

DataB00M: the canon for data science

Databrew

2021-04-29

Contents

1	Welcome!	13
	What this is, and what it isn't	13
	Who this is for	13
	What you will learn	14
	Making the most of this course	15
	Who we are	15
I	Core theory	17
2	Principles of data science	19
	2.1 What is data science?	19
	2.2 What is the data life cycle?	19
	2.3 What is a pipeline?	19
	2.4 Data science 'in the wild'	19
	2.5 The reproducibility crisis	19
3	Visualizing data	21
	3.1 Bad examples	21
	3.2 Good exaples	21
	3.3 Edward Tufte	21
	3.4 Grammar of graphics	21
	3.5 Design principles	21
	3.6 Plots & power	21

4	CONTENTS
4	Writing about data 23
5	Data ethics 25
II	Getting started 27
6	Setting up RStudio 29
7	Running R code 31
	Learning goals 31
	Tutorial video 31
	RStudio's <i>Console</i> 31
	Running code in the <i>Console</i> 32
	Use R like a calculator 34
	7.1 Using operators in R 36
	7.2 Use built-in functions within R 38
	Review assignment: 39
	7.3 Other Resources 39
8	Using RStudio and R scripts 41
	Learning goals 41
	Watch this tutorial 41
	R and RStudio: what's the difference? 41
	Two-minute tour of RStudio 42
	Scripts 43
	Your working directory 48
	Typical workflows 50
	Review assignment: 51
	Other Resources 52

<i>CONTENTS</i>	5
9 Variables in R	53
Learning goals	53
Introducing variables	53
Types of data in R	56
Review assignment	57
Other Resources	57
10 Structures for data in R	59
Learning goals	59
Introducing data structures	59
Vectors	59
Review assignment	67
Other Resources	67
11 Calling functions	69
Learning goals	69
Introducing R functions	69
Review assignment	74
Other Resources	74
12 Base plots	75
Learning goals	75
Introduction	75
Create a basic plot	76
Most common types of plots	76
Basic plot formatting	78
Plotting with data frames	86
Next-level plotting	87
Review assignment	94
Other Resources	94

13 Packages	95
Learning goals	95
Introducing R packages	95
Finding the packages already on your computer	96
Installing a new package	97
Loading an installed package	99
Calling functions from a package	99
Review: the workflow for using a package	100
A note on package dependencies	101
A note on package versions	101
Review assignment	101
Other Resources	102
 14 Basics of ggplot	 103
14.1 Learning goals	103
14.2 What is ggplot	103
14.3 The name and concept	103
14.4 A practical example	104
14.5 Learning examples	104
14.6 Review assignment:	105
14.7 Other resources:	105
 III Working with data in R	 107
 15 Importing data	 109
15.1 Working directories	109
15.2 Reading in data	109
 16 Dataframes	 111
16.1 Exploration	111
16.2 Summarization	111

<i>CONTENTS</i>	7
17 Data wrangling	113
17.1 Data transformation	113
17.2 The tidyverse and tibbles	113
17.3 Transformation with dplyr	113
 IV Exploring & analyzing data	 115
18 Exploratory Data Analysis	117
18.1 Exploring distributions	117
18.2 Variable types & statistics	117
18.3 Descriptive statistics	117
 19 Significance statistics	 119
19.1 Thinking about significance	119
19.2 Comparison tests	119
19.3 Correlation tests	119
 20 Displaying data	 121
20.1 Tables	121
20.2 Base plots	121
20.3 ggplot	121
 V Accessing & managing data	 123
21 Managing project files	125
22 Formatting your own data	127
23 Reading Excel files	129
24 Reading GoogleSheets	131
25 Reading online data	133

26 Exporting data & plots	135
 VI Your R tool bag	 137
27 Joining datasets	139
28 Writing functions	141
Learning goals	141
First steps	141
Next steps	143
Sourcing functions	150
Review assignment: Baby names over time	151
Other Resources	152
 29 for loops	 153
Learning goals	153
Basics	153
for loops in plots	155
Using for loops to process & summarize data	163
Review assignment	170
 30 Conditional statements	 177
Learning goals	177
First steps	177
Next steps	179
Review assignment	182
Other Resources	183
 31 Working with text	 185
 32 Working with dates & times	 187
 33 Working with factors	 189

<i>CONTENTS</i>	9
34 Cleaning messy data	191
35 Matrices & lists	193
36 Pipes	195
 VII Interactive dashboards	 197
37 Intro to Shiny apps	199
38 Shiny dashboards	201
39 Data entry apps	203
 VIII Databases	 205
40 Introduction	207
40.1 What	207
40.2 Why	207
40.3 When	207
40.4 When not	207
 41 Platforms	 209
41.1 PostgreSQL	209
41.2 mySQL	209
41.3 SQLite	209
 42 Alternatives	 211
42.1 NoSQL	211
 43 Practices	 213
 IX Documenting your work	 215
44 R Markdown	217

45 Reproducible research	219
46 Automated reporting	221
47 Formatting standards	223
47.1 Tables	223
47.2 Figures	223
47.3 Captions	223
 X Version control and teamwork	 225
48 What is version control?	227
49 What is Git?	229
49.1 Repositories	229
49.2 Github	229
50 Standard git operations	231
51 A git workflow	233
52 Other git platforms	235
 XI Writing about data	 237
53 Types of writing	239
53.1 Grant proposals	239
53.2 Reports and publications	239
53.3 Fundraising	239
53.4 Press releases	239
54 Elements of style	241

<i>CONTENTS</i>	11
55 Sections of a report	243
55.1 Abstract	243
55.2 Introduction	243
55.3 Methods	243
55.4 Results	243
55.5 Discussion	243
55.6 Other elements	243
 XII Creating websites	 245
 XIII Advanced skills	 247
56 Mapping	249
57 Geographic computing & GIS	251
58 Statistical modeling	253
59 Apply family	255
60 Iterative statistics	257
61 Iterative simulations	259
62 Image analysis	261
63 Machine learning	263
64 Template	265
Learning goals	265
Tutorial video	265
Basics	265
Review assignment	265
Other Resources	266

Chapter 1

Welcome!

Welcome to DataBOOM, a curriculum designed to guide you from your very first line of code towards becoming a professional data scientist.

What this is, and what it isn't

This is not a textbook or a reference manual. It is not exhaustive or comprehensive. It is a *training manual* designed to *empower researchers to do impactful data science*. As such, its tutorials and exercises aim to get you, the researcher, to start writing your own code as quickly as possible and – equally of importance – to *start thinking like a data scientist*, by which we mean tackling ambiguous problems with persistence, independence, and creative problem solving.

Furthermore, this is not a fancy interactive tutorial with bells or whistles. It was purposefully designed 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 data scientist in the real world.

Who this is for

This curriculum covers everything from the absolute basics of writing code in R to machine learning with `tensorflow`. As such, it is designed to be useful to everyone in some way. But the target audience for these tutorials is the student

who *wants* to work with data but has *zero* formal training in programming, computer science, or statistics.

This curriculum was originally developed for the **Sewanee Data Institute for Social Good** at Sewanee: The University of the South, TN.

What you will learn

- The **Core theory** unit establishes the 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 next several units comprise a *core* curriculum for tackling data science problems:

- The **Getting started** unit teaches you how to use R (in RStudio) to explore and plot data. Here you will add the first and most important tools to your toolbox: working with variables, vectors, dataframes, scripts, and file directories.
- The **Working with data** unit teaches you how to bring in your own data and work with it in R. You will learn how to format data to simplify analysis and add tools for *data wrangling* (i.e., transforming and re-formatting data to prepare it for plotting and analysis).
- The **Exploring & analyzing data** unit teaches you how to conduct basic statistics, from exploratory data analyses (e.g., producing and comparing distributions) to significance testing.
- The **Accessing & managing data** unit teaches you how to organize data for large projects, how to manage atypical data formats (e.g., Excel files), and how to access data from online data files.
- The **R toolbag** unit equips you with the tools, tricks, and mindset for tackling the most common tasks in data science. 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.

The next several units provide a suite of skills essential to any data science professional:

- The **Interactive dashboards** unit teaches you how to make dashboards and websites for projects using **shiny** in RStudio.

- The **Databases** unit teaches you how to access, create, and work with relational databases online using **SQL** and its alternatives.
- The **Documenting your work** unit teaches you to use **R Markdown** to produce beautiful, reproducible data reports.
- The **Version control & teamwork** unit teaches you how to use **Git** and **GitHub** to collaborate on shared projects and work on data science teams.
- The **Writing about data** unit teaches you to write successful grant applications and publishable research articles.

The final unit, **Advanced skills**, introduces you to a variety of advanced data science techniques, from interactive maps to iterative simulations to machine learning, that can help you begin to specialize your skillset.

Making the most of this course

(Make sure you involve pizza.)

Who we are

Joe Brew

Ben Brew

Eric Keen

Isabelle Puckette

Instructor tip! Here is some teacher content.

Part I

Core theory

Chapter 2

Principles of data science

- 2.1 What is data science?
- 2.2 What is the data life cycle?
- 2.3 What is a pipeline?
- 2.4 Data science ‘in the wild’
- 2.5 The reproducibility crisis

Chapter 3

Visualizing data

3.1 Bad examples

3.2 Good examples

3.3 Edward Tufte

3.4 Grammar of graphics

3.5 Design principles

3.6 Plots & power

The politics of graphics

(Test text)

Chapter 4

Writing about data

Chapter 5

Data ethics

Part II

Getting started

Chapter 6

Setting up RStudio

First, download and install R:

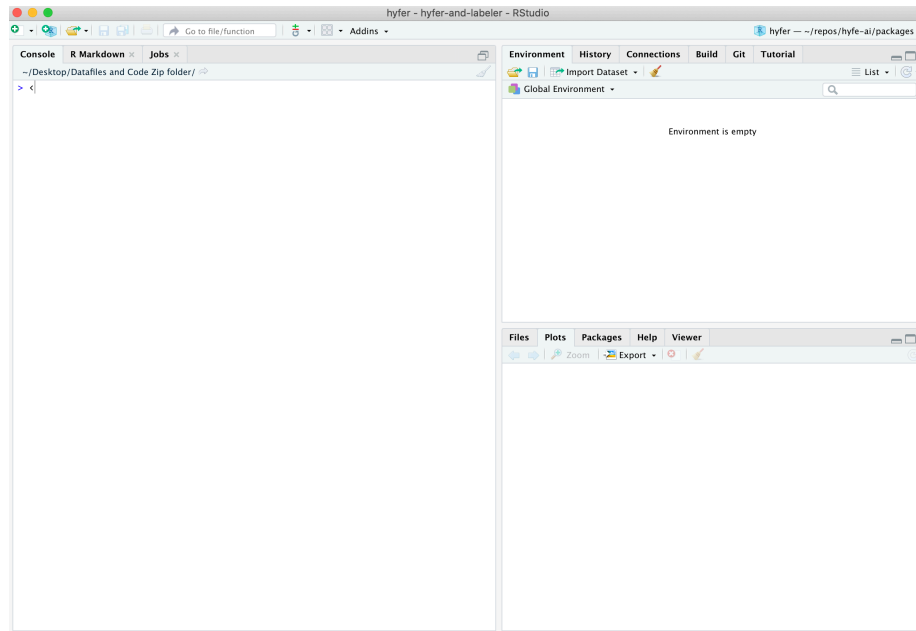
Go to the following website, click the *Download* button, and follow the website's instructions from there. <https://mirrors.nics.utk.edu/cran/>

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.

Boom!

Chapter 7

Running R code

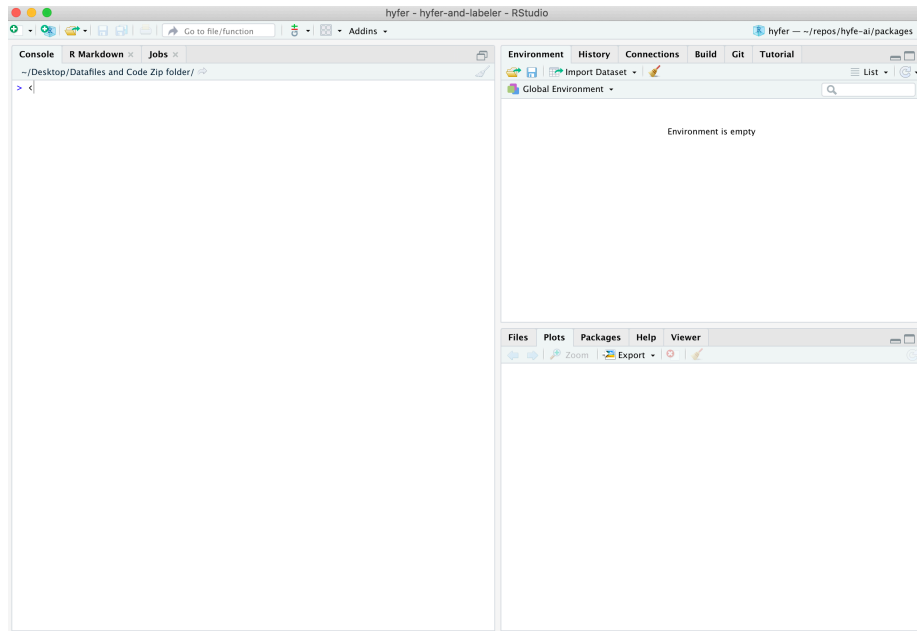
Learning goals

- 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

Tutorial video

RStudio's *Console*

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.

Running code in the *Console*

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

```
1 + 1
```

```
[1] 2
```

When you press **Enter**, R processes the command you fed it, then returns its 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 spaces.

Exercise 1

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

Re-running code in the *Console*

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.

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

Exercise 2

- A. Re-run the sum of 596 and 198 without re-typing it.
- B. Recall the command again, but this time adjust the code to find the sum of 596 and 298.
- C. Practice escaping an accidentally called command: recall your most recent command, then clear the *Console*'s command line.

Incomplete commands in R

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`).

If instead you want R to stop waiting and stop running, hit the **Escape** key.

Getting errors in R

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 + 6p
```

You probably received a message in red font stating **Error: unexpected symbol in "4 + 6p"**. That is because R did not know how to interpret the symbol `p` in this case.

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.

Exercise 3

Type a command in R that throws an error, then recall the command and revise so that R can understand it.

Use R like 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 straightforward too:

```
24 / 2
```

```
[1] 12
```

For multiplication, use an asterisk (*):

```
24 * 2
```

```
[1] 48
```

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
```

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
```

Exercise 4

- A. Find the sum of the ages of everyone in your immediate family.
- B. Now recall that command and adjust it to determine the *average* age of the members of your family.

7.1 Using operators in R

You can get R to evaluate logical tests 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 ($>$ and $<$), and greater/less than or equal to ($>=$ and $<=$).

```
100 > 100
```

```
[1] FALSE
```

```
100 >= 100
```

```
[1] TRUE
```

Exercise 5

A. 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
```

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

7.2 Use built-in functions within R

R has some built-in functions for common calculations, such as finding square roots and logarithms.

```
sqrt(16)
```

```
[1] 4
```

```
log(4)
```

```
[1] 1.386294
```

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)
```

```
[1] 2.302585
```

```
log10(10)
```

```
[1] 1
```

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

```
100/3
```

```
[1] 33.33333
```

```
round(100/3)
```

```
[1] 33
```

```
round(100/3,digits=1)
```

```
[1] 33.3
```

```
round(100/3,digits=2)
```

```
[1] 33.33
```

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

```
[1] 33.333
```

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

```
pi
```

```
[1] 3.141593
```

Exercise 6

Find the square root of π and round the answer to the 2 decimal places.

Review assignment:

NOTE: Under construction!

7.3 Other Resources

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

Chapter 8

Using RStudio and 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

Watch this tutorial

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.

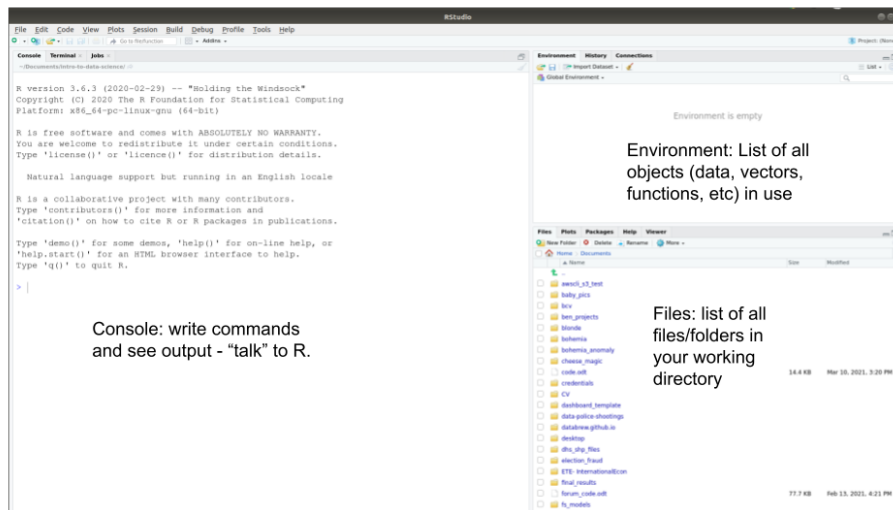
Instructor tip! At this point it may be useful to show the students what opening R looks like on its own (not through R Studio). This helps them see why RStudio is valuable, and it will also help them understand what they did wrong when they accidentally open an .R file in R instead of RStudio – which will happen a lot at first.

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 the previous 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.

This is the pane that is used the least often, and if you wish it can simplify your workspace to minimize it.

Files, Plots, Packages, & Help

You will use the bottom right pane very often.

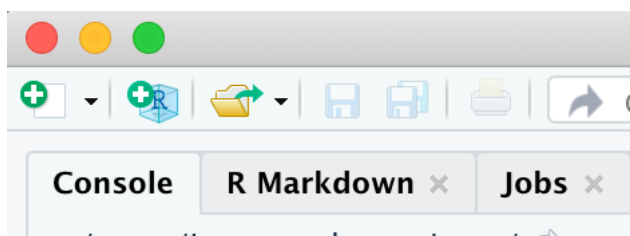
- The **Files** tab lets you see all the files within your **working directory**, which will be explained in the section below.
- The **Plots** tab lets you see the plots you are producing with your code.
- The **Packages** tab lets you see the *packages* you currently have installed on your computer. 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

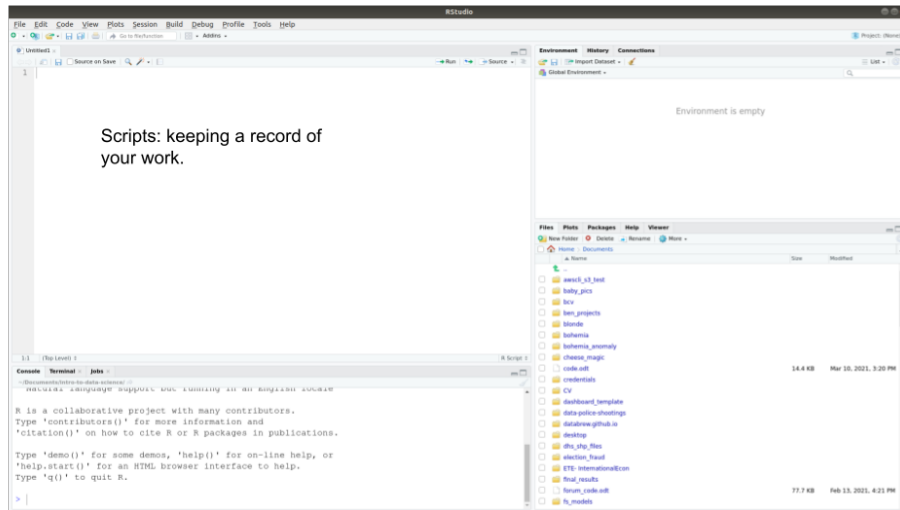
Before explaining what scripts are and why they are awesome, let's start a new script.

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**.)

Instructor tip! Get all students to practice running code at this point. The act of typing the commands themselves helps them learn and overcome their hesitation about messing up.

Exercise 1

Add a few more lines to your script, such that your script now looks like this.

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

- (A) Run all of these lines of code at once.
- (B) 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* or *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. You are 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.)

Instructor tip! Consider telling a story from your own work life before you discovered R scripts. For example: receiving versions of Excel files named DATA-final-final-final.xlsx, because tiny changes are inevitably discovered after you try to finalize a data file. Then you work all weekend on an analysis using that data, only to discover you were using the WRONG version of the data!

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 an R script, you can reproduce your own work whenever you want to. You can send your script to a colleague so that they can reproduce 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.)

Instructor tip! Having grown up in the age of GoogleDocs, many students may not be familiar with what computer files are, and may not even know that their computer operates using directories of folders. It would be useful to open up File Explorer on your demo screen and show them how these directories work.

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. We suggest making a new folder on your Desktop and naming it **databoom**, but you can name it whatever you want and place it wherever you want.

How to save your script? 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 **09_intro_to_rstudio**.

(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] "/Users/erickeen/repos/intro-to-data-science"
```

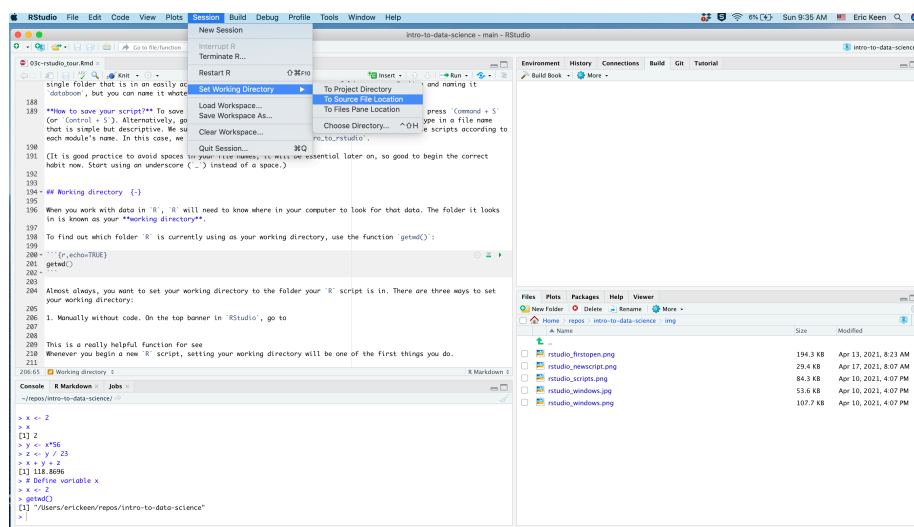
Almost always, you want to set your working directory to the folder your R script is in.

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*:

(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.

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)
```

Review assignment:

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

Part 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; (2) a **modules** folder, which is where you will keep the code you use to work on the material in these modules, and (3) a **project** folder, which is where you will keep all your work associated with your summer project.

Part 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.

Part 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 `10_variables.R` (because the next module is called *Module 10: Variables in R*).

Part 5. Modify the code in `10-variables.R` so that you are prepared to begin the next module. Change the title, and look ahead to *Module 10* 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 9

Variables in R

Learning goals

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

Instructor tip! Here is some teacher content.

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:

```
x <- 3*12 ; x
```

```
[1] 36
```

The semicolon simulates hitting **Enter**. It says: first run `x <- 3*12`, then run `x`.

You can also update variables.

```
x <- x * 3 ; x
```

```
[1] 108
```

```
x <- x * 3 ; x
```

```
[1] 324
```

You can also add variables together.

```
x <- 8
y <- 4.5
x + y
```

```
[1] 12.5
```

Naming variables

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.

You can make variable names as complicated or simple as you want.

```
supercalifragilistic.expialidocious <- 5
supercalifragilistic.expialidocious # still works
```

```
[1] 5
```

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.

Also note that variables are case-sensitive! If you name a variable `My_variable`, R will not recognize it if you refer to it as `My_Variable`.

Naming theory

Naming variables 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.

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.

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.

Exercise 1

- A. Estimate how many bananas you’ve eaten in your lifetime and store that value in a variable (choose whatever name you wish).
- B. Now estimate how many ice cream sandwiches you’ve eaten in your lifetime and store that in a different variable.
- C. 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.
- D. Who in the class has the highest ratio? Who has the lowest?

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.

Exercise 3

NOTE: UNDER CONSTRUCTION!

Review assignment

NOTE: UNDER CONSTRUCTION!

Other Resources

Chapter 10

Structures for data in R

Learning goals

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

Instructor tip! Here is some teacher content.

Introducing data structures

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.

Vectors

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
```

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
```

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
```

You can also put vectors through logical tests:

```
x <- 1:5
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("Ben", "Joe", "Eric")
x
```

```
[1] "Ben" "Joe" "Eric"
```

```
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"
```

Useful functions for handling vectors

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

```
x <- c(5,6)
y <- c(9,10,11,12)

length(x)
```

```
[1] 2
```

```
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!"
```

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
```

`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)
```

`%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
```

Exercise 2

NOTE: UNDER CONSTRUCTION!

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 `[]`.


```
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`.

Exercise 3

A. Figure out how to replicate the `head()` function using your new vector subsetting skills.

```
x[1:6]
```

```
[1] 50 51 52 53 54 55
```

B. Now replicate the `tail()` function, using those same skills as well as the `length()` function you just learned.

```
x[(length(x) - 5) : length(x)]
```

```
[1] 95 96 97 98 99 100
```

Dataframes & other data structures

A **vector** is the most basic data structure in R, and the other structures are built out of vectors.

As a data scientist, the most common data structure you will be working with is a **dataframe**, which is essentially a spreadsheet: a dataset with rows and columns, in which each column represents is a vector of the same class of data.

We will explore dataframes in detail later, but here is a sneak peak at what they look like:

```
df <- data.frame(x=300:310,
                 y=600:610)
df
```

```
      x  y
1  300 600
2  301 601
3  302 602
4  303 603
5  304 604
6  305 605
7  306 606
8  307 607
9  308 608
10 309 609
11 310 610
```

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.inches=c(75, 73, 80))
df
```

```
      name height.inches
1   Ben             75
2   Joe             73
3  Eric             80
```

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

The two other most common data structures are **matrices** and **lists**, but we will wait on learning about those. For now, focus on becoming comfortable using vectors and dataframes.

Exercise 3

NOTE: UNDER CONSTRUCTION!

Review assignment

NOTE: UNDER CONSTRUCTION!

Other Resources

Chapter 11

Calling functions

Learning goals

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

Instructor tip! Here is some teacher content.

Introducing R functions

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`.

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)
```

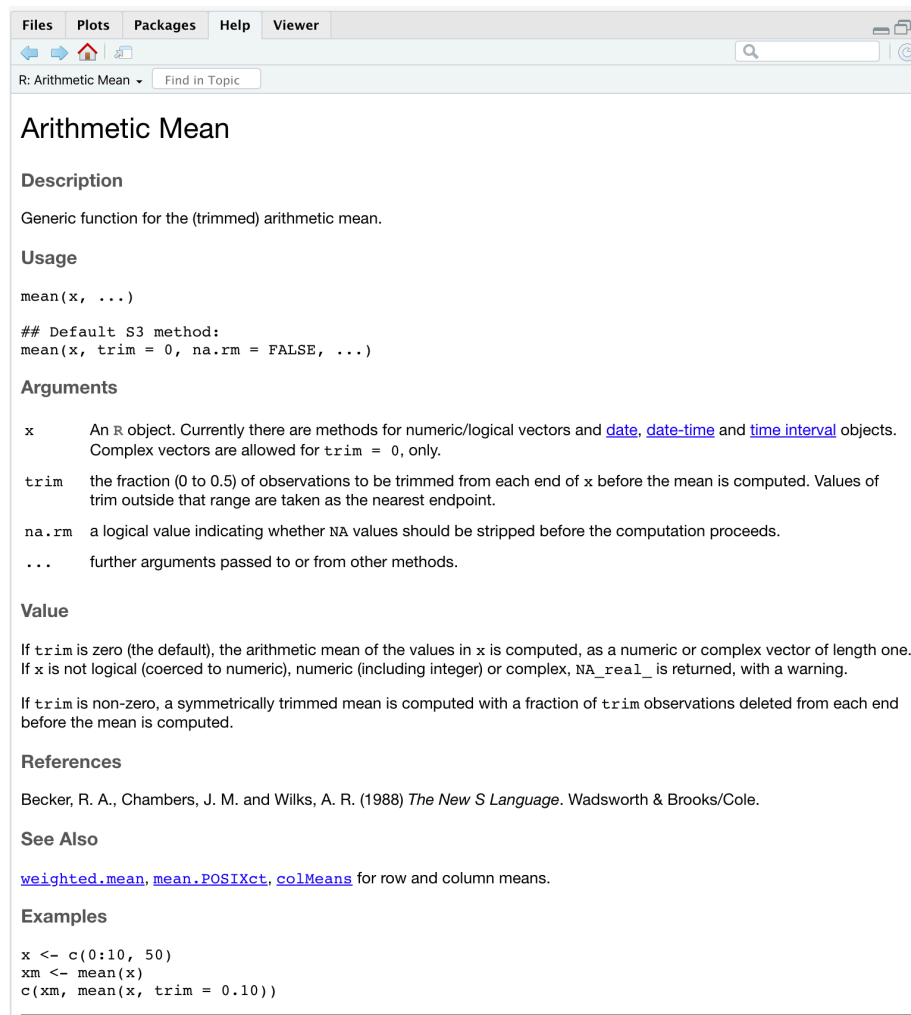
```
[1] NA
```

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:



The screenshot shows the R help viewer window for the 'Arithmetic Mean' function. The window has a menu bar with 'Files', 'Plots', 'Packages', 'Help', and 'Viewer'. Below the menu bar is a search bar and a 'Find in Topic' button. The main content area is titled 'Arithmetic Mean' and contains the following sections:

- Description**: Generic function for the (trimmed) arithmetic mean.
- Usage**:


```
mean(x, ...)
```

Default S3 method:
`mean(x, trim = 0, na.rm = FALSE, ...)`
- Arguments**:
 - `x`: An R object. Currently there are methods for numeric/logical vectors and [date](#), [date-time](#) and [time interval](#) objects. Complex vectors are allowed for `trim = 0`, only.
 - `trim`: the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of `trim` outside that range are taken as the nearest endpoint.
 - `na.rm`: a logical value indicating whether NA values should be stripped before the computation proceeds.
 - `...`: further arguments passed to or from other methods.
- Value**:

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.
- References**:

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.
- See Also**:

[weighted.mean](#), [mean.POSIXct](#), [colMeans](#) for row and column means.
- Examples**:


```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

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

Review assignment

NOTE: Under construction!

Other Resources

NOTE: Under construction!

Chapter 12

Base plots

Learning goals

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

Instructor tip! Here is some teacher content.

Introduction

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.

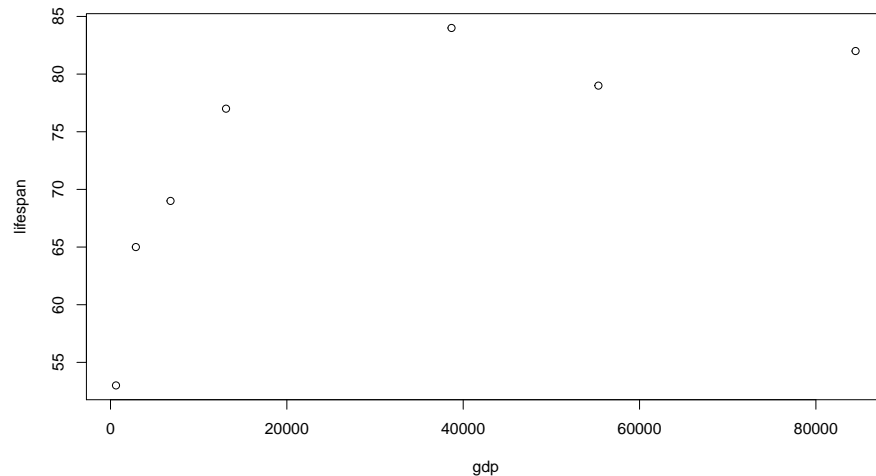
The `gdp` column presents the average GDP per capita within that country, which is a common index for the income and wealth of average citizens.

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)
```

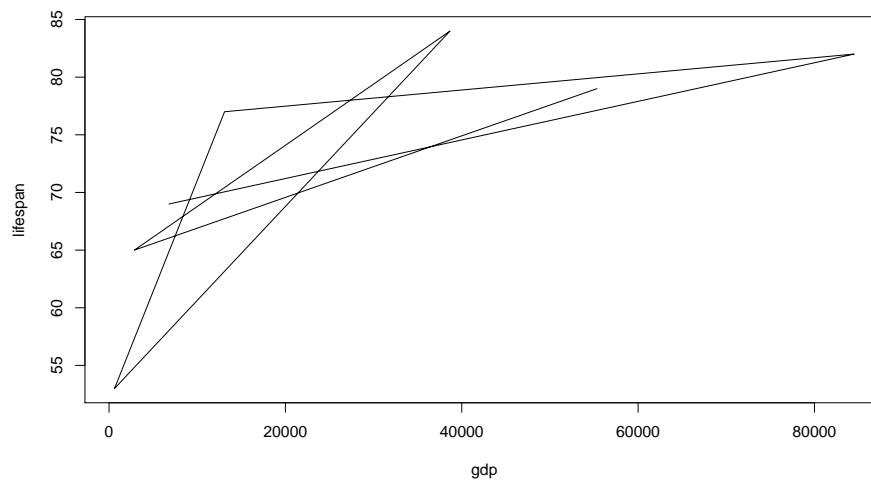
Choose whichever one is most intuitive to you.

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:

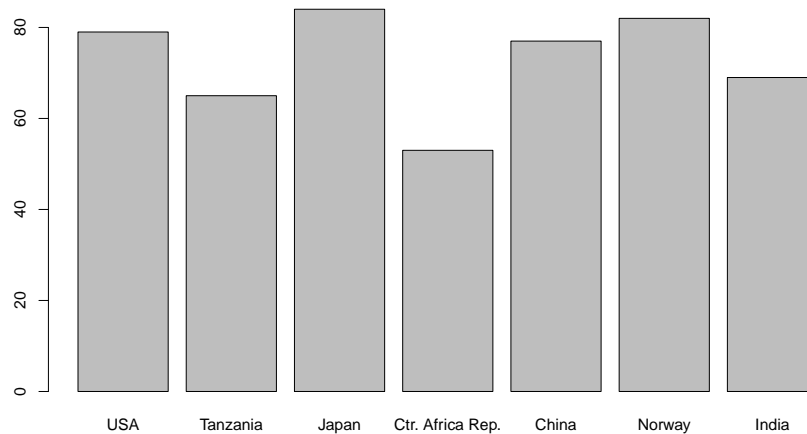
```
plot(lifespan ~ gdp, type="l")
```



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 now.

Exercise 1

Produce a bar plot that shows the GDP for each country.

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:

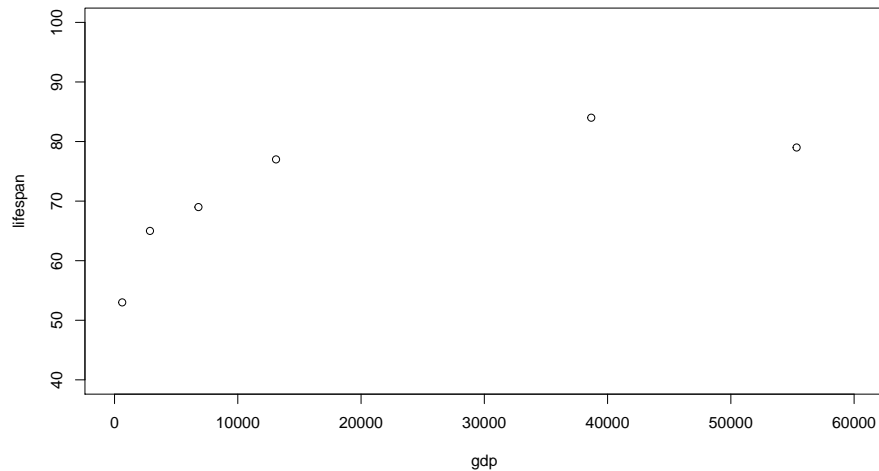
```
?plot
```

If multiple help page options are returned, select the *Generic 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)`).

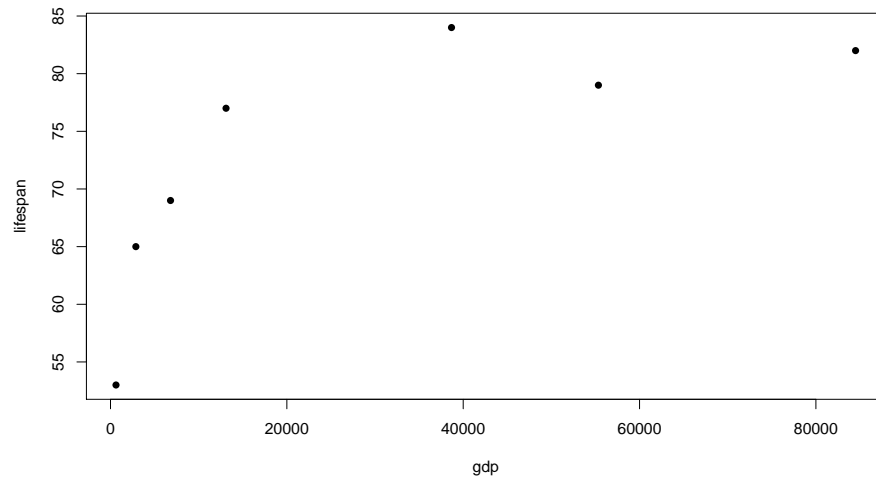
Note that it can be easier to read your code if you put each input on a new line, like this:

```
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.

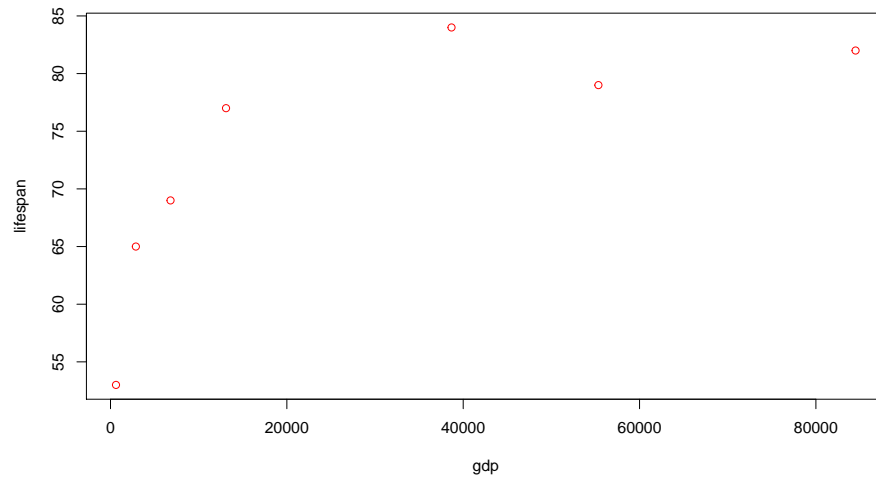
Set dot type using the input `pch`:

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



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

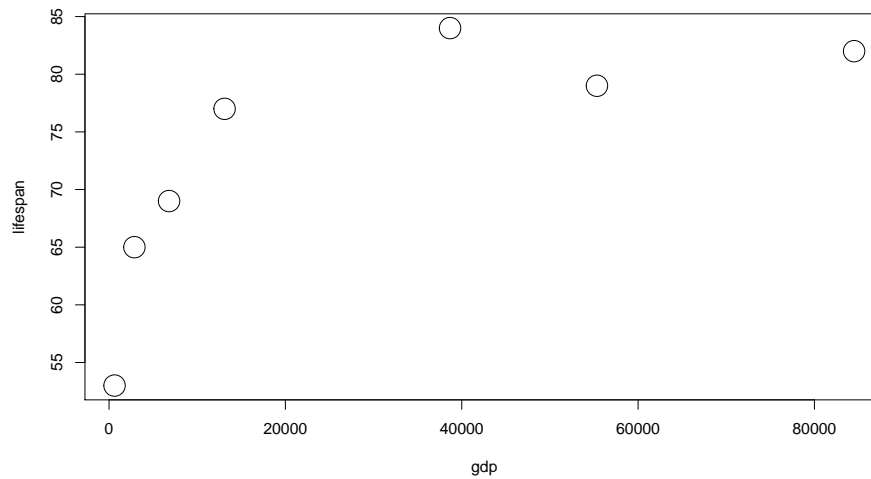
```
plot(lifespan ~ gdp,col="red")
```



Here is a great resource for color names in R.

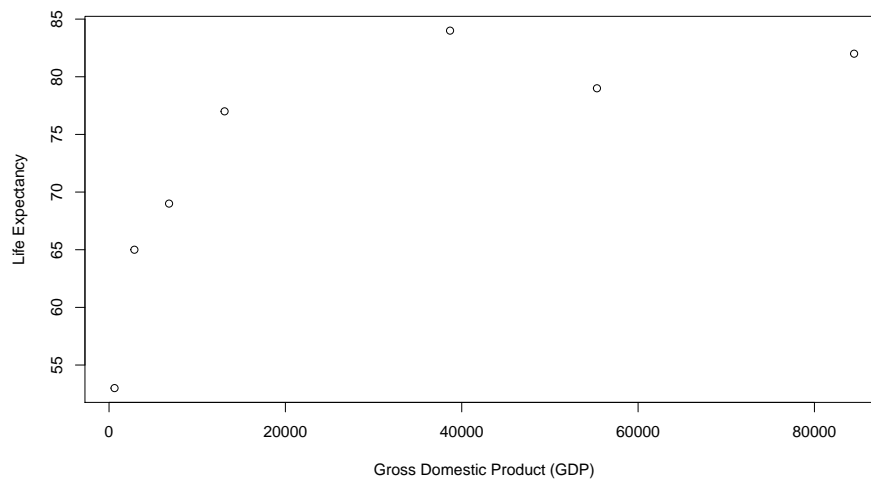
Set **dot size** using the input `cex` (the default is `cex=1`):


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



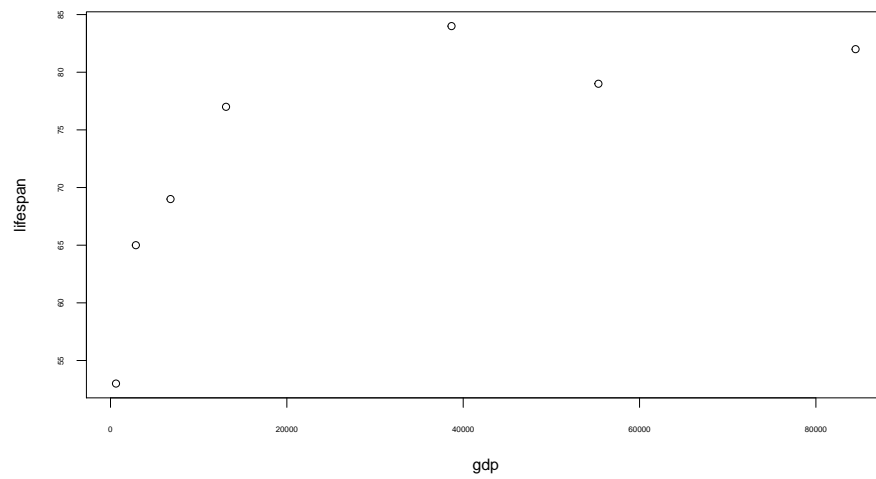
Set axis labels using the inputs `xlab` and `ylab`:

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



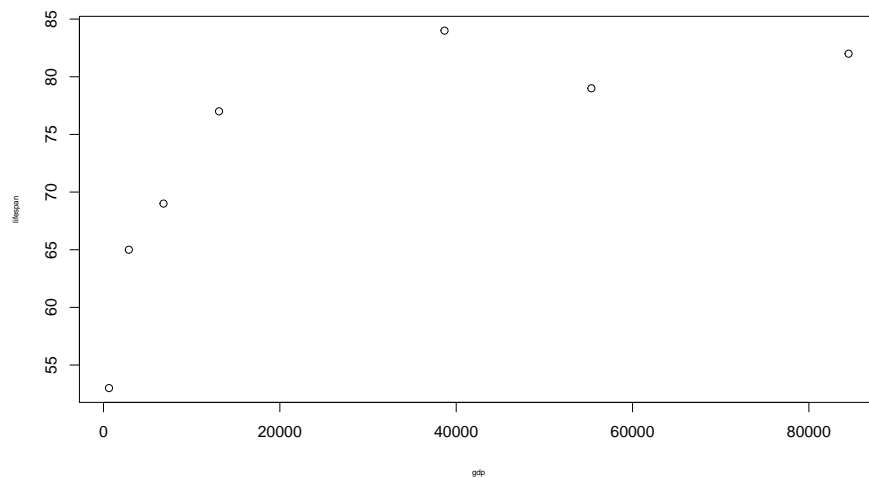
Set axis number size using the input `cex.axis` (the default is `cex.axis=1`):

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



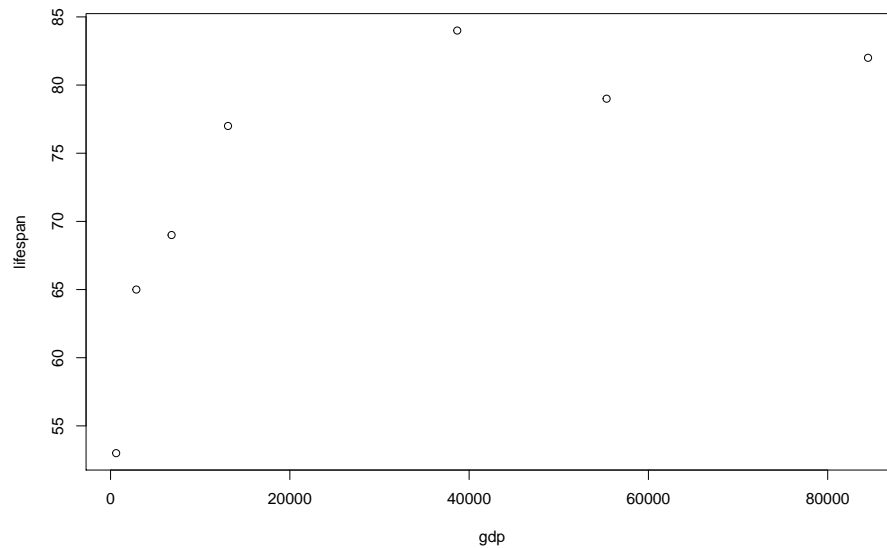
Set axis label size using the input `cex.label` (the default is `cex.lab=1`):

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



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

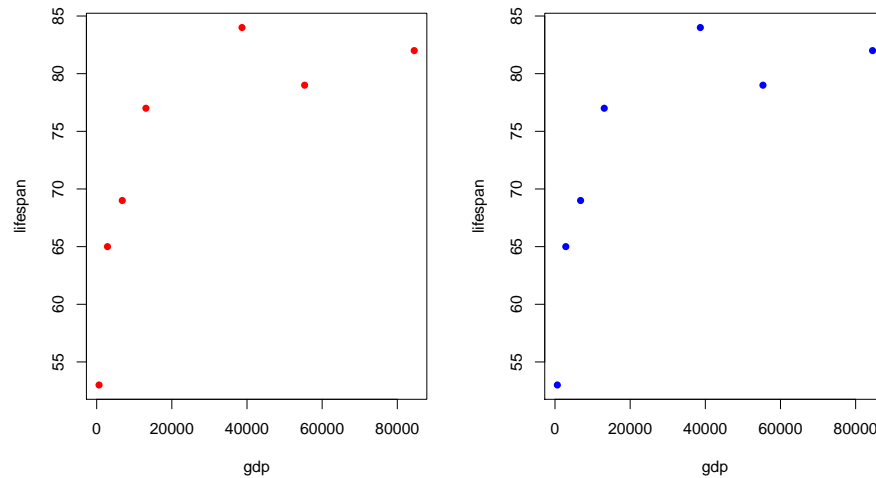
```
par(mar=c(5,5,0.5,0.5))
plot(lifespan ~ gdp)
```



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)
```



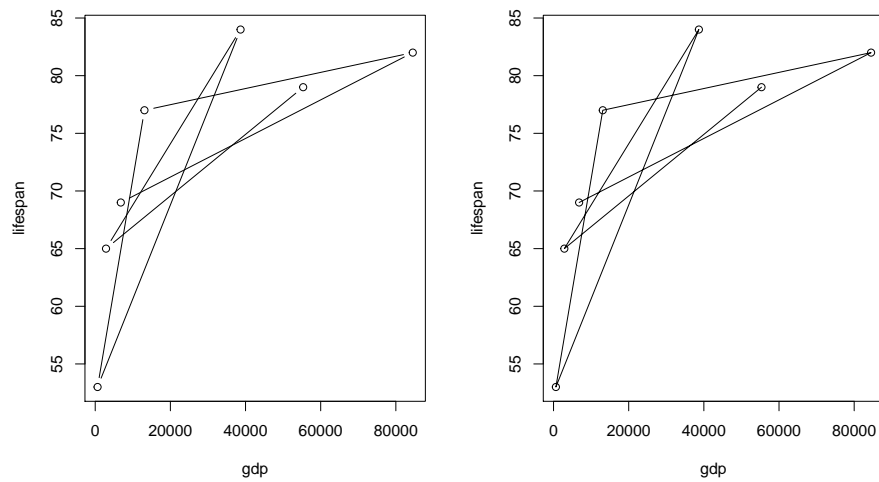
In this command, the two numbers in the vector used to define `mfrow` correspond to the number of rows and columns, respectively, on the entire plot. In this case, you have 1 row of plots with two columns.

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")
```

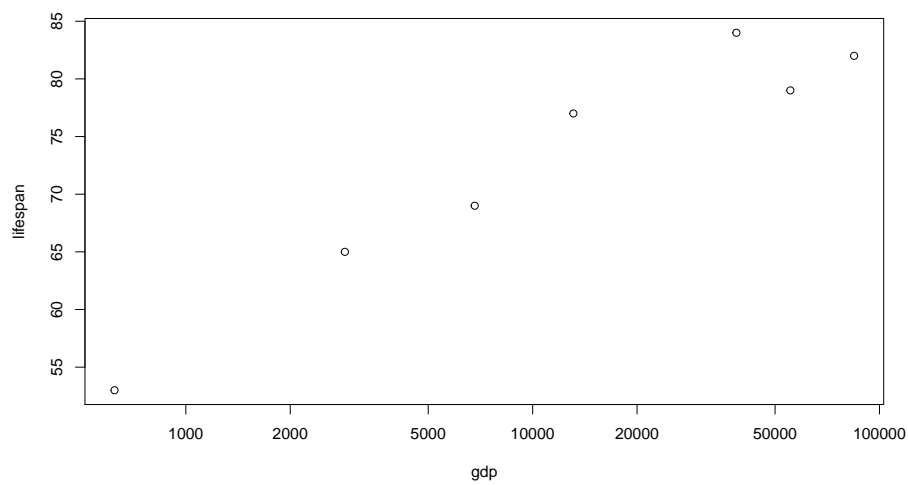


```
par(mfrow=c(1,1))
```

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")
```



Exercise 2

Produce a *beautifully* formatted plot that incorporates **all** of these customization inputs explained above into a multi-paned plot.

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.

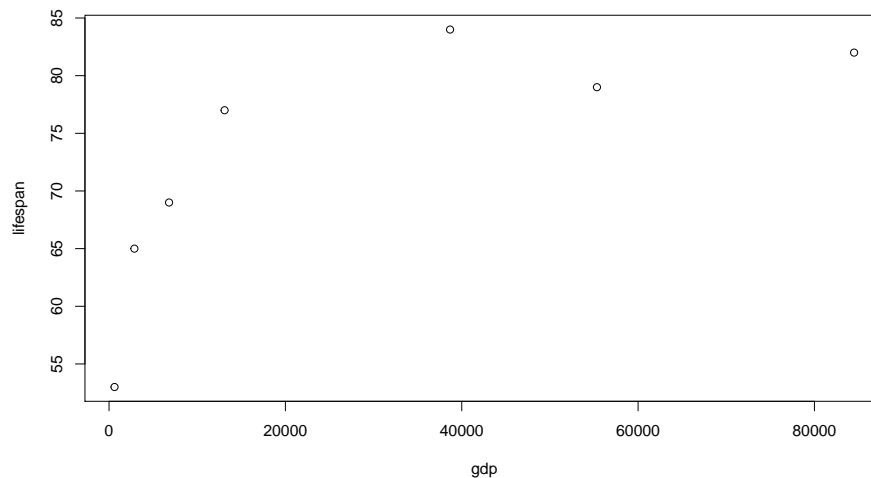
Let's turn these vectors into a dataframe:

```
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, 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. You will learn more about working with dataframes in an upcoming module.

Exercise 2

- A. Use the `df` dataframe to produce a bar plot that shows life expectancy for each country.
- B. 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.

Next-level plotting

The possibilities for data visualization in R are pretty much limitless, and over time you will become fluent in making gorgeous plots. Here are a few common

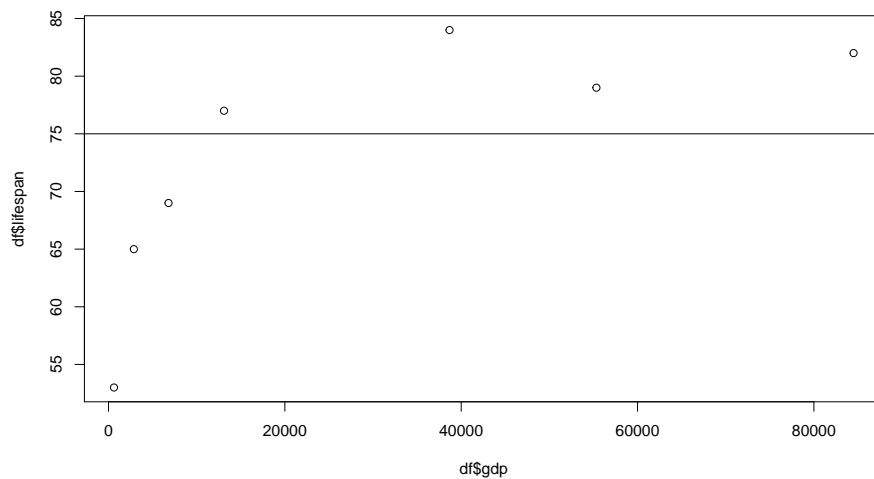
tools that can take your plots to the next level.

Adding lines

In some cases it is useful to add reference lines to your plot. For example, what if we wanted to be able to quickly see which countries had life expectancies below 75 years?

You can add a line at `lifespan = 75` using the function `abline()`.

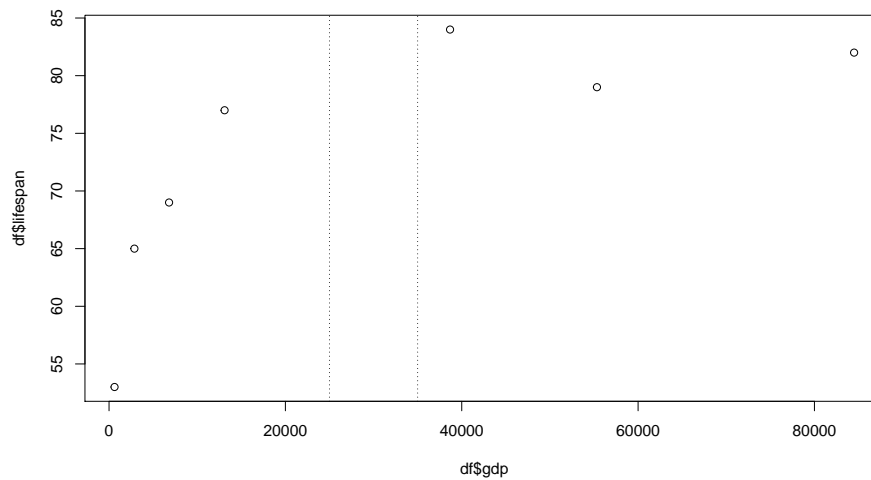
```
plot(df$lifespan ~ df$gdp)
abline(h=75)
```



In this command, the `h` input means “place a horizontal line at this y value.”.

Similarly, you can use `v` to specify vertical lines at certain x values.

```
plot(df$lifespan ~ df$gdp)
abline(v=c(25000,35000),lty=3)
```

Note here that another input, `lty`, was used to change the type of line printed. (Refer to `?abline()` for more details).

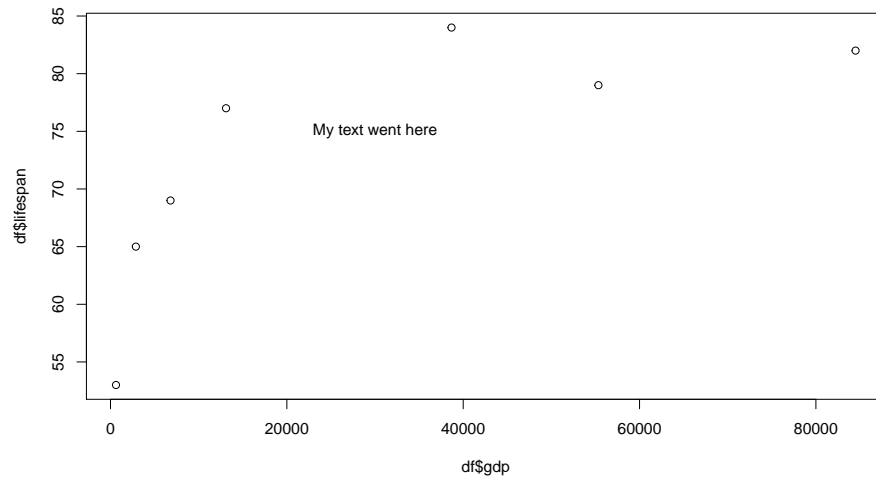
Exercise 4

Produce a plot of life expectancy as a function of GDP per capita. Then add a line to your plot that indicates which countries have per-capita GDPs that fall below (or above) the average per-capita GDP for the whole dataset. Make your line dashed and color it red.

Adding text

Use the `text()` function to add labels to your plot:

```
plot(df$lifespan ~ df$gdp)
text(x=30000,y=75,labels="My text went here")
```



Exercise 5

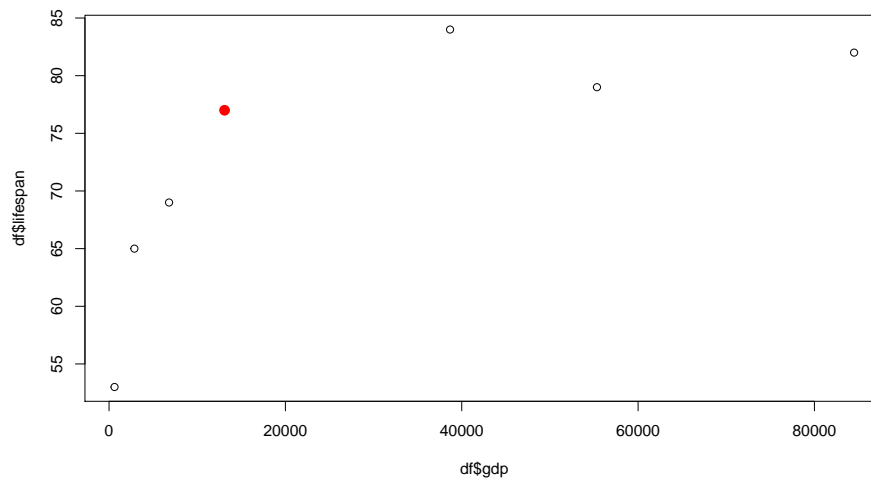
Produce a plot of life expectancy as a function of GDP per capita, then label each point by country. Make the labels small and place them *to the right* of their associated dot (Hint: use `?text` for help).

Highlighting certain data points

It can be helpful to highlight a certain data point (or group of data points) using a different dot size, format, or color.

To highlight a single data point, here is one approach you can take: first, plot all points, *then* re-plot the point of interest using the `points()` function:

```
plot(df$lifespan ~ df$gdp)
points(x=df$gdp[5], y=df$lifespan[5], col="red", pch=16, cex=1.5)
```



In this example, we re-plotted the data for the fifth row in the dataframe (in this case, China).

To highlight a group of data points, try this approach:

- First, create a vector that will contain the color for each data point.
- Second, determine the color for each data point using a logical test.
- Third, use your vector of colors within your `plot()` command.

For example, let's highlight all countries whose life expectancy is greater than 75.

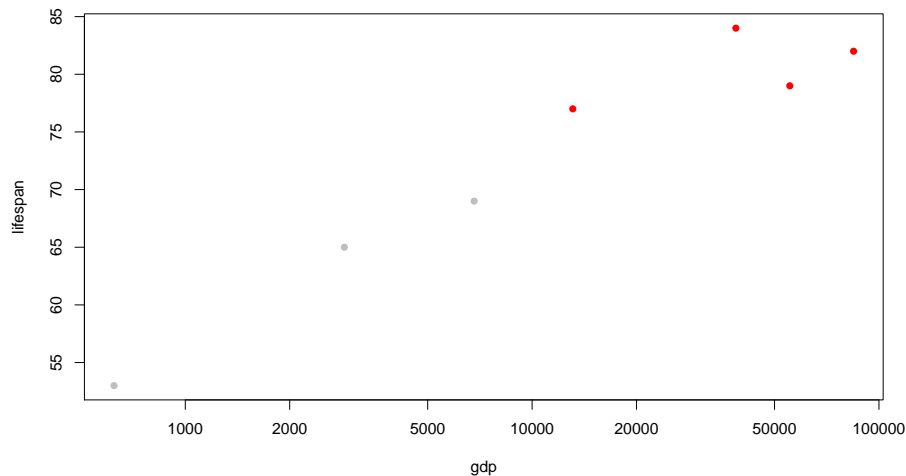
```
# First
cols <- rep("grey",times=length(lifespan)) # create a vector of colors the length of vector `lifespan`
cols
```

```
[1] "grey" "grey" "grey" "grey" "grey" "grey" "grey"
```

```
# Second
change_these <- which(lifespan > 75)
change_these # these are the elements that we want to highlight
```

```
[1] 1 3 5 6
```

```
cols[change_these] <- "red" # change the color for these elements to a highlight color
# Third
plot(lifespan ~ gdp, pch=16, col=cols, log="x")
```



Exercise 6

Produce a plot of life expectancy as a function of GDP per capita, in which all countries with GDPs below \$10,000 have larger dots of a different color.

Building a plot from the ground up

In many applications it can be helpful to have complete control over the way your plot is built. To do so, you can build your plot from the very bottom up in multiple steps.

The steps for building up your own plot are as follows:

1. **Stage a blank canvas:** A plot begins with a blank canvas that covers a certain range of values for x and y. To stage a blank canvas, add this parameters to your `plot()` function: `type="n", axes=FALSE, ann=FALSE, xlim=c(__, __), ylim=c(__, __)`. These commands tell R to plot a blank space, not to print axes, not to print annotations like x- or y-axis labels, and to limit your canvas to a certain coordinate range. Be sure to add numbers to the `xlim()` and `ylim()` commands.

2. **Add your axes**, if you want, using the function `axis()`. The command `axis(1)` prints the x-axis, and `axis(2)` prints the y-axis. This function allows you to define where tick marks occur and other details (see `?axis`).
3. **Add axis titles** using the function `title()`.
4. **Add reference lines**, if you want, using `abline()`. Do this before adding data, since it is usually nice for data points to be superimposed *on top of* your reference lines.
5. **Add your data** using either `points()` or `lines()`.
6. **Add text labels**, if you want, using `text()`.

Here is an example of this process:

```
# 1. Stage a blank canvas
par(mar=c(4.5,4.5,1,1))
plot(1,type="n",axes=FALSE,ann=FALSE,xlim=c(0,100000),ylim=c(40,100))

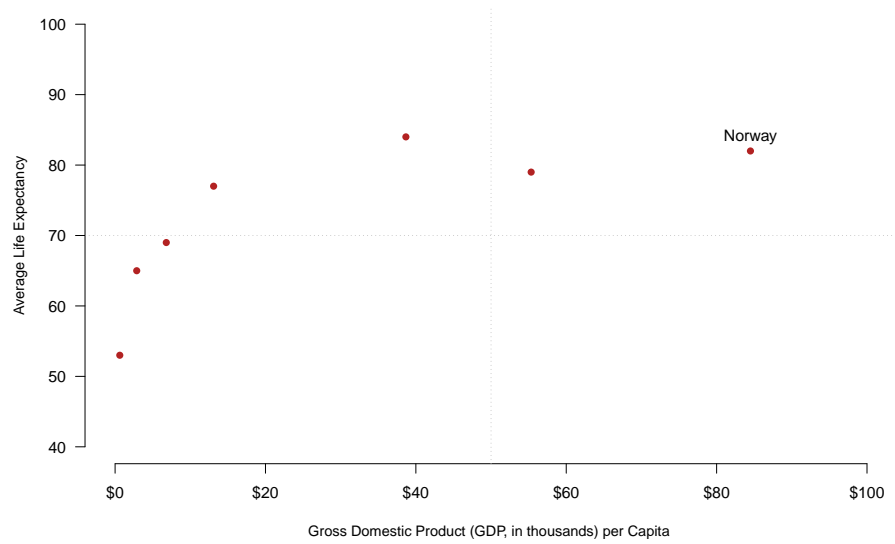
# 2. Add axes
axis(1,at=c(0,20000,40000,60000,80000,100000),labels=c("$0", "$20", "$40", "$60", "$80", "$100"))
axis(2,at=seq(40,100,by=10),las=2)

# 3. Add axis titles
title(xlab="Gross Domestic Product (GDP, in thousands) per Capita ",cex.lab=.9)
title(ylab="Average Life Expectancy",cex.lab=.9)

# 4. Add reference lines
abline(h=70,v=50000,lty=3,col="grey")

# 5. Add data
points(x=gdp,y=lifespan,pch=16,col="firebrick")

# 6. Add text
text(x=gdp[6],y=lifespan[6],labels="Norway",pos=3)
```



Review assignment

NOTE: Under construction!

Other Resources

Chapter 13

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.

Instructor tip! Here is some teacher content.

Introducing R packages

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.

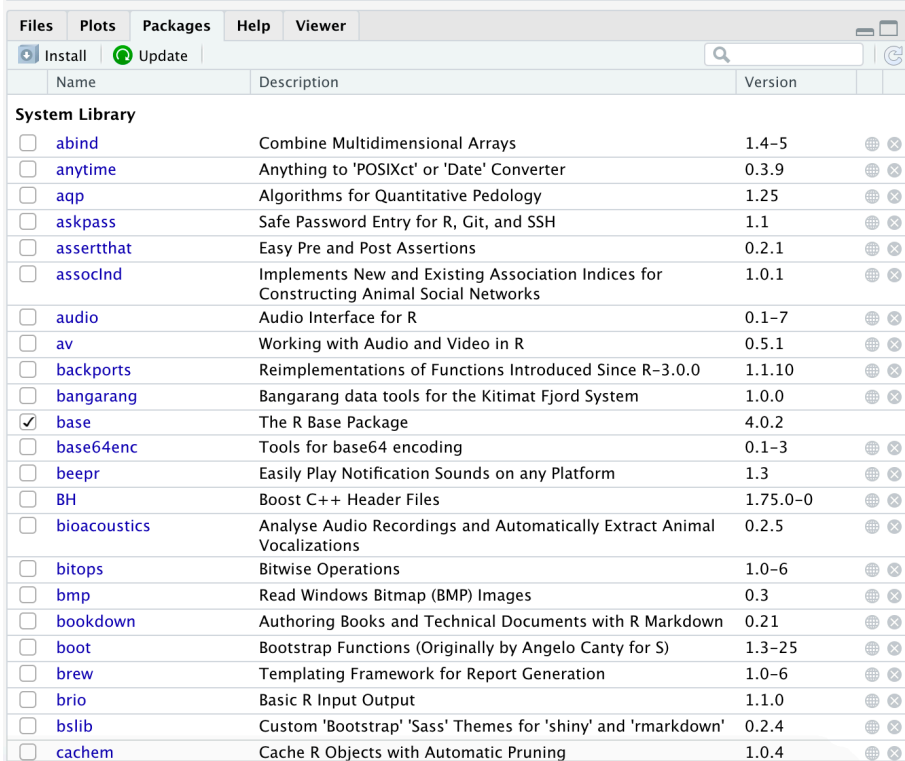
You can also access thousands of other functions and datasets through bundles of external code known as **packages**. Packages are developed and shared by R users around the world – a global community working together to increase R’s versatility and impact.

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.

Finding the packages already on your computer

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

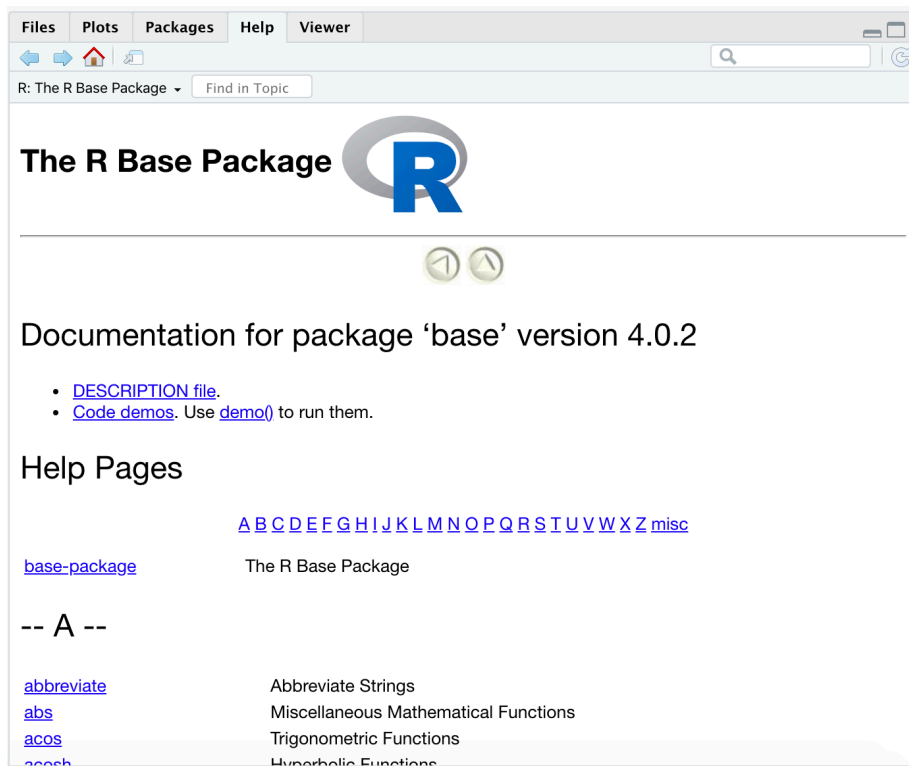


The screenshot shows the RStudio interface with the 'Packages' tab selected in the bottom right pane. The pane has a search bar and buttons for 'Install' and 'Update'. Below is a table of packages. The 'base' package is checked, indicating it is installed. Other packages like 'abind', 'anytime', 'aqp', etc., are unchecked, indicating they are available for installation.

	Name	Description	Version		
System Library					
<input type="checkbox"/>	abind	Combine Multidimensional Arrays	1.4-5	⊞	⊗
<input type="checkbox"/>	anytime	Anything to 'POSIXct' or 'Date' Converter	0.3.9	⊞	⊗
<input type="checkbox"/>	aqp	Algorithms for Quantitative Pedology	1.25	⊞	⊗
<input type="checkbox"/>	askpass	Safe Password Entry for R, Git, and SSH	1.1	⊞	⊗
<input type="checkbox"/>	assertthat	Easy Pre and Post Assertions	0.2.1	⊞	⊗
<input type="checkbox"/>	assocInd	Implements New and Existing Association Indices for Constructing Animal Social Networks	1.0.1	⊞	⊗
<input type="checkbox"/>	audio	Audio Interface for R	0.1-7	⊞	⊗
<input type="checkbox"/>	av	Working with Audio and Video in R	0.5.1	⊞	⊗
<input type="checkbox"/>	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.10	⊞	⊗
<input type="checkbox"/>	bangarang	Bangarang data tools for the Kitimat Fjord System	1.0.0	⊞	⊗
<input checked="" type="checkbox"/>	base	The R Base Package	4.0.2		
<input type="checkbox"/>	base64enc	Tools for base64 encoding	0.1-3	⊞	⊗
<input type="checkbox"/>	beepR	Easily Play Notification Sounds on any Platform	1.3	⊞	⊗
<input type="checkbox"/>	BH	Boost C++ Header Files	1.75.0-0	⊞	⊗
<input type="checkbox"/>	bioacoustics	Analyse Audio Recordings and Automatically Extract Animal Vocalizations	0.2.5	⊞	⊗
<input type="checkbox"/>	bitops	Bitwise Operations	1.0-6	⊞	⊗
<input type="checkbox"/>	bmp	Read Windows Bitmap (BMP) Images	0.3	⊞	⊗
<input type="checkbox"/>	bookdown	Authoring Books and Technical Documents with R Markdown	0.21	⊞	⊗
<input type="checkbox"/>	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-25	⊞	⊗
<input type="checkbox"/>	brew	Templating Framework for Report Generation	1.0-6	⊞	⊗
<input type="checkbox"/>	brio	Basic R Input Output	1.1.0	⊞	⊗
<input type="checkbox"/>	bslib	Custom 'Bootstrap' 'Sass' Themes for 'shiny' and 'rmarkdown'	0.2.4	⊞	⊗
<input type="checkbox"/>	cachem	Cache R Objects with Automatic Pruning	1.0.4	⊞	⊗

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

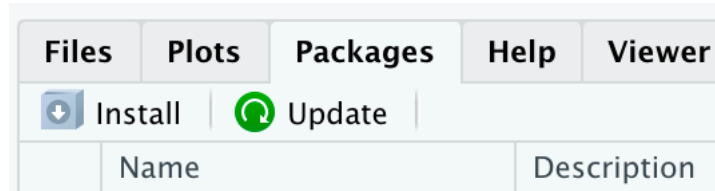
There are a couple ways to download and install a new R package on your computer. Most packages are available from an open-source repository known as CRAN (which stands for Comprehensive R Archive Network). However, an increasingly common practice is to release packages on a public repository such as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

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] "x<dr_y,Nn9EiB {38k&\"(ILO"
```

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] "JaZ=!FX{$ jz)|frHhn^BqIp"
```

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

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

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

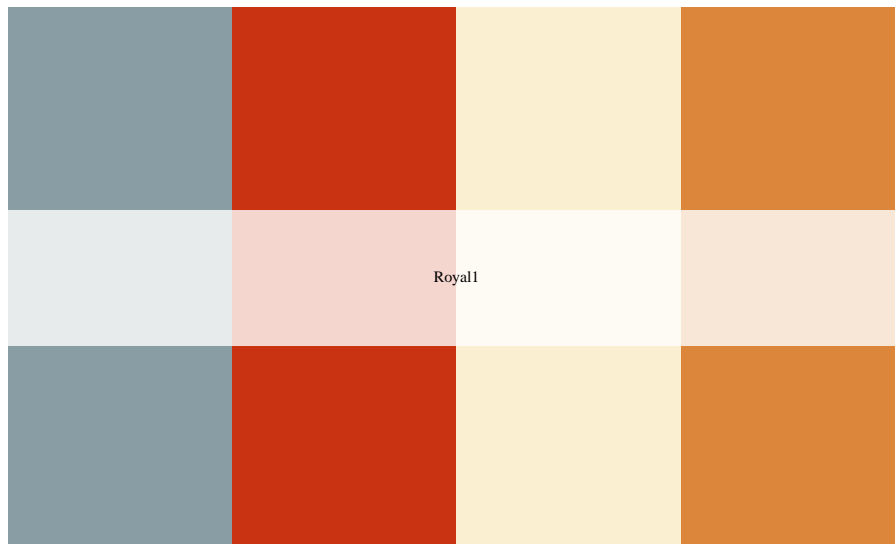
# Example from GitHub
devtools::install_github("karthik/wesanderson")
```

2. Load the package

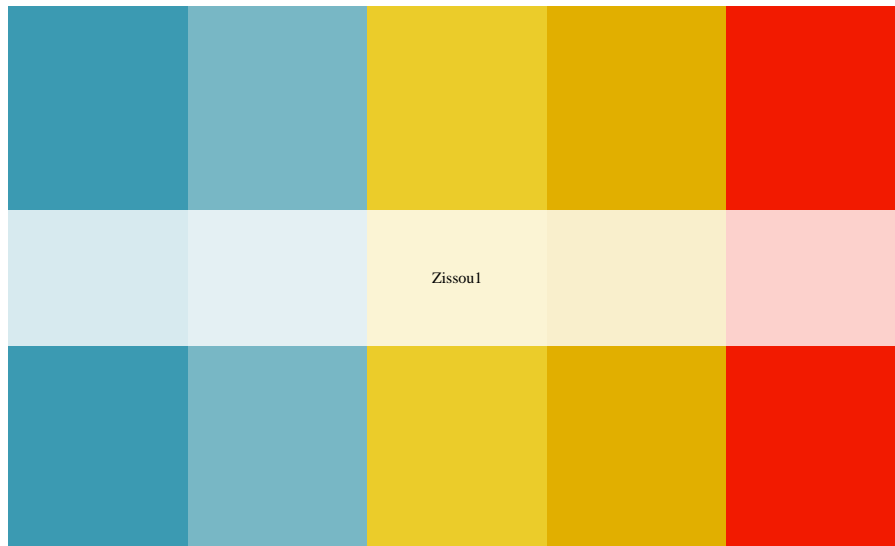
```
# Example
library(wesanderson)
```

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.)

A note on 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.

A note on 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 assignment

NOTE: Under construction!

Other Resources

NOTE: Under construction!

Chapter 14

Basics of ggplot

(Refer heavily to <https://ggplot2-book.org/introduction.html>)

14.1 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 syntax for creating different plots using `ggplot2`

14.2 What is ggplot

`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.

14.3 The name and concept

“GG” stands for “grammar of graphics”, with “grammar” meaning “the fundamental principles or rules of an art or science” (Wickham, 2010). The most

well-known “grammar of graphics” was written in 2005 and laid out some abstract principles for describing statistical graphics (Wilkinson, 2005). The basic idea is that all graphs can be described using a *layered* grammar, and that all graphs have the same general elements...

- data
- geometric objects
- aesthetics (mapping) of variables to objects

... whereas some graphs have additional elements...

- statistical transformations
- scales
- facets

14.4 A practical example

Let’s get practical (we’ll get back to the theory later).

First, let’s read in some data on health from the World Bank:

Canvas Canvas + variables (mapping) Canvas + variables (mapping) + geometric objects

14.5 Learning examples

14.5.1 Perfecting the canvas

Adjust y / x limits

Add a different background

Change x / y labels

14.5.2 Aesthetic attributes of the geoms

A scatterplot

Add a line of best fit

Add title / subtitle / caption

14.6 Review assignment:

Note: Under construction!

14.7 Other resources:

Note: Under construction!

Part III

Working with data in R

Chapter 15

Importing data

15.1 Working directories

15.2 Reading in data

Chapter 16

Dataframes

16.1 Exploration

16.2 Summarization

Chapter 17

Data wrangling

17.1 Data transformation

17.1.1 Filtering

17.1.2 Grouping

17.1.3 Joining

17.2 The tidyverse and tibbles

17.3 Transformation with dplyr

17.3.1 Filtering

17.3.2 Grouping

17.3.3 Mutating

Part IV

Exploring & analyzing data

Chapter 18

Exploratory Data Analysis

18.1 Exploring distributions

18.2 Variable types & statistics

18.3 Descriptive statistics

Chapter 19

Significance statistics

19.1 Thinking about significance

19.2 Comparison tests

19.3 Correlation tests

Chapter 20

Displaying data

20.1 Tables

20.2 Base plots

Advanced techniques

20.3 ggplot

Advanced techniques

Part V

Accessing & managing data

Chapter 21

Managing project files

Chapter 22

Formatting your own data

Chapter 23

Reading Excel files

Chapter 24

Reading GoogleSheets

Chapter 25

Reading online data

Chapter 26

Exporting data & plots

Part VI

Your R tool bag

Chapter 27

Joining datasets

Chapter 28

Writing functions

Learning goals

- Be able to write your own functions.
- Be able to *source* your own functions from separate R files.
- Be able to use functions to make your work more efficient, effective, and organized.

Instructor tip! Here is some teacher content.

First steps

You've already used dozens of functions during your learning in R so far. As you start applying R to your own projects, you will inevitably encounter a puzzle that could be solved by a custom function you write yourself. This module shows you how.

As explained in the **Calling Functions** module, 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 define your own custom function, these are the three pieces you must be sure to include.

Here is a basic example:

```
my_function <- function(x){  
  y <- 1.3*x + 10  
  return(y)  
}
```

Now use your function:

```
my_function(x=2) # example 1
```

```
[1] 12.6
```

```
my_function(x=4) # example 2
```

```
[1] 15.2
```

Let's break this down.

- `my_function` is the name you are giving your function. It is the command you will use to call your function.
- The `function()` command is what you use to define a function.
- `x` is the variable you are using to represent your input.
- `y <- 1.3x + 10` is the process that you are applying to your input.
- `return(y)` is the command you use to define what the function's output will be.

Note that you are not *required* to write out `x=2` in full when you are calling your function. Just providing `2` can also work:

```
my_function(2)
```

```
[1] 12.6
```

Exercise 1

Define your own basic function and run it to make sure it works.

Next steps

Multiple inputs

You can define a function with multiple inputs. Just separate each input with a comma.

To demonstrate this, let's modify the function above to allow you to define any linear regression you wish:

```
my_function <- function(x,a,b){  
  y <- a*x + b  
  return(y)  
}
```

Now call your function:

```
my_function(x=2,a=1.3,b=10) # example 1
```

```
[1] 12.6
```

```
my_function(x=4,a=5,b=100) # example 2
```

```
[1] 120
```

Note that you do not need to write out the name of each input, as long as you provide inputs in the correct order.

```
my_function(2, 1.3, 10) # example 1
```

```
[1] 12.6
```

```
my_function(4, 5, 100) # example 2
```

```
[1] 120
```

But note that it is usually best practice to name each input in your function call, to prevent the possibility of any confusion or mistakes. Also, when you name each input you can provide inputs in whatever order you wish:

```
my_function(x=2, a=1.3, b=10)
```

```
[1] 12.6
```

```
my_function(a=1.3, b=10, x=2) # different inout order, same output value
```

```
[1] 12.6
```

Providing defaults for inputs

Just as R's base functions include default values for some inputs (think `na.rm=FALSE` for `mean()` and `sd()`), you can define defaults in your own functions.

This version of `my_function` includes default values for inputs `a` and `b`.

```
my_function <- function(x,a=1.3,b=10){  
  y <- a*x + b  
  return(y)  
}
```

When you provide default values, you no longer need to specify those inputs in your function call:

```
my_function(x=2)
```

```
[1] 12.6
```

NULL as a default

Setting the default value for an input to `NULL` can be useful in certain use cases. For example, let's say that if `b` is not defined by the user, you want its value to be set to five times the value of `x`.

```
my_function <- function(x,a=1.3,b=NULL){  
  
  # Handle input `b`  
  if(is.null(b)){  
    b <- x*5  
    print(paste("b was NULL! Setting its value to ", b))  
  }  
}
```



```

# Now perform process
y <- a*x + b

return(y)
}

```

In this function, a conditional statement is used – `if(is.null(b)){ ... }` – to handle the input `b` when the user does not specify a value for it. When `b` is `NULL`, the logical test `is.null(b)` will be `TRUE`, which will trigger the conditional statement and case `b` to be defined as `x*5`. Conditional statements will be covered in detail in the next modules.

Try running the function with and without providing a value for `b`.

```
my_function(x=2,b=5)
```

```
[1] 7.6
```

```
my_function(x=2)
```

```
[1] "b was NULL! Setting its value to 10"
```

```
[1] 12.6
```

Conditional statements such as `if(is.null(x)){ ... }` or `if(is.na(x)){ ... }` will be helpful in dealing with all the possible values that a user can pass to your custom functions.

Complex inputs

You can pass vectors, dataframes, and any other data structure as inputs in your own custom functions. For example:

```
my_input <- 1:20
my_function(x=my_input,a=1.2,b=10)
```

```

[1] 11.2 12.4 13.6 14.8 16.0 17.2 18.4 19.6 20.8 22.0 23.2 24.4 25.6 26.8 28.0
[16] 29.2 30.4 31.6 32.8 34.0

```

Complex function outputs

At some point you will want multiple objects to be returned by your function. For example, perhaps you want both `y` and `b` to be returned now that you can define `b` according to the value of `x`.

Unfortunately, the `return()` command does not let you include multiple objects. `return(y,b)` will not work. To make it work, you have to place your output objects within a single object, such as a vector, dataframe, or list.

Here is a modification of `my_function()` that allows multiple outputs:

```
my_function <- function(x,a=1.3,b=NULL){  
  
  # Handle input `b`  
  if(is.null(b)){  
    b <- x*5  
  }  
  
  # Now perform process  
  y <- a*x + b  
  
  output <- c("y"=y,"b"=b)  
  return(output)  
}
```

Now `my_function()` works like this:

```
my_function(x=5)
```

```
      y      b  
31.5 25.0
```

To get the value of just `y` or just `b`, you can treat the output just like any other vector:

```
my_function(x=5)[1]
```

```
      y  
31.5
```

```
my_function(x=5)[2]
```

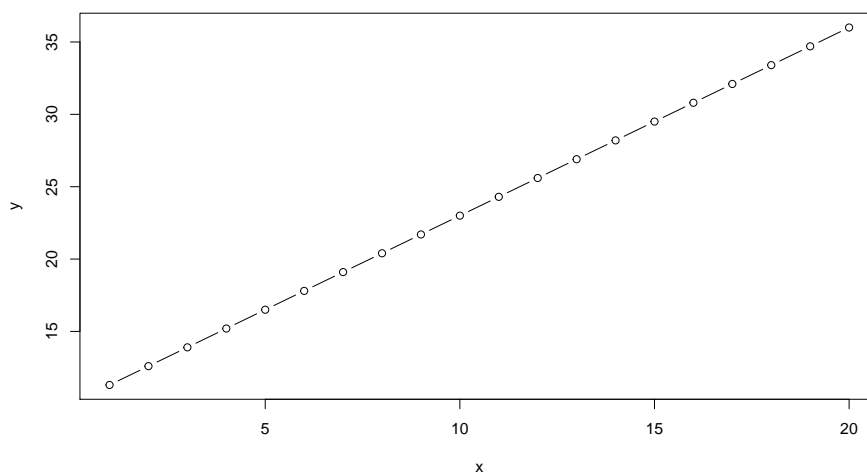
```
      b  
25
```

Adding plots

Plots can be included in the function commands just as in any other context:

```
my_function <- function(x,a=1.3,b=10){
  y <- a*x + b
  plot(y ~ x, type="b")
  return(y)
}
```

```
my_input <- 1:20
my_function(x=my_input)
```



```
[1] 11.3 12.6 13.9 15.2 16.5 17.8 19.1 20.4 21.7 23.0 24.3 25.6 26.9 28.2 29.5
[16] 30.8 32.1 33.4 34.7 36.0
```

Adding plots to functions can be super useful if you want to make multiple plots with the same formatting specifications. Rather than retyping the same long plot commands multiple times, just write a single function and call the function as many times as you wish.

Let's add some fancy formatting to our plot. Note that we will modify the name of the function to make it more descriptive and helpful. The `lm` in `plot_my_lm` stands for *linear model*, which is what is being defined with the $y=ax+b$ equation.

```

plot_my_lm <- function(x,a=1.3,b=10,plot_only=TRUE){

  # Process
  y <- a*x + b

  # Plot
  par(mar=c(4.2,4.2,3,.5)) # set plot margins
  plot(y ~ x, type="o",axes=FALSE,ann=FALSE,pch=16,col="firebrick",xlim=c(-20,20),ylim=
  title(main=paste("y =",a,"x +",b)) # print a dynamic main title
  title(xlab="x",ylab="y") # print axis labels
  axis(1) # print the X axis
  axis(2,las=2) # print the Y axis and turn its labels right-side-up
  abline(h=0,v=0,col="grey70") # add grey lines indicating x=0 and y=0

  # Return
  if(plot_only==FALSE){
    return(y)
  }
}

```

Note that we added a parameter, `plot_only`. When it is set to `TRUE`, the function will not return any numbers.

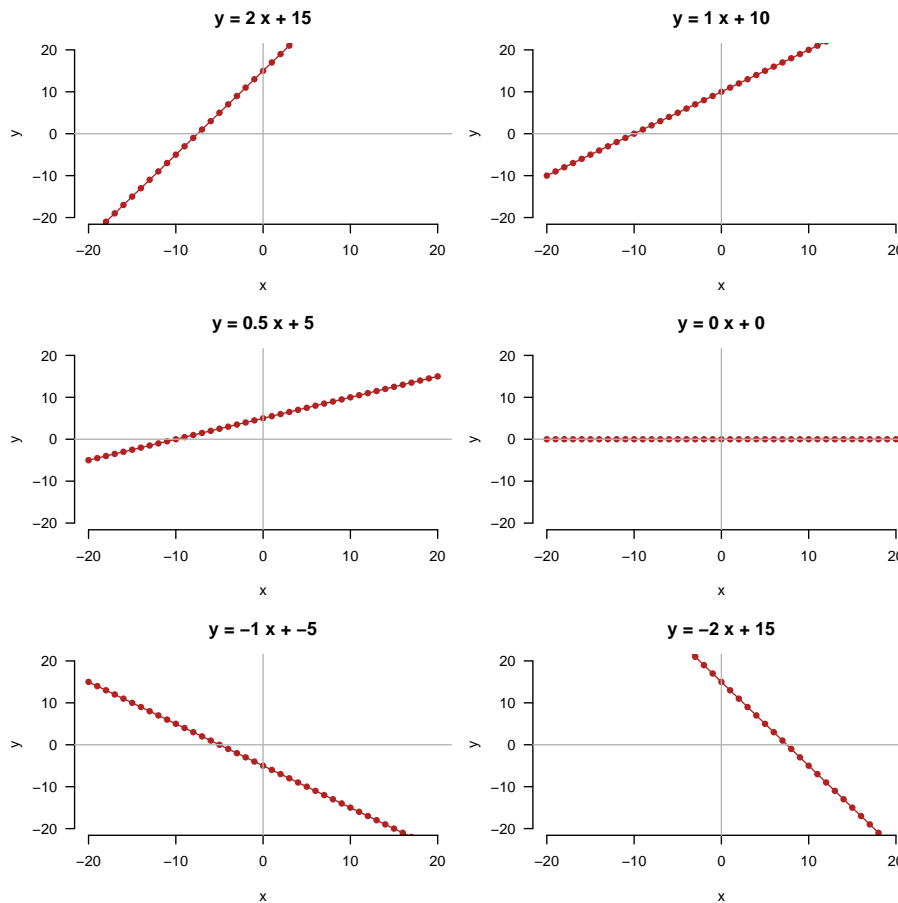
Now let's call this fancy function a bunch of times:

```

my_input <- -20:20 # define a common x input value

par(mfrow=c(3,2)) # stage a multi-paned plot
plot_my_lm(x=my_input,a=2,b=15)
plot_my_lm(x=my_input,a=1,b=10)
plot_my_lm(x=my_input,a=.5,b=5)
plot_my_lm(x=my_input,a=0,b=0)
plot_my_lm(x=my_input,a=-1,b=-5)
plot_my_lm(x=my_input,a=-2,b=15)

```



Think about how many lines of code would have been needed to write out all of these fancy plots if you did not use a custom function! Think about how cluttered and dizzying your code would look! And think about how many opportunities for errors and inconsistencies there would have been! That is the advantage of writing your own functions: it makes your work more efficient, more organized, and less prone to errors.

Another major advantage of this approach comes into play when you decide you want to tweak the formatting of your plot. Rather than going through each `plot(...)` command and modifying the inputs in each one, when you write a custom plotting function you just have to make those changes once. Again, using a custom function saves you time and removes the possibility of inconsistencies or mistakes in the plots you are creating.

Exercise 2

Modify the most recent version of `plot_my_lm` above such that you can specify the color for the plotted line as an input in the function. Then reproduce the multi-paned plot using a different color in each plot. (Here is a good reference for color options in R).

Sourcing functions

As you advance in your coding, you will likely be writing multiple custom functions within a single R script. It is usually useful to group these functions into the same section of code near the top of your script.

But for even *better* script organization and simplification, you should *source* your functions from a separate R script. This means placing your function code in a separate R script and calling that file from the script in which you are carrying out your analyses. In addition to simplifying your analysis script, keeping your functions in a separate file allows them to be shared or sourced from any number of other scripts, which further organizes and simplifies your project's code and increases the reproducibility of your work.

Here is how sourcing functions can work:

1. Open a new R script. Save it as `functions.R` and save it in the same working directory as the script you are using to work through this module.
2. Copy and paste the `plot_my_lm()` function into your `functions.R` script. Save that script to ensure your code is safe.
3. No remove the code defining `plot_my_lm()` from your module R script.
4. In its place, type this command:

```
source("functions.R")
```

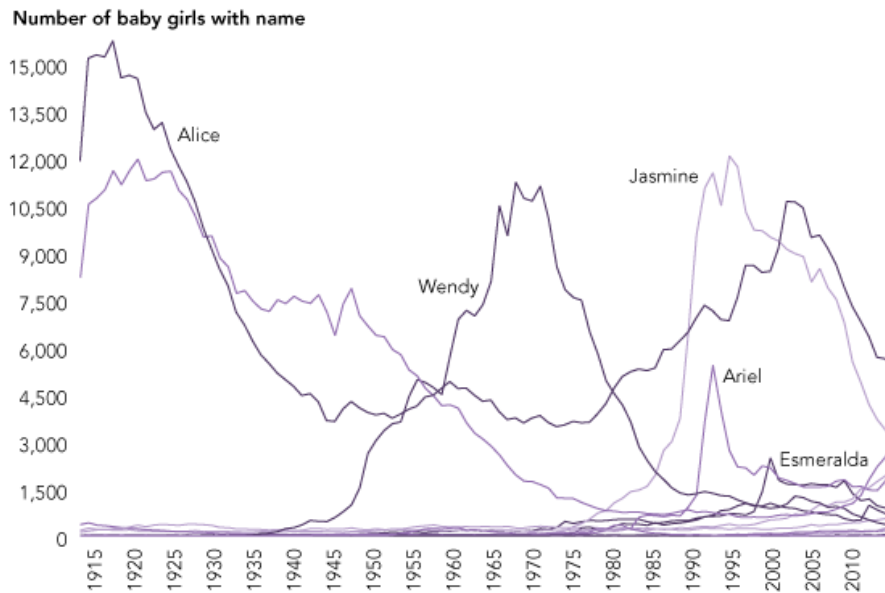
This command tells R to run the code in `functions.R` and store the objects and outputs from it in its active memory. You can now call `plot_my_lm()` from your module script.

Exercise 3

Carry out the above instructions to ensure that you know how to source a function from a separate R script.

Review assignment: Baby names over time

Female Disney Characters



Source: Social Security Administration

THE HUFFINGTON POST

In this exercise, you will investigate annual trends in the prevalence of six names for babies born in the United States.

Step 1. Decide upon five names of interest to you, in addition to your own. Create a vector of these six names.

Step 2. Install and load the package `babynames`, which includes the names of each child born in the United States from 1880 to 2017, according to the Social Security Administration.

Step 3. This package provides the dataset in a dataframe that is also named `babynames`. Familiarize yourself with this dataset using its documentation.

Step 4. Before writing any functions, do some basic exploration with this dataset.

- (a) How many different names have been used since 1880?
- (b) What is the most common name (proportionally) ever given to female babies?
- (c) To male babies?

(d) What was the most common name in 2017?

Step 5. Write a function that takes any name and plots its proportional prevalence from 1880 to 2017. Format the plot beautifully. Provide the name as the `main` title of the plot.

Step 6. Include an input that allows the user to specify the y axis range. If that input is set to `NULL`, the function should just use the maximum value contained in the data itself, scaled by 1.2 to make the plot prettier. Always use 0 as the lower bound of the y axis.

Step 7. Create a multi-pane plot containing six subplots, one for each of your names of interest.

Other Resources

NOTE: Under construction!

Chapter 29

for loops

Learning goals

- What `for` loops are, and how to use them yourself
- How to use `for` loops for plots that are tricky but super cool.
- How to use `for` loops to summarize subgroups in your data
- How to use nested `for` loops

Basics

A `for` loop is a super powerful coding tool. In a `for` loop, R loops through a chunk of code for a set number of repetitions.

A super basic example:

```
x <- 1:5
for(i in x){
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Here's an example of a pretty useless `for` loop:

```
for(i in 1:5){  
  print("I'm just repeating myself.")  
}
```

```
[1] "I'm just repeating myself."  
[1] "I'm just repeating myself."  
[1] "I'm just repeating myself."  
[1] "I'm just repeating myself."  
[1] "I'm just repeating myself."
```

This code is saying:

- For each iteration of this loop, step to the next value in `x` (first example) or `1:5` (second example).
- Store that value in an object `i`,
- and run the code inside the curly brackets. - Repeat until the end of `x`.

Look at the basic structure:

- In the `for()` parenthetical, you tell R what values to step through (`x`), and how to refer to the value in each iteration (`i`).
- Within the curly brackets, you place the chunk of code you want to repeat.

Another basic example, demonstrating that you can update a variable repeatedly in a loop.

```
x <- 2  
for(i in 1:5){  
  x <- x*x  
  print(x)  
}
```

```
[1] 4  
[1] 16  
[1] 256  
[1] 65536  
[1] 4294967296
```

Another silly example:

```
professors <- c("Keri", "Deb", "Ken")  
for(x in professors){  
  print(paste0(x, " is pretty cool!"))  
}
```

```
[1] "Keri is pretty cool!"  
[1] "Deb is pretty cool!"  
[1] "Ken is pretty cool!"
```

Exercise 1

Use this space to practice the basics of `for` loop formatting.

First, create a vector of names (add at least 3)

```
# Add your names to this vector
famous.names <- c("Lady Gaga","David Haskell","Tom Cruise")
```

Using the examples above as a guide, create a `for` loop that prints the same silly statement about each of these names.

```
# Do your coding here
for(i in famous.names){
  print(paste0(i," has cooties!"))
}
```

```
[1] "Lady Gaga has cooties!"
[1] "David Haskell has cooties!"
[1] "Tom Cruise has cooties!"
```

for loops in plots

These silly examples above do a poor job of demonstrating how powerful a `for` loop can be. You can use `for` loops with data to do really cool things with relatively simple code.

Thinking in `for` loops, however, can take some getting used to. Learning how to design code with `for` loops is one of the most important milestones on your way to thinking like a data scientist.

Efficient multi-panel plots

For example, a `for` loop can be a very efficient way of making multi-panel plots.

Let's use a `for` loop to get a quick overview of the variables included in the `airquality` dataset built into R.

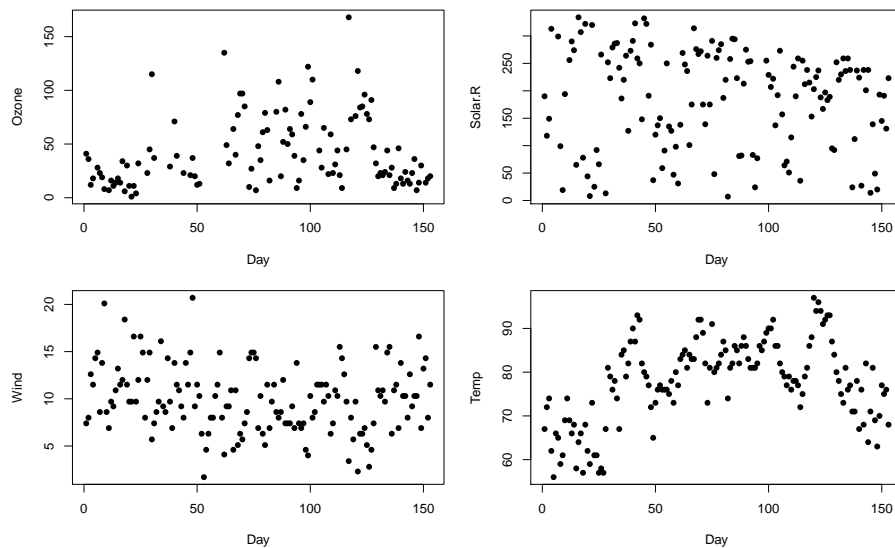
```
data(airquality)
head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

Looks like the first four columns would be interesting to plot.

```
par(mfrow=c(2,2)) # Setup a multi-panel plot # format = c(number of rows, number of co
par(mar=c(4.5,4.5,1,1)) # Set plot margins

# Loop through the first four columns ...
for(i in 1:4){
  y <- airquality[,i] # Select data in column i
  var.name <- names(airquality)[i] # Get name of that column
  plot(y,xlab="Day",ylab=var.name,pch=16) # Plot data
}
```



```
par(mfrow=c(1,1)) # restore the default single-panel plot
```

Using for loops to plot subgroups of data

for loops are also useful for plotting data in tricky ways. Let's use a different built-in dataset, that shows the performance of various car make/models.

```
data(mtcars)
head(mtcars)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Let's say we want to see how gas mileage is affected by the number of cylinders a car has. It would be nice to create a plot that shows the raw data as well as the mean mileage for each cylinder number.

```
# Let's see how many different cylinder types there are in the data
ucyl <- unique(mtcars$cyl) ; ucyl
```

```
[1] 6 4 8
```

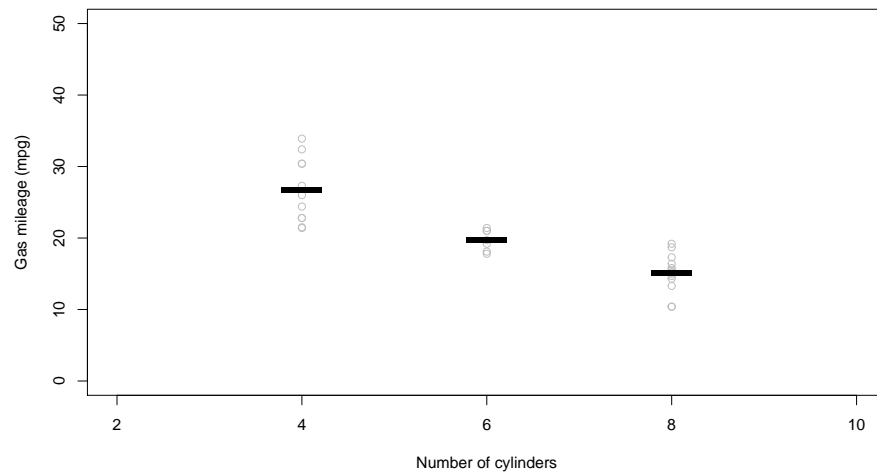
```
# Let's make an empty plot
plot(1,type="n", # tell R not to draw anything
     xlim=c(2,10),ylim=c(0,50),
     xlab="Number of cylinders",
     ylab="Gas mileage (mpg)")

# Write your for loop here to add the actual data
i=ucyl[1] # It's always good to use a known value of i as you build up your for loop
for(i in ucyl){

  # Subset the dataframe according to number of cylinders
  cari <- mtcars[mtcars$cyl==i,]

  # Plot the raw data
  points(x=cari$cyl,y=cari$mpg,col="grey")

  # Superimpose the mean on top
  points(x=i,y=mean(cari$mpg),col="black",pch="-",cex=5,)
}
```



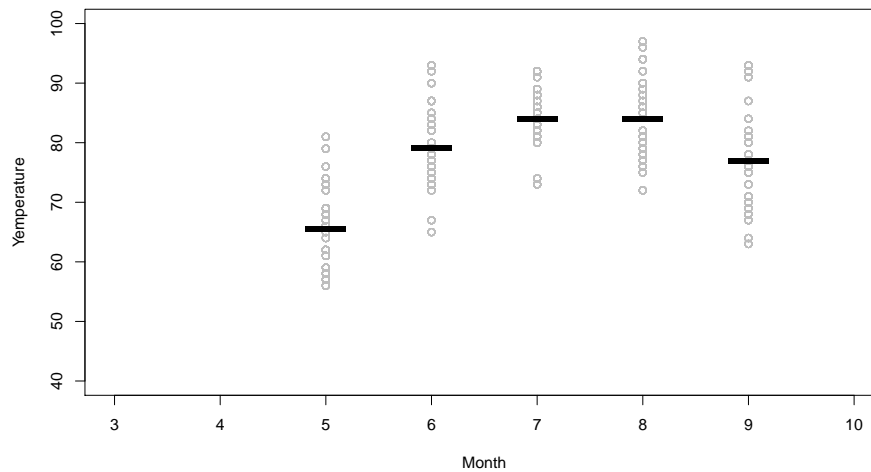
Exercise 3

Now try to do something similar on your own with the `airquality` dataset. Use `for` loops to create a plot with `Month` on the x axis and `Temperature` on the y axis. On this plot, depict all the temperatures recorded in each month in the color grey, then superimpose the mean temperature for each month.

We will provide the empty plot, you provide the `for` loop:

```
plot(1,type="n",
     xlim=c(3,10),ylim=c(40,100),
     xlab="Month",
     ylab="Temperature")

# Write your for loop here to add the actual data
for(i in airquality$Month){
  airi <- airquality[airquality$Month==i,]
  points(x=airi$Month,y=airi$Temp,pch=1,col="grey")
  points(x=i,y=mean(airi$Temp),pch="-",cex=5,col="black")
}
```



Using for loops to layer cyclical or repetitive data

Here's another good example of the power of a good `for` loop.

First, read in some cool data.

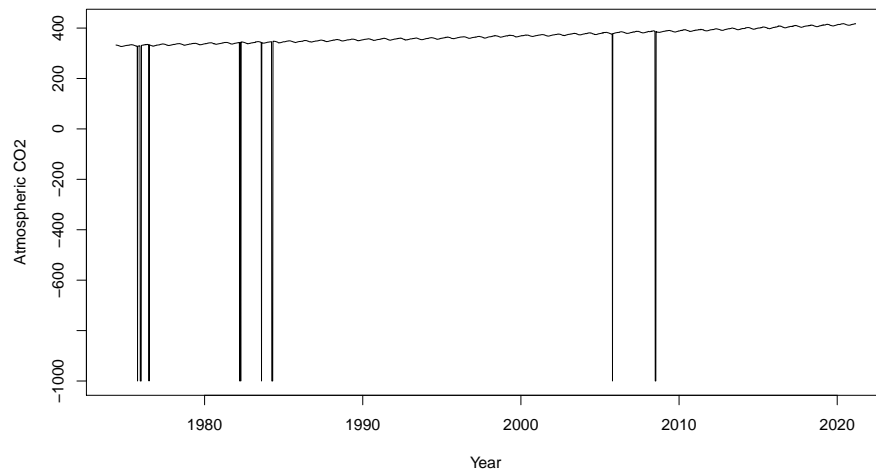
```
kc <- read.csv("./data/keeling-curve.csv") ; head(kc)
```

	year	month	day_of_month	day_of_year	year_dec	frac_of_year	CO2
1	1974	5	26	145.4890	1974.399	0.3986	332.95
2	1974	6	2	152.4970	1974.418	0.4178	332.35
3	1974	6	9	159.5050	1974.437	0.4370	332.20
4	1974	6	16	166.5130	1974.456	0.4562	332.37
5	1974	6	23	173.4845	1974.475	0.4753	331.73
6	1974	6	30	180.4925	1974.495	0.4945	331.68

This is the famous Keeling Curve dataset: long-term monitoring of atmospheric CO₂ measured at a volcanic observatory in Hawaii.

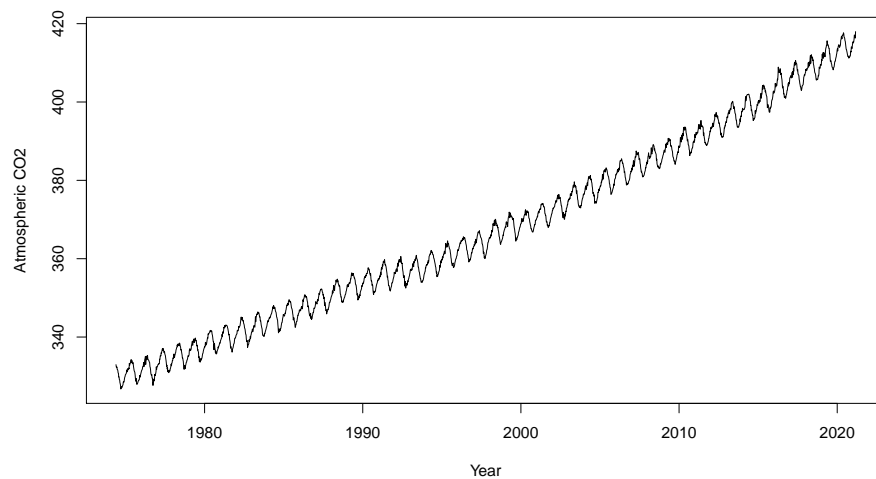
Try plotting the Keeling Curve:

```
plot(kc$CO2 ~ kc$year_dec, type="l", xlab="Year", ylab="Atmospheric CO2")
```



There are some erroneous data points! We clearly can't have negative CO2 values. Let's remove those and try again:

```
kc <- kc[kc$CO2 > 0,]  
plot(kc$CO2 ~ kc$year_dec, type="l", xlab="Year", ylab="Atmospheric CO2")
```



What's the deal with those squiggles?

They seem to happen every year, cyclically. Let's investigate!

Let's look at the data a different way: *by layering years on top of one another.*

To begin, let's plot data for only a *single* year:

```
# Stage an empty plot for what you are trying to represent
plot(1, # plot a single point
     type="n",
     xlim=c(0,365),xlab="Day of year",
     ylim=c(-5,5),ylab="CO2 anomaly")
abline(h=0,col="grey") # add nifty horizontal line

# Reduce the dataset to a single year (any year)
kcy <- kc[kc$year=="1990",] ; head(kcy)
```

	year	month	day_of_month	day_of_year	year_dec	frac_of_year	CO2
816	1990	1	7	6.4970	1990.018	0.0178	353.58
817	1990	1	14	13.5050	1990.037	0.0370	353.99
818	1990	1	21	20.5130	1990.056	0.0562	353.92
819	1990	1	28	27.4845	1990.075	0.0753	354.39
820	1990	2	4	34.4925	1990.094	0.0945	355.04
821	1990	2	11	41.5005	1990.114	0.1137	355.09

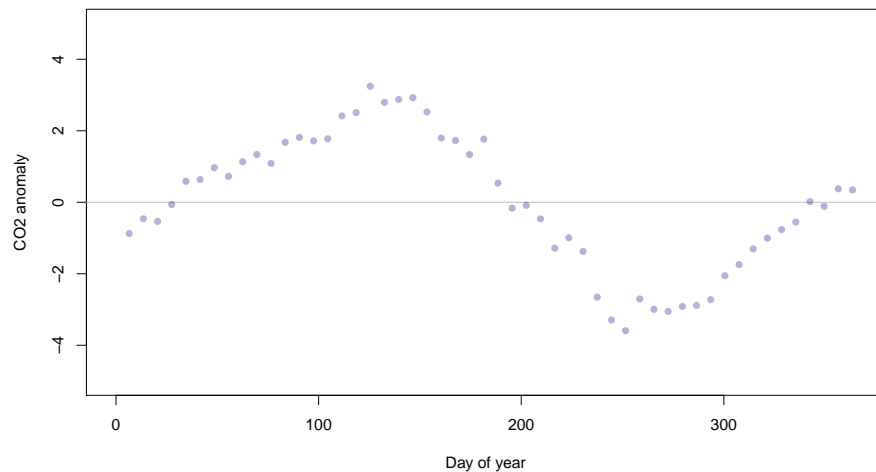
```
# Let's convert each CO2 reading to an 'anomaly' compared to the year's average.
CO2.mean <- mean(kcy$CO2,na.rm=TRUE) ; CO2.mean # Take note of how useful that 'na.rm=TRUE' is
```

```
[1] 354.4538
```

```
y <- kcy$CO2 - CO2.mean ; y # Translate each data point to an anomaly
```

```
[1] -0.87384615 -0.46384615 -0.53384615 -0.06384615  0.58615385  0.63615385
[7]  0.96615385  0.72615385  1.13615385  1.33615385  1.08615385  1.67615385
[13]  1.81615385  1.71615385  1.77615385  2.41615385  2.50615385  3.24615385
[19]  2.79615385  2.87615385  2.92615385  2.52615385  1.79615385  1.72615385
[25]  1.33615385  1.76615385  0.53615385 -0.16384615 -0.08384615 -0.46384615
[31] -1.28384615 -0.99384615 -1.37384615 -2.65384615 -3.29384615 -3.59384615
[37] -2.70384615 -2.99384615 -3.05384615 -2.91384615 -2.88384615 -2.72384615
[43] -2.05384615 -1.74384615 -1.30384615 -1.00384615 -0.76384615 -0.55384615
[49]  0.01615385 -0.11384615  0.37615385  0.34615385          NA
```

```
# Add points to your plot
points(y~kcy$day_of_year,pch=16,col=adjustcolor("darkblue",alpha.f=.3))
```



But this only shows one year of data! How can we include the seasonal squiggle from other years?

Let's use a `for` loop!

OK – let's redo that graph and add a `for` loop into the mix:

```
# First, stage your empty plot:
plot(1,type="n",
     xlim=c(0,365),xlab="Day of year",
     ylim=c(-5,5),ylab="CO2 anomaly")

abline(h=0,col="grey")

# Now we will loop through each year of data. First, get a vector of the years included
years <- unique(kc$year) ; years
```

```
[1] "1974" "1975" "1976" "1977" "1978" "1979" "1980" "1981" "1982" "1983"
[11] "1984" "1985" "1986" "1987" "1988" "1989" "1990" "1991" "1992" "1993"
[21] "1994" "1995" "1996" "1997" "1998" "1999" "2000" "2001" "2002" "2003"
[31] "2004" "2005" "2006" "2007" "2008" "2009" "2010" "2011" "2012" "2013"
[41] "2014" "2015" "2016" "2017" "2018" "2019" "2020" "2021" NA
```

```
# Now build your for loop.
# Notice that the contents of the `for loop` are exactly the same
# as the single plot above -- with one exception.
# Notice the use of the symbol i
```

```

for(i in years){

  # Reduce the dataset to a single year
  kcy <- kc[kc$year==i,] ; head(kcy)

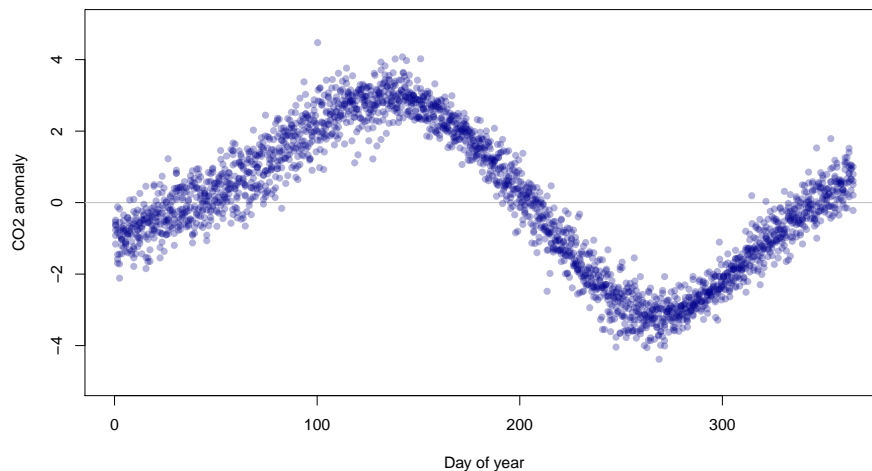
  # Let's convert each CO2 reading to an 'anomaly' compared to the year's average.
  CO2.mean <- mean(kcy$CO2,na.rm=TRUE) ; CO2.mean # Get average CO2 for year

  y <- kcy$CO2 - CO2.mean ; y # Translate each data point to an anomaly

  # Add points to your plot
  points(y~kcy$day_of_year,pch=16,col=adjustcolor("darkblue",alpha.f=.3))

}

```



Beautiful! So how do you interpret this graph? Why does the squiggle happen every year?

Using for loops to process & summarize data

Using for loops to summarize data subgroups

This is one of the most common uses of for loops: you want to summarize information about subgroups in your dataset.

Example use cases

- You want to summarize sample counts for each day of fieldwork.
- You want to summarize details for each user in your database.
- You want to summarize weather information for each month of the year.

Workflow

All of these summaries can be achieved with the same basic workflow.

1. Determine how to define the basic unit of summarization (*e.g., each day of fieldwork, each user in your database, or each month of the year*).
2. Stage an empty data object (or multiple objects) in which you will store your summary.
3. Build a `for` loop to iterate through for each unique subgroup value (*e.g., each sampling date or each user or each month*).
4. In each iteration of the loop, subset your data to only the subgroup value of interest.
5. At the end of each iteration, update your data objects with the result from that iteration.
6. Run your ‘for loop!’
7. After the `for` loop, combine your data objects into a dataframe.

As you are building for loops, it can be very helpful to first start thinking about what needs to happen in each iteration of the loop. So first think about what needs to happen in *Step 4*. Then you go through each step of this workflow to build a loop around that core code.

Example

The following scenario provides a concrete example of how to do this:

Scenario: You participate in a survey of flightless birds in the forests of New Zealand. You conduct thirty days of fieldwork on four species of bird: the kiwi, the weka, the kakapo (*the world’s heaviest parrot*), and the kea (*the world’s only alpine parrot*).

Your data look like this:

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

```
[1] 406
```

```
head(df)
```

```
  day species group
1   1    Kiwi     3
2   1     Kea     4
3   1 Kakapo     4
4   1    Weka     2
5   1    Kiwi     1
6   1     Kea     3
```

```
tail(df)
```

```
  day species group
401  30    Weka     3
402  30    Kiwi     3
403  30    Kiwi     3
404  30 Kakapo     3
405  30 Kakapo     1
406  30     Kea     2
```

Each row contains the data for a single bird group detection.

Your supervisor has tasked you with writing a report of your findings, and she wants to see a table with the number of each species seen on each day of the fieldwork.

You can use a `for` loop to make this table.

But first think about what needs to happen in each iteration of your loop. For each date in the dataset, you want to count the number of sightings for each species.

Let's build our core code using day 10 as a practice date:

```
# Subset to day of interest
dayi <- 10
dfi <- df[df$day == dayi,]

# Tally up counts for each species
kiwi <- length(which(dfi$species=="Kiwi"))
```

```
weka <- length(which(dfi$species=="Weka"))
kakapo <- length(which(dfi$species=="Kakapo"))
kea <- length(which(dfi$species=="Kea"))
```

Okay! So our `for` loop needs to contain that basic code, with some minor adjustments in order to save the results from all iterations of the loop instead of just one.

Here is the full code for this `for` loop problem:

```
# 1: determine the unique days in the dataset:
udays <- unique(df$day)

# 2: stage empty results vectors
kiwi <- weka <- kakapo <- kea <- c()

# (create a variable named i to help you test your code as you build it)
i=1

# 3: Start for loop. Loop through each unique day
for(i in 1:length(udays)){

  # 4. Subset to day of interest
  dayi <- udays[i]
  dfi <- df[df$day == dayi,]

  # 5. Tally up birds, add to each vector
  kiwi[i] <- length(which(dfi$species=="Kiwi"))
  weka[i] <- length(which(dfi$species=="Weka"))
  kakapo[i] <- length(which(dfi$species=="Kakapo"))
  kea[i] <- length(which(dfi$species=="Kea"))

}

# 6. Run for loop

# 7. Combine results into a dataframe
results <- data.frame(day=udays, kiwi, weka, kakapo, kea)

# Check it out!
results
```

	day	kiwi	weka	kakapo	kea
1	1	5	6	3	3
2	2	1	3	1	2

3	3	2	4	5	3
4	4	2	3	2	0
5	5	6	6	3	2
6	6	6	2	3	6
7	7	3	1	3	7
8	8	2	6	2	2
9	9	4	1	1	1
10	10	2	0	3	3
11	11	2	2	3	5
12	12	2	3	4	4
13	13	3	3	5	1
14	14	7	3	4	6
15	15	3	4	4	5
16	16	4	2	1	4
17	17	3	4	6	2
18	18	3	4	1	1
19	19	4	6	1	6
20	20	2	6	3	1
21	21	4	3	5	6
22	22	4	2	5	5
23	23	4	6	2	4
24	24	1	2	5	2
25	25	4	6	3	1
26	26	4	2	3	5
27	27	8	5	6	7
28	28	8	1	0	2
29	29	6	1	1	3
30	30	4	2	3	4

Nested for loops

Sometimes you need to summarize your data in such a specific way that you will need to use *nested for* loops, i.e., one **for** loop contained within another.

For example, your supervisor for the New Zealand Flightless Birds Survey has now taken an interest in associations among the four bird species you have been monitoring. For example, are kiwis more abundant on the days when you detect a lot of kakapos?

To answer this question, your supervisor wants to see a table with each species combination (Kiwi - Kakapo, Kiwi - Weka, ... Kakapo - Kea, etc.) and the number of dates in which both species were seen more than 5 times.

You can produce this table using a nested **for** loop. Here is how it's done:

```

uspp <- unique(df$species) # get set of unique species

A <- B <- c() # empty vector for names of each species in the pair
X <- c() # empty vector for number of dates in which both species were common

i=1 ; j=2

# For loop 1
for(i in 1:length(uspp)){
  spi <- uspp[i] # species i
  dfi <- df[df$species == spi,] # subset df to only this species
  counti <- table(dfi$day) # get number of sightings on each day
  dayi <- names(counti)[which(counti >= 5)] ; # get the dates on which this species was seen

  # For loop 2
  for(j in 1:length(uspp)){
    spj <- uspp[j] # species j
    dfj <- df[df$species == spj,] # subset df to only this species
    countj <- table(dfj$day) # get number of sightings on each day
    dayj <- names(countj)[which(countj >= 5)] ; # get the dates on which this species was seen

    dates_ij <- which(dayi %in% dayj) # get the dates on which both species were seen
    Xij <- length(dates_ij) # count the number of these dates

    # Add results to staged objects
    X <- c(X,Xij)
    A <- c(A,spi)
    B <- c(B,spj)
  }
}

# Combine results into a dataframe
results <- data.frame(A, B, X)

# Check it out!
results

```

	A	B	X
1	Kiwi	Kiwi	7
2	Kiwi	Kea	3
3	Kiwi	Kakapo	1
4	Kiwi	Weka	3
5	Kea	Kiwi	3
6	Kea	Kea	10


```

7      Kea Kakapo 3
8      Kea  Weka 2
9  Kakapo  Kiwi 1
10 Kakapo  Kea 3
11 Kakapo Kakapo 7
12 Kakapo  Weka 1
13  Weka  Kiwi 3
14  Weka  Kea 2
15  Weka Kakapo 1
16  Weka  Weka 8

```

Note that the code for adding the results to the staged objects X, A, and B is contained within the second for loop. This is necessary for producing our results; if we put that code in the first for loop *after* the code for the nested loop, our results would not be complete.

Note that each for loop *must* use a different variable to represent each iteration. In this example, the first loop uses `i` and the second uses `j`. If we used `i` for both loops, R would get very confused indeed.

Also note that we used `i` and `j` in the variables specific to each loop (e.g., `dayi` and `dayj`), as a simple way to help us keep track of what each variable is representing.

Exercise 2

Your supervisor is happy with your pairwise species association dataframe, and wants to use it in an analysis for a publication. However, the R package she wants to use requires that the data be in the format of a square *matrix* with four rows – one for each species – and four columns. Like this:

```
results_matrix <- matrix(data=NA, nrow=4, ncol=4, dimnames=list(uspp,uspp))
results_matrix
```

	Kiwi	Kea	Kakapo	Weka
Kiwi	NA	NA	NA	NA
Kea	NA	NA	NA	NA
Kakapo	NA	NA	NA	NA
Weka	NA	NA	NA	NA

You have not yet worked with matrices in this curriculum (you will in a few modules), but for now think of them as simple dataframes with a single type of data (e.g., all numeric values, like this one). You can subset matrices just as you would a dataframe: `matrix[row,column]`.

The values in this matrix should represent the number of dates in which each species pair was seen 5 times or more. For example, `result[1,2]` would be 3, since the Kiwi and Kea were seen 5+ times on only 3 dates.

She asks you to use the dataframe you just created to create this matrix. Use a nested for loop to do it.

```
# Stage empty results objects
results_matrix <- matrix(data=NA, nrow=4, ncol=4, dimnames=list(uspp,uspp))

# Get unique species
uspp <- unique(c(results$A,results$B))

# Loop 1: each species (row)
for(i in 1:length(uspp)){
  spi <- uspp[i]
  resultsi <- results[results$A==spi,] # subset data to rows where column A equals spe

  # Loop 2: each species (column)
  for(j in 1:length(uspp)){
    spj <- uspp[j]
    resultsj <- resultsi[resultsi$B==spj,] # subset data from i loop where column B eq

    Xij <- resultsj$X # Find result

    results_matrix[i,j] <- Xij # Add result to staged object
  }
}

results_matrix
```

	Kiwi	Kea	Kakapo	Weka
Kiwi	7	3	1	3
Kea	3	10	3	2
Kakapo	1	3	7	1
Weka	3	2	1	8

Boom!

Review assignment

First, read in and format some other cool data. The code for doing so is provided for you here:

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

This dataset, freely available from World Bank, shows the renewable electricity output for various countries, presented as a percentage of the nation's total electricity output. They provide this data as a time series.

Summarize columns with a for loop

Task 1: Use a for loop to find the change in renewable energy output for each nation in the dataset between 1990 and 2015. Print the difference for each nation in the console.

```
# Write your code here
i=2
for(i in 2:ncol(df)){
  dfi <- df[,i] ; dfi
  diffi <- dfi[length(dfi)] - dfi[1] ; diffi
  print(paste0(names(df)[i], " : ",round(diffi,"% change."))
}
```

```
[1] "World : 3% change."
[1] "Australia : 4% change."
[1] "Canada : 1% change."
[1] "China : 4% change."
[1] "Denmark : 62% change."
[1] "India : -9% change."
[1] "Japan : 5% change."
[1] "New_Zealand : 0% change."
[1] "Sweden : 12% change."
[1] "Switzerland : 7% change."
[1] "United_Kingdom : 23% change."
[1] "United_States : 2% change."
```

Task 2: Re-do this loop, but instead of printing the differences to the console, save them in a vector.

```
# Write your code here
diffs <- c()
i=2
for(i in 2:ncol(df)){
  dfi <- df[,i] ; dfi
  diffi <- dfi[length(dfi)] - dfi[1] ; diffi
  diffs <- c(diffs,diffi)
```

```
}
diffs
```

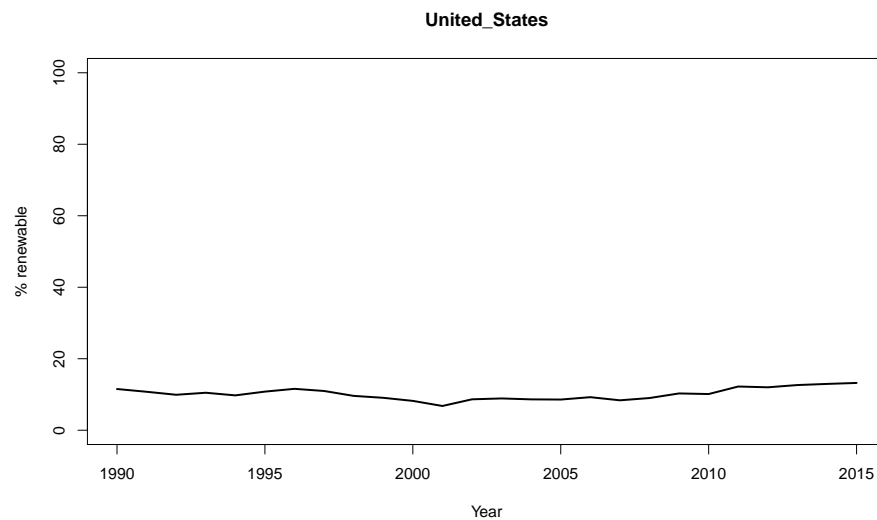
```
[1] 3.49241703 3.98181045 0.63273122 3.51887728 62.33064943 -9.14624362
[7] 4.73004321 0.07524008 12.26263811 7.21543884 23.01128298 1.69994636
```

Multi-pane plots with for loops

Practice with a single plot

Task 3: First, get your bearings by figuring out how to use the `df` dataset to plot the time series for the United States, for the years 1990 - 2015. Label the x axis “Year” and the y axis “% Renewable”. Include the full name of the country as the `main` title for the plot.

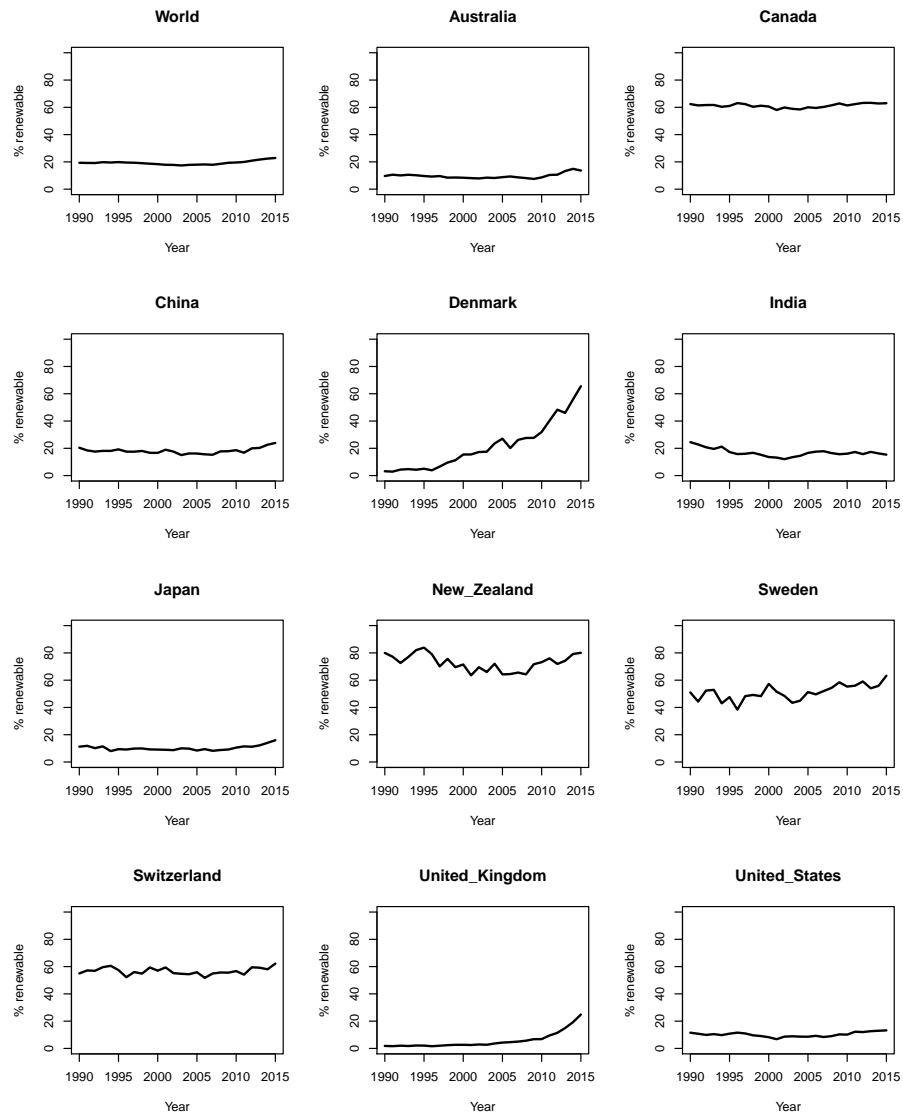
```
# Write code here
dfi <- df[,c(1,13)]
plot(x=dfi[,1],
     y=dfi[,2],
     type="l",lwd=2,
     xlim=c(1990,2015),ylim=c(0,100),
     xlab="Year",ylab="% renewable",
     main=names(dfi)[2])
```



Now loop it!

Task 4: Use that code as the foundation for building up a `for` loop that displays the same time series for every country in the dataset on a multi-pane graph that with 4 rows and 3 columns.

```
par(mfrow=c(4,3))
i=3
for(i in 2:ncol(df)){
  dfi <- df[,c(1,i)] ; dfi
  plot(x=dfi[,1],
        y=dfi[,2],
        type="l",lwd=2,
        xlim=c(1990,2015),ylim=c(0,100),
        xlab="Year",ylab="% renewable",
        main=names(dfi)[2])
}
```



Now loop it *in layers!*

Task 5: Now try a different presentation. Instead of producing 12 different plots, superimpose the time series for each country on the *same single plot*.

To add some flare, highlight the USA curve by coloring it red and making it thicker.

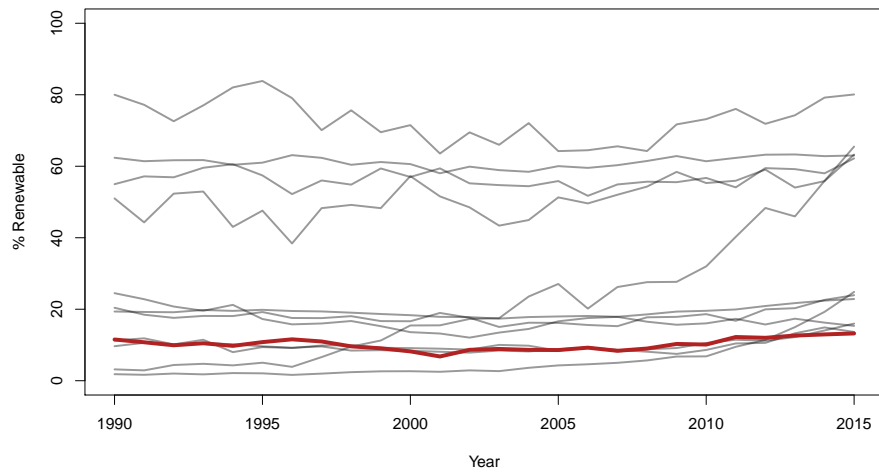
```

par(mfrow=c(1,1))
plot(1,type="n",lwd=2,
      xlim=c(1990,2015),ylim=c(0,100),
      xlab="Year",ylab="% Renewable")

for(i in 2:ncol(df)){
  dfi <- df[,c(1,i)] ; dfi
  lines(dfi[,2]~dfi[,1],lwd=2,col=adjustcolor("black",alpha.f=.4))
}

lines(df$United_States~df$year,lwd=4,col="firebrick")

```



Chapter 30

Conditional statements

Learning goals

- Understand what conditional statements are, and why they are so awesome.
- Be able to write your own conditional statements in R.

First steps

An example of a conditional statement is, “*If _____ happens, do _____. Otherwise, do _____.*”

In R code, conditional statements work a similar way: they let a variable’s value determine which process to carry out next.

Here is a basic example:

```
x <- 4

if(x==3){
  message("x is equal to 3!")
}else{
  message("x does NOT equal 3!")
}
```

Let’s break this `if` statement down.

- The `if` command opens up a conditional statement.

- The parenthetical (`x==3`) is where the logical test happens. If the result of this test is `TRUE`, then the `if` statement will be processed; if not, the `else` statement will be run instead.
- The curly brackets, `{ }` serve to contain the code that will be run, depending on the outcome of the logical test.
- The `else` command indicates the start of the code that will be run if the logical test's result is `FALSE`.

This code ran the logical test `x==3`, determined its outcome to be `FALSE`, and so it skipped the `if` code and ran the `else` code instead.

Here is another example of the same idea:

```
x <- 4

if(x==3){
  y <- 100
}else{
  y <- 0
}

y # see what y is
```

```
[1] 0
```

Since `x==3` returned `FALSE`, `y` was defined as 0 instead of 100.

Note that you do not need to define an `else` statement. If you do not, `R` will simply do nothing if the logical test is `FALSE`.

```
x <- 4

if(x==3){
  message("x is equal to 3!")
}
```

(Nothing happened)

You can also feed the `if` statement a logical object instead of a test.

```
x <- 4
x_test <- x == 3
x_test # check out the value of x_test
```

```
[1] FALSE
```

```
if(x_test){
  message("x is equal to 3!")
}
```

Not that, since `x_test` is a logical value (`TRUE` or `FALSE`), you do not need to write out a logical test within the `if` parenthetical. But you are free to do so if you wish:

```
x <- 4
x_test <- x == 3

if(x_test == FALSE){
  message("x is not equal to 3!")
}
```

This `if` statement is saying that, if it is `TRUE` that `x_test` is `FALSE`, print a message saying so.

Exercise 1

Write out your own basic `if...else` statement and ensure that it works.

Next steps

Nested conditions

You can nest conditional statements within others as many times as you wish:

```
x <- 6

if(x==3){
  message("x equals 3")
}else{
  if(x < 3){
    message("x is less than 3")
  }else{
    message("x is greater than 3")
  }
}
```

Note that every open bracket, `{`, needs a corresponding closing bracket `}`. Most errors associated with `if` statements involve missing brackets.

Handling NAs, NaN's, Infs, and NULLs

`if` statements can be particularly helpful when your dataset contains missing or broken values. R includes base functions that help you carry out logical tests concerning missing values. These three test below can be very helpful within `if` statements.

`is.finite()` tests whether a numeric object is a real, finite number.

```
x <- 0
is.finite(x)
```

```
[1] TRUE
```

```
y <- 10/x
y
```

```
[1] Inf
```

```
is.finite(y)
```

```
[1] FALSE
```

Here is an `if` statement making use of `is.finite()`:

```
x <- 0
y <- 10/x
if(is.finite(y)){
  message("y is indeed a finite number!")
}else{
  message("y is not a finite number!")
}
```

`is.na()` tests whether a variable contains a missing or broken object.

```
x <- "eric"
is.na(x)
```

```
[1] FALSE
```

```
y <- as.numeric(x) # try converting `x` to a numeric value
is.na(y)
```

```
[1] TRUE
```

```
is.finite(y)
```

```
[1] FALSE
```

NA stands for “*Not Available*”. NaN is also a common sign of a broken value. It stands for *Not a Number*.

`is.null()` tests whether a variable is empty. That is, it has been initiated, but it contains no data.

```
x <- c(4,7,3,1,5) # make a vector of numbers
is.null(x)
```

```
[1] FALSE
```

```
y <- c() # make an empty vector
is.null(y)
```

```
[1] TRUE
```

Joint conditions

if statements can also accommodate joint logical tests. For example, the follow if statement only returns if a message if *two* tests are true:

```
x <- 6
if(is.finite(x) & x==6){
  message("x is real and equals 6")
}
```

The next if statement returns a message if either of the logical statements are true:

```
x <- 6
if(x==3 | x==6){
  message("x is equal to either 3 or 6")
}
```

Exercise 2

Write a nested `if` statement that produces a message reporting the hemisphere for any GPS position you provide it (a pair of latitude and longitude coordinates, in decimal degrees). The four hemisphere options are as follows:

- *Northwest* (positive latitudes and negative longitudes, e.g., USA)
- *Northeast* (positive latitudes and longitudes, e.g., Russia)
- *Southwest* (negative latitudes and longitudes, e.g., Brazil)
- *Southeast* (negative latitudes and positive longitudes, e.g., New Zealand).

Include the ability to handle missing values (i.e., if an `NA` is provided, return a message saying that values are missing and the hemisphere cannot be determined.)

Provide five examples that demonstrate the functionality for all the possible message options.

Review assignment

Note: This exercise will combine all the skills you’ve learned for `for` loops, `if` statements, *and* writing functions into a real-world data science scenario. Buckle up!

You are working at the Center for Disease Control. Your supervisor has asked you to take a look at state-level data on infectious diseases within the United States in the last century. Specifically, she wants you to address the following questions and requests:

1. In the last 90 years, which states have had the highest average prevalence of **measles**, **pertussis** (whooping cough), and **smallpox** in proportion to their population sizes? Which have had the lowest?
2. Provide beautiful plots showing trends in the prevalence of these diseases over the last century. Produce a single plot for each state, with three lines representing the time series for each disease of interest. Save each plot as a `pdf` into a folder named `state-level-summaries`. Name each `pdf` using the state’s name.

To do this work, your supervisor asks you to use the `us_contagious_diseases` dataset contained within the R package `dslabs`, which contains disease data from 1928-2011 for all states. To make sure the numbers reflect actual patterns, she asks you to only use prevalence numbers for years in which counts were made in more than 20 weeks out of the year.

Other Resources

NOTE: Under construction!

Chapter 31

Working with text

Chapter 32

Working with dates & times

Chapter 33

Working with factors

Chapter 34

Cleaning messy data

Chapter 35

Matrices & lists

Chapter 36

Pipes

Part VII

Interactive dashboards

Chapter 37

Intro to Shiny apps

Chapter 38

Shiny dashboards

Chapter 39

Data entry apps

Part VIII

Databases

Chapter 40

Introduction

40.1 What

40.2 Why

40.3 When

40.4 When not

Chapter 41

Platforms

41.1 PostgreSQL

41.2 MySQL

41.3 SQLite

Chapter 42

Alternatives

42.1 NoSQL

Chapter 43

Practices

Spinning up a local DB

Part IX

Documenting your work

Chapter 44

R Markdown

Chapter 45

Reproducible research

Chapter 46

Automated reporting

Chapter 47

Formatting standards

47.1 Tables

47.2 Figures

47.3 Captions

Part X

Version control and teamwork

Chapter 48

What is version control?

Chapter 49

What is Git?

49.1 Repositories

49.2 Github

Chapter 50

Standard git operations

Chapter 51

A git workflow

Chapter 52

Other git platforms

Part XI

Writing about data

Chapter 53

Types of writing

53.1 Grant proposals

53.2 Reports and publications

53.3 Fundraising

53.4 Press releases

Chapter 54

Elements of style

Chapter 55

Sections of a report

55.1 Abstract

55.2 Introduction

55.3 Methods

55.4 Results

55.5 Discussion

55.6 Other elements

55.6.1 Acknowledgments

55.6.2 Literature Cited

55.6.3 Tables

55.6.4 Figures

55.6.5 Supplementary Materials

Part XII

Creating websites

Part XIII

Advanced skills

Chapter 56

Mapping

Chapter 57

Geographic computing & GIS

Chapter 58

Statistical modeling

Chapter 59

Apply family

Chapter 60

Iterative statistics

Chapter 61

Iterative simulations

Chapter 62

Image analysis

Chapter 63

Machine learning

Chapter 64

Template

Learning goals

- Item 1
- Item 2
- Item 3

Instructor tip!Here is some teacher content.

Tutorial video

Bangarang - Crew Briefing from Luke Padgett on Vimeo.

Basics

Exercise 1

Review assignment

Introduce data

Introduce task(s)

Other Resources

<https://desiree.rbind.io/post/2020/learnr-iframe/>

<https://rstudio.github.io/learnr/>

Bibliography

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Wilkinson, L. (2005). *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg.