# java.util.helpmepls

*Learning to Code from a Subpar Programmer*

Aditya Rao

February 7, 2023

*"Dedicated to Arnav's fat juicy ass"*

- Adi

# Contents

# Prologue/Introduction

Hi there, my name is Adi. At the time of writing this book, I recently have graduated from highschool and am on my way to university soon. Over the last four or five years, I've developed a deep interest in the ever elusive world of code. My main language is Python but I've recently diversified into many other avenues like Java, JavaScript, HTML, CSS, C++, Dart, and a few others. However, Python is definitely the language I have the most experience in. As such, that will be the basis for most of the actual code snippets in this book. I will also provide a few examples in psuedocode whenever I do something in a certain language.

I don't consider myself to be any authority on Computer Science/Engineering or anything like that, I'm just a guy who thinks he's got a decent method to help some people pursue their interests. My objective with this book is not really to help you learn a certain language, rather, it gives you the tools and knowledge to help you learn whatever language you want.

Many people who want to learn how to code already have an objective in mind and simply want the tools to make that dream a reality. Others just are interested in the subject area as a whole, some just want to get ahead of the curb and learn

the skills required in a world ever dominated by tech. All of these are valid reasons for someone to learn to code. However, there is a difference between a person who knows how to program, and a programmer. Thea former can give instructions to a computer and knows what to expect as an output. The latter knows how and why a computer is doing what it is doing to make their instructions more and more efficient.

Therefore, these are the topics that we'll delve into in this instructional guide. First, we'll start with why you should be interested in code. Next, we'll progress into what coding actual is. Then, we'll take a turn and look at the inner workings of a computer and what components like the CPU and RAM actually do and how a computer understands the instructions that you give it. Then we'll learn some of the basics of any major high-level programming language (examples will be given in *Python, Java, and psuedocode*) and techniques to go along with it. Lastly, we'll go into some of the most important skills for a coder, the way you need to think about problems. This is by far the most important - and one of the most overlooked - skills that you develop when programming.

From there, we'll go over the future and some good resources to look at to progress your skills past what you have been taught here. Good luck on your coding journey.

# Chapter 1

# Why learn how to code

There are a variety of reasons why one would want to learn how to code. First and foremost, ever second, there are millions of developers around the world trying to automate something new. Everyone is searching for problems to try and solve - every hour of every day. That means that by not learning how to code, you may be putting yourself on the back-foot as it's a skill more and more employers are looking for in candidates.

## 1.1 Interest

For a lot of us, personal interest is what's motivating us to learn how to code. You could be intrigued by the software side and how a machine can be controlled and worked with to do what you want. Essentially, we wanted to see how we make a computer go bleep bloop using words.

## 1.2 Education

You may be looking for a new route in your education career. Coding is a skill that can help you in almost any field. Even something like history where you almost definitively will not be writing any code, skills like Decomposition will help you breakdown your tasks and methodically complete them.

## 1.3 Freedom

The last opportunity that coding gives you is freedom. In our technology enhanced world almost everything is in some way shape or form dependent on some sort of tech. If you want to start a business or some sort of initiative, tech skills can help you start your concept, be a better manager by understanding the work being conducted, or by simply allowing you to automate monotonous tasks giving you more time to do what you want.

# Chapter 2

# What is coding?

What actually is coding? Many people don't seem to understand what someone who codes actually does. We've all seen those scenes in movies and TV shows where the protagonist (or antagonist) says a bunch of tech words, sometimes throws a "quantum" or two in there for good measure, and then how is able to stop a software nuke (whatever that is) from annihilating downtown Brooklyn. It's not nearly as glamours. When you code, you're simply giving instructions that a machine can understand and execute.

Now, the machine doesn't actually understand the words you write, instead it translates them into machine code/language. This is essentially a giant list of Ones and Zeros which the computer can use to turn switches off and on. That's basically how a computer works - turning switches off an on - zero and one.

By manipulating these switches, we as programmers are able to make the computer do what we want. However, it would be incredibly laborious and confusing

to type out said 1s and 0s by hand - though this was done using "punch cards" in the early days of the computer. Instead, we write code in what are known as "high-level languages." These essentially make it easier for people to read and write code, allowing a machine to convert the words (aka syntax) into machine code.

That is all that coding is, giving a machine a set of instructions and hopeing it does what you want. My friend likes to joke around, saying that "Computer Science should be considered an experimental science" because it can be mostly trial and error at times as you try to figure out how something works.

There are a few different types of coders: front-end and back-end. There are of course a few other flavors - i.e. mobile and game development - but these are the two most prominent ones. By no means do you have to stick to one once you choose it, but the skill sets for each are quite different. A front-end developer are generally people who focus on the **user experience** aka **UX** and **user interface** aka **UI**. If you're look at a website, that's typically the work of a front-end developer. On the flip side, we have back-end developers. Back-end developers are responsible for delivering content and internal logic of how some system works. For example, if you login to a website the process by which your account is authenticated and validated is the work of a back-end developer.

# Chapter 3

# How does a computer work?

Knowing how a computer works is what separates the good from the great. When you understand how a computer works you, it's easier to make code more efficient and understand how your code works on the lower-levels (machine and assembly language).

## 3.1 Ones and Zeros

When you write code, you're essentially writing a bunch of words which get turned into 0s and 1s. These 0s and 1s subsequently are used to tell the computer whether to turn a switch on or off (also known as high or low as if the switch is *on* then a *higher* current is sent across the wire and vice versa). This number system is known as *binary* or *base-2*. On the other hand, our number system is called *decimal* or *base-10* because we have 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9).

### 3.1.1 Binary

Binary only has two digits - hence base-2 - which are 0 and 1. Each place value is a multiple of 2 with the first being $2^0 = 1$, then $2^1 = 2$, $2^2 = 4$, $2^3 = 8$ and so on as can be seen in Table **??**.

| **Decimal Value** | 8 | 4 | 2 | 1 |
|---|---|---|---|---|
| **Binary Value** | | | | |

To write a number from decimal to binary you simply take the biggest possible number from left to right. For example, if I have 11, first I'd subtract: $11 - 8 = 3$ since $11 \geq 8$and then add a 1 under the 8 in place value. Then I'd check if $3 \geq 4$ - since that's false, we add a 0 underneath 4 in place value and move to the next digit. We check if $3 \geq 2$, which is true, therefore we $3 - 2 = 1$. Finally we end by checking if $1 \geq 1$ if it is, then we add a 1 in the ones place. This can be seen in Table **??**:

| **Decimal Value** | 8 | 4 | 2 | 1 |
|---|---|---|---|---|
| **Binary Value** | 1 | 0 | 1 | 1 |

Table 3.1:
*Decimal Number:* **11**
*Binary Number:* **1011**

An alternative method is you keep dividing by 2. If you get the remainder as 0, you add a 0 to your number, if you get a 1, you add a 1 (don't change the number you're diving by). Lastly, you write the entire number our backwards. This can be illustrated in Table **??** below with 29:

| Working | Remainder |
|---|---|
| $29 \div 2 = 14$ | 1 |
| $14 \div 2 = 7$ | 0 |
| $7 \div 2 = 3$ | 1 |
| $3 \div 2 = 1$ | 1 |
| $1 \div 2 = 0$ | 1 |

Table 3.2:
*Decimal Number:* **29**
*Binary Number:* **11101**

Now in Computer science we have a few terms for certain sets of data. A single digit is called a bit. If we have a binary number that's 4 bits, it's called a *nibble*; if it's 8 bits long, it's a byte. From here it's easier to understand. 1024 bytes= 1 kibibyte (KiB). **However,** this is only if we're being extremely technically accurate. Generally 1024 bytes is taught - and learned - as a kilobyte (kB); in reality a kilobyte is actually 1000 bytes (hence kilo). 1024 KiB = 1 MiB (Mebibyte) 1000 kB = 1 MB (Megabyte) and so on.

I additionally want to add some general information that can be useful in your everyday life. There's a difference between KiB, kB, and kb (and MiB, MB, and Mb and others). The first is a **Kibibyte** which is 1024 bytes. The second is a Kilobyte which is 1000 bytes. The last is 1024 ***bits*** **not** *bytes*. Internet service providers like to use this trick to make their speeds seem faster than they actually are - specifically with kB and kB (or MB and Mb). Computer/Mobile manufactures do something similar with KiB and kB (MiB and MB) to make it seem like

their devices have more storage than they do.

There are more complex things in binary, such as binary addition, subtraction, multiplication, and division. However, I don't think it's really necessary or practical most of the time. If you're interested, I recommend taking a look at this video

### 3.1.2 Alternative Number systems

Alternatively, hexadecimal - base-16 - is commonly used as a representation for binary numbers as it can easily be used to show nibbles with a single character. Hex has 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F). This means we can represent any byte with two digits. This is quite convenient whenever we do need to look at the raw binary of some data.

An example of 255 in hexadecimal is FF. You can quickly convert between decimal and hex by multiplying each place value by the corresponding power of 16. For example, AF is $10 \times 16^1 + 15 \times 16^0 = 160 + 15 = 175$. If you want to convert decimal to hex, I find the easiest way is first to convert the number into binary, then convert that into hex.

If you want to convert binary to hexadecimal, you separate your binary number into nibbles. Then you convert each nibble into decimal (adding all bit values) and then turn that into the corresponding hexadecimal number. The following is an example with 10011110

| **Binary Value** | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Decimal Value** | 8 + | 0 + | 0 + | 1 | 8 + | 4 + | 2 + | 0 |
| **Hex Value** | 9 | | | | 14 | | | |
| **Hex Number** | 9 | | | | E | | | |

Table 3.3:
*Binary Number:* **1001110**
*Hexadecimal Number:* **9E**

One practical use of hexadecimal is when reading through machine code. When you open some files, you'll be greated with a range of hexadecimal numbers. Sometimes, it can be useful to know hexadecimal to debug these files (i.e. knowing the exit byte in a document to understand why your code is failing).

Another practical use would be simply representing large numbers easily, sometimes it's just easier to read them.

### 3.1.3 ASCII and Unicode

One important implementation of binary is ASCII and Unicode. This is a certain format of binary number used to store characters also known as symbols, letters, and numbers.

The former, ASCII, stands for the "American Standard Communication for Information Interchange." Essentially, it provides a convenient way for all computers to render documents and information. ASCII characters tend to have the form "0100 0000" with different combinations of the first two bits representing different

characters (i.e. “0110 0000” for capital letters).

ASCII was revolutionay, however, overtime with the widespread global use of computers, there simply weren't enough bits to represent every characters/symbol from every language. Hence, unicode was created

## 3.2 Von-Nueman Architecture

Von-Nueman Architecture is the fundamental organization of most modern computer systems. It essentially boils the system down into smaller subsystems.

### 3.2.1 Why do I need to know this?

I found myself wondering this pretty often. No doubt it was somewhat interesting, but I initially couldn't see any practical application as a software developer.

However, I later found this application - optimization. Understanding how the computer works is crucial to writing clean, neat, and efficient code. This is important in production code as any code that is clean, neat, and efficient is generally scale-able and easy to maintain.

**If you're just going through this guide to learn how to automate tasks and do some basic scripting then you'll probably not need any of this information**. However, if you do intend to go into software development, then I

definitely recommend going through this section and doing some further research.

### 3.2.2 How is a computer structured

All the subsystems in a Von-Neuman system are interconnected by data highways called "buses." There are 3 primary buses that you need to be concerned with: the "data" bus, the "address" bus, and the "control" bus.

These buses are either uni-directional (only send data 1 way), bi-directional (send data both ways), or the can be both depending on the context.

# Chapter 4

# Starting with the basics

## 4.1 Getting started

This part is pretty simple but is important. You need to download your language of choice and get an IDE (Integrated Development Environment) or an editor of some sort - the second part isn't required, but is definitely good for quality of life.

My current go-to IDE for python is *Spyder*[1], I also love *VS Code*[2] for most other languages (and some python as well). If you're just getting started and just want to play around a bit, you can use the basic Python IDE.

[1] `https://www.spyder-ide.org/`
[2] `https://code.visualstudio.com/`

## 4.2 New file

## 4.3 if, then, else

*If* conditions make up the inner workings of conditional logic. The idea behind them is very simple: if a certain property is true then do a certain task. In other words, if the answer to a question is yes, then you get a certain response.

no

## 4.4 for and while

## 4.5 functions

# Chapter 5

# Data-structures

One crucial aspect to all programming languages is the data structures that they have to offer. We'll specifically look at Python and Java, however, these sometiems vary based on different languages.

## 5.1 Mutable vs Immutable

The first important thing to understand is mutability. In short, an **immutable** data structure/type is *unable to be changed*. Conversely, a **mutable** data structure/type can be changed.

This begs the question, why would we ever want to use an **immutable** data structure? Essentially, they take up less memory and increase speed. However, for most uses that we'll be going through today, the data type will be mutable.

## 5.2 Lists

The first important data structure is a list. In Python, these are **mutable**. Lists in Python also *do not care what type of data is in them.* This is different than other languages like C++ which can only contain one type of data in a list (i.e. you can't have a number and a word/string in the same list in C++, but you can in Python).

Lists allow us to reference and store data in many different ways and their use cases are countless. In fact, I'd go so far as to say that they're the most used data type/structure outside of simple variables.

In order to create a list in Python, we simpy use square brackets "[ ]". See Code Ex. 5.2 ‘ begdef

```
1  # Creating a list in Python
2
3  # The following is a list of integers
4  lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5
6
7  # The following is a list of characters
8  lst_2 = ['A', 'B', 'C', 'D']
9
10
11 # The following is a list of multiple data types
12 lst_3 = [1, 2.3, 'A', 'Word', True]
```

codex

## 5.3 Arrays

Arrays are like lists but are generally of set length and data type.

## 5.4 Tuples

Tuples in Python are like lists, however, they're completely immutable. This means that they *cannot be changed*; once they data they're initialized is in them, they cannot be changed.

## 5.5 Dictionary

This is just an implementation of a more complex datastructure called a hashmap. Hashmaps essentially take a **key** and associate them with a specific **value**.

# Chapter 6

# Simple Algorithms

# Chapter 7

# Practicing the basics

In order to get the hang of anything, I believe the best way to go about anything is to practice by actually implementing and applying the things you learnt. Therefore, I have provided a few practice projects in this chapter.

## 7.1 Calculator

## 7.2 Chatbot

## 7.3 Management System

## 7.4 Task Planner

# Chapter 8

# Learning some techniques

# Chapter 9

# Logical Thinking

One of the most important reasons to learn how to code is logical thinking and breaking down problems. Traditionally, when you're taught how to code, you're like a computer in itself in the sense that your instructor gives you instructions and you are expected to execute them.Q

## 9.1 Decomposition

One of the key ideas on computer science is operationalization or the decomposition of ideas. At the end of the day computer science is about solving problems. Solving a big complicated problem is just that - complicated. Therefore it becomes important that we break down big problems into smaller ones, then break those into even smaller ones, and so on. Essentially, we want a problem so simple that it's almost as easy as writing a function or even a single line of code.

This is probably the most idea in learning computer science because it applies

to life as a whole. If there is one take away in your learning computer science, let it be this.

Decomposition is a hard skill to learn, but is not difficult to master. It all boils down to practice. Hence, we will first through the "theory" on how to break an idea down. Then go through a few lay examples and then finally finish off with technical examples.

## 9.2 Planning

We can now take this idea of decomposition further and use it in planning. There are two primary "formal" methods we will cover - Psuedocode and Flowcharts. Both are important in different ways. The former allows use to *specifically* write out the logic of our code and how it works without committing it to a specific language instead being written in an English-adjacent language called pseudocode. While there are no universally specific guidelines for writing psuedocode, I will outline some traditionally accepted practices.

Flowcharts on the other hand allow us to *generally* write out the logic of our code. That is to say, we can more holistically see how are code works from a higher level. This allows others (and us) to understand how are code works. It also helps with the idea of decomposition which was covered in the previous chapter.

The overall objective of this chapter is to give you and introduction to both of these planning tools. Probably they're most important purpose is not solely helping you understand your code more but also helping others understand your work. Therefore, these methods greatly help with collaboration on bigger projects (which to many people's disdain, you're probably going to have to do eventually if you plan on doing almost anything meaningful).

### 9.2.1 Psuedocode

### 9.2.2 Flowcharts

## 9.3 Mathematics

# Chapter 10

# Two Camps

## 10.1 Functional

## 10.2 Object Oriented

# Chapter 11

# Functional Programming Crash Course

# Chapter 12

# Object Oriented Programming Crash Course

## 12.1 Classes, Methods, and Objects

## 12.2 Inheritance

## 12.3 Encapsulation

## 12.4 Abstraction

## 12.5 Polymorphism

# Chapter 13

# Hello World!

## 13.1 Where do I go from here?

## 13.2 Topic/resources to look into (by subject area)

### 13.2.1 Algorithms and Math

#### Formal Logic

#### Linear Algebra

Linear Algebra Done Right (Sheldon Axler)

### 13.2.2 Networking

### 13.2.3 Data Analysis

Sentdex