Programming Logic for Non-Programmers

Kat Koziar and Stephanie Labou

2025-02-18

Contents

1 About **5**

2 Introduction **7**

2.1 Building a Mental Model 7

2.2 A Note on Syntax 7

3 Algorithms **9**

3.1 Breakout Activity 9

4 Data types **11**

4.1 Exercise - Who knows 11

5 Loops **13**

5.1 Exercise - loop trace 13

5.2 Exercise - populate an array 14

5.3 Other Types of Loops 14

5.4 Caveats 15

6 Conditionals and Making Choices **17**

6.1 Boolean Operators 18

6.2 Exercise - `ifelse` trace 18

6.3 Caveats 19

7 Functions **23**

7.1 Arguments 23

7.2 Libraries 24

7.3 Accessing functions in libraries 25

7.4 Activity reprise for functions 27

8 Comments and Names **29**

9 Common Issues **31**

9.1 Difference between = and == 31

9.2 Spelling and capitalization matter 31

9.3 Special characters and words 31

9.4 Ending a code chunk 32

9.5 Order of operations 32

9.6 Others [work in progress] 32

10 Recap and Consultation Tips **33**

10.1 Approaching Consultations 33

10.2 Three Areas For Errors 33

11 BD Demo Introduction **35**

11.1 Keeping this below for easy reference while we get used to the bookdown format 37

12 BD Demo Methods **39**

12.1 math example 39

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 not only the logic behind programming, but 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 structural 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

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 to select 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

- enchiladas?
- quesadillas?
- cheesy mashed potatoes?

3.1 Breakout Activity

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

Sounds simple, right? "Make popcorn" is something most people have at least a general understanding of how to do. But in this scenario, you're providing step-by-step directions for a computer to understand how to make popcorn and a computer would need to be told **every action** to take, in order to

successfully make popcorn.

Your task is to write out each step needed 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?
- How much time is needed to make popcorn?
- What should the computer do if the popcorn starts burning?

Chapter 4

Data types

talk about data types and variables.

use food analogy

4.1 Exercise - Who knows

Chapter 5

Loops

Drawing from Introduction to Programming Logic(Lynne O'Hanlon, 2000) Start with `for` loop as first function term "Populate an array from a for loop" as an early example; pg 376 blank table with good early exercise. Side note about infinite loop, importance of settling bounds What happens if you don't tell loop to increase? Examples of `for` loops in multiple languages (Python, C, R)? To decide: all theory and then all practice, or theory/practice/theory/practice? Also writing out result of each iteration otherwise you'll only get the last result (a common problem)

5.1 Exercise - loop trace

Grab a pen and paper and write out the output of each iteration of the loop:

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

Loop output

1
2
3
4
5

5.2 Exercise - populate an array

Let's use a `for` loop to populate an array.

We'll start with an empty array called `table` with 4 rows and columns named A, B, C, D, and E:

A	B	C	D	E

We want to populate this table with numbers, starting with 1 and increasing sequentially (1, 2, 3, 4, etc.). We want to fill each row completely before moving on to start filling the next row.

Our `for` loop:

[[adapt programming logic book exercise for loop]]

Grab and pen and paper (or a spreadsheet program like Google Sheets or Excel) and manually fill in the empty cells in the array.

- What is the number in cell A1?
- What is the number in cell B2?
- What is the number in cell D4?

Your filled array should look like this

A	B	C	D	E
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

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 at the start of the loop to determine, in advance, when the loop will stop. For example, recall the initial for loop:

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

An equivalent `while` loop would look like:

Another type of loop is a `do while` loop (check at end) and an `until` loop (check at end).

5.4 Caveats

Loops are a frequently encountered concept in programming and come with their own set of common issues.

5.4.1 Infinite loops

When condition is never false or otherwise has no break

5.4.2 Overwriting outputs

```
for x in 0 to 5
  answer = x

write answer
```

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 program to check a condition and the next step depends on whether the condition is true, or false. We can think of this as a narrative statement: "If condition A is true, do action X. If condition A is not true, do action Y."

In programming format, this narrative statement would look like:

```
IF conditionA == TRUE, do X
ELSE do Y
```

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

We are not limited to a single `TRUE/FALSE` check in an `if else` statement, where actions are limited to "x if true, y in all other scenarios".

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. Our updated narrative statement might be: "If condition A is true, do action X. If condition

B is true, do action Z. If neither condition A or condition B are true, do action Y."

In programming format, this updated narrative statement would look like:

```
IF conditionA == TRUE, do X
ELSE IF conditionB == TRUE, do Z
ELSE do Y
```

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.)

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.

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

For the first trace, we will set the values of `x` and `y` as:

```
x <- 37
y <- 42
```

What will be the printed output from this `ifelse` section?

Answer

x must be less than y

What if we reset `x` and `y` to:

```
x <- 75
y <- 9
```

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.

```
[SL:insert pseucode to calculate mean without using sum() or len(), use a for loop maybe to call
```

7.1 Arguments

In a function, "arguments" are the inputs that you can specify. So for a `mean()` function, 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.

For example, 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` exlucding `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.

You can 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 Chapter 2 activity where you created an algorithm to make popcorn.

Call back to popcorn - if function is make_popcorn - what is that for? What are your arguments - is it kernels or is it a bag, what time? make_popcorn(type = kernels, time = 15 minutes, butter = TRUE, salt = TRUE) vs make_popcorn() with defaults import cookbook; cookbook.make_popcorn() ### From stephanie import make_popcorn() ### From kat import make_popcorn()

Libraries, packages, modules Example: min (R), max (Python), mean (SQL), regression (R, Python, Stata) Can look up documentation, most should specify arguments in each function and syntax, as well as defaults for arguments

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

Recap and Consultation Tips

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

10.1 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

10.2 Three Areas For Errors

Code not running at all → often a syntax error

Running unexpectedly / unexpected output

input

logic

output

Chapter 11

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 12.

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

```
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 11.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 11.1.

```
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 (Xie, 2025) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

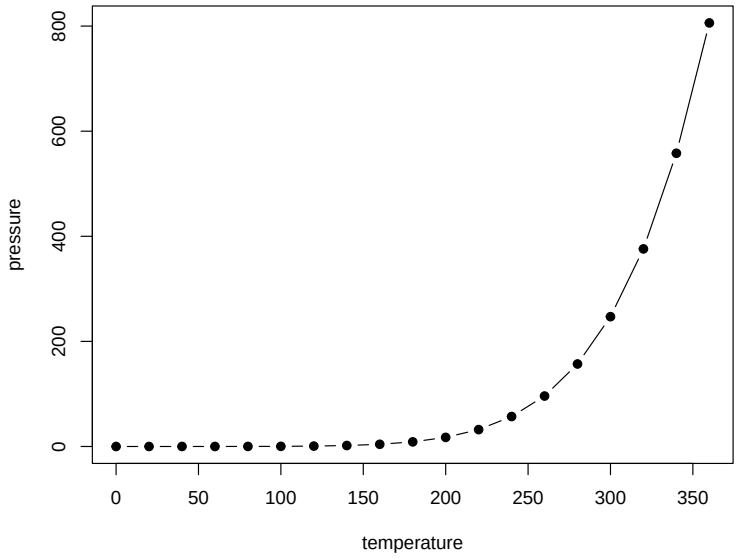

Figure 11.1: Here is a nice figure!

Table 11.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

11.1 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:

```
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/.

Chapter 12

BD Demo Methods

We describe our methods in this chapter.

Math can be added in body using usual syntax like this

12.1 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.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2025). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.42.