also be written a simpler way:

```
seq(10, 20, 2) %>% head %>% sum %>% sqrt
[1] 9.486833
```

When you see the **%>%** pipe symbol, think of the word **"then"**.

The above code could be read aloud like so: "First, get a sequence of every second number between 10 and 20. **Then**, take the first six values. **Then**, sum those samples together. **Then**, take the square root of that sum."

Using the %>% pipe framework, your code turns from a nonlinear series of parentheses and brackets to a linear progression of steps, which is a closer fit to how we tend to think about working with data. Instead of working from the inside of a command outward, we thinking linearly: take the data, **then** do things with it, **then** do more things with it, etc.

Here's another example:

```
mean(sd(1:100))
[1] 29.01149
```

… could also be written as:

```
1:100 %>% sd %>% mean
[1] 29.01149
```

dplyr verbs

To practice the `dplyr` verbs, let's make a small dataframe named `people`:

```
people <- data.frame(who = c('Joe', 'Ben', 'Xing', 'Coloma'),
                     sex = c('Male', 'Male', 'Female', 'Female'),
                     age = c(35, 33, 32, 34))
people
     who    sex age
1    Joe   Male  35
2    Ben   Male  33
3   Xing Female  32
4 Coloma Female  34
```

filter()

The `filter()` function is used to subset a dataframe, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of `TRUE` for all conditions.

```
people %>% filter(sex == 'Male')
  who  sex age
1 Joe Male  35
2 Ben Male  33
```

```
people %>% filter(sex == 'Female')
     who    sex age
1   Xing Female  32
2 Coloma Female  34
```

You can also filter according to multiple conditions. Here are three ways to acheive the same thing:

```
people %>% filter(sex == 'Female' & age < 33)
   who    sex age
1 Xing Female  32
```

```
people %>% filter(sex == 'Female', age < 33)
   who    sex age
1 Xing Female  32
```

```
people %>% filter(sex == 'Female') %>% filter(age < 33)
   who    sex age
1 Xing Female  32
```

Note that when a condition evaluates to `NA`, its row will be dropped. This differ from the base subsetting works with `[ ... ]`.

`arrange()`

Arrange means putting things in order. That is, `arrange()` orders the rows of a data frame by the values of selected columns.

```
people %>% arrange(age)
     who    sex age
1   Xing Female  32
2    Ben   Male  33
3 Coloma Female  34
4    Joe   Male  35
```

```
people %>% arrange(sex)
     who    sex age
1   Xing Female  32
2 Coloma Female  34
3    Joe   Male  35
4    Ben   Male  33
```

```
people %>% arrange(who)
     who    sex age
1    Ben   Male  33
2 Coloma Female  34
3    Joe   Male  35
4   Xing Female  32
```

To reverse the order, use `desc()`:

```
people %>% arrange(desc(age))
     who    sex age
```

```
1    Joe   Male  35
2 Coloma Female  34
3    Ben   Male  33
4   Xing Female  32
```

You can also arrange by multiple levels:

```
people %>% arrange(sex, age)
     who    sex age
1   Xing Female  32
2 Coloma Female  34
3    Ben   Male  33
4    Joe   Male  35
```

select()

Select only certain variables in a data frame, making the dataframe skinnier (fewer columns).

```
people %>% select(age)
  age
1  35
2  33
3  32
4  34
```

```
people %>% select(sex, age)
     sex age
1   Male  35
2   Male  33
3 Female  32
4 Female  34
```

As you select columns, you can rename them like so:

```
people %>% select(sex, years = age)
     sex years
1   Male    35
2   Male    33
3 Female    32
4 Female    34
```

You can also select a set of columns using the `:` notation:

```
people %>% select(who:sex)
     who    sex
1    Joe   Male
2    Ben   Male
3   Xing Female
4 Coloma Female
```

rename()

The function `rename()` changes the names of individual variables.

This verb takes the syntax `<new_name> = <old_name>` syntax.

```
people %>% rename(gender = sex, years = age, first_name = who)
  first_name gender years
1        Joe   Male    35
2        Ben   Male    33
3       Xing Female    32
4     Coloma Female    34
```

mutate()

The function `mutate()` adds new variables and preserves existing ones.

New variables overwrite existing variables of the same name.

```
people %>% mutate(agein2020 = age - 1)
     who    sex age agein2020
1    Joe   Male  35        34
2    Ben   Male  33        32
3   Xing Female  32        31
4 Coloma Female  34        33
```

```
people %>% mutate(is_male = sex == 'Male')
     who    sex age is_male
1    Joe   Male  35    TRUE
2    Ben   Male  33    TRUE
3   Xing Female  32   FALSE
4 Coloma Female  34   FALSE
```

```
people %>% mutate(average_age = mean(age))
     who    sex age average_age
1    Joe   Male  35        33.5
2    Ben   Male  33        33.5
3   Xing Female  32        33.5
4 Coloma Female  34        33.5
```

You can call `mutate()` multiple times in the same pipe:

```
people %>% mutate(average_age = mean(age)) %>%
  mutate(diff_from_avg = age - average_age)
     who    sex age average_age diff_from_avg
1    Joe   Male  35        33.5           1.5
2    Ben   Male  33        33.5          -0.5
3   Xing Female  32        33.5          -1.5
4 Coloma Female  34        33.5           0.5
```

You can also remove variables can be removed by setting their value to NULL.

```
people %>% mutate(age = NULL)
     who    sex
1    Joe   Male
2    Ben   Male
3   Xing Female
4 Coloma Female
```

A similar function, `transmute()`, adds new variables and drops existing ones, kind of like a combination of `select()` and `mutate()`.

```
people %>% transmute(average_age = mean(age))
  average_age
1        33.5
2        33.5
3        33.5
4        33.5
```

group_by()

Most data operations are done on groups defined by variables. The function `group_by()` takes an existing table and converts it into a grouped one where operations are performed "by group".

```
people %>%
  group_by(sex) %>%
  mutate(average_age_for_sex = mean(age))
# A tibble: 4 x 4
# Groups:   sex [2]
  who    sex      age average_age_for_sex
  <chr>  <chr>  <dbl>               <dbl>
1 Joe    Male      35                  34
2 Ben    Male      33                  34
3 Xing   Female    32                  33
4 Coloma Female    34                  33
```

```
people %>%
  group_by(sex) %>%
  mutate(average_age_for_sex = mean(age)) %>%
  mutate(diff_from_avg_for_sex = age - average_age_for_sex)
# A tibble: 4 x 5
# Groups:   sex [2]
  who    sex      age average_age_for_sex diff_from_avg_for_sex
  <chr>  <chr>  <dbl>               <dbl>                 <dbl>
1 Joe    Male      35                  34                     1
2 Ben    Male      33                  34                    -1
3 Xing   Female    32                  33                    -1
4 Coloma Female    34                  33                     1
```

Note that a similar verb, `ungroup()`, removes grouping.

summarize()

`summarize()` or `summarize()` creates an entirely new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarizing all observations in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

```
people %>%
  summarize(average_age = mean(age))
  average_age
1        33.5
```

```
people %>%
  summarize(average_age = mean(age),
```

```
            standard_dev_of_age = sd(age),
            oldest_age = max(age),
            youngest_age = min(age))
  average_age standard_dev_of_age oldest_age youngest_age
1        33.5            1.290994         35           32
```

```
people %>%
  group_by(sex) %>%
  summarise(avg_age = mean(age),
            oldest_age = max(age),
            total_years = sum(age))
# A tibble: 2 x 4
  sex    avg_age oldest_age total_years
  <chr>    <dbl>      <dbl>       <dbl>
1 Female      33         34          66
2 Male        34         35          68
```

```
people %>%
  group_by(sex) %>%
  summarise(sample_size = n())
# A tibble: 2 x 2
  sex    sample_size
  <chr>        <int>
1 Female           2
2 Male             2
```

Note the use of the function, `n()`. This simple function counts up the number of records in each group.

```
Error in file(filename, "r", encoding = encoding): cannot open the connection
```

```
Error in teacher_tip(tip): could not find function "teacher_tip"
```

Exercises

Answer these questions using the new `dplyr` verbs you just learned:

Baby names over time

1. Run the below code to load a dataset about baby names given in the USA since the 1800's.

```
library(dplyr)
library(babynames)
bn <- babynames
```

2. Check out the first and last six rows of `bn`.

3. What are the names of the variables in this dataset?

4. How many rows are in this dataset?

5. What is the earliest year in this dataset?

6. Create a dataframe named `turn_of_century`, which contain data only for the year 1900.

7. Create a dataframe named `boys`, containing only boys.

8. Create a dataframe named `moms_gen`. This should be females born in the year of birth of your mom.

9. Order `moms_gen` by `n`, in ascending order (i.e., with the least popular name at top). Look at the result; what is the least popular name among women the year your mom was born?

10. Reverse the order and save the result into an object named `moms_gen_ordered`.

11. Create an object named `boys2k`. This should be all males born in the year 2000.

12. Arrange `boys2k` from most to least popular. What was the most popular boys name in 2000?

13. What percentage of boys were named `Joseph` in 2000?

14. Were there more Jims or Matthews in 2020?

15. Create an object named `tot_names_by_year`, which contains the total counts for boy and girl names assigned in each year of the dataset. You should have four columns: `year`, `boys`, `girls`, and `tot`.

16. How many people were born with *your* name in 2020?

17. Was your name more prevalent in 2020 than it was in the year you were born?

18. What if you account for the changing overall population size? In other words, is the *proportional prevalence* of your name greater in 2020 or your birth year?

19. In which year was your name the most prevalent?

20. Create a basic plot of the proportional prevalence of your name since the earliest year of this dataset.

21. Update this plot with lines for your parent's names and your siblings names, if you have any.

22. Format that plot so that it is gorgeous and well-labelled.

23. Screenshot it and email it to your family.

```
Error in file(filename, "r", encoding = encoding): cannot open the connection
```

```
Error in teacher_tip(tip): could not find function "teacher_tip"
```

Part III

EXERCISES

We can then add more advanced material here that can serve as practice or we can use these for the more advanced group when we have to separate.