Programming Logic for Non-Programmers

Kat Koziar and Stephanie Labou

2025-04-15

Contents

Chapter 1

About

```
BEGIN pitch()
```

Have you ever wondered how some of your colleagues can look at a computer programming script, with little prior knowledge of the language, and not only read it, but help fix the code? It's not because they know all programming languages, but because most programming languages use the same concepts and logic.

```
STRUCTURE(workshop)
```

In this interactive workshop, attendees will gain hands-on experience to understand and interpret programming logic. We will cover fundamental topics in programming including: conditional statements, loops, order of operations and logical flow, functions and arguments, and data types. Attendees will practice formulating programming arguments to accomplish common tasks, such as subsetting data based on a set of conditions.

```
WHERE prior_experience == FALSE
```

No coding experience required! Programming logic is transferable across specific languages, so learners will focus on concepts, rather than specific syntax from a specific language. Attendees will learn to interpret programming logic and build confidence to apply their understanding to various programming languages they may encounter.

```
FOR (x in example1:example5) {annotate(x)}
```

To provide real world examples of programming logic in practice, the workshop will integrate hands-on work time with examples of sample code written in R, Python, SQL, Stata, and other languages. Attendees will practice annotating code in human understandable language and discuss the process, and any pitfalls, with their peers and the instructors.

```
IF attendee_need == "learn_programming_logic": print("register
for this workshop!")
```

Chapter 2

Introduction

This is a different type of programming workshop. One that doesn't require a computer, but instead intends to help you build mental models of how computer programming works. You will learn the logic behind programming, and also methods for identifying errors in algorithms and code that the computer doesn't see. The only technology required, besides the ability to view this lesson, is something to write with and a piece of paper.

2.1 Building a Mental Model

Our experience with computational consultations is often student researchers will take someone else's code and try to adapt it for their own research, but they use the code without knowing how it does what it does. This means they're unable to easily update the script, will create errors they don't know how to address, and even import errors already in the script. Sometimes they can't even adapt it to use the original data which is in a different location on their computer. They can't really read the code, but are reliant on the computer reading the code.

The purpose of this workshop isn't just to introduce you to programming logic, it's to provide a safe space to practice thinking through what the computer does by tracing algorithms and code snippets instead of having the computer just do it.

2.2 A Note on Syntax

Syntax is the formal structure of a computer programming language. Assembly, C/C++, C#, Python, and R all have formal language structures so the

computer knows how to read the code. But, sometimes syntax gets in the way of learning concepts. Luckily, most programming concepts are the same across all languages.

The syntax we will use for most of the examples will be something called psuedocode. Pseudocode focuses on concepts and tasks, not syntax. You don't need to worry if there is a semicolon or parenthesis out of place when you write in psuedocode, you'll still know how to interpret what is written.

We will also introduce programming examples in different languages, so you can start to recognize the similarities and differences between the languages. You won't be an expert by end of this workshop, but we will help you build your skillset so hopefully you're more comfortable reading known (and unknown) computer languages.

Chapter 3

Algorithms

3.1 What Are Algorithms

Most people have heard the term algorithm, especially in relation to AI or social media feeds, but what is an algorithm?

An algorithm is a set of instructions for how to do something. For AI, the term *algorithm* is often used as a term to represent the overall way that whatever AI that you're working with was trained to do what it does. For social media feeds, an algorithm will determine how the platform selects which content is delivered to a user. Algorithms can be very simple or very complex.

A common analogy for an algorithm is a recipe. A recipe includes a set of ingredients, which are like the variables in your code, then lists instructions on what to do with the ingredients. But, a big difference between common recipes that you are used to seeing and an algorithm is you have to be very exact and specific when you tell a computer to do something. It isn't enough to tell a computer, "get the data." You have to tell it where the data is located, what format it is in, how to read it, and what format you want the data in for use in your script. So let's do an example with a recipe on how to make cheesy mashed potatoes.

The steps are easy enough:

1. Take potatoes and boil them until they're done.
2. Drain.
3. Add butter, milk, salt/pepper to taste.
4. Mash.
5. Fold in cheese.
6. Eat.

Most of us would know how to do this because we infer the instructions that are missing. But, computers aren't intelligent like humans. They need to be explicitly told how to do something.

How many potatoes? What does *boil* mean? What does *done* mean? (which is also a question I asked my mom when reading my grandmother's recipes.)

Computer code needs to be explicit and complete. Let's see what that looks like with the first part of an algorithm.

Algorithm for peeling potatoes for Cheesy Mashed Potatoes

1. Get three pounds of russet potatoes.
2. Get bowl large enough to hold three pounds of russet potatoes.
3. Get a vegetable peeler.
4. Take one potato.
5. Peel potato.
6. Rinse potato with water.
7. Place peeled potato in bowl
8. Repeat 4-7 until all potatoes are washed and peeled

When you start breaking down the process into specific actions, the instructions can get quite lengthy! But this is what we need to do in order to "instruct" the computer what we want to do: splitting a large complex overall task into simple tasks that can be done one action at a time.

Essentially, that's what computer processors do - one action at a time. Computers with multi-core processors can do multiple actions simultaneously, one action for each core. It's just the computer is so fast, it looks like it's doing everything at once, but it still executes commands step by step.

3.2 Breakout Activity

Timing: 15 minutes of breakout group work, followed by 10 minutes of full group discussion.

For this activity, your group will write an algorithm to make popcorn.

"Make popcorn" is something most people have at least a general understanding of how to do (even more so than making cheesy mashed potatoes) but remember: in this scenario, your group needs to provide step-by-step directions for a *computer* to understand how to make popcorn.

As you write out the steps with your group, consider:

- Are you opening a container at any point?

- Are you making microwave popcorn, or using a stove, and does that impact the directions?
- Is time needed to make popcorn consistent, or does this depend on other factors?
- Is there something the computer should do if the popcorn starts burning, or other "errors" arise?

Chapter 4

Data types

4.1 Variables

Now that we understand the structure and logic of algorithms, we can start on how the natural language of algorithms is translated into programming scripts. In order to do this, we need to introduce closely related concepts of variables, data types, and data structures.

A word that people use a lot when referring to computer code is *variable*, but what is a variable? A variable is a placeholder for some type of data that will be used in computer code.

```
ingredients <- cheddar cheese, potatoes, milk, salt, butter
```

We use variables to represent the data that are being used within a program because it's easy to change it,

```
ingredients <- smoked gouda cheese, potatoes, milk, salt, butter
```

and also easy to build a mental model of how the different variables interrelate.

```
ingredients <- smoked gouda cheese, potatoes, milk, salt, butter
ingredient_quantities <- all, 3 lb, 0.5 c, to taste, 8 tbsp
```

4.1.1 Pro-tip

A common rule to help decide what name to use for a variable is, the name should describe what the data represent. If we used the term `x` - which is commonly used as a variable in math equations - we wouldn't know what it represents. Using the word `ingredients` is descriptive, so you have an idea what the data are in the variable - it describes what the variable represents.

4.2 Data Types

As we already stated, variables hold some value. (I won't go into the details of *how* that happens on a computer, though if you're interested, I could tell you after the workshop, or at a separate time.) It's enough for us to know that whenever you see a variable, it correlates with some sort of data.

You know how before you can do math, you have to define numbers? Or, more commonly, before you can write and spell a word, you have to define the alphabet the word uses? Or, to use our recipe analogy, before you can cook, you have to define (or identify) food? Computer programming is the same way. Before you can use data, you have to define data types.

There are three data types which are ubiquitous across most programming languages.

The first two are related to **numbers**:

Integers (`int`) which are whole numbers: positive, negative, and zero.

Floats (`float`) or floating point numbers, which are real numbers, or numbers with decimals.

And the third is related to **words** and **text**.

Characters (`char`) which by themselves are a single character, but can be connected together to create a **string** (`str`), which is basically text.

Most data types specific to any particular language is a special case of one or more of these three data types (such as how a **string** is a special case of a **character**).

4.3 Data Structures

A data structure is a special framework which holds your data. Again, there are a lot of different types of data structures, but we'll describe some of the common ones.

A single value isn't strictly a data structure, but it's worth mentioning they exist.

A **list** is just that, a list of data elements. The nice thing about lists is it can contain a multitude of different data types, and also different data structures.

```
cheeses <- (8, smoked gouda, 0, milk, 16, cheddar, .25, limburger)
```

An **array** is a common data structure, which at its base is a single-dimension structure. You'll commonly see an array represented with square brackets.

You may encounter text elements in an array

`ingredients = [cheese, potatoes, milk, salt, butter]`

or an array of integers

`quantities <- [16, 32, 8, 0, 4]`

or floats

`quantities = [16, 32, 8, 0.125, 4]`

The data in all cells (or elements) of an array needs to be the same type, which is part of what makes an array different than a list. If an array is created with different data types, many computer languages, such as R or Python, will automatically convert the data to a single data type. This is formally called *coercion*. Numeric values with text will be coerced to all text, and integers with real numbers to floats, but not the other way around.

You may also see this data structure called a **vector** *(which is different than vectors used in math or physics).*

A **matrix** is a multi-dimensional array and has the same restrictions as an array, in terms of same data type used throughout. Honestly, it depends on who you're talking to for how they refer to arrays and matrices, and it's likely highly dependent on the type of math they're using in their research.

A matrix will have both rows and columns. The number of elements in each row needs to be the same, and the same with the number of elements in each column.

Below is a matrix with size i by j, which means that there are i number of rows, and j number of columns. The matrix below also demonstrates what is called an *index* which is basically the address of any particular element in the matrix.

$$\begin{bmatrix} index_{1,1} & index_{1,2} & \cdots & index_{1,j} \\ index_{2,1} & index_{2,2} & \cdots & index_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ index_{i,1} & index_{i,2} & \cdots & index_{i,j} \end{bmatrix} \tag{4.1}$$

Using an index for a one-dimensional array is pretty easy.

for `ingredients <- [cheese, potatoes, milk, salt, butter]`

the value in `ingredients[1]` is `cheese`

if we expand it into a matrix

$$ingredients = \begin{bmatrix} potatoes & milk & heavy\ cream & butter \\ Cheddar & Gouda & Parmesan & Muenster \\ salt & rosemary & thyme & sage \end{bmatrix} \tag{4.2}$$

the value in `ingredients[2,4]` is $Muenster$

I want to share that I've been programming for a long time, and have done all of the math, and I *still* will have to double-check that the index I'm referring to in multidimensional arrays or matrices is listed in the correct order.

A **dataframe** is a tabular form with rows and columns, much like a matrix. However, dataframes can contain mixed data types, for instance a column of integers, a column of strings, and another column of floats. Dataframes can also have column names and row names.

ingredients	quantities	importance
smoked gouda cheese	all	5
potatoes	3 lb	1
milk	0.5 c	3
salt	to taste	4
butter	8 tbsp	2

4.3.1 To make things more confusing: A note on indices

A fun *"feature"* of programming languages is different languages will start their index with different numbers, either 1 or 0. Python is a zero-indexed language, meaning the index of the first element of an array or list is 0, while R is a normal language, I mean, not a zero-indexed language, meaning the index of the first element of an array in R is 1.

I like to pretend that this is an extension of mathematicians having arguments over which is the first number, zero or one (yes, I've seen these debates). But, I think it has to do with ease of memory management in computers, which is something most researchers who come to us for consultations will not have questions on. So right now, it's just a quirk.

For our workshop, we will use 1 as the first element of an array, because that's easier to learn logically.

4.4 Review

For the next items in review, we want you to type the answer, but don't press enter until we tell you. We'll say, *3, 2, 1, enter* then you press enter when we say *enter*.

Let's practice this first, where we all type in chat `Hello World`.

1. In chat, assign a variable a value, then type the data type in parentheses. *e.g. quantity = 1.5 (float)*

2. For the following array `cheese <- [Cheddar, Gouda, Parmesan, Muenster]`, what is the value of `cheese[4]`

3. For the following matrix,

$$potatoes = \begin{bmatrix} Yukon\ Gold & Red\ Gold & Gala \\ Yellow\ Finn & Russet & Fingerling \\ Sweet & Kennebec & Red\ Pontiac \\ Yam & German\ Butterball & Purple\ Viking \\ Carola & Nicola & Canela\ Russet \end{bmatrix} \tag{4.3}$$

   What is the value of `potatoes[3,2]`?

4. *Challenge*: For the following dataframe, named `df`,

ingredients	quantities	importance
smoked gouda cheese	all	5
potatoes	3 lb	1
milk	0.5 c	3
salt	to taste	4
butter	8 tbsp	2

What do you think `df$ingredients[3]` would return?

Answers

1. Your answer will vary! Some examples:

- `temperature = 75 (integer)`
- `precipitation = 1.2 (float)`
- `location = California (string)`

2. The value of `cheese[4]` is `Muenster`, the fourth item in the `cheese` array.

3. The value of `potatoes[3,2]` is `Kennebec`, which is the value in row 3, column 2.

4. `df$ingredients[3]` would return `milk`. We can guess that `df$ingredients` means the `ingredients` column in the dataframe `df` (even though the `$` symbol is new). From there, we can use the same indexing approach as the other questions, such that `df$ingredients[3]` indicates the third rows down in the `ingredients` column, which is `milk`.

Chapter 5

Loops

The nice thing about the terms used for the control structures in most programming languages is they are usually self-describing. We're going to talk about loops now.

Loops are common events in programming scripts. It is what the computer uses to listen for user-input. It's also what researchers use when they want to apply a statistical method over several data files, or several columns within a data table.

5.1 `for` loops

The `for` loop is probably the most common type of loop. It executes a chunk of code *for* a certain number of times. The common structure of most `for` loops is

```
for variable in a collection
  do something
```

Now, a collection can be a list of text items, such as cheesy mashed potato ingredients, but it can also be a range of numbers, like `1 to 5` (which is programming speak for 1 2 3 4 5) or `7 to 13` (which is 7 8 9 10 11 12 13).

Let's see what this looks like in action using a basic list of cheesy mashed potatoes ingredients:

```
ingredients <- cheddar cheese, potatoes, milk, salt, butter

for ingredient in ingredients
  write(ingredient)
```

Before we look at any real world code, we will manually go through some example loops, without the computer doing the work for us. Practicing a trace on tasks that are simple, like writing each ingredient in a collection, will help us build skills for, and comfort with, reading complex code.

5.1.1 Activity: Practicing a Trace

Timing: 10 minutes
You will need: a piece of paper and writing implement

Let's set up our trace exercise. The example loop is:

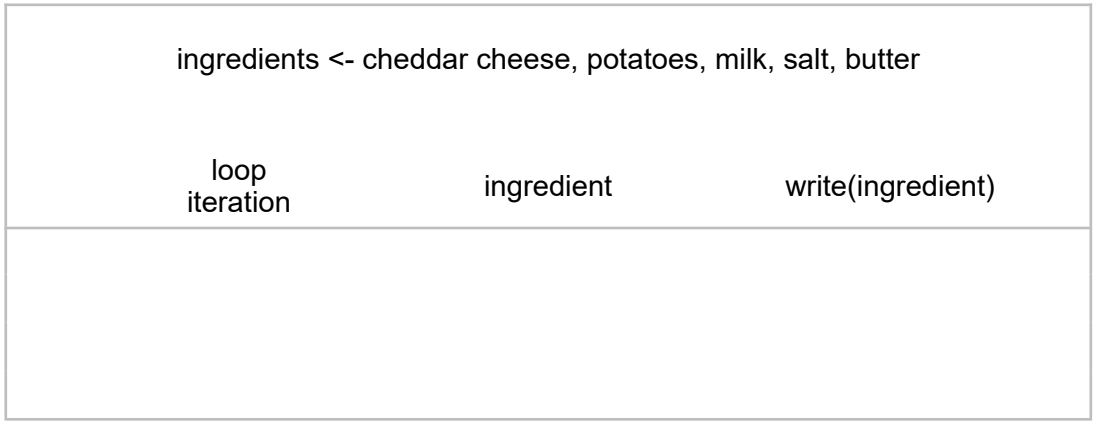
```
ingredients <- cheddar cheese, potatoes, milk, salt, butter

for ingredient in ingredients
  write(ingredient)
```

At the top of your paper, write out the `ingredients` assignment (`ingredients <- cheddar cheese, potatoes, milk, salt, butter`).

On the line below that we're going to create headers for all of the variables used within the `for` loop code block. Starting on the left, write the term `loop iteration`, then `ingredient` in the center of the line, then the term `write` on the right of the line.

Your paper should look something like this:



ingredients <- cheddar cheese, potatoes, milk, salt, butter		
loop iteration	ingredient	write(ingredient)

Your paper is formatted so it's easy to see what's going on with variable during each loop iteration. This is called a trace, and is often used in software development to help debug the program. It is also a good exercise to help build mental muscles to read code.

The next step is to fill in the values for each iteration of the loop.

Your paper will now look something like this:

```
## Warning: package 'ftExtra' was built under R version 4.4.2
```

ingredients <- cheddar cheese, potatoes, milk, salt, butter		
loop iteration	ingredient	write(ingredient)
1	cheddar cheese	cheddar cheese
2	potatoes	potatoes

ingredients <- cheddar cheese, potatoes, milk, salt, butter		
loop iteration	ingredient	write(ingredient)
3	milk	milk *(yes, this is a little redundant)*
4	salt	salt *(acknowledge the redunancy)*
5	butter	butter *(is there an equivalent to horizontal ditto marks?)*

Note: It's okay if you didn't completely fill out both the variable `ingredient` and the function `write ingredient` because in this case they are the same. But, it's important to check what actions are happening to variables when you do a trace.

You've now completed the first trace exercise of a pseudocode example. Congratulations!

In this workshop, we're focused on the programming logic underlying code, rather than specific syntax, but it can be helpful to look at "real" code doing the same task, to further build out your mental model.

If we asked a computer to do the same task we did, using the R language, it would look something like this:

```r
ingredients <- c("cheddar cheese", "potatoes", "milk", "salt", "butter")
for(ingredient in ingredients){
  print(ingredient)
}
```

```
## [1] "cheddar cheese"
## [1] "potatoes"
## [1] "milk"
## [1] "salt"
## [1] "butter"
```

5.1.2 Activity - Loop Trace

Timing: 10 minutes
You will need: a piece of paper and writing implement

Now that we've practiced a trace using text, let's practice using number ranges Remember that to a computer, the range written as `2 to 7` represents numbers 2 3 4 5 6 7.

For this loop trace, you'll write out the output of each iteration of this loop:

```
for x in 0 to 5
  write x
  write x*2
```

Like in the ingredients trace example, you'll create a table with headers for all of the variables used within the `for` loop code block. Starting on the left, write the term `loop iteration`, then `write(x)` in the center of the line, then the term `write(x*2)` on the right of the line.

Then, fill out the table for each iteration of `for x in 0 to 5`.

Loop Trace

Range 0 to 5 is 0 1 2 3 4 5		
loop iteration	write(x)	write(x*2)
1	0	0
2	1	2

Range 0 to 5 is 0 1 2 3 4 5		
loop iteration	write(x)	write(x*2)
3	2	4
4	3	6
5	4	8
6	5	10

By this point, you're probably thinking that doing a manual trace is a bit tedious. And it is! But again - this is a good way to strengthen your mental model of how loops work, and get you comfortable with thinking through what a chunk of code is trying to do.

5.1.3 Exercise - Trace Nested `for` Loops

Timing: 15 minutes You will need: a piece of paper and writing implement

So far, we've tackled one loop at a time. But loops are flexible and can be "nested", or embedded within each other.

Let's look at an example the following matrix, which is created using two `for` loops

Index	Column 1	Column 2	Column 3
Row 1	2	4	6
Row 2	5	7	9
Row 3	8	10	12

Index	Column 1	Column 2	Column 3
Row 4	11	13	15

The pseudocode for how this matrix was created is as follows:

```
num_rows = 4
num_col = 3

matrix <- []

for (i in 1 to num_rows)
  for (j in 1 to num_col)
    matrix[i,j] <- (3*(i-1))+(j*2)
```

Before creating the table to help trace the code, answer the following

- What variables need to be traced?
- What are the ranges that are used?
- What calculations do you need to keep track of?
- Is there anything else?

Answers

- What variables need to be traced?
  - `i`, `j`
  - you might think that you need to keep track of `loop iteration`, but it's redundant because both `i` and `j` start at one. it's enough to keep track of just their values.
- What are the ranges that are used?
  - 1 to 4 and 1 to 3
- What calculations do you need to keep track of?
  - `(3*(i-1))+(j*2)`
- Is there anything else?
  - With a small enough matrix, it's okay to make an empty one and fill it in

Label columns and sketch out a matrix to fill in the variables as the loop iterates for the first five lines *(you won't fill in the whole matrix, but only a portion).*

Loop Trace

num_rows = 4 & num_col = 3		
Range 1 to num_rows is 1 2 3 4		
Range 1 to num_col is 1 2 3		
i	***j***	***(3*(i-1))+(j*2)***
1	1	(3*(1-1))+(1*2)
1	2	(3*(1-1))+(2*2)
1	3	(3*(1-1))+(3*2)
2	1	(3*(2-1))+(1*2)
2	2	(3*(2-1))+(2*2)

Index	j = 1	j = 2	j = 3
i = 1	2	4	6
i = 2	5	7	
i = 3			
i = 4			

5.2 `while` Loops

There are other types of loops in addition to the commonly used `for` loop.

The `while` loop uses a condition that the computer checks if it is `TRUE` or `FALSE` at the start of the loop to determine if the code chunk inside the loop will execute.

```
while (condition is TRUE)
  do something
```

This is different from a `for` loop, because while a `for` loop will execute a predetermined number of times, the `while` loop will execute upon a condition being met.

For this loop to work, you need to set the condition *before* the loop, and change it *in* the loop.

You can write a `for` loop into a `while` loop by using a variable which will increment by one for each time the loop is run. This usually isn't done, but we want to show something you're already familiar with.

For example, recall the initial for loop:

```
ingredients <- cheddar cheese, potatoes, milk, salt, butter

for ingredient in ingredients
  write(ingredient)
```

An equivalent `while` loop would look like:

```
ingredients <- cheddar cheese, potatoes, milk, salt, butter

i = 1
condition_check = length(ingredients) + 1
while i < condition_check
  write ingredients[i]
  i <- i + 1 // change for condition check
```

Here's what it look like in R

```
ingredients <- c("cheddar cheese", "potatoes", "milk", "salt", "butter")

i <- 1
condition_check = length(ingredients) + 1

while(i < condition_check){
  print(ingredients[i])
  i <- i + 1
}
```

```
## [1] "cheddar cheese"
## [1] "potatoes"
## [1] "milk"
## [1] "salt"
## [1] "butter"
```

When we identify the variables to keep track of in a `while` loop trace, we need to add one to check if the condition for entering the loop is met.

ingredients <- cheddar cheese, potatoes, milk, salt, butter

condition_check = length(ingredients) = 6

loop iteration	i	ingredients[i]	i < condition_check

Try filling out this table for `condition_check = length(ingredients) + 1` (condition_check = 6). What is the last value of `ingredients[i]` that is printed?

Now let's try a thought exercise:

- What would be the last value of `ingredients[i]` printed if `condition_check = 3`?
- What would be the last value of `ingredients[i]` printed if `condition_check = 8`?

Answers

If `condition_check = 3`, then the loop would stop after the `i = 2` iteration. Once `i = 3`, `i < condition_check` would be `False`, and the loop would not continue. So, the last printed value of `ingredients[i]` would be `ingredients[2]`, which is `potatoes`.

In this case, we would be all out of values in `ingredients` before `condition_check` was `False`, meaning the loop would keep going even if there was nothing to return. Since there are only 5 values in `ingredients`, when we got to `i = 6` and `ingredients[6]`, the code would return `NA` (missing value). The same would happen for `i = 7`. So, our full results would look like:

```
[1] "cheddar cheese"
[1] "potatoes"
[1] "milk"
[1] "salt"
[1] "butter"
[1] NA
[1] NA
```

5.2.1 Other types of loops

Two other types of loops are a `do while` loop (check at end) and an `until` loop (check at end).

```
do
  something
while (condition is TRUE)
```

The logic in the `do while` loop is while a condition is true, the loop will run. As with the `while` loop, the condition needs to be set before the loop, and changed within the loop.

For the `until` loop, it's slightly different. The loop structure is like this

```
do
  something
until (condition is TRUE)
```

The logic is opposite of a `do while` loop. It will run while the condition is false, and will only exit when the condition is true.

These aren't used often, in fact IMO it's better to just re-write the logic to use a `while` loop.

5.2.2 Exercise - While Trace

Timing: 10 minutes You will need: a piece of paper and writing implement

Let's practice a `while` combined with a `for` trace

```
condition_check <- 5
i <- 1
while (i < condition_check):
  for j in 1 to 3:
    write (i*j)+i
```

I'll give you all a few minutes to write out your trace. Once you're done, type the largest number you traced in the write statement, but don't press enter until I give the signal.

5.3 Caveats

As you've just experienced, loops come with their own set of common issues. Since loops are a frequently encountered concept in programming, we'll go over the common problems.

5.3.1 Infinite loops

What you all just experienced in the previous exercise is called an infinite loop. Infinite loops happen when the condition check is never false. In the case of the above while loop, the check variable was never incremented, so the loop would go through the same process until the programmer interrupts it, the computer fails, or time ends (whichever comes first).

5.3.2 Overwriting outputs

```
for x in files(1 to 500)
  rename_file(x, "test-case")
```

Often when researchers are developing a script, they will use test case to develop their algorithm and work out the bugs. In this way, if mistakes are made, it's on a small scale and easy to correct. Most development is the following.

1. Develop code for a single case
2. Test on a few cases
3. Use on all of the cases

If a step is missed, it can be disastrous. I can attest. I wrote a script to rename about 500 files, and in my hubris of being an awesome coder, forgot to test it on a few cases before using it on all of the files. I forgot to change the single rename `("test-case")` to something that incorporated the loop `("test-case"+x)` and when the loop was finished, wondered why I went from 500 to only one file in the folder. Luckily, I always backup my data files, so it was an easy mistake to remedy *(and makes for a good story on the importance of data backup).*

5.4 Language examples: Formatting may vary

Programming languages may have specific formatting for loops. This may mean certain brackets must be used, tab indents are needed, or ranges are specified a certain way.

Formatting of outputs may also look slightly different, with some languages printing each output iteration on a new line, and some not.

Python requires indentation for code blocks. Code blocks in different structures must be aligned with the same indentation.

Programming structures such as `for` expect `:` at the end of each statement with the subsequent code block indented by one.

```
for x in range(0, 5):
  print(x)
```

R does not require indentation for code blocks; however, indentation is used because it's easier for humans to read!

Instead of `:` **R** makes use of curly brackets `{}` to indicate the main body of a `for` statement.

```
for (x in 0:5) {
  print(x)
}
```

The following example is **C/C++** syntax. One big difference between C/C++ and Python or R is C/C++ is a compiled programming language, which must first be compiled into a file (often *.exe, but can be anything) that contains the machine-language instructions to be executed.

C/C++ syntax looks a lot like R, but requires semicolons at the end of each line to be executed in a code block, while R does not require the semicolon (although R will also run if there are semicolons).

Note that **C/C++** also uses the `cout` command to specify what should be returned to the console, rather than `print` like in **Python** or **R**.

```
for (int i = 0; i <= 5; i++) {
  cout << i;
}
```

PHP is very similar to C/C++, but is used for web development. Note that **PHP** uses `echo` rather than `cout` or `print` to specify what should be returned to the console.

```
for ($x = 0; $x <= 5; $x++) {
  echo $x";
}
```

Chapter 6

Conditionals and Making Choices

6.1 `if` Statements

Many times in programming, we want to take a certain action only if a certain condition is satisfied.

To do this, we can use conditional statements. The most commonly used format of a conditional statement in programming is an `if` statement, which is often combined with an `else` statement.

This structure tells the computer to check a condition and the next step depends on whether the condition is true or false.

It takes the basic form of:

```
IF (condition A is TRUE)
  do something
```

In terms of our cheesy mashed potatoes algorithm, an `if` statement might look like this

```
ingredient <- cheddar cheese

IF (ingredient is (a type of hard cheese))
  grate(ingredient)
```

In that example, we've told the computer what to do *if* the condition is true. We can also specify what to do if the condition is false.

For our cheesy mashed potatoes algorithm, we may want to cut cheese into cubes if it is not a hard cheese:

```
ingredient <- cheddar cheese

IF (ingredient is (a type of hard cheese))
  grate(ingredient)
ELSE
  cube(ingredient)
```

In pseudocode, a complete `if else` statement would look like:

```
IF (condition A is TRUE)
  do something
ELSE
  do something different
```

Here, `else` is used equivalent to "if not true", meaning `A == FALSE`.

Furthermore, we are not limited to a single `TRUE`/`FALSE` check in an `if else` statement, where actions are limited to *"do something if true, in **all** other scenarios do something different"*. The `else if` (written as `elif` in some programming languages) concept allows us to add another sequential check if the `if` statement is not true.

An updated cheesy mashed potato narrative statement could be that if we have a hard cheese, we want to grate it, if we have a soft cheese we want to cube it, and if we have something that is neither hard nor soft cheese, we don't put it in the potatoes.

```
IF (ingredient is (a type of hard cheese))
  grate(ingredient)
ELSE IF (ingredient is (a type of soft cheese))
  cube(ingredient)
ELSE
  don't use
```

In pseudocode, this updated statement would look like:

```
IF (condition A is TRUE)
  do something
```

```
ELSE IF (condition B is true)
  do something different
ELSE
  do something even more different
```

6.2 Exercise - Popcorn `if else` pseudocode

Timing: 5 minutes

Let's try a similar example with the popcorn algorithm from section 2.

Write out an `if else` statement specifying what appliance to use in the case of different popcorn types: bagged popcorn (`bagged`), popcorn in an aluminum pan (`jiffy_pop`), and loose kernels.
Then, write out an `if, else if, else` statement that includes use of an air fryer for loose kernels.

Example statement

```
IF (popcorn_type is `bagged`)
  use microwave
ELSE (popcorn_type is `jiffy_pop`)
  use stove

IF (popcorn_type is `bagged`)
  use microwave
ELSE IF (popcorn_type is `jiffy_pop`)
  use stove
ELSE
  use air_fryer
```

Your version may look different, and that's fine! The fun of pseudocode is practicing with the logic of code, rather than getting lost in the details.

6.3 Comparing Things and Booleans

In programming, most things boil down to `true` or `false`. (Sometimes you may see true/false capitalized as `TRUE` and `FALSE`, but the concept is the same. We'll use both ways throughout.). `TRUE` and `FALSE` are formally a boolean data

type. Programming languages use boolean operators and comparison operators to determine if a statement is `TRUE` or `FALSE` in order to make actions.

Common Boolean operators are:

- `and` (may also see the ampersand, `&` or `&&` used)
- `or` (may also see the pipe symbol, `|` or `||` used)
- `not` (may also see `!` to indicate negation, for instance `!=` for "not equal")

Common comparison operators are:

- `>` Greater-than
- `<` Less-than
- `>=` Greater-than or equal-to
- `<=` Less-than or equal-to
- `==` *is equal to*
  - you should note that `==` is very different from `=`. `==` is the comparison operator, while `=` is an assignment operator. Another assignment operator you may have noticed us using is `<-` which is commonly used in R.
  - To practice, you can replace the words "is equal to" or "is assigned the value of" appropriately.
  - `A = 6` (`A` is assigned the value of `6`)
  - `B <- 8` (`B` is assigned the value of `8`)
  - `B == 7` ("`B` is equal to `7`", where the statement is then evaluated to be `true` or `false`. In this case, it is `false`)

6.4 Conditional Statements in Practice

Let's look at some examples of conditional statements in practice, first by using our cheesy mashed potatoes example, and then with actual code.

First, for our recipe example:

```
if cheese == hard:
  print("Grate the cheese.")
else if cheese == hard:
  print("Cube the cheese.")
else:
  print("Do not use this in this recipe.")
```

If we have `cheese <- parmesan` we would expect this statement to print out `Grate the cheese`. If we have `cheese <- brie` we would expect the output to

be `Cube the cheese`. And if we have `cheese <- broccoli` we would expect the output to be `Do not use this in the recipe`.

Sidenote: yes, this assumes that somewhere we have specified a list of hard cheese and soft cheeses. For this pseudocode example, we're using our cheese-based expertise to draw conclusions, but as we've covered, we would never assume a computer would know which cheeses were hard or soft!

6.4.1 Exercise: `ifelse` Trace

Timing: 5 minutes

For this exercise, want you to type the answer in the chat, but don't press enter until we tell you. We'll say, 3, 2, 1, enter *then you press enter when we say* enter.

For the code below, what will be the printed output?

```
x <- 37
y <- 42

if x == y:
  print("values are equal")
else if x > y:
  print("x greater than y")
else:
  print("x must be less than y")
```

Answer

x must be less than y

What would be the answer if we instead set `x` and `y` to:

```
x <- 75
y <- 9

if x == y:
  print("values are equal")
else if x > y:
  print("x greater than y")
else:
  print("x must be less than y")
```

Answer

x greater than y

6.5 Caveats

We're going to cover some more formal processes and syntax. While this won't impact working with pseudocode, it is important to know when working with the computer.

6.5.1 Order of operations in conditionals

Order of operations is critical for conditionals. The computer will go through each condition *in order*, so if an early condition is found as `true`, the statement will conclude there and not check the other conditions.

```
if (cheddar == cheese) OR (brie == cheese) OR (broccoli == cheese):
  print("This is cheesy")
```

In this case, because `cheddar == cheese` is `true`, the computer doesn't bother checking the other two statements.

6.5.2 Clarity and Order of Operations in math statements

Most of us know the standard order of operation in a math problem (PEMDAS), and most computer languages do, too.

`3 + 10 * 2` is solved as `23` (and not `26`)

and so we don't need parentheses, but it is good practice to use parentheses to make your code more readable and clear.

`3 + (10 * 2)` is preferred over `3 + 10 * 2`

just as

`(cheddar == cheese) OR (brie == cheese) OR (broccoli == cheese)`

is easier to read than

`cheddar == cheese OR brie == cheese OR broccoli == cheese`

6.5.3 Matching parentheses

For complex nested conditionals, be sure to use parentheses, and be sure parentheses are matched properly.

The code below, without a closing parentheses after "equal", will continue to expect input.

```
if (x == y)
  print("values are equal"
```

As far as the programming language is concerned, you haven't finished this `if` statement. So, depending on the language, it might wait to run until you have "completed your thought", so to speak, and provided the syntax (here the end paren `)`) indicating that this statement is complete and the program is ready to run.

Or if you have mismatched parentheses, the computer might give you an error message.

Here's another example of fun with parentheses:

```
if (x == y)
  print("values are equal"))
```

In this case, there is an extra closing parentheses `)` after `print("values are equal")` that doesn't have a matching `(` anywhere in the statement. The resulting error would look like:

```
Error: unexpected ')' in:
"if (x == y) {
    print("values are equal"))"
```

6.6 Language examples: Formatting may vary

Much like with `for` loops, programming languages may have specific formatting for conditional statements. This may mean certain brackets must be used, new lines are required between sections, or tab indents are needed.

Python requires indentation for code blocks. Code blocks in different structures must be aligned with the same indentation.

Programming structures, such as `if`, `else`, `elif`, or `for` expects `:` at the end of each statement with the subsequent code block indented by one

```
if x == y:
  print("values are equal")
elif x > y:
  print("x greater than y")
else:
  print("x must be less than y")
```

R makes use of curly brackets `{}` to indicate each section of an `if else` statement. R does not require indentation for code blocks; however, indentation is used because it's easier for humans to read!

```
if (x == y) {
  print("values are equal")
} else if (x > y) {
  print("x greater than y")
} else {
  print("x must be less than y")
}
```

C/C++ syntax looks a lot like R, but requires semicolons at the end of each line to be executed in a code block and uses `cout` rather than `print` to specify what should be returned in the console.

```
if (x == y) {
  cout << "values are equal";
} else if (x > y) {
  cout << "x greater than y";
} else {
  cout << "x must be less than y";
}
```

PHP is very similar to C/C++ with curly brackets and semicolons, but uses `echo` rather than `cout`.

```
if (x == y) {
  echo "values are equal";
} elseif (x > y) {
  echo "x greater than y";
} else {
  echo "x must be less than y";
}
```

Here's a new language: **Bash**. This language is often used from the command line to perform operations on files. With **Bash**, you need to be careful of spaces

around your operators and parentheses. (cheatsheets https://ss64.com/bash/if.html https://devhints.io/bash#conditionals)

This Bash script example compares numbers

```
if (( $x == $y )) ; then
  echo "values are equal"
elif (( $x > $y )) ; then
  echo "x greater than y"
else
  echo "x must be less than y"
fi
```

This Bash script example compares text

```
if  [[ "$x" == "$y" ]] ; then
  echo "the text in x is the same as y"
else
  echo "the text in x is not the same as y"
fi
```

6.6.1 Challenge

- What are some of the differences you can spot between the different languages?
- What are the similarities?

Chapter 7

Practice

In this section, we'll practice applying our understanding of `for` loops and `if else` conditional statements to unfamiliar code.

The code example below may look intimidating! But remember, we're less concerned here about understanding every detail about what the code is doing, and more about using what we've learned about programming logic as a lens through which to begin interpreting code.

7.1 Nested `for` loops and `if else` statements (in Python)

One way to contextualize unfamiliar code is to break down a big chunk of code into self-contained sections. For instance, if the code is doing a lot of things - looping through some task, with some conditions - we want to understand the order of operations of each thing that is being done.

Below is an example of code that is scraping content from a set of website pages and saving that content locally. This code chunk uses nested `for` loops, and nested `if else` statements with the `for` loops and within other `if else` statements.

Exercise: Review the code chunk below and identify on which line each `for` loop and each `if else` statement starts and ends.

- First `for` loop
  - Starts on line:
  - Ends on line:

- Second `for` loop
  - Starts on line:
  - Ends on line:
- First `if else` statement
  - Starts on line:
  - Ends on line:
- Second `if else` statement
  - Starts on line:
  - Ends on line:
- Third `if else` statement
  - Starts on line:
  - Ends on line:
- Fourth `if else` statement
  - Starts on line:
  - Ends on line:

```
# Code to scrape data from website

1.  base_url = "https://nces.ed.gov/ipeds/datacenter/"
2.  data_url = "DataFiles.aspx?"
3.  year_base = "year="
4.  years = ["1995", "1996", "1997", "1998"]
5.  session_id = "&sid=4f8f293f-df75-42cd-9cc0-ed184270cf17&rtid=7"
6.
7.  # what type of files do you want to save?
8.  file_type = ["zip","csv"]
9.
10. # where do you want to save locally?
11.
12. # save_loc is redundant, but a reminder that you should have locally folders that
13. # because that's where below is saving to
14. save_loc = years
15.
16. # what kind of prefix do you want on your files
17. save_prefix = "ipeds_"
18.
19. # scrape
20. for year in years:
21.     url = base_url+data_url+year_base+year+session_id
22.     webpage = requests.get(url)
23.     soup = BeautifulSoup(webpage.content, 'html.parser')
```

```
24.
25.
26.     if os.path.isdir("./"+year):
27.         # notice output path has "year" as a variable indicating folders
28.         output_path = "./"+year+"/"+save_prefix+year+"_"+".html"
29.
30.         if os.path.exists(output_path):
31.             print(output_path+" already exists. Did NOT save.")
32.         else:
33.             # saving the html file
34.             print("saving "+output_path)
35.
36.             with open(output_path, 'wb') as file:
37.                 file.write(webpage.content)
38.
39.         # saving files linked on original html page which meet file type requirement
40.         for link in soup.find_all('a')[0:16]:
41.             data_target = link.get('href')
42.             # data_target[-3:]
43.             if any(extension == data_target[-3:] for extension in file_type):
44.                 wget_url = base_url+data_target
45.                 wget_save = "./"+year+"/"+save_prefix+data_target.replace("/","-")
46.
47.                 if os.path.exists(wget_save):
48.                     print(wget_save+" already exists. Did NOT save.")
49.                 else:
50.                     print("SAVING "+ wget_url+ " at local location: "+wget_save)
51.                     wget.download(wget_url,wget_save)
52.                     time.sleep(.25) # be kind, don't look like a DDOS attack
53.     else:
54.         print("\n!!! save directory "+year+" does NOT exist. please create\n")
```

Start and end locations

- First loop
  - Starts on line: 20
  - Ends on line: 54 (encompasses entire code block line 20 until end of line 54)
- Second loop
  - Starts on line: 40
  - Ends on line: 52 (encompasses entire code block line 40 until end of line 52)

- *You may have noticed another loop on line 43. If so, great job! That one originally snuck by the instructors when putting together this answer key. We're leaving it as is though, as a reminder that even more experienced programmers miss things and make mistakes. So if you didn't catch this additional loop, you're in good company!*

- First `if else` statement

  - Starts on line: 26
  - Ends on line: 54 (the matching `else` is on line 53, with end of action on line 54)

- Second `if else` statement

  - Starts on line: 30
  - Ends on line: 37 (matching `else` is on line 32, with end of action on line 37)

- Third `if else` statement

  - Starts on line: 43
  - Ends on line: 45 (no matching `else`, ends on line 45 and goes into next `if`)

- Fourth `if else` statement

  - Starts on line: 47
  - Ends on line: 52 (matching `else` is on line 49, with end of action on line 52)

Chapter 8

Functions

8.1 What is a function?

A "function" in programming is a piece of code that does a specific task, but packages it so it can be called with a single line of code. Functions can be predefined by developers, and also may be custom to any script.

We've already been using examples of functions in our psuedocode - such as when we would include the `write` statement - and formal functions, such as the `print`, `cout`, or `echo` statements we saw in the conditional examples at the end of the previous chapter.

The `print` function does something very simple, which is provide output, either on the R or Python console or interface, that people can read. It is simple to use, but internal code that creates the print function is a little more complex (to say the least). Since communicating output is ubiquitous in all computer languages, each language has some sort of print function as a built in function so it makes it easier for programmers to focus on their particular code.

Functions can do something very simple or very complex. But in essence, functions are used when you know you're going to do the same task over and over again.

Take our cheesy mashed potatoes example. A simple function might be something like `peel_potato()`, which would take our first example algorithm on peeling potatoes, and create a function.

The original algorithm **Algorithm for peeling potatoes for Cheesy Mashed Potatoes**

1. Get three pounds of russet potatoes.
2. Get bowl large enough to hold three pounds of russet potatoes.

3. Get a vegetable peeler.
4. Take one potato.
5. Peel potato.
6. Rinse potato with water.
7. Place peeled potato in bowl
8. Repeat 4-7 until all potatoes are washed and peeled

But, we don't want to create a function for the whole thing. We want to create a function that allows us to easily execute what needs to be repeated, Steps 4-7. So the function might look like

```
define function peel_potato(potato, peeler):

  while(potato has skin):
    location <- on potato, locate skin
    potato <- at location on potato, with peeler remove strip of skin
    potato <- rotate potato
  // end-while (this is a comment, it begins with // )

  potato <- rinse potato with water

  return (potato)
// end function definition
```

A bonus of functions, is you can make them generic if you know the action can be applied over many objects. Perhaps you want to peel both potatoes and carrots, you can make a function for that.

```
define function peel_vegetable(vegetable, peeler):

  while(vegetable has skin):
    location <- on vegetable, locate skin
    vegetable <- at location on vegetable, with peeler remove strip of skin
    vegetable <- rotate(vegetable)
  // end-while

  vegetable <- rinse_with_water(vegetable)

  return (vegetable)
// end function definition
```

Now the `peel_vegetable` function can be used with multiple objects. (Well, it could before, but using the function name `peel_potato` would be confusing when trying to peel `carrot` or `zucchini`. Naming matters, especially for understandability!)

Now that we've created a function for peeling vegetables, our algorithm might look like this

```
potatoes <- 16
peeler <- "swiss peeler"
bowl <- [] // empty array

for potato in potatoes:
  bowl <- bowl+peel_vegetable(potato, peeler)

// end for-statement
```

8.1.1 New Programming Concepts

Sometimes, we may want to *add* a variable, or function output, to an existing variable type that can accept new information. Other times, we may want to *change* a variable in a completely different way.

8.1.1.1 Adding to a Variable

These two examples add to something that already exists:

- `bowl <- bowl+peel_vegetable(potato)`
- `x <- 0; x <- x + 1`

A key piece of info here is the **original variable needs to be of a data type that can accept the new piece of data**.

We defined the bowl as an array `[]`, so we can keep adding the same data type (potato, maybe vegetable) to the array.

```
potatoes <- 16
peeler <- "swiss peeler"
bowl <- [] // empty array

for potato in potatoes:
  bowl <- bowl+peel_vegetable(potato, peeler)

// bowl array after for loop finishes
[potato, potato, potato, potato,...] // 16 'potato' items in array
```

For the other example, if we tried to add two different data types

- `x <- "zero"; x <- x + 1`

The computer would return an error message, because adding text and numbers, in `"zero" + 1`, is not possible.

The task of adding content to something that already exists happens so often that some languages, such as C/C++, will combine assignment operators with math functions, as in `+=`. In this case, `x += 1` is the same as `x = x + 1`

8.1.1.2 Modifying a Variable

For this statement

```
vegetable <- rotate(vegetable)
```

the vegetable is rotating, and being reassigned to the same variable name. This is a little more tricky and in need of caution, because you're completely **rewriting the value in `vegetable` instead of modifying it**. If it were to be modified to a different data type accidentally, the program would still execute with no error message for this assignment, because it simply sees a variable assignment.

```
x <- 123
x <- as.text(x) // accidentally uses the wrong function
// now the data type of x is a string

// many lines of code later

x <- x + 1  // this will cause an error in the code
```

The error will happen because a string and a numeric value can't be added together. But, this will be tricky to troubleshoot because the error will result from the `x <- x + 1` part of the code, even though the actual error was the assignment which happened many lines prior in `x <- as.text(x)`.

8.2 Back to Functions

A complex function might be something like `make_cheesy_mashed_potatoes()`, which is the equivalent to having an in-home chef who can make you cheesy mashed potatoes on demand, as long as you provide the raw ingredients.

```
make_cheesy_mashed_potatoes(cheese, potatoes, milk, salt, butter)
```

In programming parlance, these ingredients listed `(cheese, potatoes, milk, salt, butter)` would be the "arguments".

8.2.1 Arguments

Functions will have inputs *(usually)* and outputs. The function inputs are formally called "arguments," and are specified by the programmer, entered within the parentheses as part of the function call. For cheesy mashed potatoes, this could be specifying what kind of cheese to use, or the quantity of each ingredient.

While our example is in pseudocode, luckily function calls look the same across most languages.

```
# these variables have quantities in oz
cheese <- 16
potatoes <- 32
milk <- 8
salt <- 0.125
butter <- 4

make_cheesy_mashed_potatoes(cheese, potatoes, milk, salt, butter)

paste("this function uses", cheese, "oz of cheese")
```

We already mentioned that `print` is a common built-in function. Most languages a number of built-in functions which are commonly used, such as the `paste` function in R, represented above. Let's look at another built in function that is common to both R and Python: `round()`

Built-in functions have help pages which will allow you to see the syntax and arguments. The help files will include definitions of the arguments, if they're required or have a default, and examples for use.

The syntax for Python is `round(number, ndigits=None)`

The syntax for R is `round(x, digits = 0, ...)`, where `x` is a numeric vector.

The second argument for both languages indicates the number of digits along with the default value. Since there's a default value, you don't necessarily need to include the argument in the function call.

An example in R is

```
round(c(1.4, 2.5, 3.5))
```

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

Another argument common in many functions which deal with calculating something is how to handle missing or `NA` values.

For example, the function `mean()` in R is `mean(x, trim = 0, na.rm = FALSE)`. The default value of `na.rm` (which you can think of as "NA remove") is `FALSE`, indicating that `NA` values will *not* be removed from the calculation.

Optional arguments like this may come with a pre-supplied default, in this case that `na.rm = FALSE` and any `NA`values will be retained, unless this argument is manually changed to `TRUE`.

8.3 Function Takeaways

When working with functions, which is the vast majority of programming, it is important to keep in mind two things:

1. The arguments in a function
2. Which arguments are optional, and what are their defaults

A function will run as long as the required arguments are provided, but the resulting output may not match expectations unless you recognize which optional arguments were included with default values.

8.4 Libraries

We've already discussed built-in functions. Built in functions are often limited to basic tasks and do not include more complex or custom functions that you may with to use. Now, you can code more complex functions yourself, building off of the built in functions, but this would take a lot of time and require more in-depth programming knowledge.

The good news is that most programming languages will have optional "libraries" (or packages, or modules, depending on what term your programming language of choice uses) that include additional functions, beyond the built in function. So before creating a new function from scratch, it is worthwhile to check whether a library exists that includes a function that does what you want to do.

Using our cheesy mashed potatoes example, you might use a programming library to find a recipe that uses different ingredients. You want to make cheesy mashed potatoes from scratch, but don't have the time to do so. Luckily someone else has created function that uses instant mashed potatoes instead of raw potatoes.

Libraries are also useful because they mean that your computer and programming language of choice don't need to always have every single function

on hand. This set up saves computer disk space, ensures you don't have to recreate the wheel and make every function from scratch, and provides a level of standardization (e.g., everyone uses the same reference "recipe" so output should be the same for the same input, across users).

A good rule of thumb is if it *seems* like the function you want is broadly useful, then someone has likely created a library containing it. This is also true for niche or domain-specific functions: if the task is one that comes up a lot in analysis, there is likely a library that has functions for those analysis tasks.

Finding the 'right' library for the function you need can be overwhelming, but a good starting point is the official library collection for a programming language, such as CRAN for R or PyPI for Python.

8.4.1 Accessing Functions in Libraries

The syntax for accessing functions in libraries varies by programming language but follows the general process of:

1. Install the library from the source. You only need to do this once.
2. Load or import the library. You will need to do this every time you want to access a function in a library. By convention, libraries are loaded at the top of a script, so you, and other people, can see at a glance what libraries are needed to run the script.
3. Use the functions as normal.

8.4.2 Function Names

There are only so many function names that make sense in the English language, so there may be functions from *different libraries* that have the same name. How does the programming language know which function you are trying to access? By default, the language will use the function of the more recently loaded or imported package.

Let's say we have two `mean()` functions, one from `library_A` and one from `library_B`. They differ in their default settings:

- `library_A` defauls to `na.rm = TRUE`
- `library_B` defaults to `na.rm = FALSE`

If we load libraries in order `library_A` then `library_B` and then use `mean()` as a function, we will be using the `mean()` function from `library_B`.

```
load(library_A)
load(library_B)

values <- c(2, 4, 7, 5, 9, NA)
mean(values)
```

Our result will be `NA`.

Conversely, if we load `library_B` then `library_A`, we will use the `mean()` function from `library_A`.

```
load(library_B)
load(library_A)

values <- c(2, 4, 7, 5, 9, NA)
mean(values)
```

Our value will be `5.4`.

The tricky part is that all this happens invisibly. There may or may not - depending on your programming language - be a warning that two libraries contain functions of the same name. So, keeping track of your order of loading is important. If you get any unexpected results, you can double check which library the function you are using is from.

A better method is to be explicit about which function you are calling. Most programming languages will allow a syntax along the lines of `library:function()` to specify use of a function from a stated library.

```
load(library_B)
load(library_A)

values <- c(2, 4, 7, 5, 9, NA)
library_B:mean(values)

#result will be `NA`
```

8.4.3 Aliases

You may encounter the concept of an "alias" for a library. This is common in Python, where users can set an alias for a library name, and use that going forward rather than writing out the full library name. This will mostly come up if you are looking for help online, or wondering why you are seeing abbreviations.

For example, Python uses the `import` term to load a library (or "package" as Python calls them) and allows setting an alias using `import package as alias` syntax. By convention, many Python users will use standard aliases for common packages, such as:

```
import pandas as pd
import numpy as np
```

Functions can then be called using the explicit `package:function` syntax, such as `pd.DataFrame` to designate a `pandas DataFrame` object.

8.4.4 Exercise: Create Popcorn Function

Timing: 15 minutes

Let's return to the section 2 activity where your group created an algorithm to make popcorn.

- How would you adapt your step by step narrative algorithm to be a function called `make_popcorn()`?
- What arguments would be included, and would you have any default values set?
- Does your function look different than others in your group? How would you tell the computer which function you wanted to use?

Example functions

```
# specify which library we want to get a function from,
# in this case instructor Steph

from Steph import make_popcorn()

# The arguments and defaults for this library is
#
# make_popcorn(quantity_popcorn, type = microwave, time = 2,
#              butter = movie_theater_flavor, seasoning = TRUE)
#
# where time is in minutes and quantity_popcorn is in bags

# example in use

make_popcorn(quantity_popcorn = 1, time = 1.5)
```

```
# specify which library we want to get a function from,
# in this case instructor Kat

from Kat import make_popcorn()

# The arguments and defaults for this library is
#
# make_popcorn(quantity_popcorn, quantity_oil = 1, type = stovetop,
#              butter = TRUE, salt = "light")
#
# where quantity is in tbsp

# example in use

make_popcorn(2)
```

8.5 Language examples: Formatting may vary [may not need in this section, since round() and mean() included above]

Python

R

C/C++

PHP

Bash

Chapter 9

Self-Documenting Code

We've mentioned this before, but it's worth repeating. There are certain habits which will help create self-documenting code, which will reduce the need for a large amount of outside documentation. These habits are using comments and descriptive names.

9.1 Variable Names

Variable names should be self-describing, and there are some other naming conventions and restrictions which you should know about. Yes, I've had researchers schedule a consultation and ask me why their code wasn't running when their variable names violated a restriction.

9.1.1 Naming Restrictions

- Variable and function names are case sensitive
- Variables names use alphanumeric characters and underscores. Special characters cannot be in variable names, especially since `$` and even `.` mean specific things in many languages.

- Variable names cannot start with a numeric character, but must start with an letter or underscore.
  - Underscores `_variable` usually have a predefined meaning in different languages.
- Variable names cannot contain spaces or hyphens.
  - The space will treat it as two variables: `my variable`

  - The hyphen is treated as a minus sign `my-variable` will be read as `my` minus `variable`

- Variable names cannot be reserved words, such as `for` or `while`

9.1.2 Naming Conventions

Pro Tip: Choose a convention and use it consistently

Because variable names are generally case-sensitive, it allows different types of naming conventions.

- `camelCase`
- `use_of_underscore`
- `ADifferentTypeOfCamelCase`
- `ALLCAPS` means something specific in certain languages, such as constants in C/C++. it can be used, but you should avoid it.

9.1.3 Common Variable Names

You may have noticed that some of the variable names in our examples use certain variable names, such as `i`, consistently. This is not because we are being lazy. There are some common naming conventions for variable or function names used across programming languages. Here is an incomplete list that focuses on the most commonly used variable names.

- `i`,`j`,`k` are commonly used as loop indices, iterators, and sometimes a count
- `n` commonly represents a count or number
- `df` is commonly used to represent a DataFrame
- `tmp` is short for temporary, and represents a temporary variable
- `func` or `f` will represent a function
  - these can be used as `func_sum` or `f_sum` to indicate it as a user defined function
- as shown in the previous chapter, `pd` and `np` are often used as aliases for pandas and numpy, respectively. Other libraries will have common aliases used across the community which uses those libraries.

9.2 Comments

Just like metadata, comments are love notes to your future self.

Comments in programming scripts are there for the programmer. The computer doesn't see the comments at all. It just ignores them. I use comments as

a supplement to self-describing variable names. The variable name tells you *what*; the comments tell you *why*

I've seen code from grad students without comments before, and I gently asked them if they would remember why they did certain things if they weren't working on it for six months, then suggested adding comments so it'd be easier to pick it back up in the future.

Each language will have different characters which begin a comment line or comment block. For single line comments, the comment will start at the special characters and go to the end of the line.

Python, R, Ruby, Bash

```
# This is a comment
```

Python

```
'''
This is a
multi-line comment
'''
```

SQL

```
-- This is a comment
```

C/C++, JavaScript, PHP, Java, C#

```
// This is a comment
```

```
/*
This is a
multi-line comment
*/
```

HTML

```
<!-- This is a comment, which can go multiple lines -->
```

9.2.1 Multi-Line Comment Trick

There is a trick to easily using multi-line comments

Multi-line comments are often use to block out code while testing different scripts in C/C++ and JavaScript, or block out text in html. But adding and removing the beginning/end of the code is a bit of a pain.

However, here's the trick:

Always have a beginning + end at the end of a code block you want to comment out, then you just need to add the beginning.

```
/*
this is a
multi
line
block of code
commented out
and here is the end
/* */
```

It's similar for html

```
<!--
This is a comment,
which can go multiple lines
and it doesn't matter that the end
also has the beginning of a comment block
it's still commented out
<!-- -->
```

Chapter 10

Common Issues

Here are some common issues that we've seen during consultations. While we've covered some of these in prior chapters, it's worth having a list to review.

10.1 Difference between = and ==

The former, `=` is used to set something as equal as in `x = 5` where the variable `x` is equal to, or has a value of, `5`. Conversely, the double equal `==` is used to test for equality. For instance, if we set a variable x to equal the value 5, the code `x == 5` would return `True` and the code `x == 6` would return `False`.

10.2 Spelling and capitalization matter

Programming languages are generally case-sensitive. The variable `x` is different from the variable `X`. Likewise, a function `mean()` and a function `Mean()` would be separate functions.

It's easier to list the languages which are not case sensitive than those which are:

- SQL commands are not case-sensitive, but it's common practice to see the commands as all-caps, such as `SELECT`, because it makes reading the argument easier
- Functions in Excel are not case-sensitive
- HTML tags are generally not case-sensitive, however, class names in HTML and CSS are case-sensitive.

Again, it's easier to just assume everything is case-sensitive.

10.3 Special characters and reserved words

Many programming languages reserve specific words for specific tasks. For example, a function called `sum()` is fairly common across languages. While you *could* make your own alternate function called `sum()`, this may lead to unexpected results when using the `sum()` function, so it's better to avoid using the same name.

Similarly, it is best practice to avoid naming variables or objects the same as common function names. Again, while you *could* write code like `sum = sum()`, this will get confusing for you, and may lead to unexpected results and code behavior downstream.

The same is true for certain characters. Many programming languages treat `NA` as a special class of missing value. This is *not* the same as `"NA"`, which would instead be a character string containing the letters NA.

10.4 Ending a code chunk

Certain programming languages require a character at the end of a line or statement.

C/C++, Java, and PHP all require a semicolon `;` at the end of lines or statements.

Why do we say *statement*? Because technically you could write your code

```
if(x == y){cout << "values are equal";x++;cout<<"x was increased & is now
equal to "<<fixed<<setprecision(1)<<x<<endl;}else if(x > y){cout << "x greater
than y";x--cout<<"x was reduced & is now equal to "<< fixed <<setprecision(1)
<<x<<endl;} else {cout << "x must be less than y";x++;cout<<"x was increased
& is now equal to "<<fixed<<setprecision(1)<<x<<endl;}
```

But, there's an error in that code. A required semicolon is missing. Can you find it?

Having each statement on its own line is easier to read.

```
if (x == y) {
  cout << "values are equal";
  x++;
  cout << "x was increased & is now equal to " << fixed << setprecision(1) << x << endl
} else if (x > y) {
  cout << "x greater than y";
  x--
  cout << "x was reduced & is now equal to " << fixed << setprecision(1) << x << endl;
```

```
} else {
  cout << "x must be less than y";
  x++;
  cout << "x was increased & is now equal to " << fixed << setprecision(1) << x << endl;
}
```

10.5 Order of Execution

In the chapter about functions, we gave an example where libraries which have functions with the same name will default to the last loaded library when called, so it's good habit to explicitly specify the library when using functions which might have the same name in a different library

```
load(library_B)
load(library_A)

values <- c(2, 4, 7, 5, 9, NA)

library_B:mean(values)
#result will be `NA`

library_A:mean(values)
#result will be `5.4`
```

10.6 Order of Assignment

I've had researchers execute code chunks which assign a value to a variable, then reassigns a different value to the same variable, undoing the first assignment.

```
researchVariable <- someFunction(data)
researchVariable <- aDifferentFunction(data)
plot_or_graph <- anotherFunction(researchVariable)
all of
the plot
or graph
code sometimes it's a lot
```

If you see this, ask the researcher to talk through their code, and gently point out the reassignment. If it's for different conditions, ask if others looking at the code would understand that.

But, often they'll share that they were trying different things for working with their data. If the assignment is unneeded, they should just remove it from their code. If they are using version control (e.g. git), they will be able to recover the code by looking at previous versions.

But, including code that's not used in the final script clutters the script, and makes it more difficult to trace when errors occur.

10.7 Matching Delimiters: Quotes, Parens, Brackets

Delimiters are used to indicate the boundaries for something specific in the syntax. They need to be a matching set, so the computer knows the beginning and end boundary. A missing end quote or bracket will often throw an error; review the error trace to find where to complete the pair.

Many text editors (such as Sublime Text or Notepad++) or development environments (such as Jupyter Notebooks or RStudio) will auto-complete the end delimiter, but sometimes you need to adjust that in the options.

- **Quotes `" "` or `' '`** - used for text. If used within text, you either have to escape it (often the escape character is \), or use the other (`"you'll need to  surround contractions with double-quotes"`)
- **Parentheses `( )`** - used for functions or order of operation in expressions
- **Curly-Brackets `{ }`** - often used to indicate code chunks
- **Square-Brackets `[ ]`** - used to indicate an index in a collection

10.8 Reading and Writing Data Files

Reading and writing files can be a pain point. There are two things to consider when reading from or writing to files

1. The location of the file with respect to the script that is doing the request.
2. The type of file to be read/written. Different file types require different functions.

It is useful to understand how to navigate between directories on a command line, as many of the keywords and shortcuts are used to create the *path* to the files. There are recommended lessons for learning the Bash command line in the Resources chapter.

Here is an example loading a CSV file in Python

```
import pandas as pd

# updates the event list based on previously created csv
events_df = pd.read_csv("../data/events.csv")
```

Here is the folder structure. Folders have a `/` at the end of their name, while files will have an extension (`.csv`, `.r`, etc)

```
- data/
  - events.csv
- src/
  - myPythonFile.ipynb
```

Here is an example of writing a CSV file in R

```
write.csv(mydata, file = "output/census_names2.csv", row.names = FALSE)
```

The folder structure is

```
- project folder/
  - get_names.r
  - output/
    - census_names2.csv
```

Here is an example of saving an HTML file (which is just a text file) in Python

```
import requests
from bs4 import BeautifulSoup

year = "1995"
save_prefix = "ipeds_"
url = "https://nces.ed.gov/ipeds/datacenter/DataFiles.aspx?year="+year

output_path = "./"+year+"/"+save_prefix+year+"_"+".html"
webpage = requests.get(url)

with open(output_path, 'wb') as file:
  file.write(webpage.content)
```

The folder structure is

```
- save-linked-files/
  - save-linked-files.ipynb
  - 1995/
    - ipeds_1995_.html
```

Chapter 11

Practice

In this section, we'll work on annotating and understanding unfamiliar code.

The code examples may look intimidating! But remember, we're less concerned here about understanding every detail about what the code is doing, and more about using what we've learned about programming logic as a lens through which to begin interpreting code.

11.1 New types of conditions (in SQL)

Consider the following SQL command:

```
SELECT * FROM OceanBuoys
WHERE Ocean = 'Atlantic' AND (BuoyName LIKE 'S%' OR BuoyName LIKE 'K%');
```

One option for navigating unfamiliar code is to break down the code into its component parts. Using that approach, you can differentiate functions separate from variables and names to get a better understanding of what type of data you're working with, and what the code is trying to do.

Exercise: Using what we've learned in prior sections, try to answer the following questions.

- What does the `OceanBuoys` term most likely refer to - variable or function?
  - Tip: Variables will contain a data object. Functions perform an action or task.
- Assuming that `SELECT` is a function, what might it do?
- The `*` character is new. What might `*` indicate, in combination with `SELECT`?

- Parsing the second line that starts with `WHERE`, can you make an educated guess about what `Ocean` and `BuoyName` refer to, in terms of variables?
- Can you guess (or feel free to use your favorite search engine) what `LIKE 'S%'` and `LIKE 'K%'` would indicate?
- Note the `;` at the end of the line starting with `WHERE`. What might this mean?

Do your answers match these?

- The `OceanBuoys` term here refers to a table or dataframe or matrix (really, any kind of tabular format).
- The `SELECT` function selects columns from the `OceanBuoys` table.
- The `*` is a wildcard matching symbol. It indicates that we want to return *all* columns from the table, no matter what their column names are.
- The `OceanBuoys` table must have columns named `Ocean` and `BuoyName`.
  - If we wanted to return only these columns, the syntax could be: `SELECT Ocean, BuoyName FROM OceanBuoys`
- `LIKE` is used for pattern matching. The symbol `%` is a wildcard character in SQL. This code is matching values that start with "S" or with "K".
- The conditions after `WHERE` are the row-based conditions for what will be returned, analogous to `SELECT` for the column-specific condition.
- The `;` is used to indicate the end of a statement in SQL. It tells the computer that the 'thought' is finished, and the action of *doing* the thought can commence.

Exercise, continued: Now, can you write out a narrative of what you see the code is trying to do?

Example written narrative

This code is returning a subset of data from the `OceanBuoys` table. Starting from the full `OceanBuoys` table, return all the columns from this table, but keep only rows where `Ocean` is equal to "Atlantic" *and* where the `BuoyName` value starts with "S" or "K". The returned data should be tabular, with the same number of columns as the full `OceanBuoys` table, but likely with less rows (only those that met the condition).

11.2 New syntax and terms (in R)

While the logic underlying programming languages stays consistent, a central challenge is that different languages often have their own special syntax, which can take a while to get used to. Don't panic! Familiarity comes with experience and in the meantime, Google is your friend (well, the search engine of your choice. Some of us prefer DuckDuckGo).

Here is an example of R code:

```
1.  df_new <- df %>%
2.    select(-respondent_name) %>%
3.    mutate(identifier = paste(respondent_id, survey_wave, sep = "_")) %>%
4.    mutate(survey_type =
5.           ifelse(survey_wave %in% c("first", "second"), "phone", "in person"))
```

Exercise: Go line by line and annotate, in your own words, what that line of code is doing. Then, combine these into a written narrative of what this code is doing.

Each line has a new take on the same logic we've covered in prior sections; a breakdown of new terms is below to guide your annotation.

- Line 1
  - What is `df_new <- df` doing here?
    - Hint: refer to 8.1.3, Common Variable Names
  - What does the `%>%` symbol indicate?
- Line 2
  - What is the `select()` function likely doing?
  - What might it mean that the argument in this function is preceded by `-`?
- Line 3
  - Using context clues (and your favorite search engine) what do you think the `mutate()` function does?
  - The new `paste()` function has three arguments (`paste(arg1, arg2, arg3)`). What do you think the arguments are for?
- Line 4 & 5 (note: new line not strictly necessary, only to fit in page width without needing scroll)

  - This `ifelse()` statement has a different format than we've covered so far, but the concept is the same. Assuming this `ifelse()` statement has three arguments (`ifelse(condition, mystery1, mystery2)`) what might the two mystery arguments be specifying?
  - Looking at the section `survey_wave %in% c("first", "second")`, what do you think this would translate to, as a written explanation of the task here?

- Finally, it is always important to understand the data type and structure of the data being acted upon. Keep in mind, based on the questions above, what is the structure of the data being used here?

Example narrative

Starting from the table/dataframe called `df`, we want to keep (`select`) all columns *except* the column named `respondent_name`.

Then, make a new column called `identifier`. This new column is created by pasting together the value in the `respondent_id` column and the `survey_wave` column, separated by an underscore, `_`.

Then, make another new column called `survey_type`. The values in this column are determined by an `ifelse` statement: if the value in the `survey_wave` column is any of the values specified in the list `("first", "second")` (so if the value is `first` or `second`), then the value in the `survey_type` column will be `phone`. Otherwise, the value will be `in person`.

11.3 New functions in Stata

This challenge uses Stata, a proprietary statistical analysis platform. Stata has a number of unique features and syntax that can make it challenging to interpret. (At least in this instructor's experience, Stata is not user-friendly, but your mileage may vary!)

Relying on what we've learned so far and context clues, what is the code below doing?

```
replace variable = variable[_n-1] if missing(variable)
```

What is this code doing?

This is a bit of code to replace any missing values of `variable` with with previous (`n-1`) value of `variable`.

There are many ways to replace missing values in Stata, this is one.

11.4 C++

This is a C++ file named `testScratchDoc.cpp`

Review the file then answer the following questions, annotating as you see fit.

- How is the code similar to some of the code that we've seen so far?
- What do you think the code does (generally)?
- What questions do you have about it?
- What are some suggestions to make it easier to read?

```
#include <iostream>
#include <stdio.h>
#include <cstring>


int main() {
    printf("Hello World!\n");

//    std::cout << "Hello, World!" << std::endl;

    // mbr (a,b,c,d)
    int a=6;
    int b=3;
    int c=8; //c=b+1; inner/outer test c=4
    int d=5;

    //mbr (i,j,k,l)
    int i=1;    // i=n;  inner/outer test i=1
    int j=2;
    int k=3;  // k=j+1; inner/outer test k=3
    int l=4;
    bool intersect_x = false;
    bool intersect_y = false;

    printf("\nMBR1[%d,%d,%d,%d]\n",a,b,c,d);
    printf("MBR1[%d,%d,%d,%d]\n",i,j,k,l);

    if(!((c<i) ||(k<a) ))
    {
```

```
        printf("x intersects!\n");
        intersect_x = true;
        printf("intersect_x is %d\n", intersect_x);
    } else{
        printf("x does not intersect!\n");
    }

    if(!((d<j) ||(l<b) ))
    {
        printf("y intersects!\n");
        intersect_y = true;
        printf("intersect_y is %d\n", intersect_y);
    } else{
        printf("y does not intercept!\n");
    }

    if(intersect_x&&intersect_y)
        printf("MBR1 intersects with MBR2\n");
    else
        printf("MBR1 does not intersect MBR2\n");



/*
    // testing writing to memory using pointers
    int page = 1;

    int *pData; // pointer to data
    pData = &page;
    printf("\npdata is %s; \n&pData is %p; \n*pdata is %d",pData, &pData, *pData);

    int* newP; // new pointer to data
    int a = 2;
    int b = 3;

    int m = sizeof(b);

    newP = pData+m;

    memcpy(pData,&a,sizeof(int));
    printf("\npdata is %s; \n&pData is %p; \n*pdata is %d",pData, &pData, *pData);

    memcpy(pData+m,&b,sizeof(int));
//    newP = pData+sizeof(int);
    printf("\nnewP is %s; \n&newP is %p; \n*newP is %d",newP, &newP, *newP);
```

```
*/ // end testing writing to memory using pointers



    // testing whether you can initialize a structure with an outside variable
    // not really, you can with an outside constant
/*    const int b = 100;
    const int c = 7;
    const int a = (int)(b/c);
    int i;

    struct test {
        int test[a];
    };

    test mytest;

    for(i=0; i<a; i++)
    {
        mytest.test[i]=i;
    }

    for(i=0; i<a; i++)
    {
        printf("\nmytest[%d]=%d",i,mytest.test[i]);
    }*/


    return 0;
}
/*
#include <stdio.h>

main() {
    printf("Hello World!\n");
    char sentence []="test your 12 7 42";
    char str[20];
    int a, b, c;
    sscanf(sentence,"%s %s %d %d %d", str, str, &a, &b, &c);
    printf("%d %d %d\n", a,b,c);
    printf("%s\n", str);
    sscanf(sentence,"%s your %d 7 %d", str, &a, &b);
    printf("%d %d %d\n", a,b,c);
    printf("%s\n", str);
    printf("%s\n", sentence);
```

```
}*/
```

11.5 MATLAB

This is a MATLAB file named `assignment01_leaf.m`

Review the file then answer the following questions, annotating as you see fit.

- How is the code similar to some of the code that we've seen so far?
- What do you think starts a comment line?
- What do you think the code does (generally)?
- What questions do you have about it?
- What are some suggestions to make it easier to read?

```
function assignment01_leaf
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% (C) Student Name Here %%%%%%%%%%%%%%%%%%%%%%%%%%%%
TRAIN = load('FaultsNNA_csv'); % Only these two lines need to be changed to test a dif
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

TRAIN_class_labels = TRAIN(:,1);    % Pull out the class labels.
TRAIN(:,1) = [];    % Remove class labels from training set.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute and display Default Rate and other Basic Information
%
[class,count] = mode(TRAIN_class_labels(:,1));
default_error_rate=1-(count/size(TRAIN_class_labels(:,1),1));
disp(['The dataset you tested has ', int2str(length(unique(TRAIN_class_labels))), ' cla
disp(['The training set is of size ', int2str(size(TRAIN,1)),'.']);
disp(['The time series are of length ', int2str(size(TRAIN,2)),'.']);
disp(['The dataset''s most common class is ',num2str(class),' with a total of ',num2str
disp(['The default error rate is ',num2str(default_error_rate),'.']);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Compute and display the k-fold error rate
%
k=length(TRAIN_class_labels);
k_fold_error_rate = k_fold_cross_validation(TRAIN,TRAIN_class_labels,k);
disp(['The normal k-fold error rate is ',num2str(k_fold_error_rate),' where k=',num2str
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Here is a sample classification algorithm, it is the simple (yet very competitive) one-nearest
% neighbor using the Euclidean distance.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function predicted_class = Classification_Algorithm(TRAIN,TRAIN_class_labels,unknown_object)
best_so_far = inf;
for i = 1 : length(TRAIN_class_labels)
    compare_to_this_object = TRAIN(i,:);
    distance = sqrt(sum((compare_to_this_object - unknown_object).^2)); % Euclidean distance
    if distance < best_so_far
        predicted_class = TRAIN_class_labels(i);
        best_so_far = distance;
    end
 end;
```

Chapter 12

Recap and Consultation Tips

The point of this workshop isn't to make you an expert, but to build your skillset so you can read code and be more confident in consultations. Let's first talk about where errors can pop up.

12.0.1 Clarifying variables

Something I'd commonly do when helping researchers with their code, especially when I'd see variable names which are abbreviations, is I'd ask them if the abbreviation is something which is commonly used in their field. My questions would usually go something like this:

- *So you know, I'm not an expert in your field. You're the expert. Can you help me understand a little more about what you're trying to do in the code?*
- *What does* `w_f` *represent?*
- *Is that something someone who is knowledgeable in your field would recognize?*
- *It's good practice to use self-describing variables which someone in the field would recognize. It's also a good idea to keep track of information about your data in a README or data dictionary. Honestly, I've heard a lot of different terms to describe them, but it's basically a file that describes your data. I can give you an example if you want.*
- Kat's favorite dataset example: Leaf Dataset in UCI Data Science Repository (Silva and Maral, 2013)
- *At the top of your code when you define your variables, if the variable name doesn't exactly correlate to something in your dataset's README file, you can add a comment with the exact term.*

12.1 Narrowing Down Errors

You might think there's a lot of places where errors can be introduced in code, especially if it's a hundred lines, but you can usually narrow down the errors to a few places.

12.1.1 Why Doesn't My Code Run?

The nested for loops and if else statements we practiced in the previous chapter is a useful skill and especially helpful when troubleshooting. For instance, if there is an error, a first item to check is often: are functions, especially loops and conditional statements, started and ended properly in matched fashion? If they aren't, the error will be caused by a syntax error.

Examples of syntax errors are a missing quote, paren that's supposed to close a delimiter pair, a missing semicolon, the computer trying to open a file that doesn't exist in the location indicated by the path, and a malformed function call. With syntax errors, the code generally won't even run, but will output error messages.

The nice thing about syntax errors is the code doesn't run and will give you an indication of where to look to correct the error.

Here's an example in R. The code is

```
if (x == y {
  print("values are equal");
} else if (x > y) {
  print("x greater than y");
} else {
  print("x must be less than y");
}
```

But, when we run it, the console indicates

Similarly, if we try to run something in Jupyter Notebook using Python with an error…

```
if (x == y:
  print("values are equal")
elif (x > y):
  print("x greater than y")
else:
  print("x must be less than y")
```

…we will get an error message…

…with the specific location where the error started

As stated in earlier chapters, it's important to read through the error message, usually from bottom then working your way up. Sometimes error messages can be quite long. If you want to practice this for a particular language, there are recommended lessons in the Resources chapter.

12.1.2 The Code Runs, but Something is Wrong

It's a little trickier to troubleshoot when the code runs, but does something unexpected. Unexpected output might look like incorrect results, it could be a graph or chart doesn't look right. This is trickier because it might be the *logic* of the code.

Things I scan for before looking at the logic are making sure the correct data files are being requested, and making sure the location of the images the researcher is showing me matches with the file path indicated in the code.

If those match, then it's useful to ask the researcher to walk through the code and explain it to you. Which brings us to tips for how to approach consultations.

12.2 Approaching Consultations

Consultations may seem scary, especially if you aren't familiar with a particular language, but the nice thing about programming languages is the concepts are so similar across languages, many of the issues students and researchers come with aren't specific to the language, per se. That is, sometimes it's something as simple as a variable not being assigned properly. That isn't language specific, but requires someone to be able to trace the code.

Here are some tips on approaching consultations

- **Prepare in Advance.** Include a prompt for anyone scheduling a consultation to provide as much detail as possible. *Please share anything that will help prepare for our meeting. Include your specific research question or problem.* This will help you to prepare in advance. I've followed up via email, asking researchers if they'd share their code/data in advance.
- **Be clear with what you can and cannot do.** You don't need to be an expert in order to provide consultations for programming support, but you should be clear in what you can do. For example, I have provided many Stata consultations. I have never used Stata, and I am clear with that to establish reasonable expectations. *Just so you know, I'm familiar with Stata, but not an expert. If you want, we can meet and talk through your research and code.*

- **Have the researcher talk through their code.** This does a few things: it helps you understand what they're trying to do with their code and research project. More importantly, they will get a clearer understanding of what's going on because they are describing it. Sometimes the act of them reading through and describing the code will allow the researcher to see some of the issues, and fix it on their own.
- **Programming vs statistics.** Sometimes the researcher will ask for an explanation or interpretation of a PCA, linear regression, or other analysis. *Full disclosure, I've taken some machine learning and stats classes, but I am not a statistician.* Talking through the problem often helps.
  - If they share that they're using someone else's code with their own data, the first thing to ask is if the code works with the original data. The next thing to ask is to walk through the similarities and differences with the new data, and how the variables of the different datasets relate.
  - Know where they can get statistics consultations at your institution.
- **Troubleshooting vs Consult.** Understanding the difference between these two will help you manage time and expectations.
  - Troubleshooting is often shorter, even though it can take time depending on the problem. Troubleshooting benefits from having access to the code and data beforehand, especially for things that you aren't familiar with. For example, I knew R, but for several years wasn't familiar with the Tidyverse collection of packages. Having the code/data prior allowed me to save the time of the researcher (Ranganathan, 1931), focus the time of our consultations, while giving me an opportunity to learn how to use Tidyverse.
  - Consultations are often more in-depth than troubleshooting, and will include more *how* to do something, best practices for file/variable names, managing data, sometimes even how to program.
- **It's ok to say you don't know!** As long as you clearly communicate what you can and can't do, and share that you're willing to help the researcher find assistance you can't provide, most researchers will be thankful and understanding if you *don't know.* One of the best professors I ever had would sometimes answer questions with *I don't know, but I'll look that up and get back you you.* It's a philosophy I've incorporated into my professional services, *I don't know, but I'll help you find the answers.* Have a list of go-to documentation and learning resources to point to when needed.

Chapter 13

Resources

Here's a list of resources, cheat sheets, reference materials, and stuff we really like and recommend.

13.1 Learn to Program Specific Languages

- For practical, research-focused language-specific training (R, Python, SQL, Bash, etc.), we recommend The Carpentries lessons.
  - You can also use these for reminders of how certain functions are called.
  - Pro Tip: be sure to leave enough time for the lessons. R & Python will each take about 8 hours, Bash & Git will take about 4 hours each. But, you will be about at an intermediate level when you finish these lessons.

13.2 Cheat Sheets

- Python
- SQL
- Git
- Shell
- R Project Packages Most R packages have robust documentation, and is available at CRAN (The Comprehensive R Archive Network)
  - ReadXL to Read Excel Files
- R: ggplot2 & knitr (lesson with cheatsheets linked)

13.3 Help troubleshooting code

- Check language/platform-specific message boards to see if someone else has encountered a similar problem.
  - Python
  - R
  - SAS
  - SPSS
  - Stata
- Search StackOverflow
  - Programming community forum. Specify what language/platform in your search term (ex: "R plotting change axes")
  - Pro tip: use your favorite search engine, and prioritize StackOverflow.
- Use your favorite search engine and search the error code + name of software or the name of the software and the function or operator.
  - Example: "R error default.density need at least 2 points to select a bandwidth automatically"
    - This will often surface relevant StackOverflow posts
  - "R %>%" will tell you about this operator common to the tidyverse.
- Check documentation for specific function/package/program
  - User guides/docs for "official" add-on packages to software
  - Example: error/problem implementing "bmacoefsample" in Stata? → Find documentation for bmacoefsample - Posterior samples of regression coefficients - read syntax description

13.4 Did you like this style of learning?

Do you want to learn the ins and outs of computer programming, including memory allocation, logic gates, and more?

- Introduction to Programming Logic (Lynne O'Hanlon, 2000)
  - This is a very accessible and readable programming logic book. This was written by one of Kat's computer science instructors, to whom Kat is eternally grateful for a logical and structured approach to learning how to program.

Bibliography

Lynne O'Hanlon (2000). *Introduction to computer programming logic*. Kendall/Hunt Pub. Co.

Ranganathan, S. (1931). Five laws of library science. Online; accessed 2025-03-06.

Silva, P. and Maral, A. (2013). Leaf. UCI Machine Learning Repository.