EDS 221: SCIENTIFIC PROGRAMMING ESSENTIALS

Allison Horst

Contents

Chapter 1

Scientific programming essentials for environmental data science

Material disclaimer and use

This book was created by Allison Horst for EDS 221 (Scientific Programming Essentials) in the Bren School's 1-year Master of Environmental Data Science program at UC Santa Barbara. It accompanies lecture, computational lab and discussion materials that may or may not be linked to throughout the book. This book is intended as a supplemental resource for some parts of the course. In other words, it is not intended as a standalone textbook.

All materials in this book are openly available for use and reuse by Creative Commons Attribution and Share-Alike license.

Thank you in advance for suggestions and corrections, which can be submitted as issue to this GitHub repo.

Acknowledgments

I create my courses while standing on shoulders of generous teaching and developing giants in R, data science, and education communities. The wealth and quality of open educational resources (OERs) in data science has made teaching

in the field fun, innovative, and inspiring. I've tried to thoroughly credit authors resources that I have pulled from and adapted for this book, and I welcome additions if I have missed any that should be included.

That includes leaning heavily on:

- Advanced R by Hadley Wickham (Wickham, 2019)
- Introducing Python by Bill Lubanovic (Lubanovic, 2014)

1.1 Course introduction

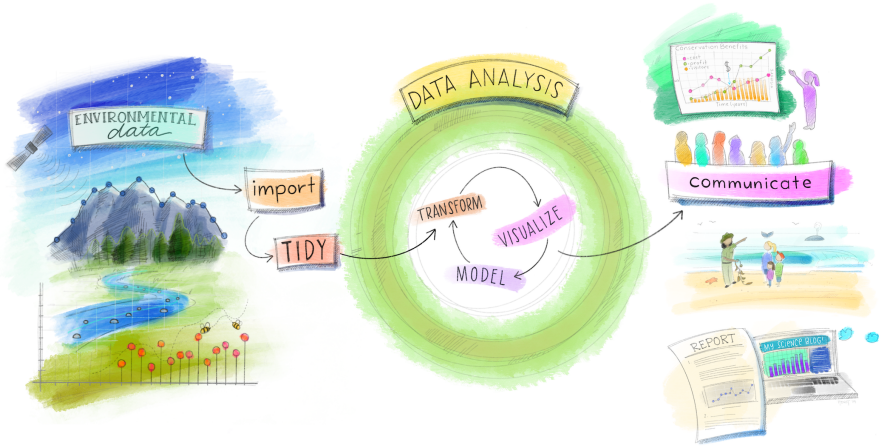


Figure 1.1: Illustration from Dr. Julia Lowndes' 2019 keynote talk at useR conference (by Allison Horst), adapted from the classic R for Data Science workflow schematic by Grolemund and Wickham

As nicely summarized in the title of a 2018 NCEAS post, **"the next generation of environmental scientists are data scientists"**. The explosion in environmental data volume, heterogeneity and availability in recent years has made managing, wrangling, analyzing and communicating with data a critical skill in environmental workplaces. Over the next year in MEDS you'll build skills to responsibly apply advanced methods in environmental modeling, spatial data analysis, machine learning, and more to investigate, analyze and communicate with complex environmental data.

To get there, you'll need a strong foundation in programming basics like: understanding types and structures of data, basic data wrangling and visualization, algorithm development with functions, loops, and conditionals, and how to troubleshoot. While working in the weeds of programming, we'll also learn

and reinforce transferable habits for reproducible workflows, robust file paths, version control, data organization, project management, and more.

In EDS 221 you'll also start building versatility by learning fundamental programming skills in different languages (R, Python) and integrated development environments (IDEs) like RStudio and PyCharm, while documenting our work in R Markdown and Jupyter Notebooks.

Upon the building blocks established in EDS 221, you'll be prepared to incrementally grow your advanced environmental data science toolkit throughout MEDS, then enter the workplace at the leading edge of quantitative methods in the field.

1.2 Links to course materials

- EDS 221 Syllabus
- Code of Conduct
- EDS 221 GitHub site

1.3 Course setup

In EDS 221 we will write code in R (in scripts and R Markdown) and Python (in scripts and Jupyter Notebooks). You should be set to go if you followed along with the MEDS Installation Guide during orientation.

1.4 Course resources

- Hands-on Programming with R by Garrett Grolemund
- R for Data Science by Garrett Grolemund and Hadley Wickham
- Advanced R by Hadley Wickham
- Learn Python Break Python by Scott Grant

Chapter 2

Programming, R, and Python

2.1 What is programming?

Programming is the process of designing then telling a computer instructions to do something useful for you, usually by taking an input and producing some meaningful output.

For example, if you've found a mean value using the `AVERAGE()` function in Excel, that's programming. But have you ever looked back at an analysis you've done in Excel weeks or months (or years) later, and tried to follow your work start to finish? It's…not great.

Throughout your MEDS courses you'll learn to programming by writing *code* to keep clear, complete and readable records of everything we do with environmental data - from accessing it to preparing final reports. In doing so, you'll build workflows to make your data science reproducible, so that you, your collaborators, or other people can re-run your code completely from start-to-finish.

Scripted code will become the core component of our reproducible data science. There are **hundreds** of coding languages - to feel immediately overwhelmed, you're encouraged to check out Wikipedia's list of notable programming languages (Wikipedia contributors, 2021). But **don't panic**, we're going to build programming skills in just two of those during EDS 221 - R and Python - because they are some of the most commonly used languages in environmental data science across sectors.

Is it possible that you'll join a team using a different language? Sure. However, a strong foundation in a few representative programming languages, along with a basic set of tools for reproducible, collaborative work, will allow you to transfer

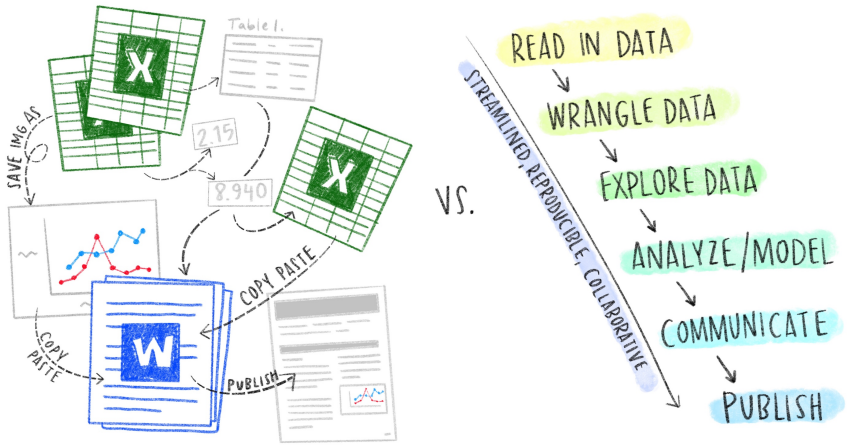


Figure 2.1: How it started, and how we hope it's going after MEDS.

your skills quickly to other languages, IDEs, or teams as they arise in your career.

So let's start with our most critical tool: our coding languages.

2.2 R

R is a programming language - a language with unique syntax and expectations for how you will give your computer instructions.

2.3 Python

You probably have Python 3 already installed if you have a recent-ish computer. If not, you'll want to install it. How can you run some Python code without anything else?

- Open the Terminal on your device (Cmd + Spacebar), open the Terminal
- Run `python` (press Enter) - this will tell you which version of Python you're using, and switch over into a Python interpreter.
- Now you're using this as a Python editor (you should see the `>>>` starting each line, prompting Python code. Try running some basic code, like:
- `print("data science")`
- `2 + 10`

- `for i in 1, 2, 3, "hooray": print(i)` [press Enter twice to run…otherwise it's like "are you really done with this loop?"]
- `help()` will open interactive Python help…then go exploring following along with some of the instructions there!

Can you run Python entirely through the Terminal? Sure. But that'd be a bummer. Instead, we'll learn some more data scientist-friendly places to write, run and explore Python code, including in Py scripts, Jupyter Notebooks, and even in R Markdown.

2.4 RStudio

2.5 Jupyter Notebooks (??? not sure)

2.5.1 Py stuff to add here or somewhere:

Random things on Allison's to-add list

- Zero index (offset) differs from one index
- Open Python interpreter (run `$ python` in Terminal to return info about your Python version, and open the interpreter starting with `>>>`)
- Pressing 'Enter' twice: doesn't know if you want to add more (R assumes you don't want to), e.g. at the end of a for loop
- Save Python script (written in plain text file) save as test.py, go to that dir, run `$ python test.py` to actually run your python script (e.g. do this in RStudio to demo)
- Type, id, value (what kind of data, what it's called, the value compatible with the type (ref Lubanovic))

Chapter 3

Data representation, types and structures in R & Python

How we work with data depends largely on the *type* and *structure* of data we're working with. That's more than just "is this a number or letters?" We need to understand how the data are stored so that we know how to access pieces of it, how our code will understand and interact with data, and so we're correctly predicting how data will change based on what we do with it.

In this chapter, we'll learn about how data is represented, different types and structures of data, and some essentials of creating and working with them.

3.1 Data representation

Data representation is how data is stored and processed by a computer. In this section, we'll learn some basics of bits and bytes. Usually when working with data you won't even need to think about this - but when you need to, you'll want to have some basic understanding of how it works.

Terms:

- **data representation:** how data is stored and processed by a computer
- **bit:** the smallest unit of data - a single digit with value 0 or 1
- **byte:** a group of bits (usually 8) to represent a value

3.1.1 Bits. Bytes. Battlestar Galactica.

Really more appropriate for EDS 221 *Bits. Bytes. Binary data.* - all terms essential for understanding how data is stored and understood by computers.

3.1.1.1 Recap: the decimal system

We're used to thinking of numbers in a decimal system. For example, the number 247 has a 7 in the ones place (10^0), a 4 in the tens place (10^1), and a 2 in the hundreds place (10^2). This value is therefore $7\times10^0 + 4\times10^1 + 2\times10^2 = 247$.

3.1.1.2 Binary values

If we only have two numbers to work with - zero and one - how can we represent values? We'll use the *binary* system, where each digit (either a one or zero) - usually called a *bit* (for **Bi**nary digi**T**)- is multiplied by 2 raised to a power, depending on the position (where the right-most digit is 2^0). A bit (a single digit having a value of either zero or one) is the smallest possible unit of data to a computer, and sometimes called an "atom" of data.

For example, let's say we have a number in binary represented as `1101`. The actual value of that number is $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 13$. That means that the *largest* value that can be represented by four binary digits (four bits) is `1111`, which is $1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0 = 15$.

Understanding check: What values do the following represent?

- `0010` =
- `0110` =
- `1001` =

Question: What is the largest numeric value we could represent with 8 digits (8-bits = 1 byte!), using binary?

`1111 1111` $= 1\times2^7 + 1\times2^6 + 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 1\times2^1 + 1\times2^0$ $= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$

...and to represent larger values, we would need more bits. For example, you will usually hear 32-bit or 64-bit operating systems and CPUs (4 or 8 byte, respectively), indicating size of chunks of memory that the processor can handle.

Understanding check:

- How would you represent the value 803 in binary?
- How would you represent the value 48 in binary?

So we have an idea of how *values* are stored using only 0s and 1s. But there are many other types of data - text, images, maps, sound, etc. Given that data must be stored as 0s and 1s, how are those represented?

3.1.2 Representing non-integer and negative values

TODO

3.1.3 How are characters stored?

Characters are also represented with binary, but are associated with a known key that converts them to the correct letter or symbol. There are established keys to help us out, like ASCII (American Standard Code for Information Interchange), Unicode, and UTF-8 (Unicode Transformation Format).

Think of these as different menus to connect bytes to the characters they represent - dependent on what you're trying to do, a different menu may be a better option. For example, ASCII only encodes English letters and a small set of symbols, so for programmers creating content in different languages or with less common symbols, a different character menu might be a better choice.

Let's just consider one: ASCII. In ASCII, here are some decimal values of different capital letters:

Character	Decimal Value
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74

So, the word CAB is `67 65 66` in ASCII. Converted to binary, that is: `0100 0011 0100 0001 0100 0010` .

Understanding check: Decipher the following word (ASCII code) written in binary:

`0100 0010 0100 0101 0100 0001 0100 0011 0100 1000`

Check your answer here using this ASCII to binary converter! RapidTables text

to binary converter.

3.2 Data types

There are more types of data, but we'll focus on four: strings, logicals, integers, and doubles. Throughout the rest of the MEDS program you may use others, but they are uncommon in environmental data science. In this section we'll learn the common types of elements, then what they become once combined.

3.2.1 Types of data elements

When I say "elements" here, I mean single units of information. Like a single number, or identifier, or name. Like `12`, or `north` or `site_12a`. Here's some language to help distinguish those types of elements:

- **strings:** (also called: character string) elements that contain letters or symbols (anything non-numeric), or numbers that have been coerced to be understood as strings, like `"MEDS"`, `"Bren School"`, `"plot 17"`, and `"19.4"`. Note that the quotations around the last one is what makes it understood as a string, instead of a number.
- **logicals / booleans:** elements that indicate `TRUE` or `FALSE`. In R, the string `TRUE` / `1` are recognized as TRUE, and `FALSE` / `0` are recognized as FALSE. In Python, these are called logicals or booleans, and `True` / `1` are recognized as TRUE, and `False` / `0` are recognized as FALSE.
- **integers:** whole numbers (no decimal places), like `10`, `-65`, and `249`. In R, these are specified with a whole number followed immediately by `L`, e.g. `12L`
- **doubles / float:** numbers that can have decimal places, like `1.275`, `10.5`, and `0.068`. Double allows for twice the precision of a float. For very small or very large numbers, this designation might be important - or just err on the side of doubles.

3.2.2 Atomic vectors in R

Combining elements creates a vector (in R), similar to a list or single-dimensional array (often used in R and Python). We'll learn later on that *lists* are vectors that can contain different data types.

Given our common element types, what type of atomic vector is the outcome of their combination?

In R, there are four (common) types of atomic vectors: character (contains strings), logical (contains only logicals), and numeric (doubles and integers). Since the entire atomic vector can only be *one* of these, what is the hierarchy that determines the output vector type? See the schematic from Section 3.2 in Advanced R, which visualizes:

- A vector consisting of any combination of integers and doubles will be *numeric*
- A vector consisting of all strings, or any combination of strings and numeric values, will be *character*
- A vector with logicals and numeric values will be *numeric*
- A vector with logicals and strings will be *character*

3.2.2.1 Try it out

In R, we create a vector using `c()`, and return the class of an object using `class()`. For example, in the Console entering `class(c(1, 2, "cat"))` will return `character`.

Similarly, find the class of the following vectors:

- `c(3, TRUE, "Teddy")`
- `c(5L, 10.6, 281L)`
- `c("blue", "purple", 4.9)`
- `c(0.91, 0.36, 0.64, "missing")`
- `c("small", "medium", FALSE, "medium")`

OK but what if you **do** want to create a sequence of elements that maintain different types? Then you'll need a list, which you can create using the `list()` function:

```
taco_price <- list(1, 2.5, "free")
```

Notice in that example we *assign* the list to a name, `taco_price`, using `<-`. In R, we tend to use the `<-` assignment operator to name things, but in Python we use `=`.

For more information, read Chapter 3 in Advanced R by Hadley Wickham

3.2.3 Lists, tuples and dictionaries in Python

In the Python interpreter (Terminal > `python` to start), type `help()` to bring up the help documentation. From the list, you'll see that `TYPES` is an option. Check it out for lots of useful information. Here's a short version:

- **list:** a sequence of elements that you **can** reassign, made with square brackets and elements separated by commas (e.g. `vec = ["one",`

`'angry', "moose"]`), or `vec = [4, 10, "banana"]`. Note that strings can be created using either double or single quotes.

An example of list creation and reassignment in Python:

```
my_list = [1, 5, 8] # Create the list

my_list[1] = 9 # Changes the second element to value 9

my_list # See that the list is now [1, 9, 8]
```

The ability to change a list directly means that it is a **mutable** object. Some objects (like lists in Python) are mutable, while others (like tuples, introduced below) are not.

- **tuple:** a sequence of elements that you *can't* reassign (i.e. they are **immutable**), made with parentheses and elements separated by commas (e.g. `my_tuple = (1, 2, 10)`).

What does it mean that elements in a tuple *cannot* be reassigned? Try running the following and see what happens:

```
my_tuple = (19, 45, 218)

my_tuple[2] = 63
# TypeError: 'tuple' object does not support item assignment
```

- **dictionary:** an unordered collection of key-value pairs

A **dictionary** is created with squiggly brackets (`{}`), defininig elements separated by commas, each containing a key (name of an element that you can refer to later) and a value associated with it.

For example:

```
# Create the dictionary:
dog_weights = {'Khora':56, 'Teddy':49, 'Waffle':22}

# Return the dictionary:
print(dog_weights)
```

```
## {'Khora': 56, 'Teddy': 49, 'Waffle': 22}
```

How can we access information from a dictionary?

Since it's *unordered*, you can't use numeric indexing to access a specific element. For example, we might expect to use `dog_weights[0]` to access Khora's weight (remember - Python uses zero-index!). But when we try that...

```
dog_weights[0]
```

...an error is returned.

Instead, to pull out a specific element from a dictionary, we will use the key (in this case, the dog names). For example:

```
dog_weights['Khora']
```

```
## 56
```

Can you make something like a dictionary in R? The closest is to create a list where each element has a name. For example:

```
dog_food <- list(Khora = "bacon", Teddy = "chicken", Waffle = "pizza")

# Then return the value for Waffle:
dog_food['Waffle']
```

```
## $Waffle
## [1] "pizza"
```

3.2.4 Side-by-side comparison of data types in R & Python

- integer (both): numbers without decimals (whole numbers)
- float (both): numbers with decimals
- string / character
- Boolean / logical
- Lists / dictionaries (Py)?
- tuples v. lists v. vectors

See: https://r4ds.had.co.nz/vectors.html

3.3 Data structures

3.3.1 Vectors

3.3.2 Tibbles

3.3.3 Matrices

3.3.4 Lists

3.4 Allison random notes

See Lubanovic Ch. 2 for Python info

- Use `=` to assign values in Py, more often `<-` to assign in R

- Store some simple values in the Py interpreter (Terminal)
- Add a basic operators section? Only difference of meaningful note is ^ versus ** (Py exponential)

3.4.1 Converting between classes

Python:

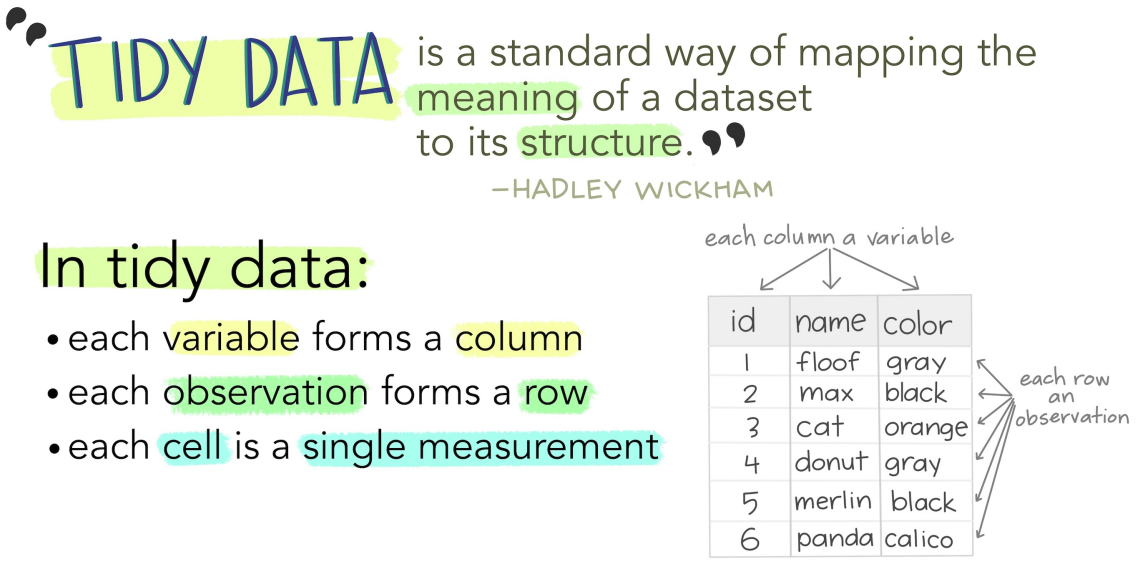
- Numeric to boolean: `bool()` anything non-zero is TRUE, anything 0 (or "False") is FALSE
- Numeric or boolean to integer: `int()` (lops off end, doesn't round)

Chapter 4

Tidy data

Note: all artwork in this chapter are from an illustrated collaborative Openscapes blog post by Dr. Julia Lowndes and Dr. Allison Horst (Lowndes and Horst, 2020), and many of the ideas here are building upon the main points in that post.

Tidy data is a predictable way to organize data that makes it more coder and collaborator friendly. As described by Hadley Wickham, **in *tidy data* each column is a variable, each row is an observation, and each cell contains a single value (measurement)** (Wickham, 2014).



Wickham, H. (2014). Tidy Data. Journal of Statistical Software *59* (10). DOI: 10.18637/jss.v059.i10

This may seem like a mundane topic, but tidy data provides a way of thinking about and organizing data that will become fundamental to how you input, wrangling, and work with environmental data - it becomes part of a systematic approach to working with data that **will make you a better data scientist and collaborator**.

4.1 Common ways data are untidy

One way to understand tidy data is to consider what makes some data sets *untidy*. Let's explore some examples of untidy data, and for each think about (1) why it's untidy, and (2) how we would wrangle it to make it tidy data.

4.1.1 Untidy example 1: A single variable across multiple columns

One of the most common ways that data can be untidy is if a single variable is broken up by group across multiple columns. For example, the following data contains the weights of three dogs, measured over four years:

In this example, there are really only 3 variables: dog name, dog weight, and year. But as organized, there are **5** columns - this should be our first indication that the data is not tidy. Instead of each variable occupying its own

Table 4.1: Dog weight (pounds) in untidy format, where a single variable (weight) is spread out across different levels of the year variable.

dog	2018	2019	2020	2021
Teddy	36.4	39.2	44.8	47.5
Khora	41.6	48.3	52.9	50.1
Waffle	NA	NA	20.4	23.7

Table 4.2: Dog weight (pounds) in tidy format, where each variable is in its own column.

dog	year	weight
Teddy	2018	36.4
Teddy	2019	39.2
Teddy	2020	44.8
Teddy	2021	47.5
Khora	2018	41.6
Khora	2019	48.3
Khora	2020	52.9
Khora	2021	50.1
Waffle	2018	NA
Waffle	2019	NA
Waffle	2020	20.4
Waffle	2021	23.7

column, the **weight** measurements have been split up across multiple columns, separated by the different levels of **year**. Sometimes you will hear this called "wide format" when a single variable is spread across multiple columns.

What would this data look like if it were tidy?

To be in tidy data, each variable (**dog**, **weight**, and **year**) should have its own column. In this example, starting from the wide format data we need to reshape **weight** observations into a single column. Year will need to populate a new column, with year values repeated as necessary to align with the long-format weights. We'll also need to repeat the dog names to accommodate the number of observations for each.

Later on, we'll hear how to reshape data from wide-to-long format (e.g. using `tidyr::pivot_longer()` in R), but for now think about the tidy format of the same data, shown below:

Table 4.3: Car descriptions in untidy format.

type	color	condition
1994 Toyota Corolla	silver	poor
2005 Subaru Outback	green	average
1977 Datsun 710	blue	excellent

Table 4.4: Car descriptions in tidy format.

year	make	model	color	condition
1994	Toyota	Corolla	silver	poor
2005	Subaru	Outback	green	average
1977	Datsun	710	blue	excellent

4.1.2 Untidy example 2: multiple values in a single cell

Another way that data can be untidy is if there are multiple "measurements" (or values) in a single cell. Keep in mind that a "value" doesn't have to be numeric - it's just a measurement or description for a recorded variable.

Sometimes raw data will contain multiple values in a single cell. For example, here we see that the make, model and year of cars are all in a single column called **type**:

An important thing is to be future-thinking about data, and expect that **even if you don't think a specific question is important now, it may be important in the future** – and having data in tidy format will make it easier to answer a wider range of questions with limited frustration. For example, maybe in the future (and if this were part of a larger data set) we would want to assess the condition of cars by year, or the color of cars by make and model. No matter how you slice those questions, having each variable in its own column will make them easier to explore and answer with code.

In the future, you'll learn how to separate components of a single column into multiple columns (e.g. using the `tidyr::separate()` function), which in this example would help to create a tidy version of the data that looks like this:

Untidy example 3: multiple observations in a single row

Occasionally, you will see environmental data where information for *multiple observations are stored in a single row.* For example, this is common when research divers are estimating numbers of a certain species within different size bins. For example, a dive record may contain information like this:

Table 4.5: Spiny lobster counts by size.

species	size_cm	count
spiny lobster	4.5	2
spiny lobster	5.0	4
spiny lobster	5.5	0
spiny lobster	6.0	1
spiny lobster	6.5	3

species	size_cm
spiny lobster	4.5
spiny lobster	4.5
spiny lobster	5.0
spiny lobster	5.0
spiny lobster	5.0
spiny lobster	5.0
spiny lobster	6.0
spiny lobster	6.5
spiny lobster	6.5
spiny lobster	6.5

So in this case, we have multiple lobster observations occupying single rows (e.g. the second row actually contains data for four lobsters). On the spectrum of untidy data, this isn't too bad - but it can make it much easier (and less risky) to visualize and analyze the data if each observation is in its own row. We'll learn how to convert a **frequency table** (like this one, which contains counts) into **case format** (which does have a single row per observation, so that the data look something like this:

Now, each individual lobster occupies its own row, and the data are in tidy format.

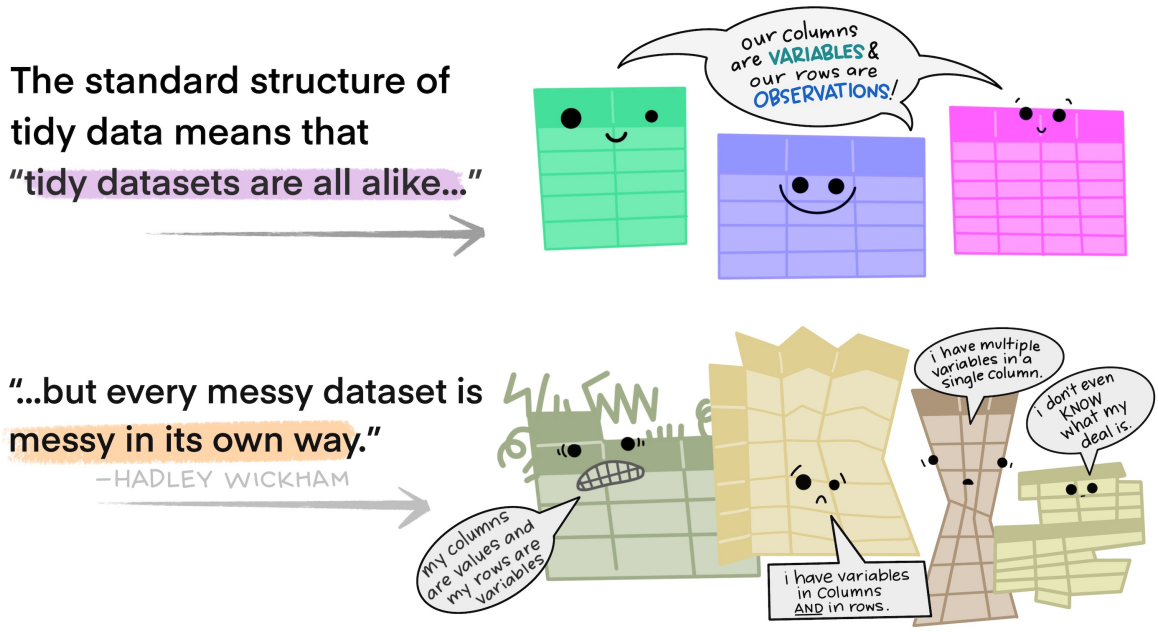
4.2 Tidy data for predictable inputs

Tidy data structure will become so ingrained into your brain over the next year that you'll almost forget a universe when you didn't automatically strive to create tidy data. Estimates for the fraction of a data scientist's time spent wrangling and cleaning data (rather than analyzing / modeling / gaining valuable insights from data) ranges from ~45 - 80%. That should make us especially inspired to create the data we hope to see in the world. In other words, on the data collection end of data science we should strive to input data in an organized, predictable format (e.g. as tidy data), and as cleanly as possible (i.e. consistent, machine-friendly values).

Question: Are there times when tidy data structure is *not* the best format?

Answer: Yup. There are very few "rules" in data science, and there will certainly be exceptions.

Tidy data gives us a predictable, organized data structure to strive for (unless we have good reason not to). Collecting data in tidy format from the get-go can reduce the time spent data wrangling / reshaping, and increase the time we can spend gaining insights to help solve environmental problems. And since all tidy data are similar in important ways, we can use similar tools in similar ways across different datasets, instead of hacking together new tools for each unique untidy dataset.



An evergreen paper to learn about data organization - in spreadsheets, and anywhere else - is *Data organization in spreadsheets* by Broman and Woo (2018). This is required reading for EDS 221, and beautifully highlights a number of other important considerations for data organization. I recommend you re-read it every year for the rest of your data science life.

4.3 Tidy data for easier analysis

The process of creating tidy data is useful in an of itself, because it requires us to be deliberate and thoughtful about how we structure our data, and makes us define our *variables*, *observations* and *measurements*. We will learn why that benefits us and our collaborators in the next section. Here, let's learn why tidy data is often code- and coder-friendly organization.

4.3.1 Code working for you

One (of many) benefits of working with tidy data is that it can save you tedious and sometimes dangerous manual subsetting and calculations for different groups. That's because in most programming languages, there are functions that can recognize different groups existing withing a variable automatically, and you can then perform operations (e.g. to calculate means, create exploratory plots, etc.) by those auto-recognized groups.

What?

OK let's break it down a bit, by considering the different ways we could tackle a summary statistics challenge.

Here is our mock data for dog hiking mileage and duration:

name	miles	time
Teddy	6.8	35:02
Khora	10.3	61:47
Teddy	3.1	15:28
Teddy	12.0	87:09
Khora	4.9	20:56
Teddy	8.5	63:34
Khora	2.7	12:01
Teddy	5.9	43:40

If we want to calculate the mean miles *for each dog*, how might we go about doing that?

1. We could manually create individual subsets for each dog (i.e., separate the two groups manually), then calculate the mean for each. That might not seem too bad for just two groups, but what if our data had 5 different dogs? Or 20 dogs? Or 1,000 dogs? Then creating and storing subsets manually before applying a function across all of them quickly becomes a big task.

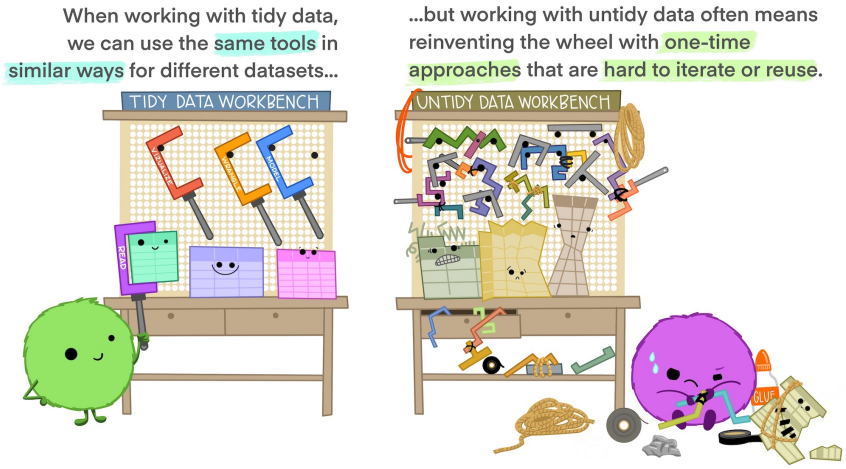
2. **Or instead**, we could expect code exists that could automatically recognize then create groups within the `name` column, and apply a function to each group. The second is more efficient - instead of actually creating separate subsets from the data, we just have our code recognize groups automatically, then apply a function (or functions) to each group, without ever separating the data in the first place. For simplest execution, it does require that your group values (here, dog names) are in a single column, which is one reason why organizing data in tidy format is useful for more efficient coding.

This isn't just true for finding summary statistics by group. We'll see that we can use groups within a variable to automatically facet graphs (present each group's data in its own plot), e.g. with `facet_wrap()` for data visualization with `ggplot2`, along with other common use cases for grouping data.

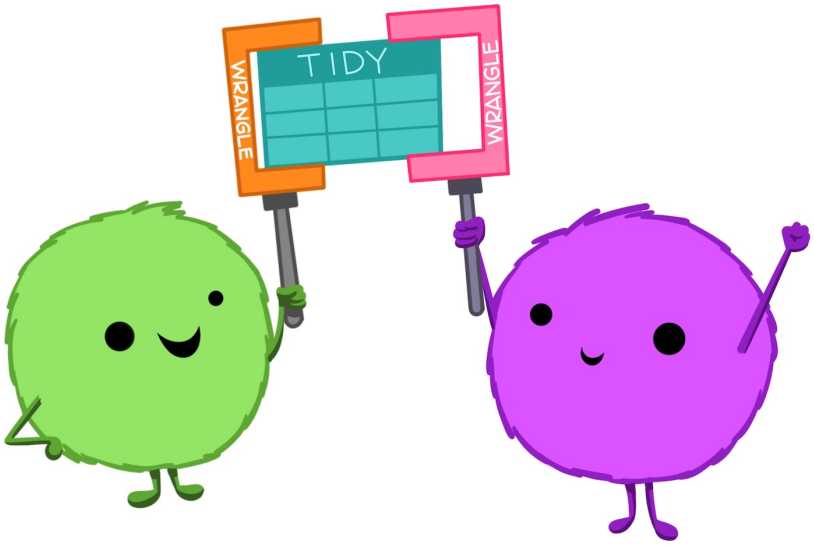
4.4 Tidy data for skills transfer & collaboration

Because tidy datasets are fundamentally and predictably similar, it will make it easier for you and your collaborators to repurpose existing tools & skills to work with new tidy data.

In other words, the data will differ, but the structure will be similar enough that we shouldn't feel like we need to reinvent the wheel every time we get a new tidy dataset.

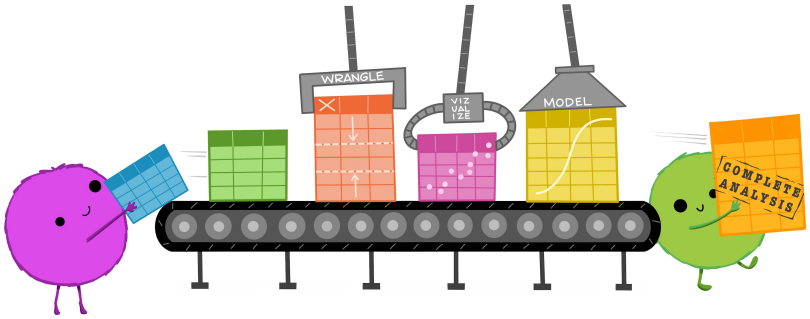


Not only will tidy data allow you to use similar tools on different datasets, but it also allows *different people* to use similar tools on the same dataset.



4.5 Tidy data for automation & repetition

The predictable and consistent structure of tidy data also helps to facilitate automation using pre-built workflows and algorithms. For example, you might imagine a report that needs to be updated annually as new data comes in. If the raw data are in the same structure as those analyzed and reported on for previous years, then you are more likely to be able to fully or partially reuse code from your previous analyses to prepare your new report.



So think carefully about how you organize your data - whether you're collecting raw data from the field, or trying to clean up data from an external source for easier, more collaborative, and more automated processing. Make the tidy data you hope to see in the world - you, future you, and all of your collaborators will thank you for it.

Chapter 5

Basic wrangling

This chapter just scrapes the surface of data wrangling tools you'll want to become familiar with as an environmental data scientist.

Subsetting, joins, pivots, and working with strings

5.1 Subsetting data

5.1.1 Subset rows based on conditions

5.1.2 Keep, omit, relocate, and rename columns (variables)

5.2 Joining data

5.3 Reshaping data

- Long versus wide data
- `pivot_longer` and `pivot_wider`

5.4 Working with strings

Chapter 6

Logical operators

Chapter 7

Conditionals

7.1 if else statements

7.2 if else if else statements

7.3 while statements

Chapter 8

Iteration

From Miriam-Webster Dictionary:

"**Iteration:** (*noun*) the action or a process of iterating or repeating, such as:

- a procedure in which repetition of a sequence of operations yields results successively closer to a desired result
- the repetition of a sequence of computer instructions a specified number of times or until a condition is met"

8.1 Iteration in programming

In programming, iteration is repeating instructions. Usually it's to spare yourself from having to manually do a repetitious thing. Well-written iteration can also make code more readable, usable, and efficient (definitely to write, sometimes to run).

For example, let's consider a few scenarios that may prompt you to use iteration:

- Your data contains 382 columns (variables), and you want to find the mean and standard deviation for each variable
- You have 250 csv files and you want to read them all in and combine them into a single data frame
- In a single data frame you have annual observations for fish passage from 1970 - 2019 recorded at 25 dams in Oregon, and you want to create and save a single graph for passage at each dam

...so basically, anything where you're like "Welp, I guess I'm going to be doing the same thing over and over and over and over..." should inspire you to consider iteration.

8.1.1 A real-world example of iteration in environmental data science

8.2 Generic for loop anatomy

When we iterate in code, most often that means we're writing some version of a for loop, which we can read as "For these elements in this thing, do this thing to each and return the output, then move on to the next element until you reach the end or a stopping point." There are a bunch of variations on that, but that's the overarching idea.

For example, in the image below our vector is a parade of friendly monsters getting passed through a for loop. There are conditions within the for loop dictating which type of accessory each monster will get, based on their shape. Then the outcome is returned with `print()`.

In words, how can we describe what's happening in this for loop? As each monster is passed individually through the loop, **if it is a triangle**, then it gets sunglasses added to it – that's why the first element in the output vector is a triangle monster with sunglasses. Then we move on to the second (orange) monster. Since they are also a triangle, they're assigned sunglasses. However, when we get to the **third** (purple) monster, it is **not** a triangle, and anything monster shape other than a triangle is assigned a **hat** - so we see the third output is the purple circle monster with a hat.

...and so on until all elements have been passed through the for loop or a stopping point is otherwise reached.

8.2.1 Anatomy of a for loop

8.3 Basic for loops in R and Python

Let's take a look at some basic for loops, written in both R and Python.

8.3.0.1 Example: A vector of very good dogs

Here's our scenario: starting with a vector of dog names "Teddy", "Khora", and "Waffle", write a for loop that returns the statement "[dog name here] is a very good dog!"

In R:

```
# Create the vector of names:
dog_names <- c("Teddy", "Khora", "Waffle")
```

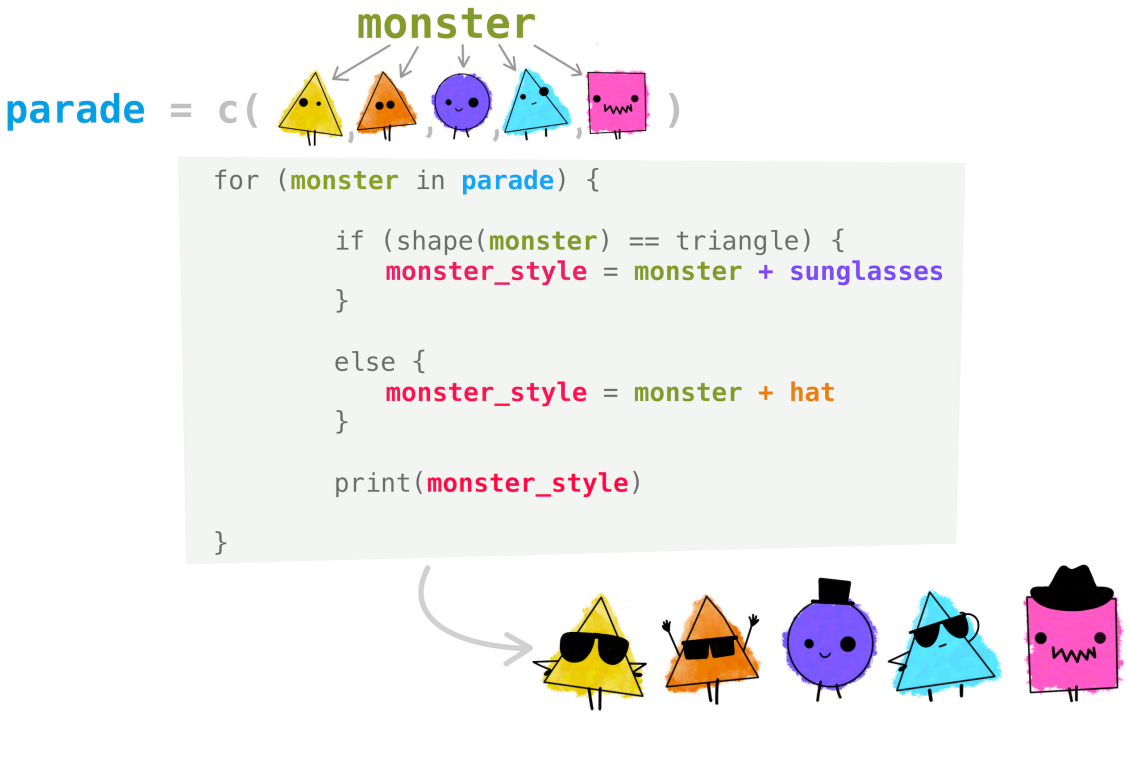


Figure 8.1: Monsters passing through a for loop, getting assigned sunglasses or a hat based on their shape.

```r
# Run it through the loop:
for (i in dog_names) {
  print(paste(i, "is a very good dog!"))
}
```

```
## [1] "Teddy is a very good dog!"
## [1] "Khora is a very good dog!"
## [1] "Waffle is a very good dog!"
```

In Python:

```python
# Create the vector of names:
dog_names = ['Teddy', 'Khora', 'Waffle']

# Run it through the for loop:
for i in dog_names:
    print(i + " is a very good dog!")
```

```
## Teddy is a very good dog!
## Khora is a very good dog!
## Waffle is a very good dog!
```

Note that there's nothing special about `i` here - that's just an identifier for "each element in this vector". It can be whatever object name you want, but make sure you're referring to the correct thing later on in the loop body. For example, that code could have been written in R as:

```r
for (treats in dog_names) {
  print(paste(treats, "is a very good dog!"))
}
```

8.3.0.2 Example: Hypotenuses!

For a vector of values `2, 3, 4, 5, 6, 7`, for any two sequential values, find the length of the hypotenuse if the two values are the lengths of sides of a right triangle. In other words, we'll find the hypotenuse length for right triangles with side lengths 2 & 3, 3 & 4, 4 & 5, etc.

Recall the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

In R:

```r
# Make the vector of values:
triangle_sides <- c(2, 3, 4, 5, 6, 7)
```

```r
# Create the loop to calculate the hypotenuses:
for (i in 1:(length(triangle_sides) - 1)) {
  hypotenuse = sqrt(triangle_sides[i]^2 + triangle_sides[i + 1]^2)
  print(hypotenuse)
}
```

```
## [1] 3.605551
## [1] 5
## [1] 6.403124
## [1] 7.81025
## [1] 9.219544
```

In Python:

Recall: Python indexing starts at ZERO (i.e., the first element in a vector is referenced with `vec[0]`), and the syntax for raising something to a power is `**` (e.g. `x**2`).

A weird one: the `range()` function in Python "…returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops **before** a specified number." So to create a sequence 0, 1, 2, 3, in Python you can use `range(4)`.

```python
# Import math library (sqrt() function is not native in Python):
import math

# Make the vector of values:
triangle_sides = [2, 3, 4, 5, 6, 7]

# Create the loop to calculate the hypotenuses:
index_no = range(0, len(triangle_sides) - 1)

for i in index_no:
  hypotenuse = math.sqrt(triangle_sides[i]**2 + triangle_sides[i + 1]**2)
  print(hypotenuse)
```

```
## 3.605551275463989
## 5.0
## 6.4031242374328485
## 7.810249675906654
## 9.219544457292887
```

What does that `index_no` vector contain? It's a sequence starting at 0 and increasing by 1 (the default increment) to a value below `len(triangle_sides) - 1`. Since the length of the `triangle_sides` vector is 6, that value is 5…and the vector continues to the value before the end value in `range()`. Frankly, it seems very weird to me, but that's what it's doing.

```
demo_vec = range(4)
for i in demo_vec:
  print(i)
```

```
## 0
## 1
## 2
## 3
```

8.4 Iteration with conditions

In the examples of for loops so far, we did the same repeated thing to each element. Sometimes, however, we'll want to change what we do to an element based on some characteristic - like in the monster parade example above, where the looped assigned a different accessory based on the monster shape.

We can add conditions within a for loop to specify **what thing** we want to to elements based on **some condition** we set.

The general anatomy of that process looks like this:

[ANATOMY OF A FOR LOOP WITH CONDITIONS]

Let's consider some examples.

8.4.0.1 Example: Feed the pets.

Given our vector of 3 pets below, write a for loop that returns "kibble" if it is a dog, but "alfalfa" if it is a horse.

The pets are: `dog, horse, dog`

In R:

```
# Make the vector of pets:
pet_type <- c("dog", "horse", "dog")

# Run it through the for loop:
for (i in pet_type) {

  if(i == "dog") {
    print("kibble")
  }

  else {
    print("alfalfa")
```

```
  }
}
```

```
## [1] "kibble"
## [1] "alfalfa"
## [1] "kibble"
```

In Python:

```
# Make the vector of pets:
pet_type = ['dog', 'horse', 'dog']

# Run it through the for loop:
for i in pet_type:
  if i == 'dog':
    print("kibble")
  else:
    print("alfalfa")
```

```
## kibble
## alfalfa
## kibble
```

8.4.0.2 Example: Bins

For a vector of values (`2, 6, 1, 18`), if the value is four or less 5, return "low", if the value is great than four and less than 12, return "moderate", and if the value is greater than or equal to 12 return "high".

In R:

```
# Make the vector:
numbers <- c(2, 6, 1, 18)

# Run it through the loop with conditions:

for (i in numbers) {

  if (i <= 4) {
    print("low")
  }

  else if (i > 4 & i < 12) {
    print("moderate")
  }

  else {
```

```
    print("high")
  }

}
```

```
## [1] "low"
## [1] "moderate"
## [1] "low"
## [1] "high"
```

In Python:

Note that we use `elif` for the *else-if* statement in the body.

```
numbers = [2, 6, 1, 18]

for i in r.numbers:
  if i <= 4:
    print("low")
  elif 12 > i >= 4:
    print("moderate")
  else:
    print("high")
```

```
## low
## moderate
## low
## high
```

8.5 While loop

A *while loop* will execute a command (or set of commands) as long as a condition is true. Once the condition is *not true*, the loop is exited.

While loop in R:

```
# Initiate
i <- 0

# Create a while loop that exits once i is NOT less than 5
while (i < 5) {
  print(i)
  i = i + 1
}
```

```
## [1] 0
## [1] 1
```

```
## [1] 2
## [1] 3
## [1] 4
```

While loop in Python:

```
i = 0

while i < 5:
  print(i)
  i = i + 1
```

```
## 0
## 1
## 2
## 3
## 4
```

8.5.1 A while loop break statement

Using a *break statement*, we can write a while loop that is exited if the break condition occurs, *even if the while condition is still true.*

8.6 R / Python side-by-side comparisons

Example

R

Python

Basic for loop

```
for (i in vec) {
```

```
print(i)
```

```
}
```

```
for i in vec:
```

```
print(i)
```

For loop with condition

```
for (i in vec) {
```

```
if i > 5 {
```

```
print("this")
```

```
}
else {
print("that")
}
}
```

python example

Chapter 9

Functions

Writing functions to implement algorithms is a fundamental skill for every environmental data scientist. Functions can reduce repetition, increase efficiency and elegance, and facilitate reuse and sharing. Functions built by other developers will be ingrained into your code, but it's also critical that you can build, test, document, and share **your own** functions.

This chapter covers:

- Function structure
- Writing basic functions
- Nested functions
- Functions with iteration and conditions
- Useful function features
- Testing
- Documentation
- Applied examples

9.1 Function components

At the most basic level, a function takes an input, does something to it (a calculation, transformation, etc.), and returns an output.

For example, we can write a function that doubles the input value. In *function notation* seen in math, that would be:

$$f(x) = 2x$$

where x is the input, and $f(x)$ is the output. The function f acts on input x by doubling the input value.

How can we create a function to do the same thing in R? An R function would look like this:

```
double_it <- function(x) {
  2*x
}
```

What are these different pieces of that function?

- **function name**: Here, the function is named `double_it`
- **formals**: The `function(x)` piece defines the function *formals* (arguments / parameters). This function expects a single input argument, `x` (you can check what the formals are using `formals(function_name)`).
- **body**: here, `{ 2*x }` is the body of the function - that's where we tell it what to do with the inputs. Note the braces (i.e. squiggly brackets) are often on separate lines from the algorithm itself.

Try out the function by inputting both a single value, and a vector of values. Note that vectorization is the default - meaning that the function is applied to each element in a vector.

```
double_it(x = 20)
```

```
## [1] 40
```

```
vec <- c(2, 4, 50) # Create a vector with multiple values

double_it(vec) # Function acts on each element in the vector
```

```
## [1]   4   8 100
```

Those are the main pieces. But don't worry, it gets a lot more interesting. Let's start by writing a few of our own functions.

9.1.1 A note on names

It's important to be thoughtful when naming functions. We generally want to follow standard practices for good names (concise, descriptive, code and coder-friendly), but you may also consider the following: - Start with a verb that describes what the function *does* (e.g. `sort`, `build`, `predict`) - End with a noun describing the thing it works with or creates (e.g. `image`, `model`, `mass`) - Combine them with a coder-friendly case (like `lower_snake_case`)

For example, here are some suggestions that may be useful function names:

`` `sum_imports` ``, `` `predict_offsets` ``, `` `plot_simulations` ``

In contrast, here are some function names that may be less useful, memorable, and intuitive for you and collaborators:

`` `fun_1` ``, `` `calc` ``, `` `x2` ``

Table 9.1: Parameter estimates for selected Hawaiian fish from Peyton et al. (2015).

Scientific name	Common name	a	b
Chanos chanos	Milkfish	0.0905	2.52
Sphyraena barracuda	Great barracuda	0.0181	3.27
Caranx ignobilis	Giant trevally	0.0353	3.05

It is likely that there will be a tradeoff between conciseness and descriptiveness. While there aren't *rules* about naming functions, I recommend erring on the side of descriptiveness to make reading and writing code a bit more intuitive. With tab-completion, the decrease in efficiency is minimal.

9.2 Writing simple functions

Let's practice writing a few simple functions using established relationships in environmental science.

9.2.1 Example 1: Fish standard weight

"Standard weight" is how much we *expect* a fish to weigh, give the species and fish length, and the nonlinear relationship is given by:

$$W = aL^b$$

where L is total fish length (centimeters), W is the expected fish weight (grams), and a and b are species-dependent parameter values.

Write a function to calculate fish weight based on a, b, and fish length, then estimate the weight of several fish based on the following parameter estimates for Hawaiian fish from Peyton et al. (2016):

Function:

```
predict_weight <- function(a, length, b) {
  a*(length^b)
}
```

Using the function:

1. Estimate the mass of a 160 cm long great barracuda.
2. Estimate the mass of a 118 cm long milkfish.

Thinking ahead: Does this pass your smell test for a user-friendly and user-helpful function? How might we make this function simpler for a user? For example, maybe a user can input the *species*, and the parameters a and b can be correctly sourced from a table? We'll learn how to add this kind of functionality in upcoming sections.

9.2.2 Example 2: Wind turbines

The full power in wind hitting a turbine is:

$$P = 0.5\rho A v^3$$

where P is power in Watts (joules/second), ρ is the air density (kg/m^3), A is the area covered by the turbine blades (square meters), and v is the wind velocity (m/s).

However, the Betz Limit means that turbines can only collect ~60% of the total wind power, which updates the theoretical "collectable" power (before accounting for inefficiencies, losses, etc.) to:

$$P = 0.3\rho A v^3$$

Write a function to calculate *maximum collectable* wind power (Watts) by a turbine requiring three inputs:

- Air density (in kg/m^3)
- Rotor radius (in meters)
- Wind velocity (in m/s)

Write the function:

```
calc_windpower <- function(rho, radius, windspeed) {

  0.3*rho*pi*(radius^2)*(windspeed^3)

}
```

Can we clean this up a bit by calculating the area first, within the function? Sure!

```
calc_windpower <- function(rho, radius, windspeed) {

  # Calculate turbine area (meters squared):
  turbine_area = pi*(radius^2)

  # Calculate collectable power:
```

```
  0.3*rho*turbine_area*(windspeed^3)
}
```

Now let's use the function we've created.

The largest turbine in the world (as of March 2021) is the GE Haliade-X, an offshore turbine prototype in Rotterdam, the Netherlands, with a 220 meter rotor diameter.

Assuming a windspeed of 7.7 m/s (based on long-term averages for North sea North Sea platforms from Coelingh et al. (1998)) and an air density of 1.225 kg/m^3 (at sea level), estimate the wind power that can be collected.

```
calc_windpower(rho = 1.225, radius = 110, windspeed = 7.7) # Watts
```

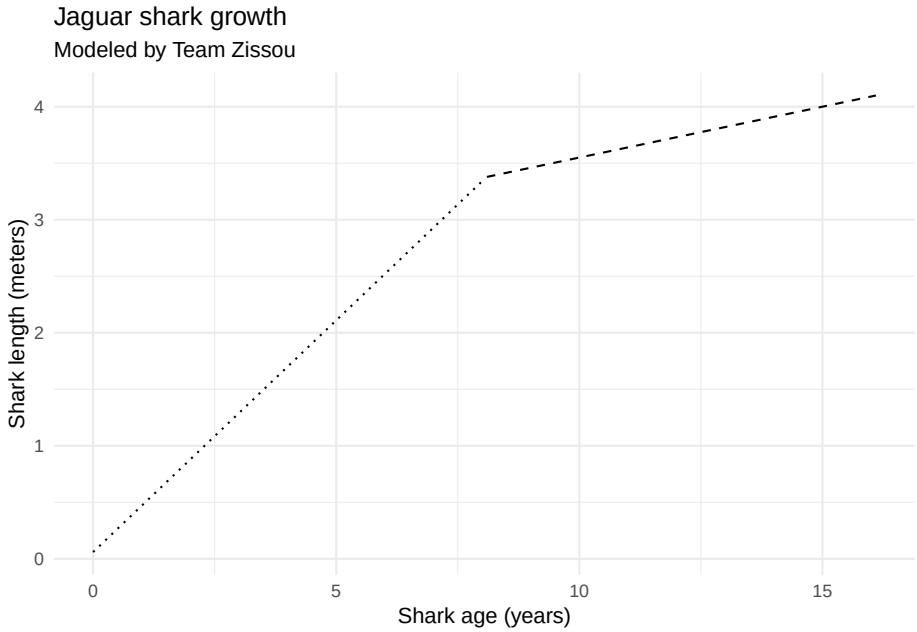
```
## [1] 6377710
```

9.3 Functions with conditionals

In the examples above, we change input values, but what the function *does* doesn't change based on those input values.

Sometimes, we'll want our function to do something different (e.g. a different calculation, use a different constant value) based on the input values.

For example, let's consider the following (made up) scenario: jaguar shark growth follows a linear bimodal pattern. From the age of 0 to 8 years, shark length (meters) is predicted by $length = 0.41(age) + 0.06$, where age is in years. When age is greater than 8 years, growth slows and is predicted by $length = 0.09(age) + 2.65$



Write a function that estimates mean shark length, based on age, using the two models shared above.

```
predict_sharklength <- function(age) {

  if (age <= 8)
    shark_length = 0.41 * age + 0.06

  else
    shark_length = 0.09 * age + 2.65

  print(shark_length)

}
```

We can then use the function to estimate the expected shark length for any age, using the correct model:

```
predict_sharklength(age = 4)
```

```
## [1] 1.7
```

```
predict_sharklength(age = 11.6)
```

```
## [1] 3.694
```

Quick check: For the example calculations above, do the predicted values align with the model visualization? Always do a quick check - even a back-of-

the-envelope calculation or visual check can catch a programming mistake!

9.4 Functions with iteration

9.5 Useful function features

9.6 Testing functions

9.7 Iterating functions

9.8 Resources on building, testing, & documenting functions

- Ch. 6 - Functions in *Advanced R* by Hadley Wickham

Chapter 10

Data wrangling with tidyverse and pandas

We've learned some strategies to subset, reshape and update data using base tools and functions. In this chapter, we'll use functions from R packages and Python libraries to wrangle our data in alternative ways that may be more efficient, intuitive and/or readable for collaborators.

Throughout this chapter, we'll see examples using the penguins data set in the palmerpenguins R package (make sure you have it installed and attached to follow along).

10.1 Meet the tidyverse and pandas

Names for things that seem weird now will be come so ingrained in your brain that you won't even realize the nonsense you're spewing to non-R or -Python users (we'll practice avoiding that throughout MEDS, but be warned...). A couple of big ones are **tidyverse** and **pandas** - what are they, and how are they useful for us?

10.1.1 R wrangling: `dplyr` & `tidyr` packages in the tidyverse

The tidyverse is a collection of R packages built to make data wrangling and visualizations easier for everyone. The packages are *opinionated* - meaning that they use similar syntax so that functions from different packages within the `tidyverse` play nicely together - especially useful once you get into burlier wrangling.

The two tidyverse packages that we'll focus on here are dplyr and tidyr. The `dplyr` package contains powerful and intuitive functions for wrangling data (subsetting, filtering, choosing or moving columns, finding summary statistics by group, etc.). The `tidyr` package contains functions useful for reshaping data into tidy structure (e.g. converting from long-to-wide or wide-to-long format so that each variable is a single column, each row is a single observation, and each cell contains a single value).

To get `dplyr` and `tidyr`, along with all other packages in the tidyverse, install the `tidyverse` by running `install.packages("tidyverse")` in the R Console.

10.1.2 Python wrangling: the `pandas` library

The pandas library is a "fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language" (from https://pandas.pydata.org/). With tools for reading, wrangling, subsetting, and reshaping data, it is the go-to for data manipulation in Python.

If you already have Anaconda installed, you're all set! The `pandas` library comes along with it (in addition to a bunch of other libraries useful for data wrangling and analysis, like `NumPy` and `matplotlib`).

10.2 Selecting columns

Make a subset of *columns* (variables, if data are tidy) using:

- tidyverse: `dplyr::select()`
- pandas:

10.2.1 `dplyr::select()`

Use `dplyr::select()` in R to choose columns by numbered position or column name (recall: use `names()` to return a vector of all column names, or `View()` to bring up the data frame in a new viewing tab).

Example: Starting with the penguins data frame, create and store a subset that only contains variables `species` and `island`.

```
penguin_sp_isl <- penguins %>%
  select(species, island)

# Alternative (but I like the formatting above...):
penguin_sp_isl <- select(penguins, species, island)
```

Which return a data frame only containing those two variables. Let's check using `names()`:

```
names(penguin_sp_isl)
```

```
## [1] "species" "island"
```

10.2.2 Selecting columns in Python

We choose columns in Python using the syntax `my_df[['col_a', 'col_b']]`, which will select `col_a` and `col_b` from a pandas DataFrame stored as `my_df`.

In the Python example below, we store the `penguins` data frame as a Python DataFrame, then create a subset that only contains `species` and `island`:

```
# Store the existing R object `penguins` as a Python DataFrame:
py_penguins = r.penguins

# Then make a subset only containing `species` and `island`:
penguin_sp_isl = py_penguins[["species", "island"]]

# Check out the first few lines to see variables included:
penguin_sp_isl.head()
```

```
##   species     island
## 0  Adelie  Torgersen
## 1  Adelie  Torgersen
## 2  Adelie  Torgersen
## 3  Adelie  Torgersen
## 4  Adelie  Torgersen
```

10.3 Subsetting rows based on conditions

Creating subsets that match conditions you specify will absolutely be one of the most common data wrangling things you do.

10.3.1 `dplyr::filter()`

Create subsets of data based on…

10.3.2 Subsetting rows in Python

10.4 Adding columns

10.4.1 `dplyr::mutate()`

10.5 Finding grouped statistics

10.6 Joining data

Chapter 11

Troubleshooting

Chapter 12

Data visualization with `ggplot2` and `seaborn`

GET SEABORN WORKING

```
# import seaborn as sns
```

Visualizing data is a critical skill in every step of data science - from exploratory visualization to look for major patterns, groups or outliers in the data to preparing highly designed infographics for stakeholders, every data scientist must have technical and conceptual skills in data viz.

In this chapter, we'll learn about two powerful packages for data visualization:

- `ggplot2`: the go-to data visualization package in R (part of the tidyverse)
- `seaborn`: one of several popular data viz libraries in Python

12.1 The grammar of graphics

The "gg" in `ggplot2` is for the **grammar of graphics** - a way to describe parts of a graph (Wickham, 2010), which in turn provides us an organized and predictable plan to build data visualizations. Understanding the grammar of graphics, and some vocabulary, can help to demystify how we build visualizations with code and the role of some specific functions.

12.2 Basic anatomy of graphs with ggplot2 in R, and seaborn in Python

12.2.1 ggplot2 bare minimum

Making the most basic ggplot requires 3 pieces:

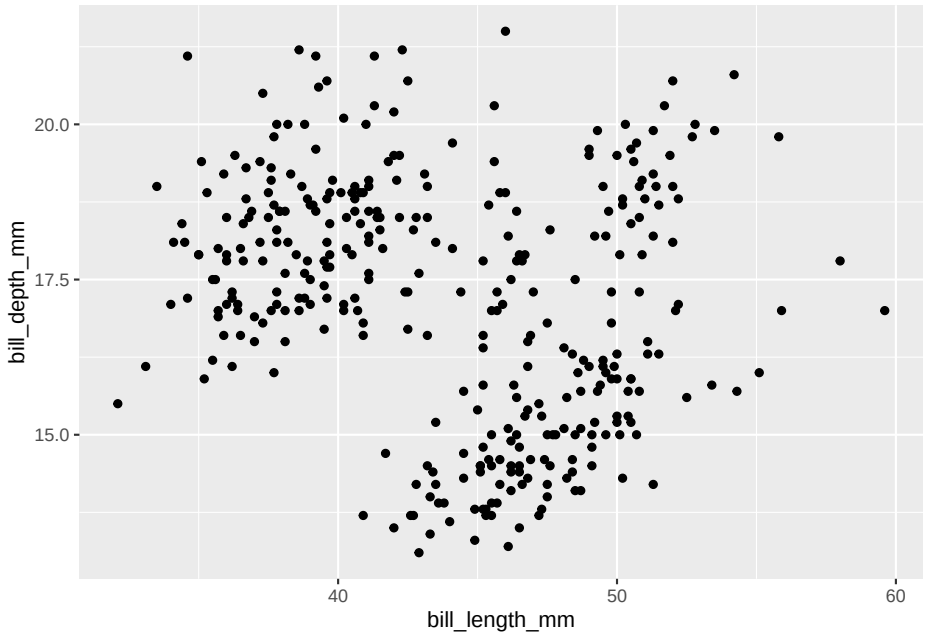
1. Tell R you're making a ggplot graph (`ggplot()`)
2. Tell it what data you're plotting (including what the x- and/or y-axis variables are)
3. Tell it what type of plot you want to make

Note: it is very important to carefully consider the type of data you're trying to visualize when making a graph. Some visualizations are not appropriate for some types of data. We'll discuss this a bit more throughout the class, but one of my favorite resources to consider appropriate visualizations based on data type is From Data to Viz.

For example, let's make a basic scatterplot of bill length versus bill depth from the `penguins` dataset (in the `palmerpenguins` package).

```
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point()
```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```



Let's break down how that code maps onto the three essential pieces of a ggplot

graph listed above.

1. We use `ggplot()` to let R know we're making a ggplot graph (note: the *package* name is `ggplot2`, but the function we use is just `ggplot()`).
2. Within the `ggplot()` function, we have two arguments: The first, `data = penguins`, specifies which object in our environment (in this case, a data frame called penguins) we will get data from for our plot. The second argument, `aes(x = bill_length_mm, y = bill_depth_mm)`, tells us which variables map on to the x- and y-axis aesthetics (hence the `aes()`, for aesthetics).
3. We then add a layer (note the `+` symbol*) containing the specific geometry, or type of graph we're trying to make. A standard scatterplot is made with `geom_point()`.

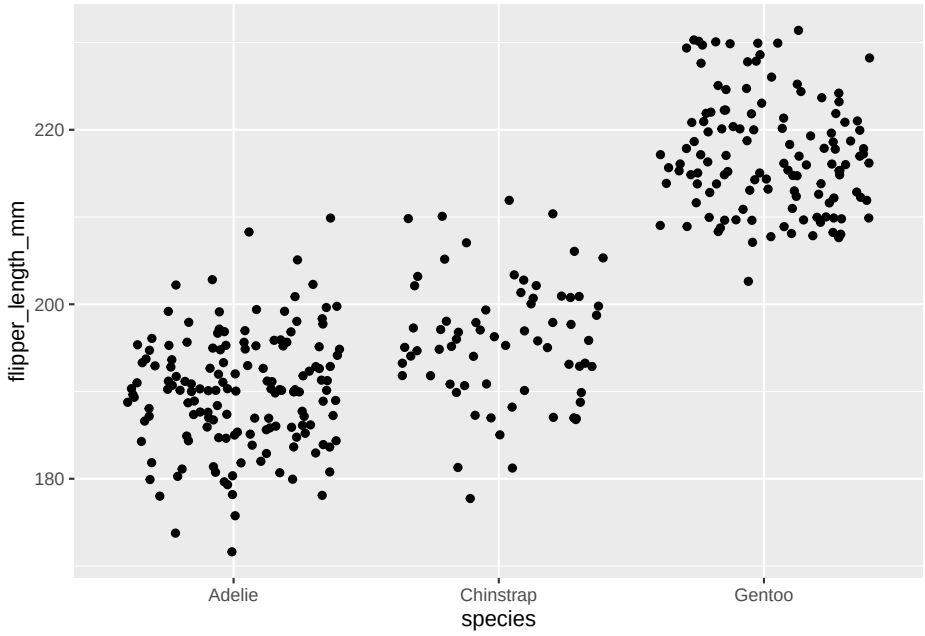
*Note: A common mistake is using the pipe operator (`%>%`) between ggplot layers instead of a `+` sign. Try to remember that we build a ggplot graph by adding layers to it piece by piece (`+`), not feeding graph pieces *into* a subsequent layer (which the `%>%` implies).

Let's make a few more bare minimum ggplots. For each, ask yourself: how does this example code map onto the three main pieces of a basic ggplot graph listed above?

Example: a jitterplot (`geom_jitter()`) of flipper length by penguin species.

```
ggplot(data = penguins, aes(x = species, y = flipper_length_mm)) +
  geom_jitter()
```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```



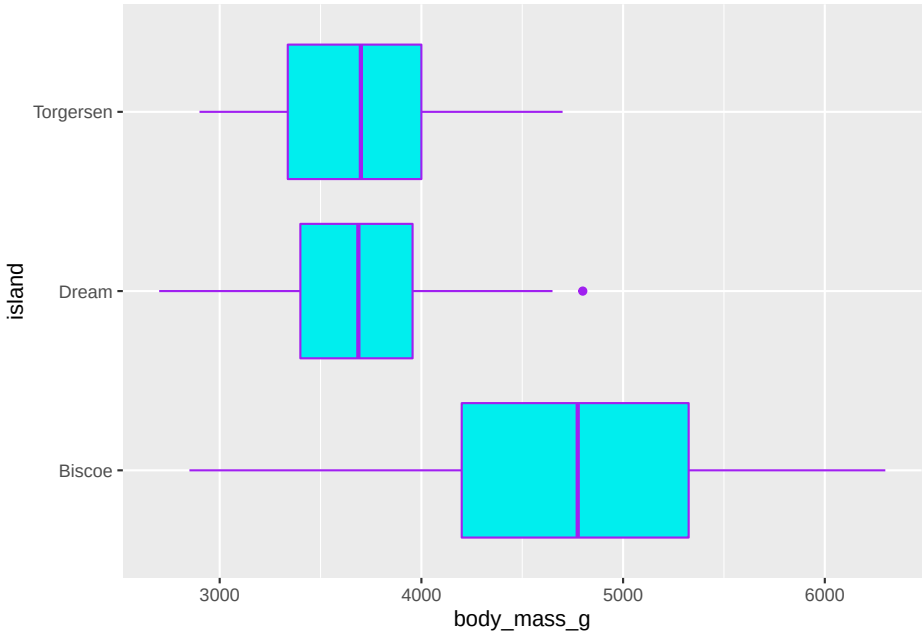
Example: A boxplot of penguin body mass by island.

Let's add some color, too, just for the halibut. Generally:

- Update line and point colors with `color =`
- Update shape fill colors with `fill =`

```
ggplot(data = penguins, aes(x = body_mass_g, y = island)) +
  geom_boxplot(color = "purple", fill = "cyan2")
```

```
## Warning: Removed 2 rows containing non-finite values (stat_boxplot).
```



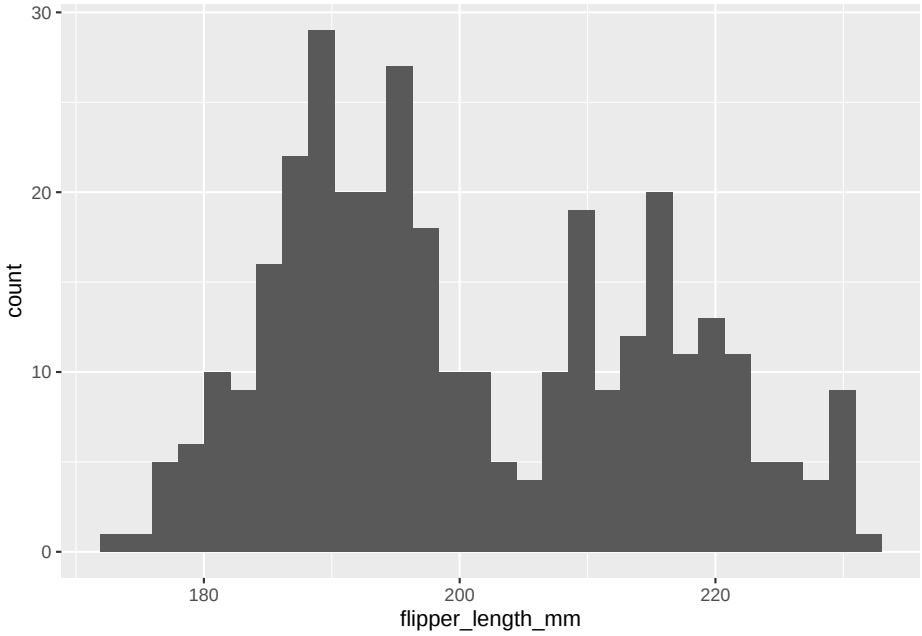
Critical thinking: What might a person unfamiliar with this data conclude if they only saw the boxplot above? Why should this concern us? What critical information is missing that is required for responsible consideration? See also: correlation is not causation, omitted variable bias.

Not all ggplot graph types require two variables. For example, a histogram only wants one numeric variable - it will find the frequencies (plotted on the y-axis) for you. For example, code to make a histogram of penguin flipper lengths is:

```
ggplot(data = penguins, aes(x = flipper_length_mm)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 2 rows containing non-finite values (stat_bin).
```



12.2.2 `seaborn` bare minimum

NOTE TO SELF: I used reticulate::py_install("seaborn") while in my project to get seaborn in the right place...

```
import seaborn as sns
```

12.3 Mapping variables onto graph aesthetics

In the examples above, we specified the x- and/or y- axis variables, but didn't update any other aesthetics that would indicate groupings or values within the data. For example, we may want to:

- Differ point color by penguin *species*
- Increase point size by penguin *bill depth*
- Change the boxplot fill color by *island*

When we want to change an aesthetic based on variable values (remember: when I say "values" here, that can mean characters like penguin species), we are *mapping a variable onto a graph aesthetic*. In other words, we are changing an aesthetic of the graph to depend on values within a variable.

12.3.1 Mapped variable aesthetics in ggplot2

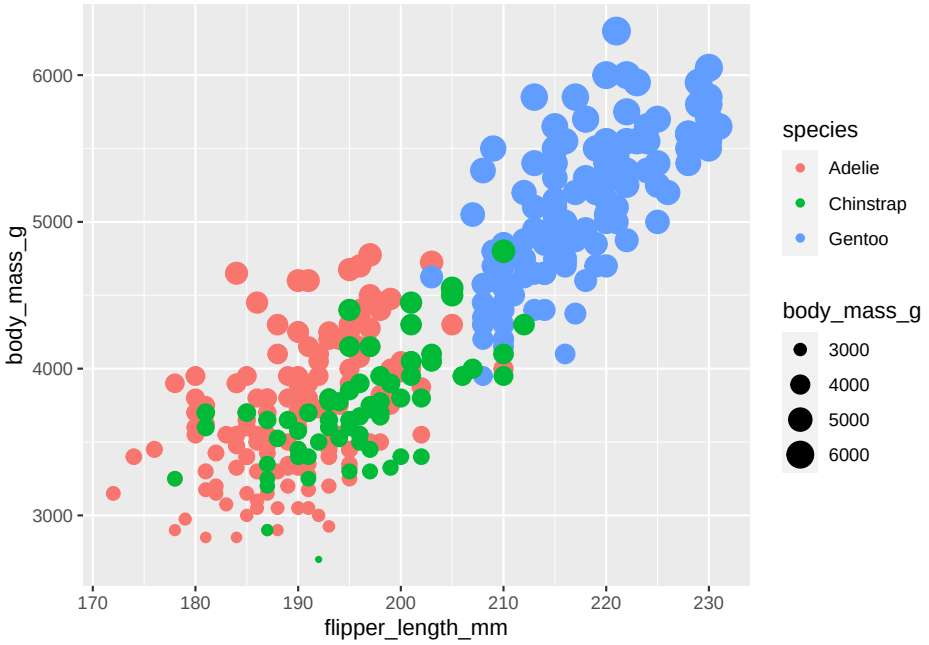
We map additional variables onto graph aesthetics with additional `aes()`.

Important: Any time you update a graph aesthetic based on a variable value, it will be within `aes()`. Any time you update a graph aesthetic based on a constant (e.g. `color = "blue"`), it should NOT be within `aes()`.

Example: Create a scatterplot of penguin flipper length versus body mass, with color indicating penguin *species* and with point size changing based on the body mass variable value (using the `size` element).

```
ggplot(data = penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(aes(color = species, size = body_mass_g))
```

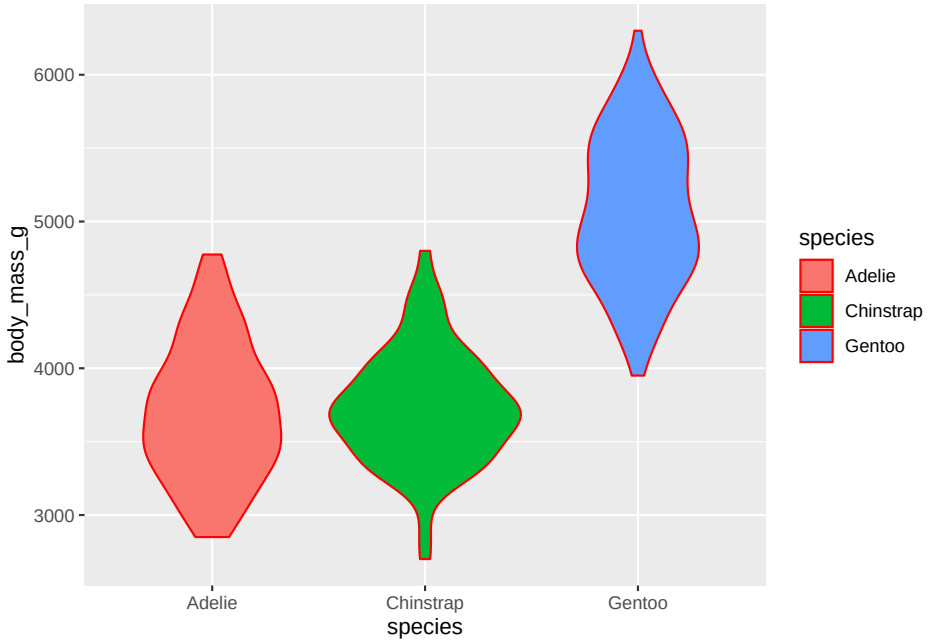
```
## Warning: Removed 2 rows containing missing values (geom_point).
```



Example: Create a violin plot of body mass by species, with the violin fill color dependent on species *and* the violin outline color set to "red".

```
ggplot(data = penguins, aes(x = species, y = body_mass_g)) +
  geom_violin(aes(fill = species), color = "red")
```

```
## Warning: Removed 2 rows containing non-finite values (stat_ydensity).
```



12.4 Essential customization

The section above introduces the bare minimum elements needed to create a graph. However, they are far from complete until we update some essential graph elements for clarity.

Those include:

- Update axis labels (include label and units)
- If necessary, add a title
- Change the theme

Chapter 13

Py-R differences

13.0.1 Overarching things

- zero indexing in Py (versus 1-index in R)
- ** instead of ^ for exponents
- elif versus else if
- type() versus class()

13.0.2 Strings differences:

- `+` to combine strings in Python
- `*` to duplicate a string in Python (e.g. versus rep() in R)
- `len()` versus `nchar()` for number of characters in a string
- df %>% function() versus df.function()
- replace() versus str_replace()
- strip() versus str_trim() or str_squish() to remove whitespace
- f-string versus glue or paste?

13.0.3 Number sequences:

- range() versus seq()

Bibliography

Broman, K. W. and Woo, K. H. (2018). Data Organization in Spreadsheets. *The American Statistician*, 72(1):2–10.

Coelingh, J., van Wijk, A., and Holtslag, A. (1998). Analysis of wind speed observations on the North Sea coast. *Journal of Wind Engineering and Industrial Aerodynamics*, 73(2):125–144.

Lowndes, J. S. S. and Horst, A. (2020). Tidy data for efficiency, reproducibility and collaboration.

Lubanovic, B. (2014). *Introducing Python: modern computing in simple packages*. O'Reilly Media, Sebastopol, CA, first edition edition. OCLC: ocn890938061.

Peyton, K. A., Sakihara, T. S., Nishiura, L. K., Shindo, T. T., Shimoda, T. E., Hau, S., Akiona, A., and Lorance, K. (2016). Length–weight relationships for common juvenile fishes and prey species in Hawaiian estuaries. *Journal of Applied Ichthyology*, 32(3):499–502.

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

Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, 59(10).

Wickham, H. (2019). *Advanced R*. Chapman and Hall/CRC, 2 edition.

Wikipedia contributors (2021). List of programming languages — Wikipedia, the free encyclopedia. [Online; accessed 29-March-2021].