# **What type of interviews might you expect**

## Introduction

Getting a job involves jumping through several hoops. After compiling your resumé, identifying a few suitable job opportunities, filling in applications and researching companies, you will begin to receive interview offers. The aim of an interview is to find a good match between the company and the candidate. Every company is different, and every company will conduct the interview process in its own way.

In this reading, you will learn about what is involved in a coding interview, such as:

* What the screening process entails.
* The type of quiz questions you might encounter.
* What an online assignment might consist of.
* The details of a technical interview.
* And any take-home assignments that you might be asked to complete.

## Screening

The first contact you will have with a company will be a screening interview. A recruiter, having read your application, will usually arrange a phone call to discuss potential suitability. This stage aims to determine if you are genuinely interested in the role and whether you would be a good fit for the company. It is an excellent opportunity for you, as a candidate, to learn more about the role, the company and their hiring process. Some great questions to ask might include the following:

* What is the role?
* What coding language does the company typically use?
* What is the interview process, what types of interviews will be conducted, and how many will there be?
* What is the nature of the project the proposed role is for?
* What is the typical makeup of the team?
* Who will be on the interview panel, and what is their role in the company?

Remember that the interviewer wants to give you the opportunity to make a compelling case about why you should be hired. They should be more than willing to provide you with any information that might help you better prepare.

From the company perspective, the screening will typically look at your soft or interpersonal skills. There will be no technical questions, and a Human Resources (HR) representative will typically conduct it. Presenting yourself well and demonstrating your social abilities, like listening and question-answering, is essential.

## Quiz

A quiz is a very straightforward way of establishing if a candidate knows how to code. Typically, they are conducted early in the interview process and act as a filter to reduce the number of interviews a company must conduct. You should not expect deep or extended questions; instead, the purpose is to determine whether the candidate has a good grasp of the basics. Some questions could include the following:

* How do you test for nulls in an array in a language of your choice?
* What is the space complexity for Quicksort?
* Which data structure would you use to store a list of keys and values?
* What is a collision in hashing?
* What is the syntax to select a column name using SQL?
* What testing processes are you familiar with?

The questions will be job-specific, so the content may vary. However, they will be questions that should be answered with concise, direct responses. When preparing for an interview, it is prudent to identify which programming languages the company uses and any other technical insights that can be gleaned from the job posting and initial screening.

## Online coding assignment

You may be asked to complete an online coding assignment, depending on the job you apply for. This generally happens before a technical interview, as the results can be used as a point of discussion during subsequent interviews. However, the assignment could also occur after a successful technical interview to compare candidates.

This step allows you to approach and solve a clearly defined task in a non-pressurized environment. Similar in difficulty level to the quiz, it is a filtering step to ensure that you can work with the target technology and grasp the basics, like loops, Booleans, checks, and so on. Some coding assignments could be timed; in this situation, you would have access to Google and other reference material.

## The technical interview

This is also called the coding challenge. Typically, this will happen after the initial screening. It can either be virtual or in-person. The interviewer will present you with a challenging problem and give you a brief period to solve it. It is always best to discuss your thoughts aloud as you engage with the challenge. For example, before deciding between two data structures, explain why you feel one is more appropriate than the other for the task. Take the opportunity to demonstrate your general understanding of the topic and the task-specific expertise.

The difficulty level of the technical interview should be low. Interviewers know that you don't have access to Google and may have had to memorize some of the required syntax. Instead, these interviews are designed to demonstrate your problem-solving skills. Some technical interviews may be conducted using pseudocode. Additionally, technical interviews allow future colleagues to assess your ability to reason with code. They often include short, specialized prompts like:

* How would you explain technology X to a non-technical person?
* What is your favorite technology and why?
* What databases have you worked with?
* Tell me about a technical challenge you have overcome in a project.
* What projects have you worked on in your spare time?

These are generic prompts, and a technical interview will be tailored for a specific role. However, preparing a few answers describing your journey as a programmer is good practice. Review your projects and be clear on the various decisions you have made and why particularly when selecting one technology over another.

## The take-home assignment

If you get this far, you are doing very well. This may happen before or after the technical interview, as the interviewers may want to discuss your thought process around the solution you provided earlier in the process. If there are many rounds of interviews for the post, then it may happen between two technical interviews. A company should not ask you to do a take-home assignment unless they feel you are potentially a good fit.

You will be given several days to expand on a solution to the challenge. You could be asked to build an application or programmatically solve a traffic routing problem. Other tasks might be project specific, such as, if you are going to work with vision processing, you may be asked to demonstrate that you can automatically detect objects in a video feed.

## Top tips

Conduct your research before every interview and review each one on completion. Most companies are happy to provide feedback after your interview, so always ask for this and use the points made as a starting point to be better prepared for the following interview. Remember, practice makes perfect.

## Conclusion

In this reading, you learned about what is involved in a coding interview, such as what the screening process entails, the type of quiz questions you might encounter, what an online assignment might consist of, the details of a technical interview, and any take-home assignments that you might be asked to complete.

Several types of interviews have been explained. However, the company you are applying to will have its preferred format, which could take one or more of these approaches. The most important takeaway is that “every interview is a learning process.” Suppose you missed a question or were queried on a technical concept you aren't familiar with. In that case, the opportunity is there for you to learn more about it post-interview.

# Pseudocode step by step

## Introduction

A valuable tool in a programmer's toolbox is pseudocode. In this reading, you will learn about why you should use pseudocode, when it should be used, and how to write pseudocode.

## Why should you use pseudocode?

Pseudocode is a legitimate first step when starting to devise a solution. Pseudocode is a high-level representation of ideas written in a way that looks like code. Fundamentally it can be an aid to highlight for yourself what elements a program should include.

Each line will give you a moment to pause and consider what is required to achieve a given outcome. When writing an application, a decision made in step 3 could impact how step 6 should be coded. Each stage has an element of constraint that informs how a subsequent step will be completed.

Consider that you are writing an application that must store data in step 3. You settle on an array before continuing. At step 6, you realize several lookups are required for an application. While writing pseudocode, it is easy to revisit step 3 and change the data structure to something more compatible with step 6, such as using a dictionary instead of an array. Slight changes could increase your overhead if an implementation has already begun because it alters how steps 4 and 5 operate.

## When should you write pseudocode?

* As a beginner, when plotting out the planned progress of your approach.
* As an experienced programmer, when attempting to wrestle with a complex problem.
* If you are trying to convey a concept to an influential audience, such as a team or potential clients.
* If you are signposting your work for future coders who may be maintaining the code or application you wrote.
* In an interview, when demonstrating your ability to reason out a problem.

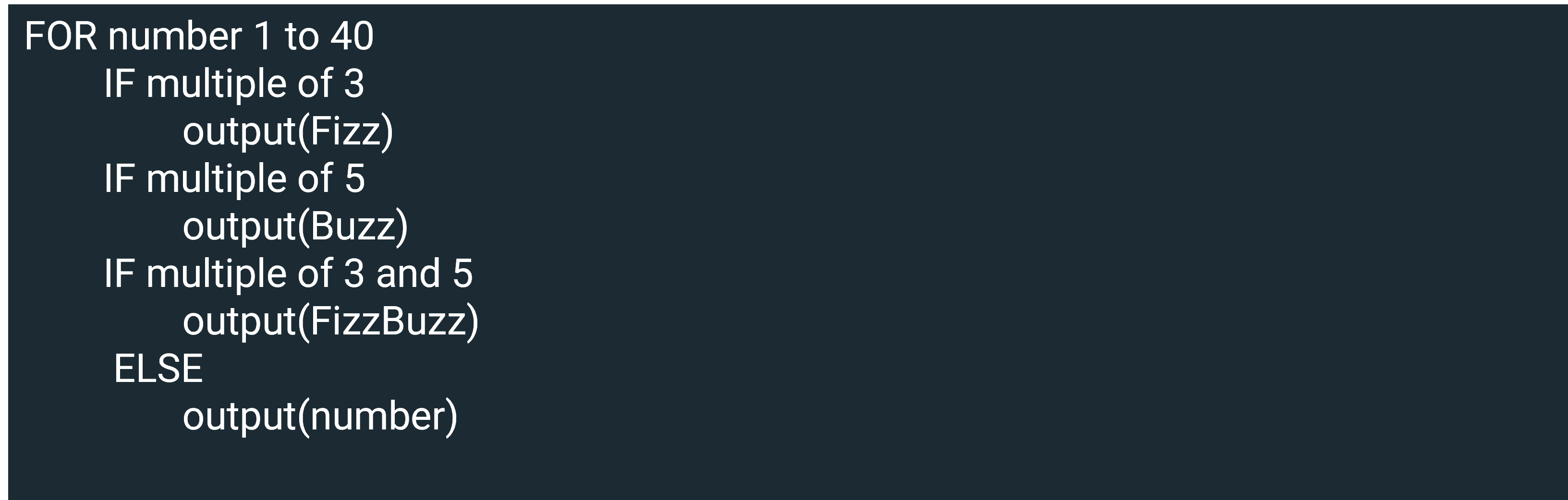
## How to write pseudocode

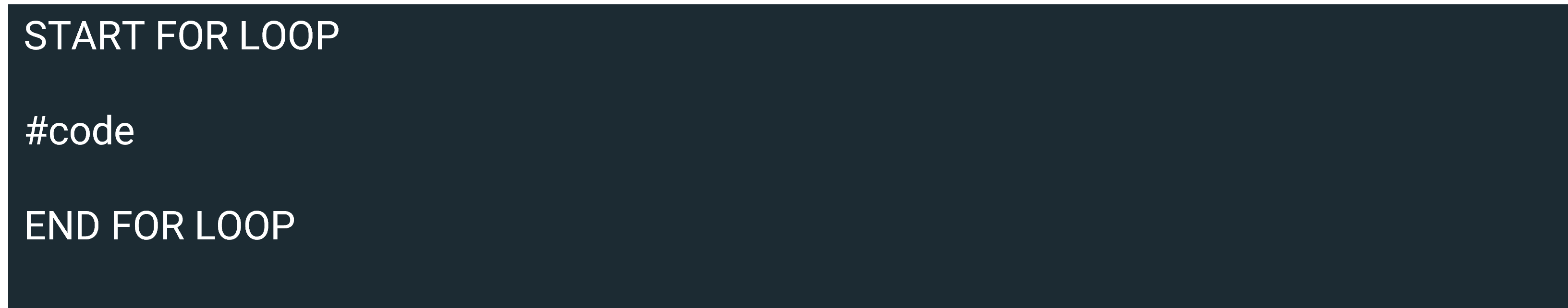
There is no one set way to write pseudocode. Each organization may have its own standard. In general, you can say that pseudocode can be considered to be any textual representation that outlines a program's operation.

Consider how to represent the FizzBuzz challenge used to test candidates' ability to reason in code:

Write a program in a given language that iterates over numbers 1 to 40. Print out a number for every number except multiples of three, in which case output Fizz. For multiples of five, output Buzz, and for multiples of 3 and 5 output FizzBuzz.

You might start by representing each requirement as a line of pseudocode:

The key to this assignment is knowing how to order the conditional statement. Line 1 indicates that there will be an iteration. In this instance, an indent demonstrates that the subsequent eight lines of code are part of this iteration. Alternatively, the following block could have been added:



It is clear that there are three conditional statements and then a catchall else clause. It is possible to visualize the code's outcome by looking at the pseudocode. The else statement will work fine, it is only to print out non-multiples of 3 and 5, but there is an issue with how the code will handle the number 15, printing an instance of Fizz, Buzz and FizzBuzz.

Examining the question's first instance as pseudocode enables you to spot where the issues might be. Visually, it is far easier to see how the conditional statements should be organized. First, the crux of the question is in what order the conditional statements should be placed and to use exclusionary conditional statements. In the above instance else if was used in place of just if. Second, the ordering of the statements has an important impact. You can try the output of above by running the code found below.

for number in range(40):

if number % 3 == 0 and number % 5 == 0:

print("FizzBuzz")

elif number % 3 == 0:

print("Fizz")

elif number % 5 == 0:

print("buzz")

else:

print(number)

## Conclusion

In this reading, you learned about why you should use pseudocode, when it should be used, and how to write pseudocode.

The scope for pseudocode goes beyond your remit to grasp a new programming language. It is a valuable practice that helps visualize the code's flow. It outlines what skills or libraries may be required to complete a task.

Software engineers use it at the start of their journey to gain insight into how a program might flow. Senior engineers use it to demonstrate ideas to a team. There is no one way of writing it, but the style you settle on will resemble the structure of the programming language you like best.

# Interview tips

## Introduction

In this reading, you will learn how to prepare for an interview and get some essential tips about presenting yourself in person and at virtual interviews.

Before successfully starting a job at a company, you must demonstrate that you are a suitable candidate for the role. Getting a job can be challenging and may require attending several interviews before finding a position suited to your skill set. Ideally, you are looking for one where you are the strongest candidate for the role. It is essential to realize that even if you are well-suited to a position, there is always the possibility that a more experienced candidate is also being interviewed. So, even though you did not get a role, you could still have had a good interview.

It may help to think about interviews as a stepping stone on your journey to a rewarding full-time career. Regardless of the outcome, always ask yourself, did I present myself to the best of my ability? And, can I learn things from this interview that can help me in subsequent interviews?

## Prepare

Here are some of the standard questions that you may encounter in any given interview:

* Tell me about yourself.
* Why do you feel that they should hire you?
* What are your major strengths?
* Or, what are your major weaknesses?
* What pay are you expecting?
* How do your previous experiences make you suitable for this role?
* What do your friends say about you?
* Why do you want this role?
* How have you dealt with conflict in the past?

Knowing these questions are coming is advantageous because you can prepare for them. After reading this list, can you write a concise answer for each? You will find the answers to many of these questions in your resumé. A good tactic is to review your resumé before an interview to see which aspects are related to the company and role.

Additional answers worth preparing for may relate to the company itself. Where is the company based, and what do they do? How does your previous employment overlap with their current needs? Research the same role in other companies. Know the market salary for the position if you are asked about a pay rate. The interviewer will know the answer, so being able to answer this successfully demonstrates your preparedness. Learning about a company displays an eagerness to work there. Finally, consider that a weakness may also be a strength. "I focus excessively on details" can be damaging if there are tight deadlines; however, it is a strength if the job requires meticulous attention to detail.

## Soft skills

Conflict resolution is an essential soft skill to possess. Invariably in your working life, you will encounter a situation where your approach or goals do not align with those of a colleague, employer, or customer. Managing these situations helps create a harmonious working environment. Be cognizant of when you have clashed with people and how you resolved it. Did you find this approach to be successful? If not, then do some research on how to deal with conflict in the workplace.

## Presentation

It's frequently said, "dress for the job you want, not the job you have." How you present yourself can subconsciously convey much information about you. Be neat and respectable. Your future work colleagues may be on the panel, and you want to make a good impression. The attire you wear should be suitable for the role. Pride in appearance can be equated with pride in a role.

## Demeanor

Display readiness for the role. Colleagues will want to see someone interested in being there. Be calm and relaxed when answering questions. It is never a good policy to interrupt a question. Listen until the question is finished, and take a breath. Competency questions require a STAR answer (Situation, Task, Action, Result). Display pride in previous work-related experiences you've had. And have some questions prepared. In an interview, the interviewer should always allow you to ask them a question. Take this time to learn something about what is being sought. What conditions might be more favorable for you? Are the company's goals aligned with yours, and can you convey that?

Be authentic in your actions. It is better to be truthful and honest when applying for a post. Exaggeration may lead to a follow-up interview, but it is better to be capable of any position you intend to work in.

## Virtual interviews

Many interviews are conducted virtually. Virtual interviews are challenging as it can be more difficult to read body language over video. Due to sound quality, it is essential to refrain from speaking over someone. Ensure that all your tech is working. Have your computer plugged in and a working set of earphones in case of sound issues. Ensure that your video is working and compatible with the platform. Sometimes, a computer needs to change permissions before you can use a camera. And remember to test the connection speed.

As with an in-person interview, appearance is essential. Take some time to make sure you are suitably attired. Even though you have not gone into the office, dress as though you have. As with an in-person interview, be aware of your body language. Sit up straight and be attentive. Though you may be in your home, it is still essential to conduct yourself with work decorum.

Find a quiet location that is free from distraction. It can be helpful to have a dedicated space to work in. This is necessary to ensure that during the interview, there will be no surprise guests doing housework in the background.

## Conclusion

In this reading, you learned how to prepare for an interview and got some essential tips on presentation and how to conduct yourself in virtual interviews. Despite your best efforts, you may not be successful if there is a more suitable candidate. You may also find that the role isn't as aligned with your skillset as you first thought. It is good to remember that regardless of the outcome, an interview is a learning experience. If treated as such, it is a stepping stone to a career that aligns with your goals and abilities. Being prepared and ready to pounce when the opportunity arises is the key to success.

# Testing your solution

## Introduction

In this reading, you will learn about how you can build testing into your take-home assignments and a coding interview, you’ll also explore the practice of writing unit tests.

Testing is a key part of software development. Ideally, the scope of your tests is comprehensive; however, there are always external considerations that can affect this ideal outcome. Historically, testing has often been done at the end, so if a production deadline is looming, tests can be the first thing to be reduced. Test Driven Development (TDD) is an approach developed to avoid this outcome. This discussion of tests will focus on two instances, a coding interview and a take-home assignment.

## Take-home assignment

If you have been allowed to take some code home, you may have time to implement more comprehensive tests.

There are many types of tests that you can choose to run. For example:

* Integration tests: These test how various components of an application interact with one another. An integration test will not be in-depth because the external dependencies will be simulated rather than using actual instances. Further, unit tests assess individual lines of code, while integration tests take more of a global approach.
* Functional tests: This is an automated test that proves that a system operates as expected. This test is concerned with the capabilities of the system.
* Regression tests: This is to test that a change does not cause an error in the existing code. It ensures that when a system change is made, it does not affect its operations.

The level of testing you employ will always be time-dependent. A crucial part of the process is creating a working application. Testing shows your thoroughness and ensures that the application will work. The degree to which you can include these strategies may be limited depending on how much time you have.

## Coding interview

Unit testing is an approach that confirms your code is working as expected. While many tests are written before code goes into production, you will not have an opportunity to write all these when engaging in a coding interview. Unit testing is a manageable approach that can be incorporated into your solution, demonstrating your attention to detail. It will demonstrate your good practice of testing your code.

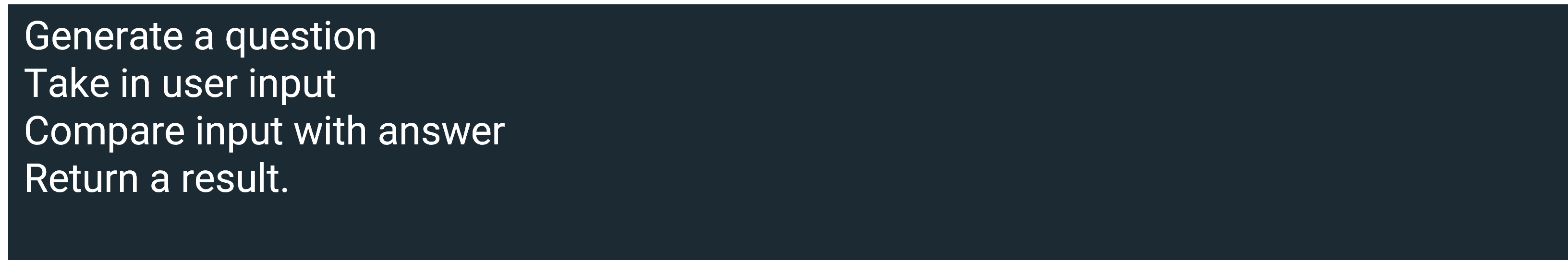
A unit test is less concerned with an application's overall operation. Instead, it tests that each of the individual components works as expected.

### Considerations when writing tests

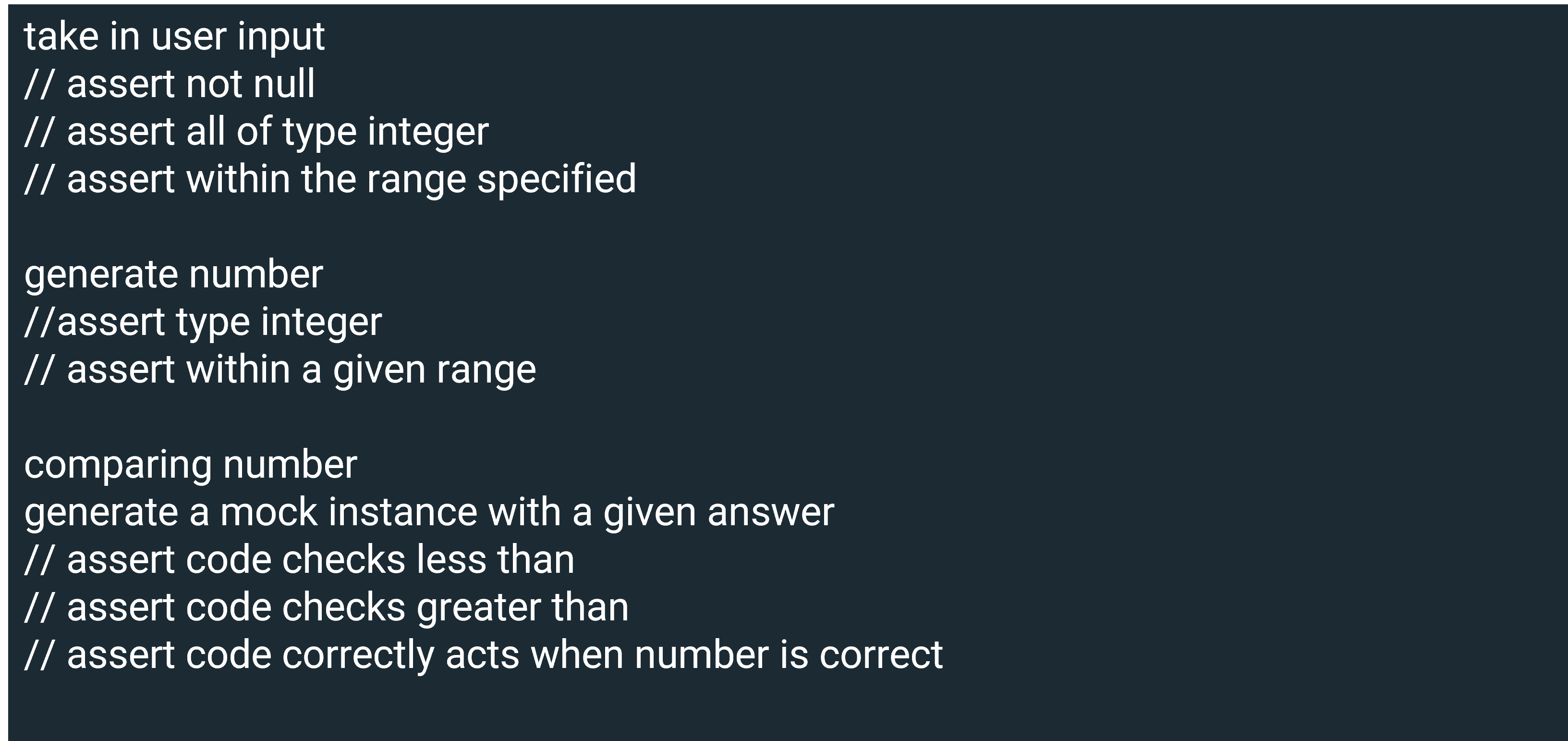
* Readability: Make the tests readable for other developers. This habit should be internalized regardless of whether you work in a team. Tests with a clear purpose identify problems should they arise. They also signpost what a section of code is supposed to achieve. This has the added effect of increasing maintainability.
* Clear outcomes: Your tests should, whenever possible, be deterministic. This is to say that the result should always be the same regardless of the conditions. A deterministic test will always fail when buggy code is written. Tests dependent on a combination of different conditions can be challenging to debug.
* Automation: Tests should always have the potential to be automated so that they can be run quickly whenever a change has been made to code. This is a cornerstone of CI/CD (Continuous Integration/ Continuous Development).

### Putting unit tests into practice

The defining solutions reading outlines the steps needed to make a number-guessing game. Below is a screenshot of the pseudocode used to develop a viable solution.



Taking this example, imagine that there are only a few minutes left in the allotted time, what would you focus on? The most basic unit tests that can be written are assert cases, which determine that what is given is as expected. Libraries of these are available in most coding languages.



The first step may be to write a test case to determine whether the user input is valid. Lines 2 – 4 outline some tests that can be applied to validate user input, checking that something was entered before the user pressed return. You should determine that it is of the correct type and will not crash the program. Further, ensure that it is within the specified bounds. Next, you should consider checking the random number generated. Is it of the correct type and does it fall within acceptable parameters? Finally, it would help if you thought about including checks that ensure the right outcome when the conditional statements are activated.

The process for writing unit tests is standardized. It is a good idea to practice implementing them when you are writing code. This will help you quickly familiarize yourself with the process and can highlight your mindfulness in an environment where you are testing your code.

## Conclusion

In this reading, you learned about how you can build testing into your take-home assignments and a coding interview and you also explored the practice of writing unit tests. After reading all of the above content you should now know that assessing your code is particularly important.

Often the practice of writing tests can be marginalized compared to the time spent implementing an application. To avoid this coding pitfall, ensure that you write tests as you code. While some testing can be in-depth and time-consuming, writing unit tests can be done quickly. This habit can lend weight to the argument that you are a fit candidate for a post. If you can incorporate unit tests into an assignment, particularly ones with time constraints, you will be well-prepared for a coding interview.

Working in binary

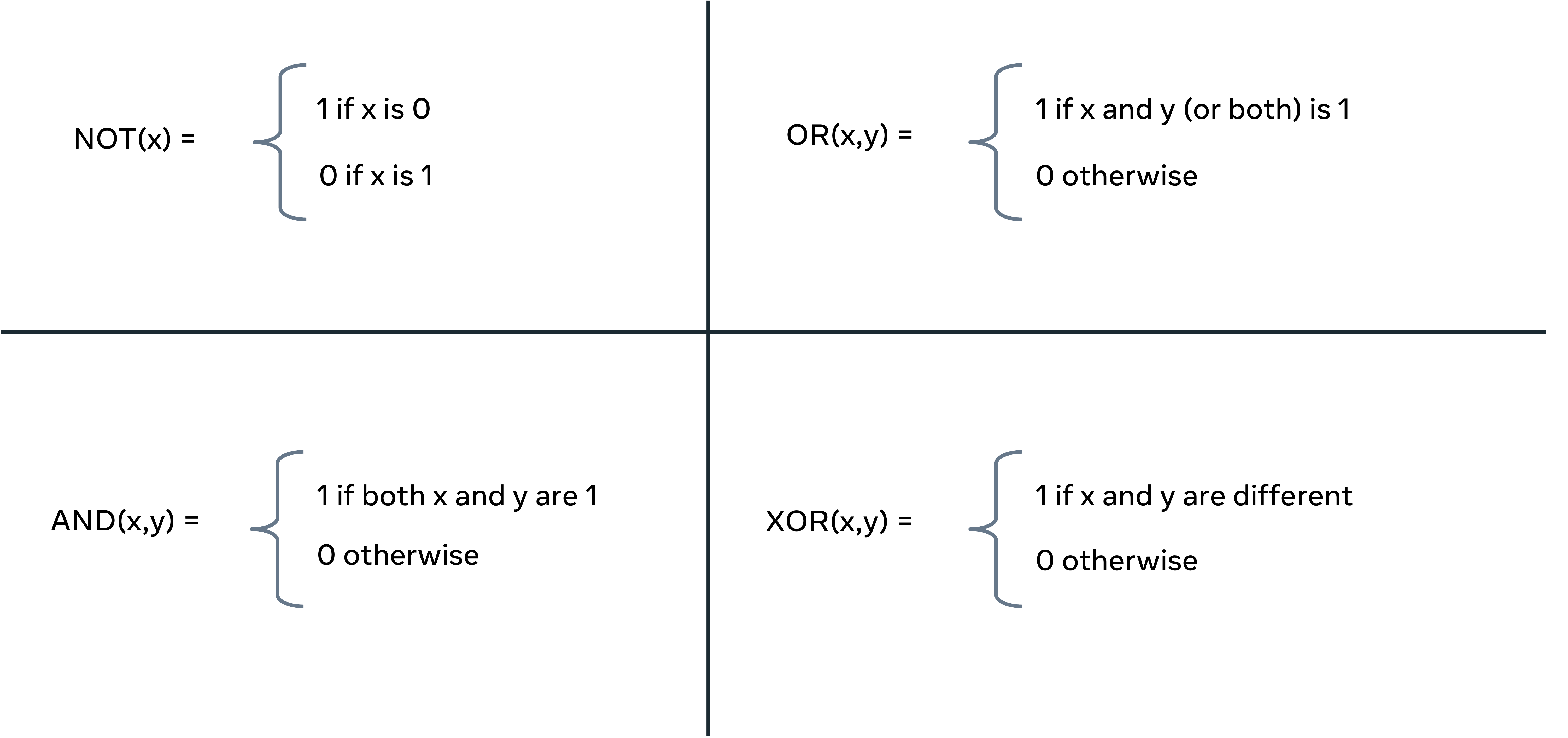
## Introduction

It is common knowledge that computers think in zeroes and ones. But, how this works in practice is a little more complex than one might initially think. You previously explored how computers utilize binary data to store and process information. In this reading, you will learn more about binary, including how to work with Boolean logic, truth tables and gates.

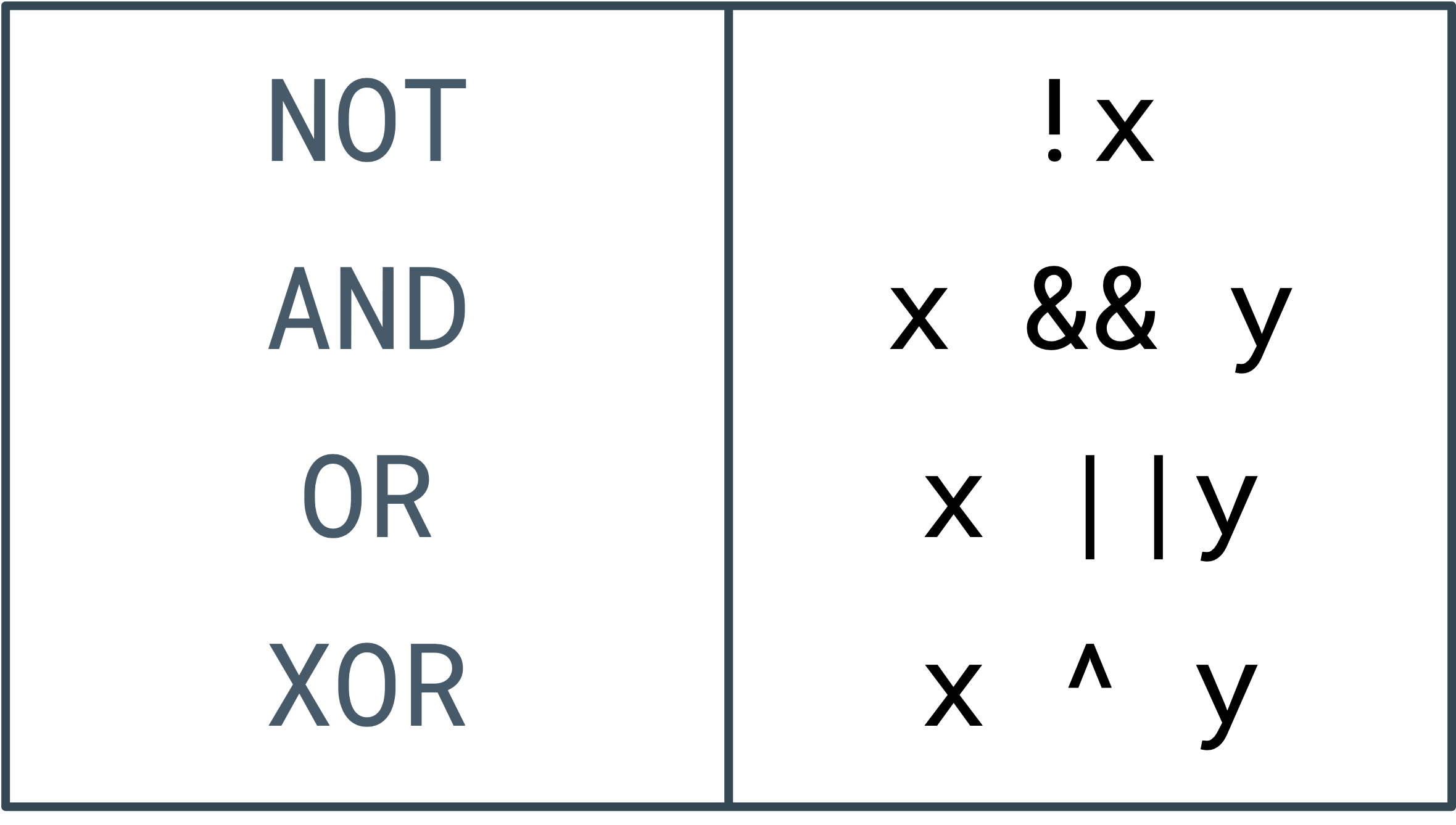
You may think that using only two inputs is restrictive. However, it is extraordinarily versatile and offers a broad range of options if combined with Boolean algebra and circuits.

## Boolean logic

A Boolean function maps two inputs to a value. These inputs are limited to two states. These states can be considered: on/off, true/false or 1/0. To define a Boolean function, you only need to specify the output from its inclusion. Here is an example of 4 functions:



NOT takes in one value x and resolves it to either a 0 or 1. OR takes two arguments (x, y) to generate an output of 1. Only one of the x or y values must be 1. If both values are 0, the output is 0. AND requires both inputs to be 1 before it can generate a 1. Finally, XOR will take any two values and determine if they are the same; if so, the output is 1. In all other cases, the result is 0.



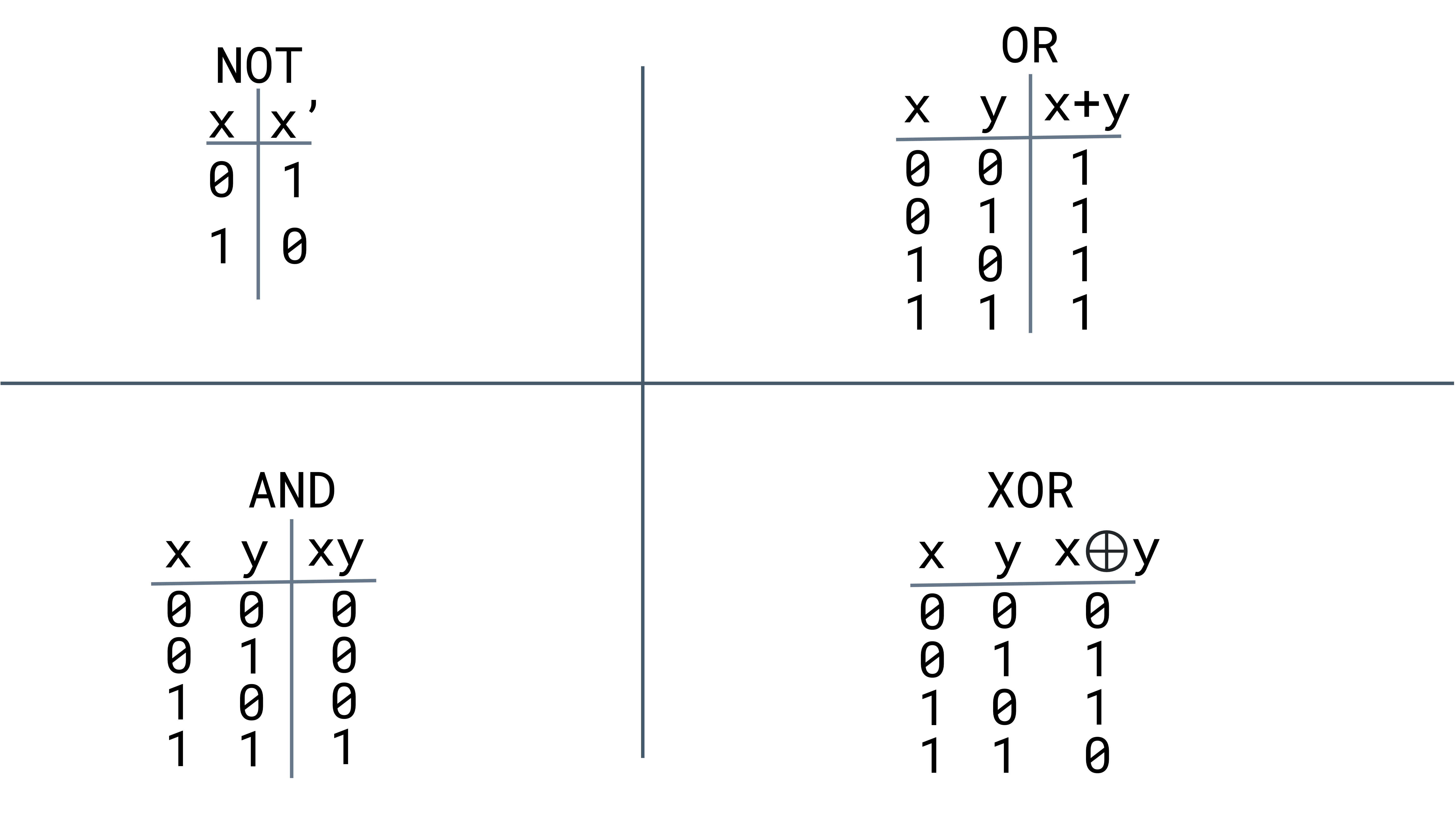
The same concepts can be represented computationally. In this table, the Boolean expressions are on the left, and their computational symbols are on the right. Most commonly, they are combined with conditional statements or iterators. So, you could expect code that resembles the following code snippet:



In the above code, a Boolean has been set to false. A do/while loop then continually executes the code found within the do until the value for x switches to true. Generally, you would be looking for a specific outcome, and using Boolean logic enables you to keep looking until the result becomes true. Notice how the NOT function is applied in the while loop to test if another iteration should exist.

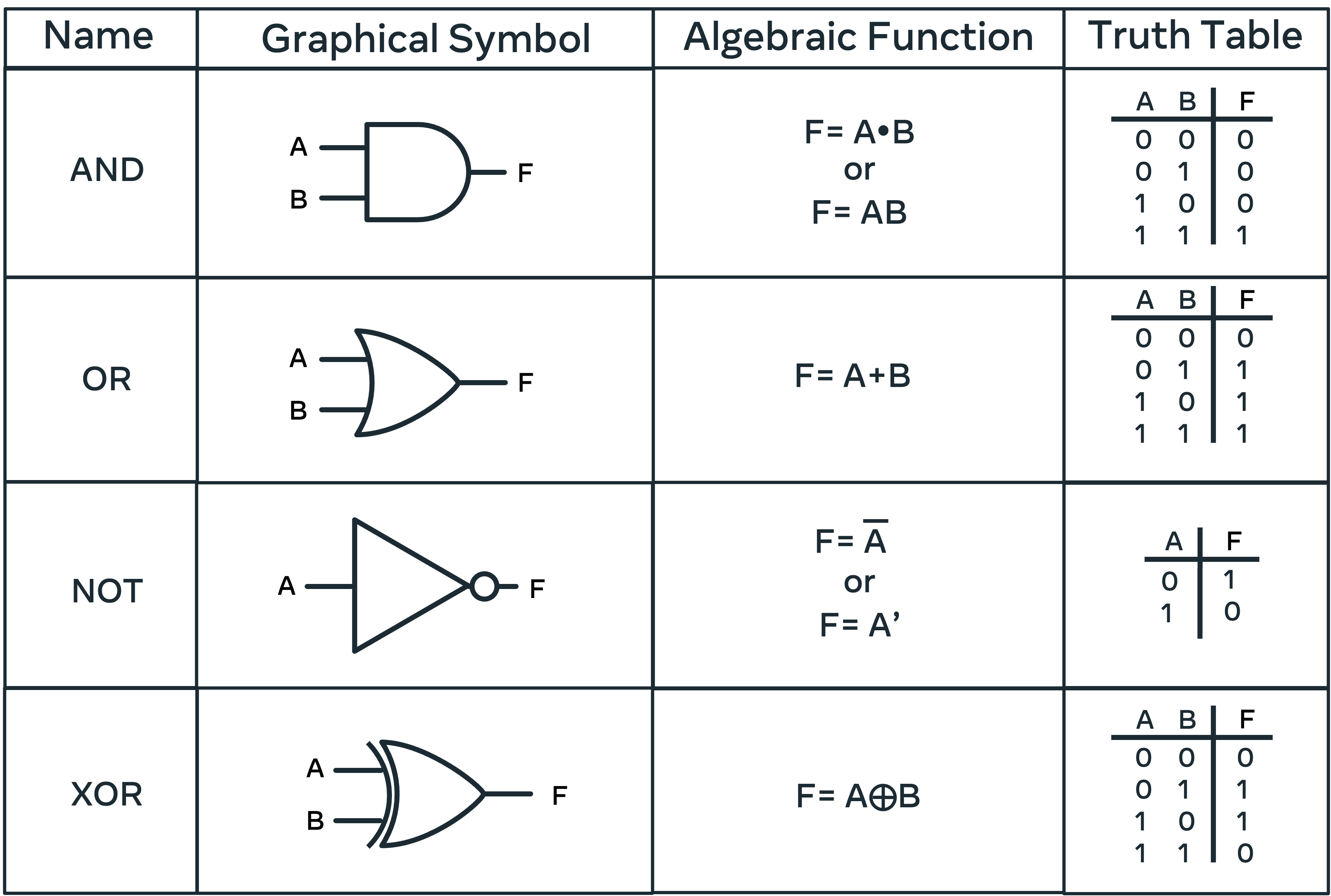
## Truth tables

Given the finite number of outputs to the Boolean functions, it is possible to plot all permutations to a table.



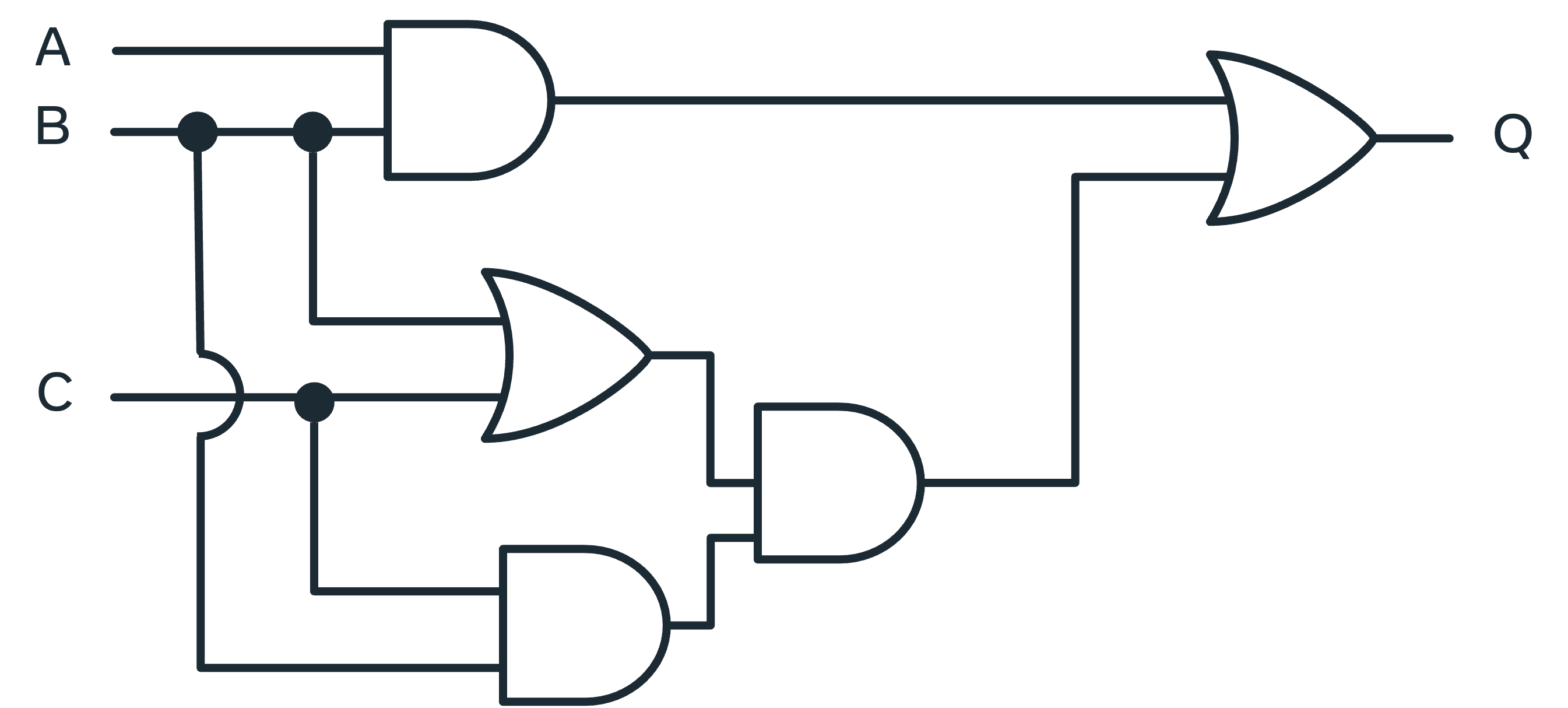
This image demonstrates all the permutations for each of the Boolean functions. The X and Y columns denote whether it receives a 0 or 1, and the XY indicates the eventual outcome. The NOT function only has one output and input, so the table is only 2 x 2. In comparison, the other three examples all take two inputs and generate a single result.

## Gates

The fundamental building block for digital logic circuits is the gate. The logical functions are then implemented through their interconnectedness. A gate is an electronic circuit that generates a Boolean output from its inputs.

In the above image, the four basic Boolean functions are illustrated. Graphical Symbol is how these gates are represented on diagrams. You will often encounter circuit diagrams that host a series of interconnected gates. The Algebraic function demonstrates how these gates are described when written as formulae.

Finally, the truth tables outline the outcomes from a given input to the gate. On a primitive level, each input denoted by A and B represents a current that can be passed through the circuit. These inputs may be connected to a switch, or a button, that can be activated causing a 0 or 1 to be transmitted. The expression of this Boolean logic builds when circuit gates are combined to form complex circuits. The final output Q is determined from the machinations performed on the inputs A, B, and C.



## Conclusion

This reading demonstrates how a simple 0 and 1 can be combined and amplified through Boolean logic, truth tables and logic gates to generate more complex outputs. You have learned more about binary, including how to work with boolean logic, truth tables and gates.

# Defining solutions

## Introduction

Computer science is all about solving problems. The invention of the computer has given us an unprecedented ability to conceptualize and overcome problems, real and imaginary. Before, you'd have to drop a plane from the sky to test if it will fly. Or gather weather data and make predictions based on patterns. Now it is possible to model scenarios realistically and create a solution virtually before sending up a weather balloon or attaching even one wing to an airplane. This reading is about how to define solutions to problem statements by articulating the problem, formulating a model and finetuning the solution.

## Problem statement

The first step to solving a problem is to articulate it. I need to achieve A, with tools T, given constraints C. By way of fleshing out the notion of good practices when engaging with designing a solution, let's consider a real-world problem. Say you have to design an app that will entertain a child by playing a guessing game. On the face of it, this seems like a straightforward task: computer + game = happy child. But, before you start, it is worth it to determine some details.

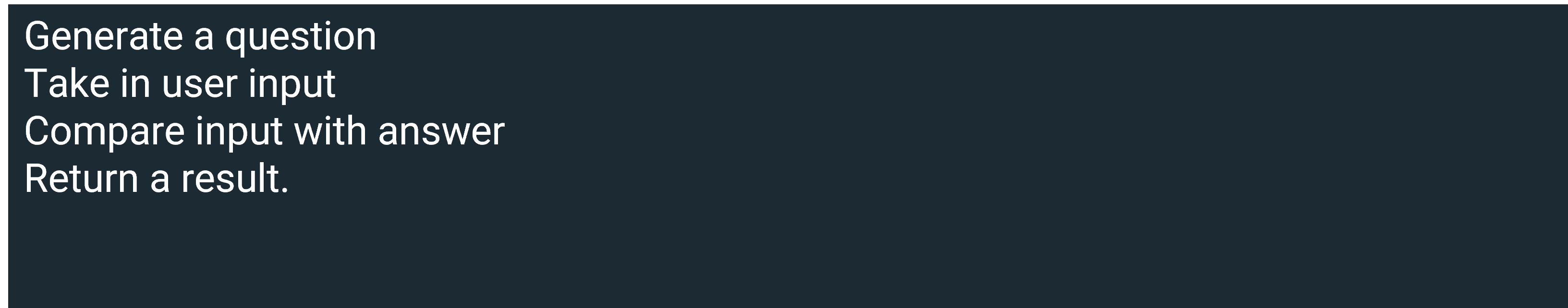
1. How will the child interact with the computer?
2. What level of questions should be asked?
3. From where are these questions generated?
4. How will they be stored?
5. How will the answers be checked?
6. How will the program start and end?

In this scenario, the issues are both hardware and software related. First, is the child of an age where safety measures are not required? In other words, it is important that the ideal solution is contemplated with consideration to how the final application is to be used. Giving a toddler an expensive laptop might buy a morning of peace, but then it might cost a month’s wages! This same thinking must be extended to the application that is visualized. Will your player be engaging with your application with a strong internet signal? Are there other constraints like the output needing to be compatible with different types of mobile phones? Is there a specific operating system or browser that must be catered to? These are all questions you need to consider before you start creating the app.

## Formulate a model

Now that there is an idea of some of the constraints on the project, potential solutions can be proposed. Let's imagine the child is of a computer-friendly age, the application need only be run on a laptop, all storage will be done locally (so there are no internet connectivity issues) and the input can be done using a keyboard on the command line. Great, now it is possible to further explore the project.

Having established a scope for the project it is time to consider what it is the computer is required to do. An algorithm can be defined as a precise sequence of instructions to solve a problem. It can be helpful to first generalize the problem and then more precisely outline the sequence of required instructions before coding an implantation. This is like taking an algorithmic approach to solving the issue. A programmer will often gain an intuition on what the solution should look like by first sketching out the problem using pseudocode. This can be a text-based description that details the requirements.

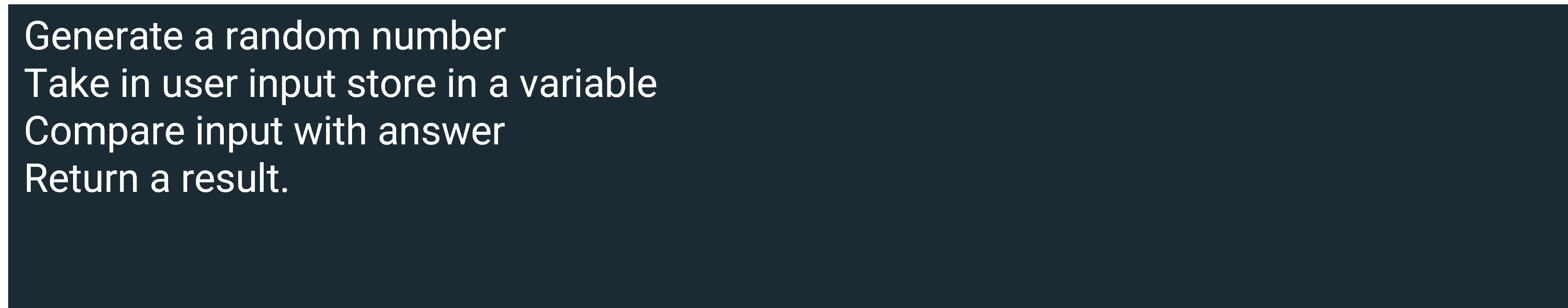


Plotting out the steps in a list is a good first step. Now, consideration can be placed on how to generate a question that is age appropriate. Dynamically generating questions from an online encyclopedia or some other online source might be time-consuming. Alternatively, you could identify a source of already compiled questions and answers.

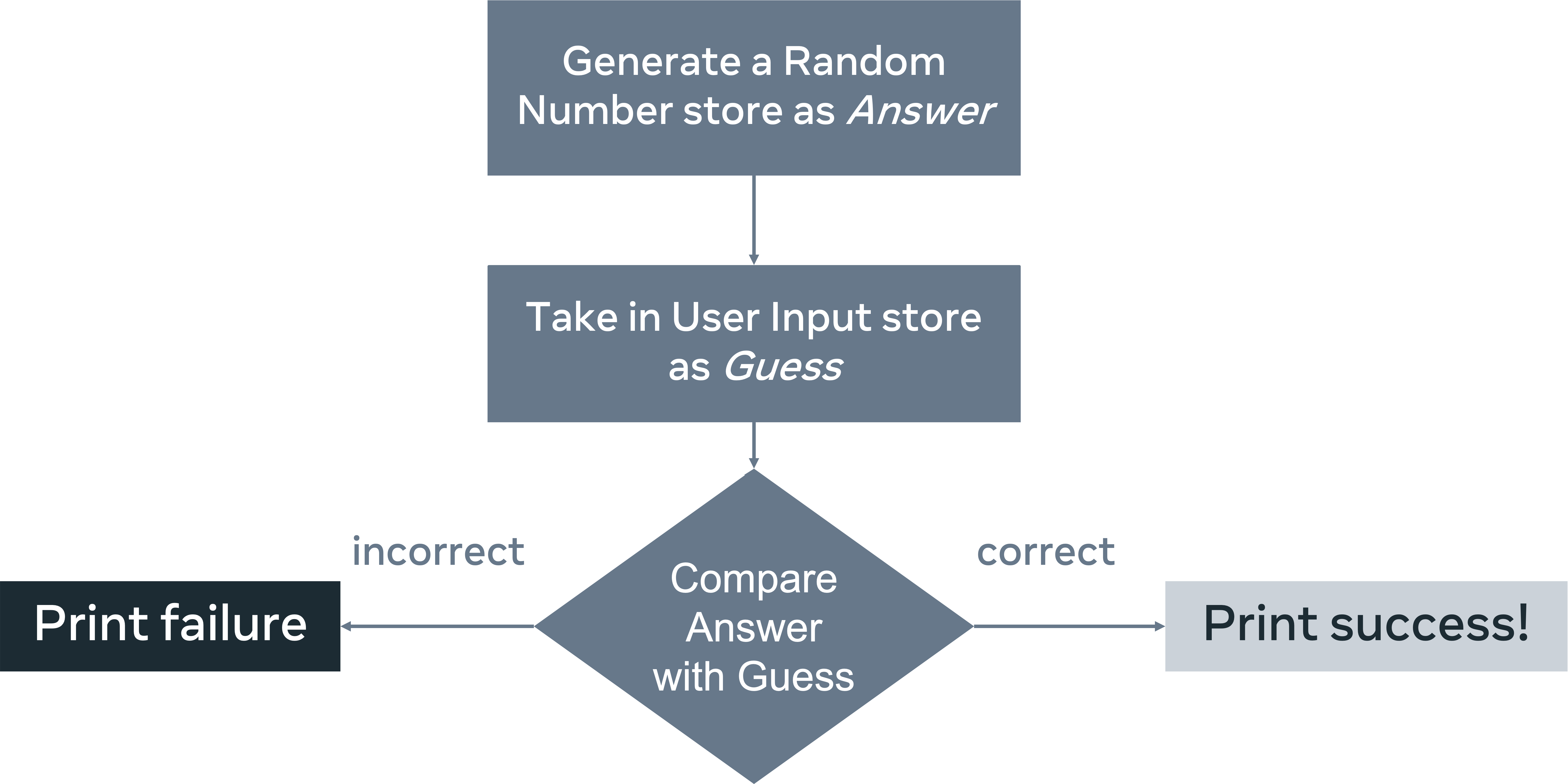
Second, how is user input taken? It has been established that the keyboard is viable, but how capable is the child at typing? Comparing a solution with text offers its own challenges. Would it need to be a direct match, would spelling, punctuation and capitalization be considered? Potentially, the questions could be multiple-choice. Raising awareness of all of these considerations is the purpose of outlining the problem statement clearly rather than charging into a solution.

## Finetuning the solution

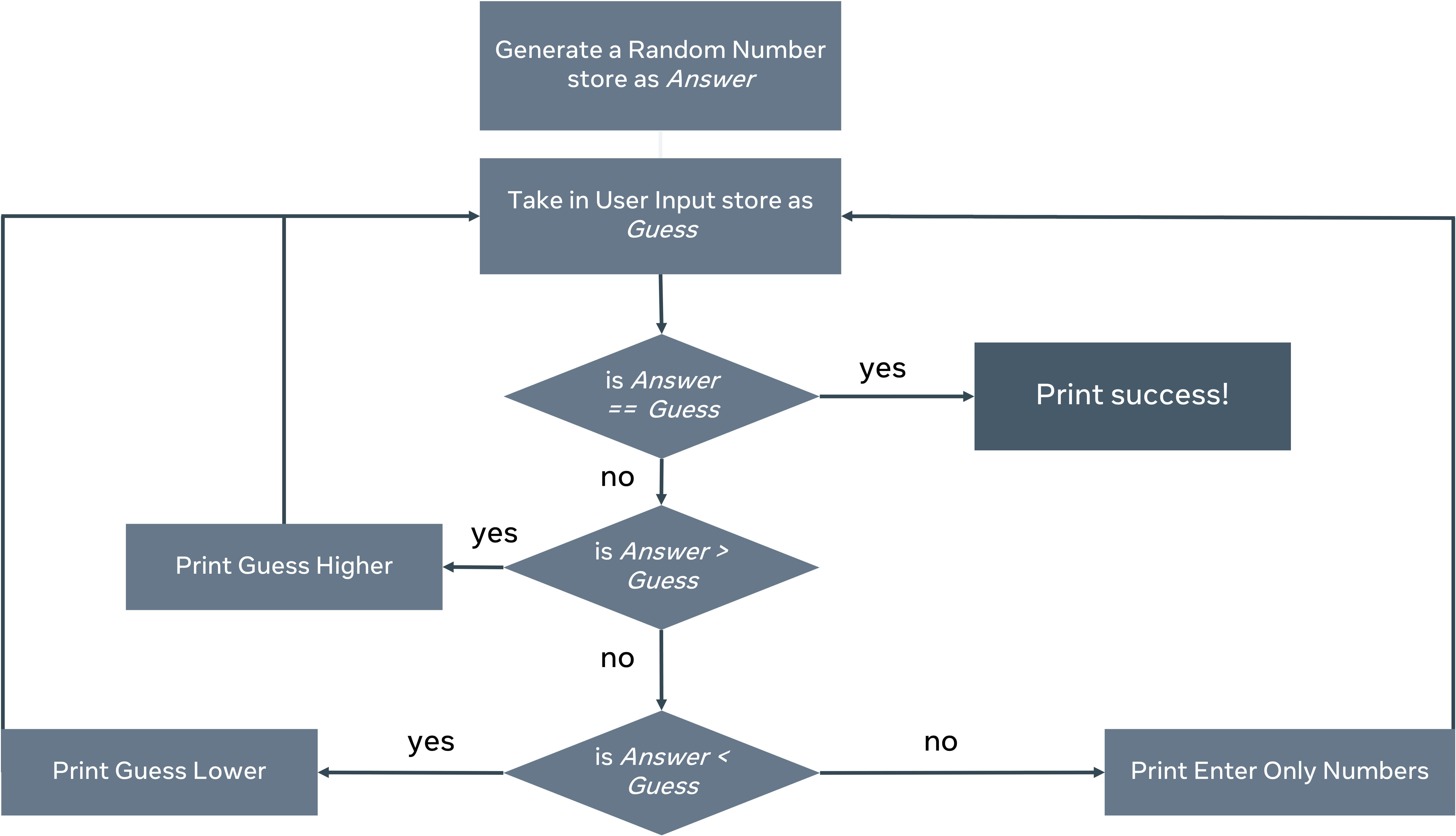
On reflection, you determine that a trivial knowledge approach will offer too many obstacles so the project is further defined as a number-guessing game. It's appropriate for all ages and there's no need to use external sources. And it should be suitable to implement with most programming languages.



Having outlined the general requirements, it is possible to fill in the finer points. Below is a flow chart of the pseudocode and it can give further insights into how to best frame the solution.



Examining the flowchart identifies further considerations. Will the program run only once? What can be added to enhance the user experience? Printing out a failure message might not be a great output. Instead, an option might be to offer another chance to guess. Further considerations might be to provide prompts that could be used to steer the user toward the correct answer. And, you should decide what happens if the user enters a non-number into the program.



Now that the various conditions have been outlined it's possible to pick a language and implement a solution. By taking a systematic approach to problem-solving you identified some potential issues before the project began and you could add some course corrections. Upon evaluating the solution the project might now change scope. In place of creating a game for a child to implement, the program might instead establish an environment so that the child can create the game themselves, using the clear-cut descriptions as a guide.

## Conclusion

Computers are a great way to model programs and a means of implementing a solution. It's the programmer's responsibility to employ a systematic approach in developing solutions. The same procedure used to identify a good project for a child can be applied to creating an application for work. It's imperative to establish these good working habits early in your career and to maintain them as your coding proficiency progresses.

#### **Working with time complexity**

## Introduction

In this reading, you will explore a worked example of a piece of code written in Python, along with how you would evaluate it using Big-O notation.

Evaluating an application's performance ensures that the code written is good and fit for purpose. The question is how do we evaluate efficiency? When we measure electricity, we use kilowatt-hours, which means how many kilowatts an appliance will use if it runs for an hour. The appliance will not always run for an hour, and it may have different requirements depending on the setting used, it is more of a general rule-of-thumb for evaluating cost.

When evaluating coding solutions, Big-O notation is used. So, Big-O notation is the kilowatt hour of code evaluation. It can be applied to measuring how much time a piece of code will take or how much space it will use in memory. Not all processors will run at the same speed, so instead of timing an application, you count the number of instructions an application initiates.

## Which measurement reflects the quickest possible execution of some code?

Let's explore which measurement reflects the quickest possible execution of some code.

### O(1)

You use a constant time algorithm that takes O(1) (O-of-one) time to compute. This determines that it will only take one computation to complete a task. An example of this is to print an item from an array.

# An array with 5 numbers

array = [0,1,2,3,4]

# retrieve the number found at index location 3

print(array[3])

In this instance, no matter how many values exist in the array, the approach has a Big-O of one. This means that running this code is considered O(1).

### O(n)

Next, let's explore an example of O(n). Taking the same array, an if statement is written that looks for the number 5. To establish that 5 is not there, it has to check every item in the array.

# An array with 5 numbers

array = [0,1,2,3,4]

if 5 in array:

print("five is alive")

In the above example, there is no 5, so there is no printout. To establish this, five checks were made on this array. As the input n = 5, this code is said to have a Big – O of O(n). To better understand this, let's extend the array to 10, leaving out the 5.

# an array with 10 numbers

array = [0,1,2,3,4,6,7,8,9,10]

if 5 in array:

print("five is still alive")

By extending the array 10 integers, the number of computations has now become 10. This is still called O(n) because the input size is 10, which is how many checks must be made before the program ends.

### O(log n)

This search is less intensive than O(n) but more work than O(1). O(log n) is a logarithmic search and it will increase as new inputs are added but these inputs only offer marginal increases. An excellent example of this in action is a binary search. Binary search is covered in more detail later in the course.

Now, imagine playing a guessing game with the following prompts: too high, too low, or correct. You are given a range of 100 to 1. You may decide to approach the problem systematically. First, you guess 50 – too high. So, you guess 25 – which is too high. You may choose then to go 12 or 13. What is happening here is that you are halving the search space with each guess.

So, while the input to this function was 100 using a binary search approach, you should come upon the answer in under 5 or 6 guesses. This solution would have a time complexity of O(log n). Even if n (the range of numbers entered) is ten times bigger. It will not take ten times as many guesses.

Here is a breakdown of those steps on the array.

array = [0,1,2,3,4,6,7,8,9,10]

print("##Step One")

print("Array")

print(array)

midpoint = int(len(array)/2)

print("the midpoint at step one is: " , array[midpoint])

print()

print("##Step Two")

array = array[:midpoint] # 6 is the midpoint of the array

print("Array")

print(array)

# running this shows the numbers left to check

# is 5 < 3

# no

# so discard the left hand side

# so the array is halved again

midpoint=int(len(array)/2)

print("the midpoint is: ", array[midpoint])

print()

print("##Step Three")

array = array[midpoint:] # so the array is halved at the midpoint

print(array)

# check for the midpoint

midpoint=int(len(array)/2)

print("the midpoint is: " , array[midpoint])

# is 4 < 5

# yes look to the right

print()

print("##Step Four")

print(array[midpoint:])

# check for the midpoint

array = array[midpoint:] # so the array is halved at the midpoint

midpoint=int(len(array)/2)

print()

print("##Step Five")

array = array[midpoint:]

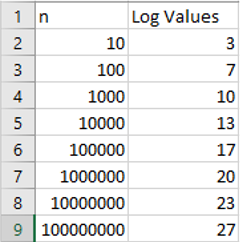
print(array)

print("only one value to check and it is not 5")

You will notice that to determine if 5 is present, it took 5 steps. That is a big-O score of O(5). You can see that this is bigger than O(1) but smaller than O(n). Now, what happens when the array is extended to 100? When looking for a number in an array of 10, it took 5 guesses. Looking at an array of 100 will not take 50 guesses; it will take no more than 10. Equally, if the list is extended to 1000, the guesses will only go up to 15-20.

From this, we can see that it is not O(1) because the answer is not immediate. It is not big-O(n) because the number of guesses does not go up with the size n of the array. So here, one says that the complexity is O(log(n)).

To gain greater insight into how the log values are only a gradual rise, look at a log table up to 100,000,000. This lens shows that O(log n) incurs only a minimal processing cost. Running a binary search on an array with any n values will, in a worst-case scenario, always make the number of computations found in the log values column.



O(n^2) is heavy on computation. This is quadratic complexity, meaning that the work is doubled for every element in the array. An excellent way to visualize this is to consider that you have a variety of arrays. In keeping with the earlier example, let's explore the following code:

new\_array=[] # an array to hold all of the results

# array with five numbers

array = [0,1,2,3,4]

for i in range(len(array)): # the array has five values, so this is n=5

for j in range(len(array)): # still the same array so n = 5

new\_array.append(i\*j) # every computation made is stored here

print(len(new\_array)) #how big is this new array ?

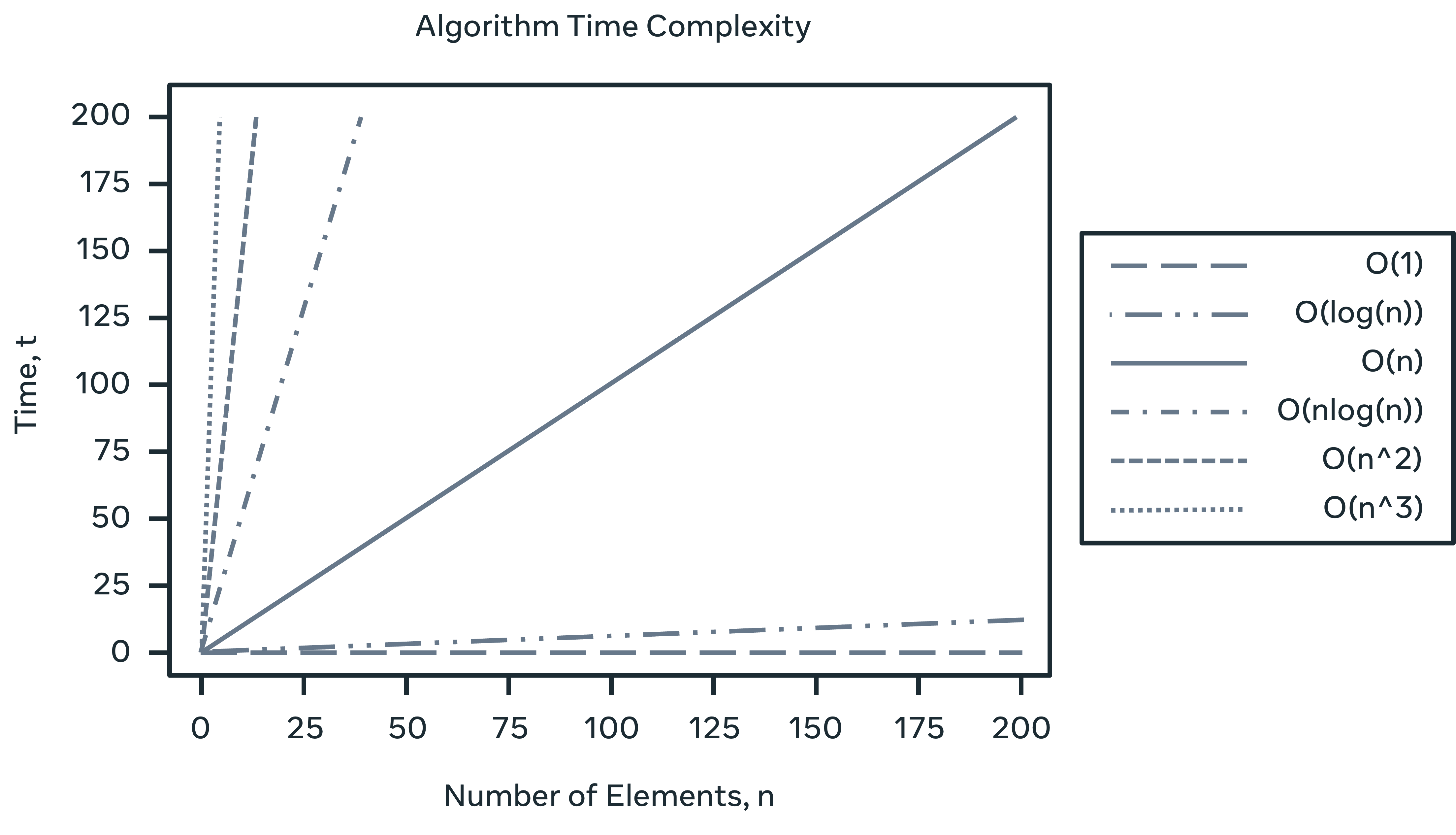
The first loop will equal the number of elements input, n. The second loop will also look at the number of input elements, n. So, the overall complexity of running this approach can be said to be n\*n which is n^2 (n-squared). To find out how many computations were made, you have to print out the number of times n was used in the loop as below.

n = 5 #size of array

print(n\*n) # how big is this new array ?

If you know that the array has 25 elements, then you understand the principles of calculating Big-O notation. To further test your knowledge, how many computations would be required if n = 6? Meaning the array had 6 values? The answer is 6 x 6 so 36.

## Visual representation of the problem

Below is a graphical representation of how n relates to the number of computations taken.

As you can see, the best time to aim for is O(1); O(log n) is still excellent. O(n) is ok and O(n^2) is not great.

## Worst case, best case and average case

Of course, it is not always possible to tell how long an approach will take. When looking at the initial loop example, there was a search performed for an element that was absent. You can say that to search a loop takes O(n) times but this might not always be the case.

Consider that the item being searched for is the first in the array. Then the return will be pretty good in O(1) time! In the example provided every item must be searched before determining that it was absent: O(n) time. The middle case would be that it is found around the middle of the loop O(n/2). When evaluating an approach, three definitions are used: best case, worst case and average case.

## Conclusion

This reading introduced the notion of time in relation to complexity and you explored a worked example of a piece of code written in Python. You also investigated how you would evaluate it using Big-O notation.

A good question to ask yourself before you start is, "how many computations does my solution employ and is there a better way?" Now that you know how to use a metric to evaluate the solution to a given problem, you can start thinking of its efficacy concerning time complexity.

# Strings

Strings are a feature of every programming language. A string can be defined as an ordered sequence of characters or symbols encased in matching single or double quotation marks. Most languages will support primary ASCII characters as well as Unicode representations. A character will occupy one byte of memory. However, there are many additional ways of representing strings.

In this reading, you will learn about the string data structure, how strings are represented across different languages, what they are commonly used for and why programming languages make strings immutable.

## **String representation**

There are critical differences in how each language represents and supports strings. All languages support the basic operations of creating, modifying, copying, and assigning strings to variables. Additionally, everyday actions include concatenating, appending strings, finding the substring and dealing with collections of strings.

When processing strings, many languages will allow you to perform algebraic actions on them. For instance, String\_A == String\_B, which will return a true or a false or String\_A < String\_B which determines which is alphabetically first. Some languages allow strings to represent variables in a string but require a special symbol. Often this is a dollar sign $ before the variable name but might also, in some instances, include being encased in curly brackets {}. Escape flags allow the inclusion of symbols in a string. Imagine that you want to have quotations within a string.

 String = “the man said \”two more pints please\” to the barman”

Here the use of the backslash \ ensured that the string would retain the quotation marks in the sentence. Other escape symbols could be #, %,' or a double quote inside a string denoted by single quotes. While the actual implementation may differ, the general approach to dealing with strings is the same.

## **Common usage**

One area that deals heavily with strings is Natural Language Processing (NLP). How strings are encoded when reading them from various locations such as Twitter, PDF files, text documents, or Reddit can raise unexpected issues. These can create complex debugging problems if the application processes an unusual symbol representation that affects the results.

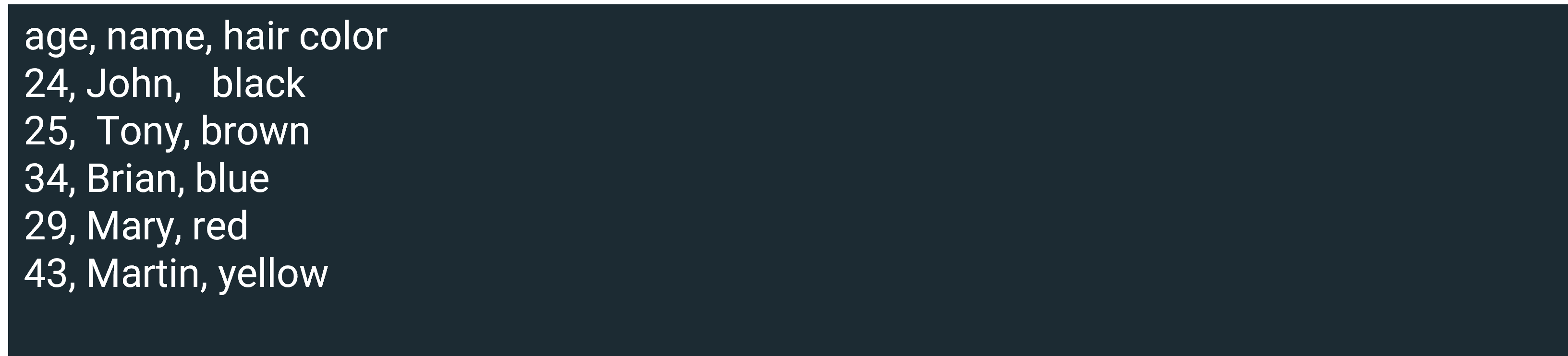
When writing strings into a program, it is common to apply tokenization. This is converting a string into an array of smaller strings. Here you would identify a delimiter and break the string into segments separated by the said delimiter. So, a paragraph might be broken into sentences through the use of the period(.) or a sentence into individual words by way of segmenting by space (" "). However, care should be taken when tokenizing text. Consider the text below that has three period symbols (.).

At 3.30 I went to the shop and spent $40.40 dollars.

Conventionally, you'd read it as a single sentence, but the periods used to indicate the time and amount of money are delimiters so the sentence is broken up into three parts.

Tokenization on processed, formatted data offers less of a challenge. A common way to store formatted string representations is through Comma Separated Values (CSV) or Tab Separated Values (TSV).

Processing such strings is known as parsing, and the returned array will equal the size of the number of delimiters found, with the delimiter removed. Here is an example of a CSV file that holds the columns' age, name, and hair color. Notice how there is no comma at the end. Unless otherwise stated, the new line symbol also functions as a delimiter.



## **Immutability**

An important concept to consider when dealing with strings is whether the string is mutable or immutable. Mutability refers to your ability to change a string after it has been created. Some languages like Ruby and PHP allow strings to be changed after creation. It is more common, however, to use immutability, such as in Java, C#, JavaScript, Python, and Go.

There are several reasons why programming languages make strings immutable. For one, it can reduce memory consumption. Instead of creating a variable that contains a string, a string pool is designed to represent all strings used. The immutable approach reuses memory allocation by having all instances of string point to one location. So, when a change happens with a string, say it becomes strings, instead of changing the value of string, the variable is pointed to another instance of strings. Or it's pointed to another immutable example representing strings, which is then added to the string pool.

This is a vast memory-saving device. Consider reading existing words instead of creating a memory location for each word, including repeating ones. Using a unique set reduces the required space. The drawback is that if your application constantly changes texts, a memory penalty is incurred for every alteration made.

## **Conclusion**

In this reading, you learned about the string data structure, how strings are represented across different languages, what they are commonly used for and why programming languages make strings immutable.

# Integers

An integer is used to hold numeric values. An integer can either be signed (holds both positive and negative numbers), or unsigned (will only hold positive numbers). Integers are represented in binary and an essential representation will need 4 bytes.

In this reading, you will learn about the integer data structure, such as how integers are represented, memory allocation for integers and the integer wrapper class in Java.

## **Integer representation**

There are several ways to represent integers in binary. One typical example is sign-magnitude. If integers are represented in binary, how can they distinguish between a positive and negative value? Sign-magnitude proposes using an indicator on the far left of the binary number to denote polarity.

| Integer | Sign-magnitude representation |
| --- | --- |
| 2 | 0010 |
| 1 | 0001 |
| +0 | 0000 |
| -0 | 1000 |
| -1 | 1001 |
| -2 | 1010 |

An integer cannot represent fractions. For this, one would use decimal or float. A fixed number of bytes is used when representing integers, the size of which can be specified in some languages. There is a standardized approach for representing numbers called the IEEE 754 standard, which outlines a common set of standards for representing all numbers.

Some high-level languages like Python and JavaScript subscribe to this approach and encapsulate the initialization of integers to this fixed representation. This makes working with numbers easy in these high-level dynamic languages but removes the ability to customize your approach to enable memory optimization.

## **Memory allocation**

Other statically typed languages like C++, Rust, Ada and C allow you to customize the size in memory. C++ allows you to customize your integer. Using unsigned short int will only take 2 bytes. This savings can mount up if your application includes many positive numbers that do not require precision to the degree of fractions.

Rust goes one better and enables the instantiation of unsigned 1-byte integers. This limited int can hold integers from 0 – 255. While that may not seem like a massive range to deal with, if you are working with pixels, you would be working with a tuple of values in this range. Image processing is a highly CPU-intensive process and being able to store 8 pixels for the standard price of 1 is a massive saving.

## **Wrapper classes**

In addition to primitive integers denoted by int, Java allows you to wrap the value of the primitive into a wrapper class Integer. This allows several methods for dealing with integers, such as converting from a string to a double, comparisons, maximum and minimum size and so on. The integer class is immutable, which makes it thread-safe. The extra functionality and safety incur a memory cost, and Integer object will take 16 bytes of memory to store.

## **Conclusion**

There is enormous freedom in controlling how your figures are stored in memory. While a high-level dynamically typed language gets up and running, it may have limited returns if your application becomes resource heavy. Suppose you are working with Arduinos or other hobby electronics. In that case, space becomes a premium, and the freedom to customize may be the difference between a functional and non-functional application. However, if you are developing an application that requires substantial amounts of available memory, using integers quickly and not having to worry about the detail may be the way to go.

In this reading, you learned about the integer data structure, such as how integers are represented, memory allocation for integers and the integer wrapper class.

# Booleans

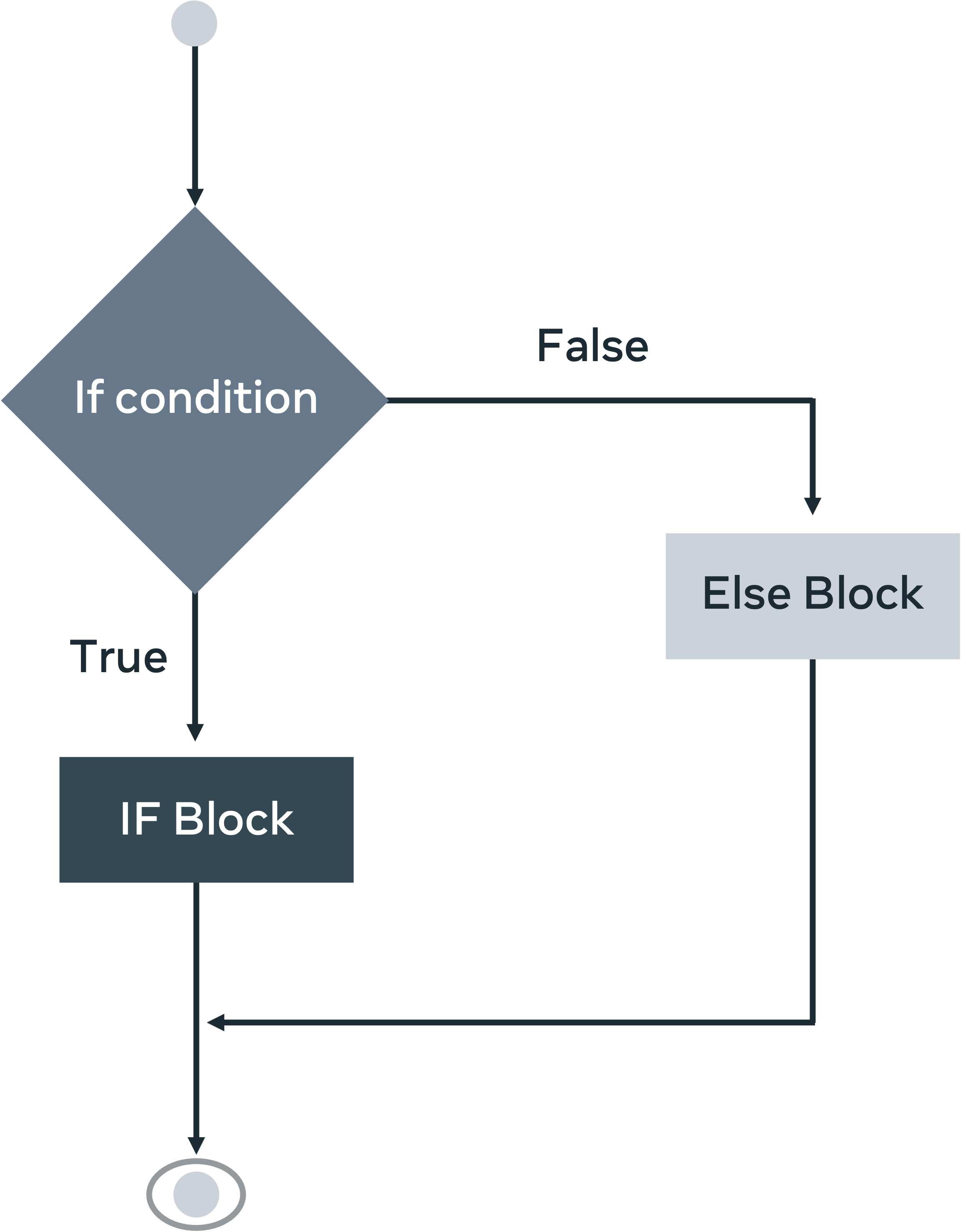
When testing your knowledge in this course, you are often presented with two options, multiple choice and true or false. The latter is an example of a Boolean; a thing is or is not. In this reading, you will learn more about Boolean data structures and the critical features for working with them, conditional statements and logical operators.

## **Conditional statements**

Boolean expressions can take a range of relational operators.

|  |  |
| --- | --- |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| == | Equal to |
| != | Not equal to |

These enable you to interrogate some data before running different code options. Therefore, Boolean expressions are often referred to as conditionals. Combine these with conditional statements, and you have an early implementation of AI.

Conditional statements are: if, else, else if, while and so on. A conditional block can enable you to execute one set of instructions in one condition and another if the conditions are different. Consider the following diagram.

That same diagram is represented in this Python code:

if right > wrong:

doTheRightThing()

elif wrong > right:

doTheWrongThing()

else:

keepResearching()

Using Boolean expressions as indicators, you can then use relational operators to determine which line of code is to be executed. In this example, you have a computer evaluating between right and wrong. This has multiple applications and is commonly used. Finally, if you have not met any of your conditions, you might employ a catch-all code block. Roombas are a prime example of how this code can be used in everyday life. In place of right and wrong, you will have rudimentary sensors that determine which motors are to be activated based on distance and impact.

## **Logical operators**

Additionally, you might want to expand the scope of your application by using logical operators.

| Logical operators |  |
| --- | --- |
| !! | Logical OR |
| && | Logical AND |
| ! | Logical NOT |

These logic expressions can be combined with Boolean expressions to give your code greater diversity.

if condition\_1 !! condition\_2:

doActionOne()

elif condition\_1 && condition\_2:

doActionTwo()

elif !condition\_1:

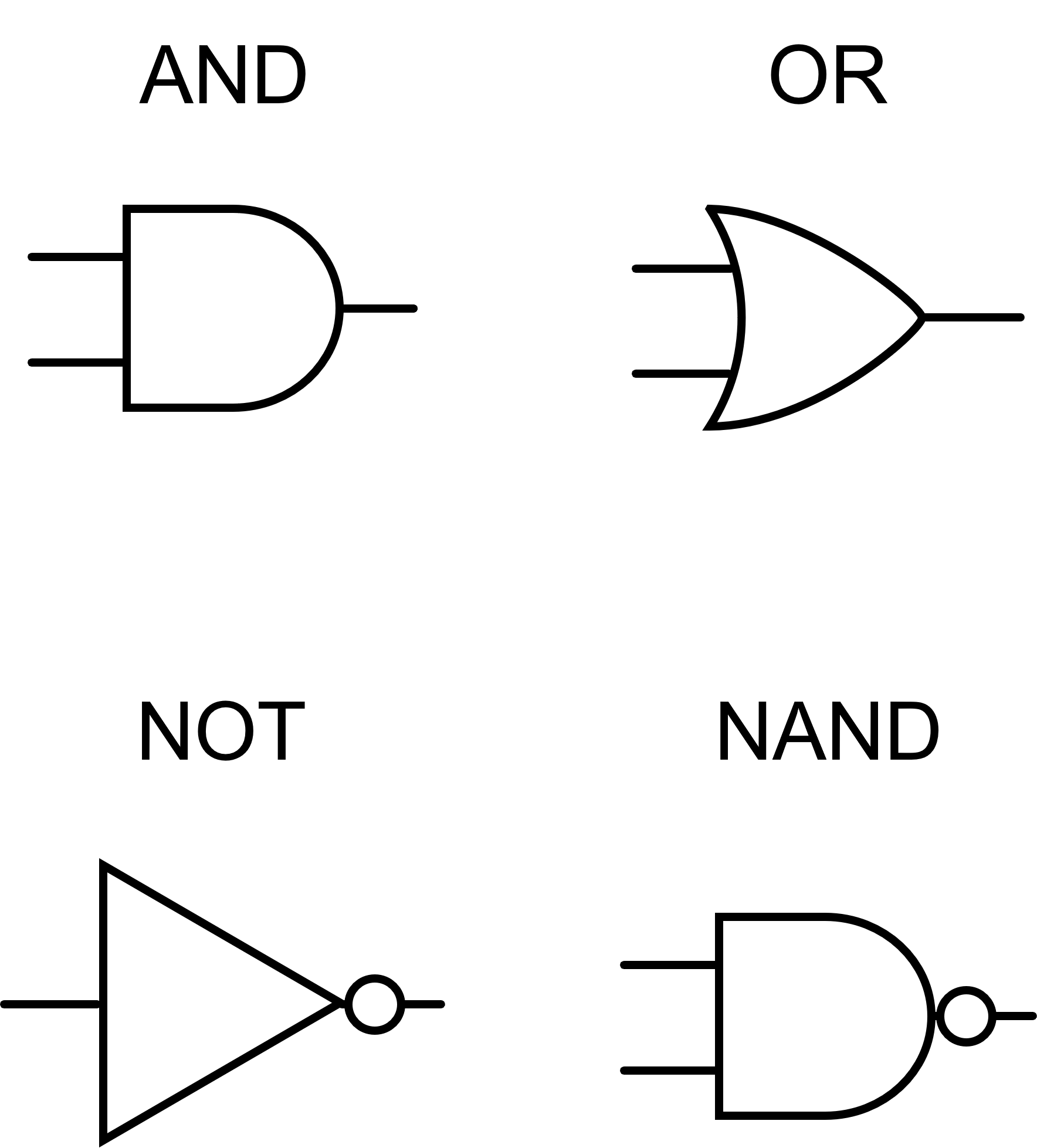
doActionFour()

else:

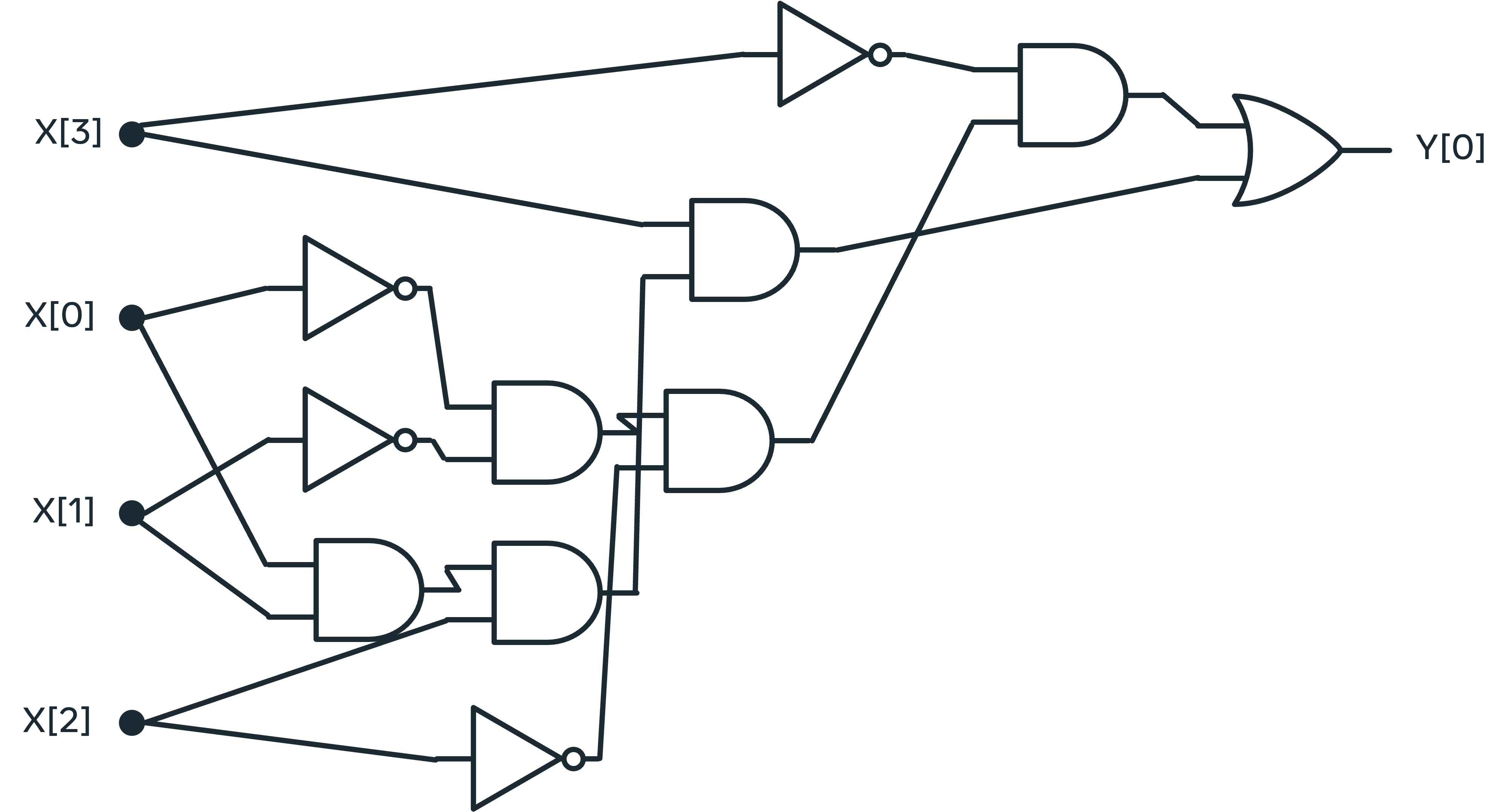
waitForInstruction()

In this code, four outcomes are checked. First, take condition\_1 or condition\_2, and let's stick with the Roomba example. If proximity detection or impact detection is true, then action one might be stop and reverse. If there's proximity and impact detection, then it'll trigger an alarm. If there is no proximity detection, it will continue to send power to the forward motors. Finally, if no conditions are met, there should be some fail-safe code ready to activate.

The use of Boolean logic is the backbone of circuit design. It is a way of interpreting Boolean operators together. The shapes in the diagram below are gates triggered when certain Boolean operators are fired. A signal is sent through the wires (depicted by the incoming and outgoing lines). The AND operator says if both wires are true, then continue this line of execution. At the same time, the NAND specifies that only two false assertions will trigger a reaction.



Combined, they can be used to make complex decisions and cause a device to operate in diverse ways, depending on the input from the sensors.



## **Conclusion**

To conclude, Boolean expressions are either true or false, and their equivalence in binary would be 0 or 1. Just like with binary, you might think that only having two values results in limited capability. But, they can achieve a surprisingly elevated level of complexity when combined with other mathematical constructs like conditional statements and logical operators. Boolean can be used in your computer programs to inform which operations should run automatedly and form the backbone of circuitry diagrams.

In this reading, you learned more about Boolean data structures and the critical features for working with them, conditional statements and logical operators.

**Arrays**

This reading demonstrates the array data structure, including how to initialize, manage and join arrays. All programming languages have some form of an array, but how they operate might have slight differences. For instance, some languages specify that an array must hold the same element type, like a string array or int array.

In contrast, other languages allow mixed elements to be stored in an array. The functionality of arrays might also differ. Sometimes, an array will be a fully-fledged object with complex functionality. At the same time, it can be treated as a storage type for primitives; operations and functionality would then be externally applied.

## **Initializing arrays**

Arrays can be created statically or dynamically. In a static language, the array would be kept on the stack and require that the array type be specified a priori. Dynamic languages offer more fluidity, sometimes calling for the size to be set and do not require the type to be specified before use. Such an instance would be stored on a heap. Arrays have indexes, which is a contiguous value starting at 0 and increasing until the end of the array. When an item is required, the array's name followed by square brackets and the index location are given.

array\_name[0]

This example will retrieve the first item in the array. Any number could be used and the element at that index location will be returned. Specifying a number greater than the size of the array would throw an out-of-bounds error. A typical action you will make with an array is to iterate over the array and investigate the elements.

n <- size of array arr

FOR (i <- 0;i<(n-1); i <-(i+1)) DO

process element arr[i]

END

The general shape of a for loop is the same in all approaches: you need the size of the array, an integer i that increases at every iteration and a general for loop for syntax. The appearance may vary slightly, but the underlying mechanism is the same. Starting at 0, go through each item in the array and do something. An array will always have some way of accessing the size. Some examples are .size(), len(array) and .length().

## **Managing arrays in memory**

One fundamental difference regarding arrays in different programming languages is where they are stored. An item can be stored in heap or stack memory. Stack memory is created when running a function. It is created for that function and discarded after the execution. Items in this memory allocation are only available for that function. In contrast, heap memory is created during the execution of instructions and is available to all.

Care should be taken when altering elements in a static environment. On instantiation of an array, memory space equal to the specified initialization size is created on the call stack. A call stack is created to execute the goal when a function is called. Altering an array and returning it from the stack may lead to corrupted memory as the stack is discarded after the function completes.

Dynamic languages avoid this issue by storing the array in a heap. Thus, the array remains unaffected when the function ends, and the stack is discarded. Stacks, by their nature, hold contiguous memory blocks, making accessing the information more manageable. A heap is less organized and it can take more time to access the elements. Once again, you should consider the trade-off between size and convenience, or accessibility versus speed.

Ordinarily, when a program is finished with an array, the memory is deallocated and becomes available for something else. How this deallocation and garbage collection is handled is down to the programming language selected and can only improve performance if done effectively. Poor memory management can lead to leaks, which may crash your application upon repeated calls.

Two memory-related concepts you may encounter are shallow copy and deep copy. The first instance does not make a copy of the array but returns an index location. A deep copy will create a new instance of an array. Making a shallow copy optimizes memory usage; however, you must ensure that no unexpected changes are inadvertently made to an array shared by two variables.

## **Joining arrays**

A matrix is a two-dimensional array (or an array composed of arrays) that can act like a table. It can be used to represent rows and columns. It gives an element a 2D (x,y) coordinate. Here x would refer to the rows and y to the columns. As before, one would use square brackets to access an element in a two-dimensional array. However, two index locations are required for a matrix, as below.

Matrix[x][y]

A matrix will exhibit a square shape with the same number of columns and rows. This need not necessarily be the case. But if you choose not to maintain uniformity, ensure that your loop acts accordingly.

int[][] matrix = new int[5][7]

for (int i = 0; i < matrix.length; i++) {

for(int j = 0; j < matrix[i].length; j++)

{

doSomethingNeo(matrix[i][j])

}

}

In this example, a 2D array called a matrix is initialized to hold integers. The outer collection can contain five elements (integer arrays). The inner arrays all have a capacity of 7. The first array in the outer array is selected with the i, the size is determined (7), and each element of that inner array is then passed to the method. Notice how the size of the inner array was accessed dynamically using the .length attribute. This ensures that there can be no out-of-bounds errors during the iteration.

## **Conclusion**

Arrays are a fundamental data structure that exists in most programming languages. They help store related data. It is worth noting that their implementation varies depending on the language, so care should be taken when using them. This reading taught you about the array data structure, including how to initialize, manage and join arrays.

# Objects

## Introduction

In this reading, objects will be discussed. Objects are the building blocks of all code and play a particularly important role in object-orientated programming (OOP). In this reading, the benefits of using objects will be outlined, as well as some important terminology relating to the use of objects.

## Definition

An object is a programming concept that means that a structure has both state and behavior. Here, behavior relates to the object's ability to perform some action. As you progress through this course, you will witness many instances of objects exhibiting different behaviors. Concretely it relates to calling the methods of an object. An example of this might be calling the sort method of an array. This has the result of re-organizing the items of an array so that they are organized in relation to one another.

State pertains to the information about an object. Another word for this that you may be familiar with is object attributes. If you create a person class and you instantiate it to an object, an example of a state might be the person's age or name. The behavior then might relate to an action that is required of this person-object, like run or tackle.

Classes are commonly described as blueprints for an object. By extension, an object can be described as an instance of a class. The most common use of objects is in OOP, where code is encapsulated into objects, and these objects then interact with one another.

## Example

One of the considerable strengths of using objects to instantiate a class is that you only need to create one template for how an object will act. Then you are free to create multiple instances of this class that can interact with one another. Consider a football team with 11 players. Only one class needs to be outlined that can hold varying speed, agility and work rate characteristics. This can seriously reduce the amount of code overhead in creating a football game. Characteristics like speed and agility are known as instance variables and can be said to relate to the state of the object. No unit outside of the object will share this instance. Two players can have a fitness instance. Changing this in one will not affect the fitness instance found within the other objects. You can change the instances of a class through the constructor or an internal method of the object. In Java, a typical instantiation of a class is as follows:

Player player1 = new Player(agility = 54, speed = 88, fitness = 90);

Player player2 = new Player(agility = 90, speed = 64, fitness = 83);

In this example, the player class creates two objects, player1 and player2. The variables within the class are set differently depending on the values found within the soft brackets. There also needs to be a method to change the variable. Over the course of a match, a player may become tired. A method can be called to reduce the variable instance to reflect this.

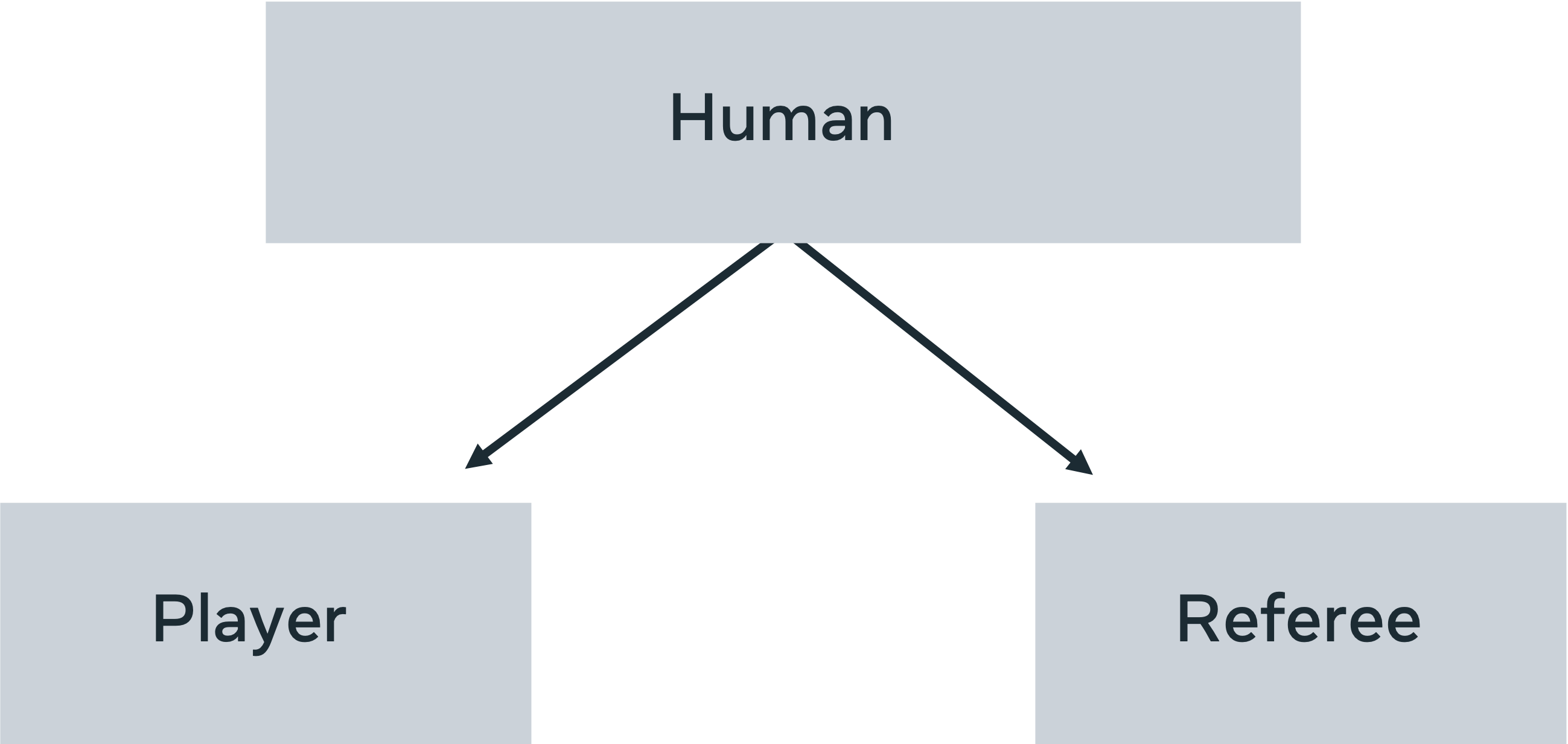
player1.set\_fitness(80)

This will alter the fitness level of player1 but will not affect the other players. The methods used to change attribute instances are generally called getters and setters.

Further control of the objects is available through the other methods found with the object. A command such as

player1.kick()

would cause the object assigned to player1 to enact the procedure of kicking the ball.

The player class allows for the creation of a team with varying abilities that can play. Another concept related to the use of objects is inheritance. Imagine there is a need to create another human on the pitch, such as a referee, that does not play but still performs actions relating to the game, such as running and general appearance. You may want to reuse some of the code found in the player without creating a player. Here you would create a class that holds all common attributes you wish to retain, and each object can inherit them, as shown.

Thus, the standard methods such as run, get-tired, and follow the ball may all be found within Humans, but how these relate to the class player and class referee can be differentiated. This notion of having one general concept(human) that can manifest in different forms (players, managers, referees) is known as polymorphism. One shape, many forms. The classic example is the use of shapes. One overarching shape will contain the area, diameter, and height concepts. Then each instance of shape (square, triangle, circle) will apply the actual implementations of how these attributes look in their given state.

## Conclusion

In this reading, the concept of objects was explored, with a particular focus on its usefulness when used in an object-orientated programming approach.

**Lists and sets**

Lists and sets are common in many programming languages. Let's get started by exploring lists. In most programming languages, lists are represented as objects. This means that in addition to storing data, they also have their own in-built methods. Here, an in-built sort method is used to arrange the numbers in a list. As with arrays, it is common to find lists that are declared as either a string, an integer, or float. In some programming languages, you can have lists with mixed element types. A list is an abstract concept that refers to a container of elements. A stable implementation of a list is done using either an array or a linked list. An array-based list is an ordered collection built using arrays as the underlying data structure. As such, they are subject to the same strengths and limitations associated with arrays. Array-based implementations relate to the initial sizing rather than simply pointing to another node as with a linked list. Some languages require that you initially determine how big a structure will be, while others allow for dynamically growing structures. It should be noted that this freedom is somewhat surface level. For many dynamic structures, there is an initial size automatically configured at instantiation. When this limit is reached, the array will copy itself into a new structure with a larger size allocation. Therefore, the decision not to arbitrarily allocate space at the onset may come at a cost at runtime when such data structures may have to expand multiple times during the execution of other operations. Consider the computation cost of a list dynamically growing while performing operations in a loop. In this case, it would help to set the initial list size to be larger rather than dynamically growing, which can be costly due to having to create and copy over values into increasingly bigger lists. A linked list works differently. A linked list contains two pieces of information, the data and a pointer to the next list item. A linked list begins with an empty list and can grow dynamically by introducing new cells to the list. To grow a linked list, you simply have to add a new node and point the list at its location. This makes them very fast for storing large amounts of data. The flexibility of linked lists is achieved by including some additional storage requirements. Notably, in each node, there must be some reference to the nodes around it. There is also a head and a tail. The head is a unique node that indicates that it is the start of the list, and the tail indicates where the list ends. This approach to growing the size of the data structure is very powerful and can lead to very large but manageable datasets.

**What does sets entail?**

Set is very similar to a list. However, a set will store its elements in an unordered way. Though there are some possible implementations of ordered sets, sets have some unusual tendencies. A set will only hold unique elements, so adding an element that already exist to a set will make no difference to the data stored there. The unordered process in which sets store their information means that printing out a set will not necessarily reflect the order in which the element was added to the set. Once a value has been added to a set, it cannot change. Instead, you would have to delete it and add a new value instead. Sets are exceptionally fast to search. This is because if it's internal mechanisms. A set uses hash tables to determine where to store the elements of a set. Therefore, each number that is passed to a set will have a hashing function applied to it. A hashing function can be defined as an algorithm that takes in some data and maps it to a fixed size value. The value is theoretically unique and every time the function is applied to the data, the same value is returned. This means that searching a set can be done in O1 time. This is due to the mechanism that is used to save values in a set. You will learn about hashing functions in more detail later in the course. A O1 approach would be to iterate over the entire data structure to check for the presence or absence of a value. Sets instead apply the mapping function to the input data and check the resulting output to see if a value exists there. If it does, then the value is returned. If it doesn't exist in the set, then the data was not stored in the set, and hence will return a false. While sets can perform an exceptionally quick search, performance degrades when dealing with very large datasets. This is due to the nature of the hashing function. The more values retained, the more risk there is a clashing. Clashing is when the hashing functions return the same unique mapping for two different values. The larger the dataset used, the more likely clashing is prone to happen.

# Lists and sets in different programming languages

Sets and lists are built-in types in many languages with a general overarching theme common to most. In the actual implementation, there are some subtle differences.

In this reading, you will explore some of the differences between lists and sets in different programming languages.

## Mutability

The concept of immutability has already been discussed. Concretely, it is how a data structure is created. Immutable objects cannot be changed when created, while mutable ones can be changed after creation. Sets are an example of an immutable structure in most programming languages. However, objects that are added to sets can be treated differently depending on the language. In JavaScript, you can add mutable objects to a set, which is not permitted in Python. Understanding this fundamental difference will give you insights into how the language uses sets.

## Why are sets as quick as they are?

The reason for the speed at which sets can find an item is down to their underlying architecture. Sets store values using a hashing approach. To hash something is to take it and generate a unique output from it. This is done by applying an algorithm to it that converts the input into a simple alpha-numeric (words and letters) output. Every time the algorithm sees the input, it will always generate the same alpha-numeric output. This is then used in a hashtable, which uses the unique hash to inform where to store an item in memory. So, it is possible to know with one computation whether the value is in memory or not. Instead of searching for every element in the list, just apply the hashing function and check if it is being used.

Therefore, sets are fast but the reason they're so fast is also why you can't change the value being stored. If you create a hash for an element and store it according to that hash, then you change the item, and the hash will no longer be able to find it. This is one reason why Python only allows you to store immutable objects that can't be changed in sets. Not all languages offer this protection. For example, JavaScript does allow the storing of mutable objects. Thus the object's protection is left to the user when coding with JavaScript sets. Suppose you wish to alter a mutable value in JavaScript. In that case, extracting and deleting the mutable object is advisable before reinserting it into the set as a new element.

Like Python, Kotlin will not allow elements to be altered once they have been added to a set. This means that there is read-only access to the set. If you want sets that can be altered, then Kotlin has another type of set called a MutableSet. Sets in Kotlin are very versatile and offer a range of built-in methods like max, min, sum and average. This makes them very handy when looking for specific items within a group of numbers.

## Lists

Another important collection type that is good to know about is lists. Lists store elements in a given order which can be accessed using an index. An index is just a number passed to the list that indicates that the element found at that number location should be returned. The index is how elements are searched in a list. If someone wants to know if something is in a list, they would have to do a search and check each item, so they are slower than a set. Additionally, a list will allow you to store all kinds of elements in them. There is no distinction made between whether an item is mutable or immutable. Lists will also allow you to store duplicates of items. There are different types of lists that are implemented differently.

However, the fundamental point that is worth remembering is that a list allows you to store repeating items and items of different types. A list is ordered, which means that it will retain the ordering of items as they are inserted. This is different from sets, which might store items in different locations from where they were initially entered.

Knowing these subtle differences can be helpful when deciding what type of data structure you want to select. Both sets and lists are useful, but because they act differently, they will be more useful in some situations than others. Making sure that a list or a set is most suitable in a situation is the trick!

## Conclusion

In this reading, a brief overview was given on lists and sets in different languages. The strengths and weaknesses of both have been discussed, as well as some interesting insights into how they work. Hopefully, you will think about these the next time you are planning to store data!

# Stacks and queues in different programming languages

## Introduction

A stack is a fundamental data structure that limits the way data is stored in an application. Namely, items must adhere to the entry or exit policy of last-in-first-out. Different languages have different implementations, though overall, the general result is the same. A queue is very similar to a stack; however, the order of entry is different, favoring a first-in-first-out policy.

In this reading, you will learn about some of the inherent differences in both stacks and queues across different programming languages.

## Stacks

Stacks are an example of an abstract data type. In computer science, this means that there are some very important characteristics that need to be enforced when implementing it, but there are no actual built-in versions that you can just import. For stacks, the important principle is LIFO or last-in-first-out. The analogy for stacks can be to visualize a pile of plates on a washing board. As each plate is cleaned, it is placed on top of the previous one. For drying them, each plate must be taken from the top. So, the last plate added to the pile is dried first, and the first plate on the pile is dried last. A common usage for stacks is to keep your browser history. Each time you hit back, the previously visited page loads and, going forward, then reads it to the pile.

A common implementation of a stack in JavaScript is through using an array and making sure that it acts only in a way that a stack would act. Taking a data structure and using it to build another type of data structure is said to use a container adapter. So here, the array is the container, and the adaptation forces it to behave as a stack should. The important features of a stack are push, pop, peek and count. Push adds an item to the top of the stack. Using an array means that the count must always know where on the array the last item went. Equally, for the pop method, having a count for the array is important, as pop only returns the last item entered on the stack.

## Queues

A queue, like a stack, is a linear data structure that retains the order in which things were entered. When one talks about a linear structure, it means that items are stored in the order in which they were added. Just like a stack, the queue has a strict implementation of how items are added and removed. While a stack implements a LIFO approach, queues work with a first-in-first-out (FIFO) approach. So, the first item that is added to the queue will be the first item that is removed. Like a customer queuing to make a purchase, queues look to implement a policy that places importance on the time of arrival. This has some real-world benefits when implementing things like CPU and disk scheduling, where it is important to deal with tasks as they arrive.

A queue is an abstract data type in Swift, which means that to use the functionality of a queue, you would first have to code up a structure that will act in that way. Two important terms associated with queues are enqueued (to add an item to the queue) and dequeue (to remove an item from the back of a list). The simplest approach for creating a queue is to use an array as the container adaptor. Because a queue only pops elements from the front of the queue, the look-up time will always be O(1), so any application built using this approach can expect very quick service. As with stacks, the available methods for a queue are kept minimal, with peak, pop, push and size being the most important ones.

Python implements several variant queue classes that are all synchronized. Synchronization is a computer science concept that means that all access to the data found there is managed so that the data structure can only be accessed by one process at a time. This is important when you think about cloud computing and having many different sources trying to look up and change data at the same time. On top of FIFO and LIFO, Python queue implementation includes a priority queue. A priority queue, as the name suggests, is about assigning importance to the elements found in the queue. So instead of a first-come-first-serve basis, service is based on importance. This is a bit like the VIP line at the concert.

## Conclusion

Selecting the appropriate data structure for the appropriate task is a very important programming decision, as it can affect the entire project. You would select a stack when there is a need to keep track of the order in which an item is entered and to return them in a very specific way. A queue is used for maintaining order, but it will abide by different rules than a stack. Finally, if you want a data structure where you can assign importance, then a priority queue is worth looking at.

**Trees**

Tree node is described by

→ Root

→ Sibling

→ Child

→ Parent

→ Leaf

# Trees in different programming languages

In this reading, you will learn about some of the differences in the implementation of trees in different programming languages.

A tree is an abstract data type (ADT) that is present in many languages. As discussed in other readings, an ADT is a blueprint for how a data structure will manifest. It relates to the restraints and requirements placed on a data structure to ensure that it will always operate in the same way. This can be very useful for programmers who are switching between multiple programming languages as it provides clear expectations of how a data structure will operate.

The key fundamental principles of a tree are that it contains a number of nodes, a root node and a selection of leaf nodes. Leaf nodes are unconnected nodes at the base of a tree. A root node is always at the top, and every value descends from this one node. A tree will always be arranged in some form of hierarchy.

## Implementing a tree

There are many types of trees; perhaps the most straightforward and common one is a binary tree. A binary tree has the following properties:

1. Every node has a maximum of two child nodes.
2. Every node must have a key so that it can be easily identified.
3. Values found to be less than the node are placed in the left child node, and values that are greater are placed in the right child node.

To create a tree, one must ensure that there is a root (starting node) and a method for determining the subsequent nodes. Each node must contain a reference to the left and right nodes so that the tree can be traversed. This can be achieved by creating a class with these three attributes (key, location of left node, location of right node). This class will need three additional methods so that it can function as a tree. These include:

1. A lookup method: It is important that the tree can be queried for the existence or absence of information.
2. Insertion method: As has already been noted, inserting a node on a tree involves finding out where it should go and placing it on the left side of the nearest higher value.
3. Removal method: This method will need to remove an item from a tree. This operation poses additional challenges when applied to a tree due to the connected nature of a tree. Consider that a tree comprises a series of connected nodes. So, removing one carelessly can result in destroying the connections in a tree. Therefore, when implementing this method, in addition to removing the node, it is necessary to check all the children nodes and ensure that a new connection is made with the node of the next highest value.

## Searching in trees

The approach outlined above for coding a tree is applicable to all the languages covered in this course. There are no concrete implementations of trees, which means that to use them, you are required to implement the code yourself. Another feature that is worth bearing in mind is how a search is performed on a tree. One can code a solution that applies a depth-first search or a breadth-first search. To better understand this concept, consider how the tree is organized. Each level has a parent node that connects to two child nodes. These child nodes, in turn, have two of their own child nodes. A breadth-first search is one that will examine each node on the same level before stepping into a deeper level. This can be pictured as scanning all nodes horizontally before checking the next level. A depth-first search will examine each node on a branch until the end node is reached before checking the adjacent branch. This can be pictured as a vertical scanning of the tree.

## Conclusion

This reading covered the topic of trees and how they appear in different programming languages. Because trees are an abstract data type, there is no built-in method for the languages used in this course. To implement a tree, it is crucial to keep in mind the important characteristics of a tree so that you can correctly implement them.

# Hash tables in different programming languages

Collection classes are specialized classes for data storage and retrieval. Depending on your application's need, selecting an appropriate data structure can significantly impact your approach to coding and the efficacy of the application. A hashtable will store key-value pairs and offers a quick lookup. Some languages include out-of-the-box implementations, while others require another collection type to act like a hashtable.

## What is a hashtable?

A hashtable offers very quick look-ups for an application. This is achieved by creating a hashing function that will create an alpha-numeric (letters and numbers) output from a given input. This hash is then used to determine where in memory to store something. This means that when you want to know if an element is in the data structure, instead of looking through every item and making a comparison, you only need to apply the hashing function and see if that item has been hashed to memory. When you consider that a data source might have millions of entries, not having to check every single one is a great time saver.

None of the languages covered in this course have a built-in hashtable implementation. So, in order to implement an instance, it is necessary to alter an existing data structure to perform the operations of a hashtable.

You are likely to find hash tables used in caches, dictionaries, database indexes and sets.

Hashtable uses speed over space

## Implementing hashtables in Kotlin

While Kotlin does not have a built-in implementation of a hashtable, a very similar data structure called a hashmap is supported. Hash-maps are very similar to hashtables as they also store key-value pairs and use a hash to determine where in memory to find the key. There are some distinct differences that should be kept note of: a hashmap will allow the use of nulls for keys and values, and it's not thread-safe.

## What is thread-safe, and how does it apply to hashtables?

Threads are processes that a computer can run. Typically, when you turn on your computer, a number of processes will begin. These processes include things like starting up a Word document, Excel, a Java application and so on. These processes are often run at the same time. To do this, the compiler will create many threads that can execute the code. So, a thread is a small executable piece of code that can run a process. So how can you say that code is thread-safe? Thread-safe means that if you are to write a program that accesses a data structure, you can duplicate this application and access the same data structure via different threads without causing an error. Having five different threads working on the same data structure might make your code run five times faster, but it is only useful if the information that is being changed is done so correctly.

## How does thread-safe relate to the Kotlin implementation of hashtables?

One feature of hashtables is that they can be synchronized. This means that if five different processes are using and changing the same information in a table, the information is always correct. Kotlin does not provide an implementation of hashtables; however, there is a very relatable data structure that can be adapted for this purpose called hashmaps. Hashmaps have the same key, value and hash lookup implementation, but they are not thread-safe. So, in implementing a hashtable in Kotlin, one can take a hashmap and add some code that ensures synchronization so multiple threads can access it at the same time. Additionally, it is important to ensure that the hashtable will not allow nulls to be added as keys or values. This can cause an issue with comparing values within the table.

## Hashtable implementation in Python

As with Kotlin, Python does not have a native implementation of a hashtable. In the previous section, it was demonstrated that a table could be mimicked using an existing structure called a hashmap. This can be done in Python as well, though the underlying structure used is a dictionary. A dictionary is an appropriate data structure to use, as it works on the same principle as a hashtable. Namely, it stores key-value pairs. The keys are hash able and used as an indicator of where in the memory to store the value. This means that it has very fast search and insertion methods. In addition, dictionaries are already thread-safe, so they don't require changing if you require operations to perform concurrently.

## Conclusion

In this reading, hashtables were discussed. The implementation of a hashtable is dependent on the language. Here it was demonstrated how a hashmap in Kotlin and a dictionary in Python are sufficiently similar structures from which one can create a hashtable.

**collisions in hash tables**

A hashing function will apply a clever algorithm that will reduce the size of the key to a manageable size. Some approaches are more intricate than others.

# Heaps and graphs in different programming languages

## Introduction

In this reading, the focus is on graphs and heaps. A graph is a non-linear data structure that can store information in a way that allows you to extract some interesting relations found in the data. Heaps can be defined as specialized graphs that place special importance on the topmost element.

## Terminology

Graphs are comprised of nodes and edges. The node is where the data is stored, and the edge is a connection between two nodes. Unlike a tree, nodes do not have to be connected and can exist independently from the other nodes. An edge connects two nodes. An edge can be said to have a weight. This is a value that is stored in the connection that infers some information on the strength of the connection between the two nodes. A graph can be said to be directed, this means that the edges are focused (like a one-way street) or undirected (like a two-way street) and the connection infers information back and forth.

## NetworkX

NetworkX is a Python-based external library that can be imported into the Python environment and utilized to model data. It is a lightweight third-party library that can work in any Python environment. Many featured graphs include directed, undirected, cyclic and acyclic. NetworkX also supports a wide range of graph-based algorithms, including:

* Distance metrics (the distance between two nodes)
* Centrality (how central a node is in relation to other nodes)
* And clique detection (relates to subsets within a graph)

A clique detection algorithm will analyze a graph to determine a range of subsets most likely to have strong internal and weak external relations. These are referred to as cliques due to their resemblance to social cliques. This can be a helpful way of identifying potentially related customers, given a sample of customer data and habits. NetworkX can be modeled using Python's matplotlib to give insight into the discoveries of the data. This is used as a back-end application by a programmer interrogating data for insights.

## Heaps

Heaps are implemented differently in different languages but are essentially graphs with specific constraints. Heaps sort information in order so that they can quickly return min and max values. Thus, they employ a binary approach, and any implementation must have a maximum of two nodes. Depending on the implementation, a heap will have the largest or smallest value as the root. Finally, each branch of the heap will follow a sequential pattern.

A heap has an O(1) lookup time because it only returns one item. The highest or lowest value depends on whether it is a min or max heap. This means that once this value is popped, the following item is pushed onto the root node of the heap. This also impacts inserting onto a heap. When a new element is added, beginning at the root, it is compared to each node until the correct position is determined. The surrounding elements are then moved to ensure that it is placed in the appropriate position.

An alternative way that some languages like Kotlin or Javascript. which lack a built-in instance, represent heaps using an array list. The same properties you use with graphs are employed, though the actual implementation may differ slightly. The efficacy of heaps will depend on the underlying data structure used, so it is worth investigating how efficient your target language implementation is before using them.

## Conclusion

Heaps can be seen as specialized graphs, and a graph is made up of nodes and edges. There are some very specialized methods available for graphs, whose purpose is to allow us to infer some connection from the data that is stored there.

# Time and space complexity in sorting algorithms

You previously learned that time and space complexity are a means of evaluating code efficiency. In this reading, you will explore time and space complexity in both the selection sort and quicksort algorithms. These are common algorithms that are used to sort data in an array.

## **Selection sort**

Selection sort is a sorting algorithm that works from a very simple principle. Take an array to items and iterate from left to right. Starting with the first place on the index, iterate over the entire array and swap this value with the lowest value found to the right of this item. Repeat until the entire array is sorted.

Selection sort has:

* Worst case time complexity is O(N^2)
* Average case time complexity is O(N^2)
* Best case time complexity is O(N^2)
* Space complexity: O(1) Auxiliary.

To perform selection sort, take the following steps:

* Find the smallest value and swap it with the first value of the array
* Find the second smallest value and swap it with the second place in the array
* Repeat until all items are changed from ordered from smallest to largest

Time complexity is determined in relation to the number of transactions enacted. Given a list of size n, the compiler must search each entry in the list to identify the smallest item, then perform a swap to index location 0. The pseudocode for the algorithm is as below.

for(i = 0; i < n-1; i++)

int min\_index=List[i]

for(j=i+1; j<n;j++)

if(List[j] < List[min\_index])

min\_index=j

swap(List[i], List[min\_index])

Line 1 says that the length of the list List must be searched n-1 times. Line 2 sets a temporary variable to hold the lowest value. Line 3 is an inner loop that must iterate through the loop n-1 times. Line 4 checks if the value found in position List[j] is smaller than the current lowest value. If so, the position of that element is recorded. At the end of each inner loop, the value found to be the lowest is swapped with position i in the index, i is incremented and the procedure begins again. Always check the next item in the list until every item has been checked.

There are four considerations to be made when evaluating this algorithm.

1. Worst case scenario: Given a list sorted in reverse order, how many comparisons are made? The inner and outer loop will have to run n times so it can be determined that worst case = O(n^2).
2. Average comparison: Regardless of the order of the list, every item must be checked against average case = O(n^2).
3. Best comparison: Given a sorted list of how many comparisons must be made. Again, regardless of the items in the list, every item must be checked, so best case = O(n^2).
4. Finally, what is the space complexity of this approach? Because an in-place swap is being performed, no temporary array is required. There are three temporary variables i, j and min\_index; however, these are not dependent on the list size. So, the image doubles the list, and the space complexity does not increase accordingly. Therefore, space complexity = O(1).

When evaluating time complexity, a good rule of thumb is to consider what will happen if the list is doubled. Naturally, the inner and outer loops will have to increase by no iterations to match the additional elements in the list. Therefore, it can be concluded that the time complexity increases with the size.

## **Quicksort**

Quicksort is a sorting approach that uses a divide-and-conquer methodology. Given an array of items, a place is determined on the array on which to split the array and this is called the pivot point. All values greater than this point go to the right and all values less than this point go to the left. In this step, you have two arrays. The same process is applied to these arrays until there are no elements left to sort.

Quicksort has:

* Worst case time complexity O(n^2)
* Average case time complexity O(n log n)
* Best case time complexity O(n log n)
* Space complexity O(n)

To perform quicksort, take the following steps:

* Select a point on the list to pivot on.
* Split the list into two lists, items to the left of the pivot and items to the right.
* Set variables i to iterate from left to right on the left of the pivot. Set variable j to repeat from right to left on the left side of the pivot.
* The variables i on the left look for a value greater than or equal to the pivot. Variables j on the right look for a value less than or equal to the pivot.
* When j < i, the values at these index locations are swapped, this is repeated until i and j meet at the pivot point.
* Partition the list values into two lists, one to the left and one to the right of the pivot. Repeat the process on each of the resulting arrays.
* Recursively apply the algorithm.

The pseudocode below for quicksort is done recursively.

* Starting at the leftmost element, each subsequent element is checked, and if it is found to be less, it is swapped.
* Line 3 calls the partition method, which begins on line 8.
* Line 10 determines the more significant element to be placed on the right side of the list. Line 10 sets a variable i to be assigned to the index of the smaller element. The variable j is then used to check the elements to the right from which to make a comparison with the current smallest element.
* Line 12 determines if there is to be a swap, a smaller element on the right will require moving to the current index position. Line 4 is for sorting the left array.
* Line 5 is for sorting the right array. At each iteration, the size of the array to be sorted is halved. The arrays will continually break down until only one element is left in the subarrays. The result of calling partition will determine the location of the current element. This location is incremented and repeated until every element rests in its naturally ordered position.

QuickSort(List, low, high)

if(low<high)

pivot=partition(List, high, low)

QuickSort(List, how, pivot-1)

QuickSort(List, pivot+1, high)

Partition(List,high,low)

pivot=arr[high]

i=(low-1)

for j = low; j <= high-1; j++)

if(List[j] < pivot)

i++

swap(List[i], List[j])

swap(arr[i+1], List[j])

return I + 1

**Things to consider when evaluating this algorithm:**

1. Worst case scenario: this happens when the most significant element is consistently chosen as a pivot point. This will cause a loop to iterate over every element n from the left. The split will cause a search of every element on the right with none on the left, O(n^2).
2. Average case scenario: an average pivot point is selected at every call. This will reduce the number of additional iterations required. So, there will be n iterations and an ever-decreasing logn iterative calls, O(n\*logn).
3. Best case scenario: The middle value is always selected, and the iteration space is halved at every iteration, O(n\*logn).
4. The iterative nature of the algorithm will impact the space complexity because the function call and variables are retained on the stack while the calculations are performed. However, the decision to use an in-place swap means no new array needs to be created, O(log n).

Conclusion

Two different sorting approaches have been broken down and analyzed through the lens of Big-O space and complexity. It has been shown that quicksort is more complex in implementation but returns overall quicker solutions. Selection sort is more simplistic and less code-heavy and requires less space, but will not generate results as effectively.

In this reading, you explored time and space complexity in both the selection sort and quicksort algorithms.

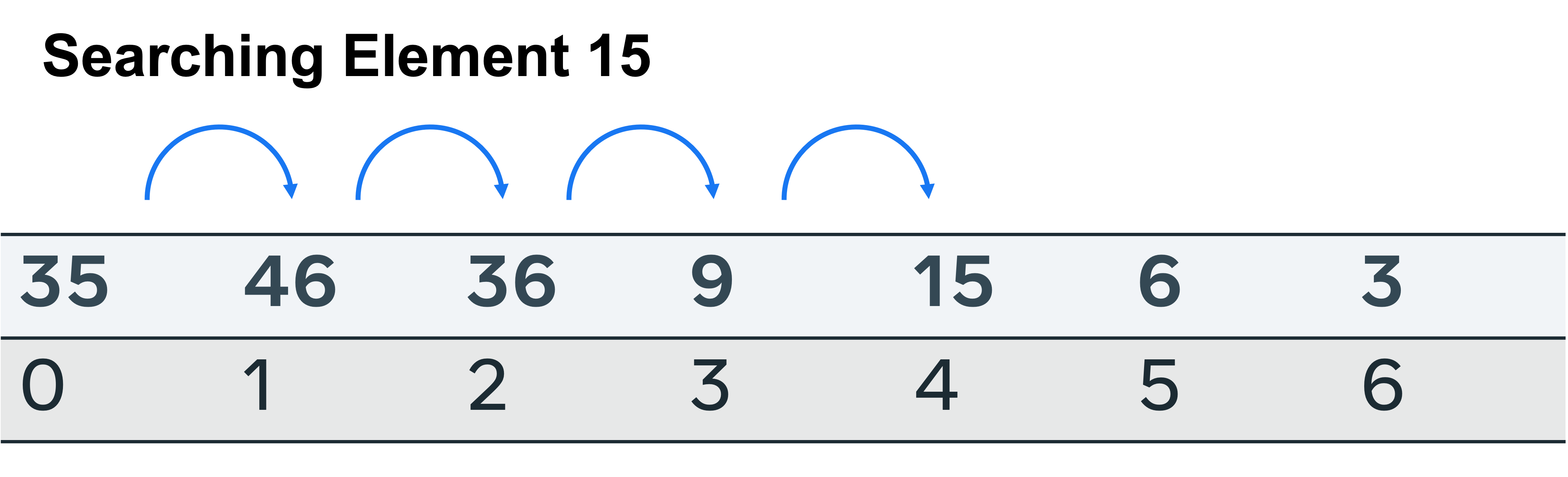
# Time and space complexity in search algorithms

In this reading, you will explore time and space complexity in both linear and binary searching algorithms. To gain a greater sense of how time and space complexity feature, these two searches are examined noting the time and space used in finding the target element.

## **Linear search**

A linear search is the most direct way of retrieving an item. It means that the search starts at the first item and iterates until either the target item is found or there are no more items left in the array to check.

Given a list of numbers, start at index location 0 and compare each item with a target variable. Return when the index location has been determined or the entire list has been checked and there is no instance of the target element.



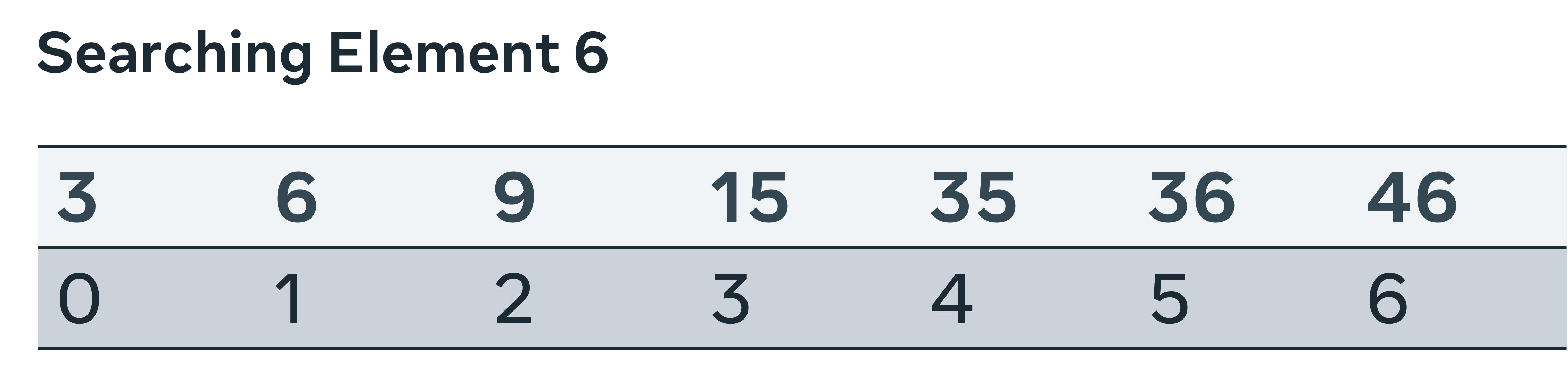
These are the outcomes to consider when evaluating the efficacy of the search.

* Worst case: The item is absent from the list. To determine this, every possible location in the list size n has to be searched. O(n) time complexity.
* Average case: The element is found in the middle. This is considered an outcome of O(n).
* Best case: The item is found at the starting index and no further checks are required, so O(1).
* Space complexity: No additional space is required to perform the search. So, the space required will only be as large as the items that have to be stored in the list, space complexity O(n).

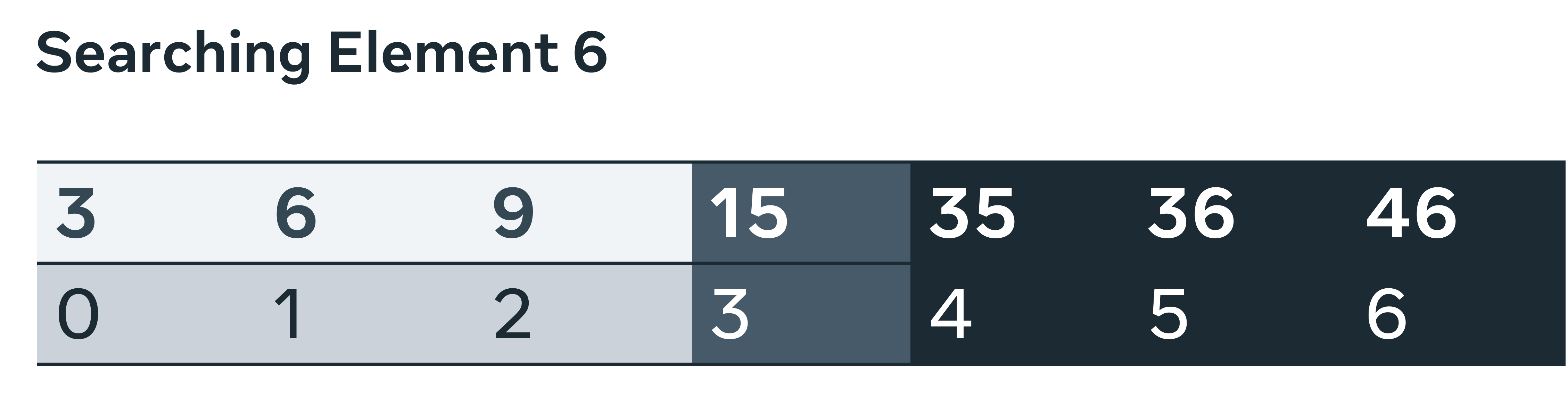
## **Binary search**

A binary search is performed by first identifying the mid-point on a sorted list, comparing the target element to it and discarding the half that is less than the target element. This halving at the mid-point is repeated until the target element is found or there is no more list to half.

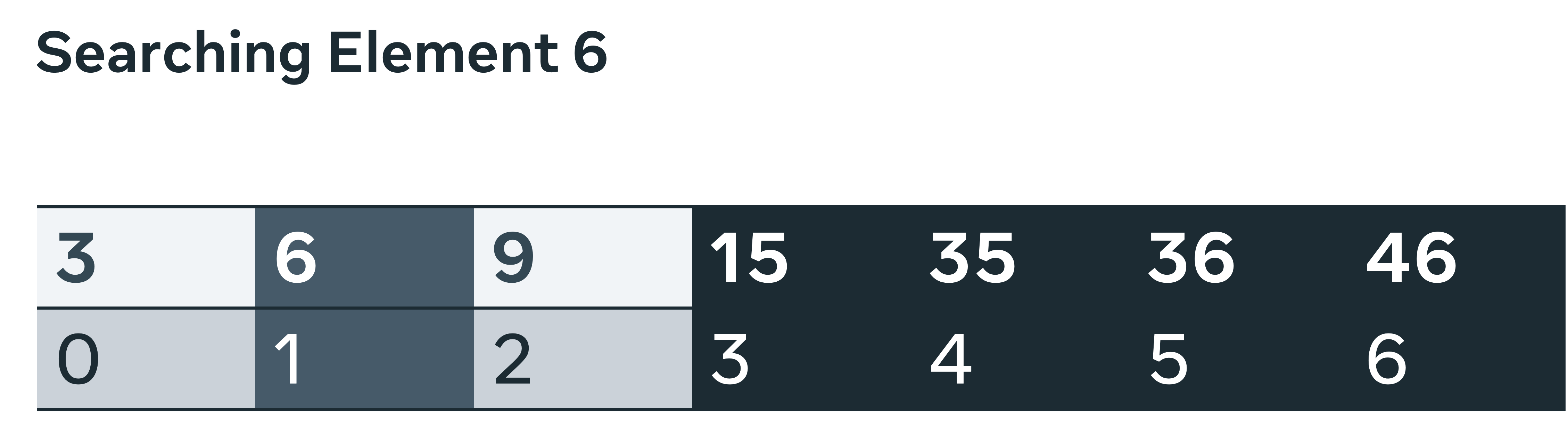
To conduct a binary search, the list must first be sorted.



First, a middle point is selected. The value at index 3, is 15. Is this greater than or less than the target element? The search space is broken in two and the left is further examined.



A new central point is selected. This time there are only three slots to check; index location 1 is at the halfway point.



The target value is found here and no further splits are required.

 These are the outcomes to consider when evaluating the efficacy of the search:

* Worst case: The item is absent from the list. Due to the nature of the approach, many items are removed with the use of the logical operators greater than and less than. This means that only n/2 is checked first, then n/4 and n/8. The overall complexity is then O(log N).
* Average case: The element is found after several iterations. Again due to the search mechanism, each subsequent call reduces the state space. So, it can be determined that after a medium number of searches, the complexity is O(Log N).
* Best case: The item is found at the starting index and no further checks are required, so O(1).
* Space complexity: No additional space is required to perform the search. So, the space required will only be as large as the items to be stored in the list, space complexity O(n).

## **Conclusion**

In this reading, you explored time and space complexity in both linear and binary searching algorithms. The time and space complexity for linear and binary search has been analyzed. Both approaches are in-place searches; therefore, there is no additional space required, so the space complexity is small. Linear search has shown itself to be more time complex in all instances, other than the correct item being found on the first attempt.

Binary search halves the state space at every iteration, making it far more efficient. However, you should consider factoring the time taken to sort the array in the overall calculations when evaluating the efficacy of selecting this approach for a general application.

**Divide and conquer**

Both are the mandatory steps and combine, which is an optional step. In the divide step, the input is split into smaller segments and processed individually. In the conquer step, every task associated with a given segment is solved. The optional last step, combine, is combining all the solved segments.

Merge sort is a sophisticated approach for sorting an array. It starts by having the array, these two halves and then halved, and halved again. This process is repeated until there is only one element remaining, then, the process reverses and each smaller list is sorted before rejoining the part it was halved from.

**Advantage of the divide and conquer paradigm**

Parallelization → Parallelism is when you have different threads or computers working on the same problem at the same time to complete it in a quicker time.

Dynamic programming involves saving the computations used to come upon a given solution.

Dynamic programming is a programming paradigm that promotes solving problems by breaking them into smaller problems and solving these. The solutions are then stored in an appropriate data structure for later use. The advantage to this is that if these sub problems require being computed again, one only searches for the answer instead of computing the problem again. The technique of solving sub problems and storing them to save time on a potential future look up is known as memorization. Dynamic programming relates to two concepts already encountered in divide and conquer.

The first is divide and conquer, that is taking one large problem and breaking it into a smaller set of sub problems and then solving these. The second is a subset of this known as Recursion. Recursion is the practice of coding a solution that avoids running loops but instead uses multiple self calls in coming upon a solution. Dynamic programming is an extension of these approaches which in addition involves keeping a record of results generated from running the sub problems each time they are newly run. In subsequent runs instead of re computing results a look up is queried for the last time the question was asked, as said, this approach is called memorization. And to reinforce the concept, this is when the results of previous calculations are stored and used in place of rerunning the calculations when the compiler identifies that the computation has been run for a previous task.

Dynamic programming involves saving the computations used to come upon a given solution.

**Steps included in the dynamic programming process**

→ Describe the optimum outcome.

When computing dynamic programming solutions, one must first determine the objective function. That is the description of what the optimum outcome is to be.

→ Break the problem into smaller steps.

Breaking the problem into smaller steps can allow recursions to take place. That is when functions call themselves repeatedly until a solution is reached.

**Greedy algorithms Dynamic programming**

→ Current most rewarding → Globally optimal solution

→ select the most valuable item and → what subset of items to pack would place this in the bag. maximize the value of filling the bag?

A greedy approach would differ in its methodology.

→ best option not guaranteed