Programming Logic for Non-Programmers

Kat Koziar and Stephanie Labou

2025-03-02

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 *(Kat's note: data structures might be too early, but maybe not.)*

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.

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.*

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

ingredients	quantities	importance

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*.

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]`?

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.

5.2 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:

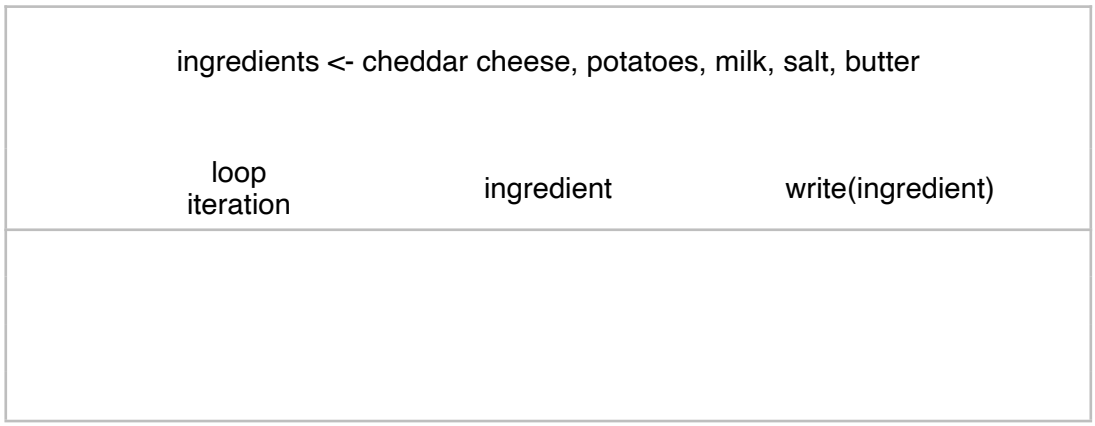
```
ingredients <- cheddar cheese, potatoes, milk, salt, butter

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:

ingredients <- cheddar cheese, potatoes, milk, salt, butter		
loop iteration	ingredient	write(ingredient)
1	cheddar cheese	cheddar cheese
2	potatoes	potatoes
3	milk	milk *(yes, this is a little redundant)*

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

```
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"
```

5.2.1 Exercise - Loop Trace

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
3	2	4
4	3	6
5	4	8
6	5	10

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

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.

5.2.2 Exercise - Trace Nested `for` Loops

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
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			

5.3 Other Types of 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

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.

5.3.1 While Trace

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.

5.4 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.

5.4.1 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).

5.4.2 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).*

Chapter 6

Conditionals and Making Choices

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 a generic programming format, 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 generic programming format, 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
```

Exercise: 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 microwave vs stovetop popcorn. Write out an `if, else if, else` statement that includes an air fry scenario.

Example statement

```
IF (popcorn_type is microwave)
  use microwave
ELSE IF (popcorn_type is stovetop)
  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 lots in the details.

6.1 Boolean Operators

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.)

Programming uses Boolean operators such as:

- `and` (may also see `&` used)
- `or` (may also see `|` used)
- `not` (may also see `!` to indicate negation, for instance `!=` for "not equal")
- `equals` (also `==`)
  - [SL note: most Boolean operator lists include AND, OR, NOT - should we include EQUALS here or keep as separate part of testing true/false?]

6.2 Exercise - `ifelse` trace

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!

Exercise

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

6.3 Caveats

6.3.1 Order of operations

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

[[**insert example here**]]

6.3.2 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, it will wait to run until you have "completed your thought", so to speak, and provided the syntax (here, `)`) indicating that this statement is complete and the program is ready to run.

Alternately, if you have mismatched parentheses, the result may be an error.

```
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"))"
```

6.3.3 Formatting may vary

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.

For example, Python expects `:` at the end of each section of an `ifelse` statement, uses `elif` for `else if`, and requires indentation to indicate action of each section:

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

In contract, R makes use of curly brackets `{}` to indicate each section of an `ifelse` statement and while indentation is a convention for readability, it is not technically required for the code to run:

```
if (x == y) {
  print("values are equal")
} else if (x > y) {
```

```
  print("x greater than y")
} else {
  print("x must be less than y")
}
```

[[Note to self: add another 1 or 2 language examples here to show similarities and differences in formatting]]

Chapter 7

Functions

A "function" in programming is a piece of code that does a specific task.

Consider the example of calculating the mean of a set of values. On the one hand, you could write out code to manually add each value and divide by the number of values.

```
values = (2, 4, 7, 5, 9)

mean_value = (2 + 4 + 7 + 5 + 9) / 5
```

On the other hand, if you plan to take the mean of a set of values often, it would be easier to have a function that could do this task without you needing manually write it out every time.

draft example, needs work

```
values = (2, 4, 7, 5, 9)

sum = 0

for x in 1 to length(values)
  sum = sum + x

mean = sum / length(values)
```

In our cheesy mashed potatoes example, a function would be equivalent to having an in-home chef who can make you cheesy mashed potatoes on demand, as long as you provide the raw ingredients. In programming parlance, these raw ingredients would be the "arguments".

7.1 Arguments

In a function, "arguments" are the inputs that you can specify. For cheesy mashed potatoes, this could be specifying what kind of cheese to use, or how many servings you want to make.

For our `mean()` function example, the primary argument would be what kind of input to put in: `mean(x)` where `x` is a list or vector of values. Another argument might be an option for how to handle missing or `NA` values: `mean(x, na.rm = FALSE)` where the default value of `na.rm` 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`.

It is important to understand (a) what arguments are in a function and (b) what arguments are optional and what the defaults are. A function will run as long as the non-optional arguments are completed (that is, input is specified), but the resulting output may not match expectations unless you understand what other optional arguments were included, and what the default values were.

Putting this into practice, let's return to the `mean()` function where our arguments are input (mandatory) and handling of `NA` values (optional).

If we only input the mandatory argument of specifying input:

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

The output of this function would be `NA`, because if we retain the `NA` values - which we do by default in the `mean()` function, then the mean will necessary be `NA`. For a computer, the sum of a set of values plus `NA` will always be `NA`, and taking the mean of `NA` will also return `NA`.

Conversely, if we specify that we want to remove `NA` values from our mean calculation:

```
values <- c(2, 4, 7, 5, 9, NA)
mean(values, na.rm = TRUE)
```

Now our output is `5.4` which is the mean of `values` excluding `NA`.

7.2 Libraries

Programming languages will have a variety of functions for common tasks ready to use without any additional work. These are called "built in" functions and are available to use right away.

However, 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.

It gets a bit tricky to extend our cheesy mashed potatoes example to more complex topics like this, but libraries are kind of like this: You want to make cheesy mashed potatoes from scratch, but don't have the time to do so. Luckily someone else has created instant mashed potatoes, so all you have to do is add boiling water and stir. You can add your own flair to the resulting potatoes if you want, but the important part is that someone else has done a lot of the work and you can build off it, rather than doing it all yourself.

You can also think of programming libraries as serving a role similar to actual libraries. For instance, you don't need to memorize every historical event, or write your favorite novel from scratch - you can check out a book from a library to read and learn more!

In the same way, your computer and your programming language of choice doesn't need to always have every single function on hand, which would take up a lot of space. Instead, it can "check out" (load) a "book" (collection of functions) created by another person. You can then use those additional functions the same way you would any function.

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 "book" 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.

7.3 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.

7.3.1 Caveats

7.3.2 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 `A` then `B` and then use `mean()` as a function, we will be using the `mean()` function from library `B`.

```
load(A)
load(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(B)
load(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.

An alternative 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(B)
load(A)

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

#result will be `NA`
```

7.3.3 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.

7.4 Activity reprise for functions

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()

# set arguments and defaults

make_popcorn(type = microwave, time = 2 minutes, flavor = movie_theater_flavor, seasoni

# example in use
# keeping all defaults, but changing time because this microwave
# is known to burn popcorn at recommended time

make_popcorn(time = 1.5 minutes)
```

Chapter 8

Comments and Names

Concept of comments, naming of variables and variables Briefly touch on common conventions (like `df` for dataframe); ask about disciplinary conventions for abbreviations or naming

[SL: kind of moved this into 'caveats' of functions, flagged to discuss]

Chapter 9

Common Issues

Some common issues

9.1 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`.

9.2 Spelling and capitalization matter

The variable `x` is different from the variable `X`. Likewise, a function `mean()` and a function `Mean()` would be separate functions.

9.3 Special characters and 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

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.

9.4 Ending a code chunk

Ending a statement (needing ; or other conclusion)

9.5 Order of operations

overwriting/ variable will be whatever most recently set as

9.6 Others [work in progress]

Closing quotes and parenthese
Direct comparison of multiple syntax, so like the same task in R, Python, C, SQL, Stata, Java You don't need to memorize specifics! Reading and writing data / files
Syntax is going to be specific to a language, or package within a language
Show some examples of reading/writing data in R, Python, Stata, SQL

Chapter 10

Practice

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

10.1 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 separate out functions (as common tasks across programming languages) from data-specific 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 - object/variable, or function?
- 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 Google) 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. 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.

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).

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

Another useful 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, maybe subsetting - we want to understand the order of operations of each thing that is being done.

This is a useful skill and especially helpful when troubleshooting. For instance, if there is an error or unexpected result, a first item to check is often: are functions, especially loops and conditional statements, started and ended properly in matched fashion?

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.

The code may look intimidating! But remember, we're less concerned here about understanding every detail about what the code is doing, and more about understanding where each loop and conditional statement begins and ends.

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:
- (sneaky third for in line 43?)
- 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 requireme
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")
```

This needs a color coded image to clarify

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)
- 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)

10.3 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.

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 = ifelse(survey_wave %in% c("first", "second"), "phone", "in p
```

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?
  - 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 Google) 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
  - 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".

10.4 add example for C/C++

10.5 New functions in Stata

We'll conclude with a challenge in 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!)

So, 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 Stata 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.

Chapter 11

Recap and Consultation Tips

Recap - you won't be an expert, the idea is to build up your skillset

11.1 Approaching Consultations

How you may approach consultations - prepare in advance knowing specific question, evening seeing code in advance; have student talk through their code Be clear with what you can and cannot do. Helping with programming vs statistics (for when they ask for help with interpreting something Full disclosure, I am not a statistician) Troubleshooting vs consult Ok to say you don't know! Point to documentation and learning resources

11.2 Three Areas For Errors

Code not running at all → often a syntax error

Running unexpectedly / unexpected output

input

logic

output

Chapter 12

Flextables

This reflects nested for loops of `row+column`

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

This reflects nested for loops of `(5*(row-1))+column`

Index	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	1	2	3	4	5
Row 2	6	7	8	9	10
Row 3	11	12	13	14	15
Row 4	16	17	18	19	20
Row 5	21	22	23	24	25

Chapter 13

Resources

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

- Introduction to Programming Logic (Lynne O'Hanlon, 2000)

Chapter 14

BD Demo Introduction

Bookdown reference: https://bookdown.org/yihui/bookdown/usage.html

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 2. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 15.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

{r nice-fig, fig.cap='Here is a nice figure!', out.width='80%', fig.asp=.75, fig.align='center'}

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure `\@ref(fig:nice-fig)`. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table `\@ref(tab:nice-tab)`.

***Kat's note**: This is not a nice table. Flextables are better.*

{r nice-tab, tidy=FALSE}

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package `[@R-bookdown]` in this sample book, which was built on top of R Markdown and **knitr** `[@xie2015]`.

14.1 Keeping this below for easy reference while we get used to the bookdown format

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

{r eval=FALSE}

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.org/tinytex/.

{r include=FALSE}

```
# automatically create a bib database for R packages
knitr::write_bib(c(
  .packages(), 'bookdown', 'knitr', 'rmarkdown'
), 'packages.bib')
```

Chapter 15

BD Demo Methods

We describe our methods in this chapter.

Math can be added in body using usual syntax like this

15.1 math example

p is unknown but expected to be around 1/3. Standard error will be approximated

$$SE = \sqrt{\frac{p(1-p)}{n}} \approx \sqrt{\frac{1/3(1-1/3)}{300}} = 0.027$$

You can also use math in footnotes like this[1].

We will approximate standard error to 0.027[2]

[1] where we mention $p = \frac{a}{b}$

[2] p is unknown but expected to be around 1/3. Standard error will be approximated

$$SE = \sqrt{\frac{p(1-p)}{n}} \approx \sqrt{\frac{1/3(1-1/3)}{300}} = 0.027$$

Bibliography

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

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