TECH TODAY

# UNLOCKING PYTHON

## A Comprehensive Guide for Beginners

RYAN MITCHELL

WILEY

# UNLOCKING PYTHON

# Unlocking Python

## A COMPREHENSIVE GUIDE FOR BEGINNERS

Ryan Mitchell

WILEY

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada and the United Kingdom.

ISBNs: 9781394288496 (Paperback), 9781394288519 (ePDF), 9781394288502 (ePub)



**Library of Congress Control Number:** 2024950121

Cover design: Wiley
Cover image: © CSA-Printstock/Getty Images

# ABOUT THE AUTHOR

**RYAN MITCHELL** is the author of *Unlocking Python* (Wiley) and *Web Scraping with Python* (O'Reilly). She has six LinkedIn Learning courses, including *Python Essential Training*, the leading Python course on the platform. An expert in web scraping, application security, and data science, Ryan has hosted workshops and spoken at many events, including Data Day Texas and DEF CON.

Ryan holds a master's degree in software engineering from Harvard University Extension School and a bachelor's in engineering from Olin College of Engineering. She is currently a principal software engineer at the Gerson Lehrman Group, where she does back-end development and data science on the AI & Data Platform team.

# ABOUT THE TECHNICAL EDITORS

**PETER HENSTOCK** is now a Senior Machine Learning Engineer at Incyte, after leading AI/ML efforts at Pfizer. His work has focused on the intersection of AI, visualization, statistics, and software engineering applied to drug R&D. Peter holds a PhD in artificial intelligence from Purdue University, along with seven master's degrees. He was recognized as being among the top 12 leaders in AI and pharma globally by the Deep Knowledge Analytics Group. He also currently teaches graduate AI and software engineering courses at Harvard.

**DAVIN POTTS** became a CPython Core Developer in March 2016. When he is not fielding questions about or helping folks do more with the Python language, he can be observed visiting large, dangerous snakes at the local zoo or watching episodes of *Monty Python's Flying Circus* on repeat.

# ACKNOWLEDGMENTS

**THE BEST TECHNICAL EDITORS ARE** the ones you dread bringing onto a project because you know it's going to be a lot of work. My extreme gratitude to Dr. Peter Henstock and Davin Potts for their voluminous and accurate feedback chapter after chapter. If any errata are sent in for this book, it's almost certainly because I overlooked one of their notes.

Thanks to James Minatel at Wiley for suggesting this book and seeing the project through. Thanks to Elizabeth Britten for cracking the whip over many months of writing and editing. Her flexibility and occasional crisis management were crucial.

My thanks and love to Edson Lacerda, John Mitchell, and Valinda Mitchell. You all hold my life together because I apparently haven't developed the skill of doing it myself yet. Love also to Viola and Hazel, who are my life.

Finally, thanks to Jim Waldo, who started this whole project many years ago when he mailed a Linux box and a copy of *The Art and Science of C* to a young and impressionable teenager.

# CONTENTS

# PART I
# Programming

# 1

# Introduction to Programming

*The computer programmer is a creator of universes for which he alone is the lawgiver. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or field of battle and to command such unswervingly dutiful actors or troops.*

— Joseph Weizenbaum

The programming book has a curious place in the modern world. With Python documentation, tutorials, and entire courses available online, what is the purpose of long-form text on the subject?

Similarly, what is the purpose of, say, historian John Keegan's book *The First World War* when a straightforward timeline of battles is readily available? This question is, obviously (at least, I hope), rhetorical. Books convey the author's point of view, insights, colorful commentary, and perhaps even add a little entertainment to the mix. They present facts, expand on those facts, and bring them to life.

For most people, the concept of "bringing history to life" probably makes more visceral and immediate sense than the concept of "bringing programming to life." This is unfortunate.

The goal of this book is to teach you to write Python programs, yes, and also to discuss the history, culture, and context of Python, the machines that run it, and the people who write it. The first step to understanding a programming language isn't simply copying that first line of code and hitting the Go button—it's learning what the computer is doing while that first line of code is running.

Whatever your motivations for learning Python, I hope this text can inspire in you the same enthusiasm I have for Python, and programming in general. In some small way, perhaps it can bring it to life.

# PROGRAMMING AS A CAREER

"If you could be anyone in King Arthur's court, who would you be?" was the question posed to me one weekend afternoon in the kitchen of my childhood home by my uncle, a software engineer visiting from Massachusetts.

As a teenager, my knowledge of King Arthur was mainly limited to the film *Monty Python and the Holy Grail*, so even thinking of suitable answers was difficult. There was Lancelot, Galahad, and all the rest of the knights, but none of them seemed especially worth becoming. "King Arthur" himself would have been a fun way to respond, but surely, there must be a more interesting character. What about the Lady of the Lake—the kingmaker herself?

Of course, in some versions, such as the 1963 Disney classic *The Sword in the Stone*, Arthur becomes king, not by women in ponds distributing swords, but by pulling Excalibur out of a stone. I thought about that movie, too.

And, certain that my answer was cheating somehow, I finally gave my uncle the snarky response: "Merlin."

This, of course, is the correct answer. Years later, my uncle said that it was because of this answer that he knew my future profession as a programmer was inevitable.

The association between programming and wizardry is a long one. And the association between programmers and snarky responses is even longer.

Programmers shape the world we view through our screens and create new realities seemingly out of thin air. They realize that the boundaries of the virtual world are artificially constructed. They either implement the rules passed down by kings or subvert them as they choose. They perform feats by mastering the arcane languages.

The power of King Arthur depends on external factors: his fame, wealth, political clout, and societal recognition of his authority. The power that Merlin has is intrinsic.

Of course, even if my response had been "King Arthur," I still would have become a programmer. Software engineers giving single-question personality tests to teenagers is no basis for career determination, just like yanking a sword out of a rock doesn't make you a king.

## Myths About Programmers

Let's be honest: Many aspiring programmers get interested in the field tempted by big paychecks, job security, great benefits, and favorable working conditions. These things are all nice, to be sure. But getting hundreds of thousands of dollars a year from a big-name company is far more challenging than most programming bootcamps would have you believe. Also very common (but less lauded in the media) are entry-level programmers working long hours for small companies, nonprofits, and start-ups whose funding allows them a runway measured in months.

As a programmer you will experience both good times and bad times. Times when you are working from your couch on very easy and enjoyable projects for lots of money. And times when you are, perhaps volunteering (or working for so little in return that it's essentially volunteering), on stressful projects in an office with a long commute and a strict dress code.

So, what really differentiates programmers and nonprogrammers? The people who stick it out and have long, happy careers, and the ones who give up in frustration? I want to address a few industry misconceptions about what's "required" to be a programmer:

**They Are Very Smart** I can't blame you if you think that programmers need to be very smart, based on typical media depictions of programmers. Programmers are not required to be geniuses, or even to be particularly nerdy.

For the vast majority of you learning to program, there are going to be certain problems you're not going to understand right away, certain concepts you're just not going to get. And these have nothing to do with some fundamental deficiency in your brain.

Programming is a marathon, not a sprint. Determination and consistency with always be more valuable in programing than genius.

**They Work Very Hard** This stereotype is shifting somewhat thanks to an increase in remote work and contract jobs available for programmers. However, there is still a perception that programmers are glued to their computers all day, every day, working insane hours under lots of pressure to fix a bug or add a feature.

There is also the complicating factor that it can be difficult to estimate how much time is required to add a feature or fix a bug. If you are working for a small and/or mismanaged company, you may occasionally find yourself in a situation where a feature is *required* to be added by a certain deadline; complications arise, ballooning the hours required to add it; and you are the only one who can rescue the project.

But this is not a common situation, and if you find yourself doing this more than once or perhaps twice a year, consider finding new employment. In general, working as a programmer is the same as working anywhere else; in a well-managed company with good work-life balance, the hours are about the same as they are for any other profession.

Programmers do sometimes face pressure at "FAANG" companies: Facebook, Apple, Amazon, Netflix, and Google.[1] Depending on your team, company, position, and the current politics and/or economic environment, you may find yourself coasting into an easy retirement or under constant pressure to perform at the threat of losing your job. I have heard many versions of both stories, even from employees within the same company! But, again, a high-stress job is not a given, even at a large and competitive company while getting a large paycheck.

**They Have Meticulous Attention to Detail** If a program is off by even a single character, it will not perform as expected. No typos or errant logic is allowed. Even an insignificant-seeming bug can bring down a system that people's lives depend on.

Technically, the above is all true. However, it's also a very disingenuous view of the realities of modern programming. For starters, we have software that finds and highlights errors as we write the code (for more information, see Chapter 12, "Writing Cleaner Code"). It's somewhat difficult to make a simple syntax error or to forget a character—the code must still be syntactically valid in order for it to run at all. At the very least, you should probably make sure it runs—not that it produces the correct output but merely runs at all—before it goes anywhere important.

[1] There are a variety of acronyms for these companies, and definitions of the "top" tech companies, and even the names of the companies themselves, change frequently. Some have also suggested replacing FAANG with MAMAA for Meta, Amazon, Microsoft, Alphabet, and Apple, reflecting the parent companies of Facebook and Google, as well as replacing Netflix with Microsoft. With this in mind, feel free to mentally replace "FAANG" with whatever definition of "popular tech giants" you see fit.

Programs are usually run in various testing environments over many scenarios. New features get multiple stages of testing and review before they go to production. And that's just for the code that doesn't have any lives depending on it!

I know many meticulous, detail-oriented programmers. But to be honest, I don't know anyone who would describe me similarly.

The great thing about computers is that, when something does go wrong, it will tell you what went wrong and where (at least to some extent). You see an error message, a stack trace, or at least an output that's different than what you were expecting. Nothing blows up, nobody dies, you're just sitting there at your computer with a puzzle to solve and something to fix.

Can extreme conscientiousness be an asset in programming? Absolutely! Is it required? Not at all.

**They Are Good at Math** The nice thing about computers is that they do the math for you. If you frequently find yourself struggling with arithmetic, you can rest assured that "mental math" has no bearing on programming aptitude.

However, there is one intersection between programming and what most people may have encountered in a high school math class, and that is *word problems*. Programming is, in many ways, the art of modeling real-world scenarios with algorithms. If you had an aptitude for word problems, you may enjoy programming.

**They Start Young** Many well-educated and affluent parents are flocking to programming toys, books, and games as a way to get their children—even toddlers—a leg up in programming and computer science. Toy manufacturers are, of course, extremely happy to meet this demand.

But is it necessary or even beneficial to give a three-year-old a Fisher Price Code-a-Pillar? Will your programming career suffer because you never learned Scratch in middle school or played with toy robots?

STEM (science, technology, engineering, and math)-branded toys are relatively new, rising in popularity in the early 2010's. Because of their relatively-recent appearance, there aren't any studies (at least none that I could find), that link early use of programming toys with successful programming careers in later life.

A study comparing groups of children playing with robot programming toys to groups playing with blocks[2] found that the "robot" group did slightly better on tests of computational thinking afterwards. However, both groups improved overall. You don't need robots to teach computational thinking—you can use wooden blocks.

When I think back on formative experiences that helped me become a better programmer, I think of volunteering at my school library. I had to reshelve books (where I invented the insertion sort), use a card catalog (a secondary index), and look up things in the giant dictionary (binary search).

It isn't surprising to me that some of the best programmers I know majored in library science, philosophy, and journalism rather than computer science or software engineering and only started programming as part of a career change in adulthood.

[2] Yang, W., Ng, D. T. K., & Gao, H. (2022). Robot programming versus block play in early childhood education: Effects on computational thinking, sequencing ability, and self-regulation. *British Journal of Educational Technology*, 53, 1817–1841. https://doi.org/10.1111/bjet.13215

> Programming is simply applied logic, with a smattering of easy-to-learn syntax. This book will teach you the syntax (and a few best practices), but programming is really a skill that takes a lifetime to master. I think that childhood is better spent practicing logic, algorithmic thinking, and creative problem-solving outside the context of programming than it is spent learning some invented and infantilizing syntax for a nonsense programming language that will never be used again.

## HOW COMPUTERS WORK

I do not know how computers work. I've met very few people who might know how they work, but I'm sure that even those rare individuals have blind spots. Computers work because of math, yes, but also because of physics and chemistry. Silicon impregnated with boron has interesting interactions with electrons, which ultimately allow us to do math.

As you might suspect, you don't need to know any of this to do programming, which is why not very many people know how computers work.

What this section is really about is the mental models that programmers rely on to understand what's appearing on the screen in front of us and how to control it. The shorthand, the analogies, and even the occasional lore. While none of these stories may be completely precise, my goal is that you find them useful.

The first computers were essentially calculators that took input in the form of switches and dials that could be set to one value or another. You input the numbers you want with the dials, and the operation you want performed on them (addition, subtraction, multiplication, etc.), press the "go" button, and the output is calculated.

Then, an innovation: What if, instead of setting these with switches and dials each time we wanted to use them, the computer could "write the numbers down" for later use? The numbers would be stored in some section of the computer's memory, and then we tell the computer to perform an operation on those stored numbers.

But what is an "operation"? Could that be written down as well, alongside the numbers and stored until we want to access it? Perhaps we could write down and save whole lines of numbers and operations and call them "instructions."

```
Retrieve value at memory location 31415
Retrieve value at memory location 27182
Add the two values together
```

It's important to understand these two concepts of storing and retrieving data as well as performing operations on that data. These two things are at the core of what computers do. In fact, they're really *all* that computers do. A computer is simply a machine that stores and retrieves values and performs operations on those retrieved values.

The values have changed significantly in the last 90 or so years that digital computers have been around. Instead of numbers we want to add, they may also be images, keyboard input, mouse clicks, streaming data from a remote server on the internet, and so on.

The operations have changed as well. Instead of adding two numbers together, we are compressing files, doing machine learning, or running Adobe Photoshop. However, at their core, computers and the programming languages that control them are still doing the same thing: setting values, retrieving values, and performing operations on those values. Take for example, this line of Python:

```
a = 2
```

The equals sign here is called an *assignment* in programming. It assigns values into variables. Here, the integer value `2` is being assigned to the variable `a`. The computer stores the value `2` in memory and then records the variable `a` as pointing to the location in memory where that value `2` is stored.

Not only is the value, `2`, stored, but stored alongside it is information that tells us that it is a number, as opposed to a piece of text, or an MP4 file, or some other data. This tells the computer what things can be done with it, as an integer. We can use Python to perform addition and multiplication on the integer `2`, but cannot, say, play it in a video player.

We can perform an operation:

```
a = a + 1
```

This tells the computer to retrieve whatever value the variable `a` is referencing, add the number `1` to it, and then store it back in memory so that the variable `a` is pointing to the new value.

As we're programming, we don't literally need to think about these things all of the time. But having this mental model does come in handy. Take this situation, for example:

```
a = [1,2,3,4]
```

The square bracket notation with comma-separated numbers denotes a *list* in Python. Rather than a single number, like 1 or 2, we can store a whole array of them. This entire list, `[1,2,3,4]`, is assigned to the variable `a`.

Let's make another assignment:

```
b = a
```

This introduces a new variable, `b`, and assigns it to the same location in memory that `a` is pointing to. They both point to this same exact list of numbers. This is illustrated in Figure 1.1, which shows the variables `a` and `b` in a piece of Python code, pointing to a shared location in memory.

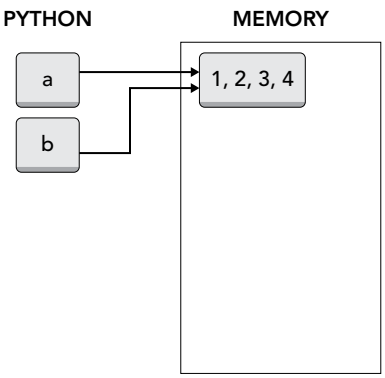


**FIGURE 1.1:** Variables `a` and `b` both point to the same location in memory when `a` is assigned to `b`.

Let's create a new instruction, written in the English language for the sake of example, and which we can pretend the computer understands how to execute:

```
append the number 5 to the end of list b
```

This adds the number 5 to the end of the list that variable `b` is referencing. Now, what happens when we view the list referenced by variable `a`? Because both variables are assigned to the same location in memory, both `a` and `b` now contain the list value:

```
[1,2,3,4,5]
```

This is because variables contain references, or *pointers*, to values rather than containing intrinsic values themselves. Every variable references a location in the computer's memory, and variables can also reference the same locations in memory if they're assigned to each other.

Files work in much the same way as values in memory. When you store a file, such as an image or text document, you may think that you are storing it at a "location" like `C:\Users\RMitchell\Documents` or `/Users/RMitchell/Documents`.

However, what this file path directly references is not the file data itself, but a disk location for where that file data starts and also a file size—how many bits must be read past the starting point in order to load the entire file.

The contents of a file in your `Documents` folder might be mixed in on disk right next to a bunch of files from your `Pictures` folder or `Downloads` folder. It's only because of the organized way your computer tracks all of these references to disk locations and accurately retrieves them that gives the appearance of an orderly nested filesystem.

Although data in memory and files on a disk work in similar ways, using pointers and references, they are very different things. Files are stored on your computer's hard drive; this is often called *storage*. Files in storage persist when the computer turns off. Data in memory is faster to access and use, but it does not persist when the power turns off. Data in a running program is lost if the computer reboots and the program is not able to save it to storage.

Granted, as technology progresses, the line between "storage" and "memory" does get somewhat blurred. Some computers may have nonvolatile or persistent memory that does not get erased when the computers turn off. Some computers may have persistent file storage that is faster than the memory of most other computers. Hardware technicalities aside, most operating systems still carry these concepts of memory and storage forward, and so this is how we tend to think of them as programmers.

Importantly, the data that computers place into both memory and storage is in the form of binary values, 1s and 0s. If you open a computer, you're not literally going to see the character "1" and the character "0" written into a disk anywhere, but they're stored as an electrical or magnetic charge. You can think of a 1 as "high charge" and a 0 as "low" or "no charge."[3]

[3] As with many of the concepts we encounter in this section, exactly how these are stored with electricity, magnets, and silicon is highly dependent on the technology and is of little importance for our purposes. But it may be helpful to think of a 1 as "charged" and a 0 as "uncharged."

Human language is made up of more than just 1s and 0s, so if you want to store letters, books, or just about anything else, you're going to need some sort of substitution. The clever computer scientists came up with just that:

```
A - 01000001        a - 01100001
B - 01000010        b - 01100010
C - 01000011        c - 01100011
```

And so on. This is called the ASCII (American Standard Code for Information Interchange) alphabet, and it's how text files are stored.

Each one of these letters (or *characters* as computer scientists call them) is composed of 8 binary values, or 8 bits. The measurement 8 bits is so commonly used that it has its own special name: a byte. Count all the possible combinations you could make out of a byte: `00000000`, `00000001`, `00000010`, `00000011`, etc. You will arrive at the value 256. There are 256 possible combinations in 8 bits, which means there are 256 possible characters you can represent with an 8-bit block of memory.

It's not a coincidence that $2^8 = 256$, and the number of bits we are working with is 8. In general, if you have $n$ bits there are $2^n$ possible combinations. This is because the first bit gives you two options (0 or 1), the second bit doubles that (0 or 1 with a 0 or a 1 in front of it), the third bit doubles it again, until you end up with $2 \times 2 \times 2 \times \ldots$ however many bits you have.

When computers store numbers, rather than characters or letters, they also have to use a binary representation of the number. When we write numbers, we use the base 10 or decimal system to do it.

In the decimal system, the number 1,234 has a 1 in the thousands place, a 2 in the hundreds place, a 3 in the tens place, and a 4 in the ones place. Or, more mathematically, it has a 1 in the $10^3$'s place, a 2 in the $10^2$'s place, a 3 in the $10^1$'s place, and a 1 in the $10^0$'s place (see Figure 1.2).

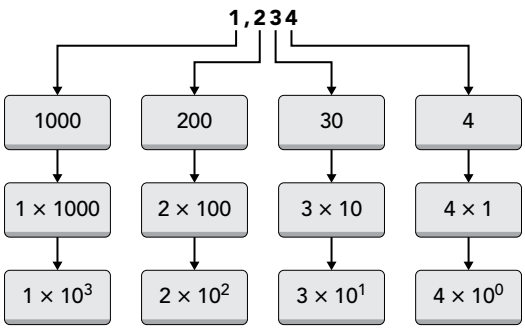


**FIGURE 1.2:** The number 1,234 broken down into the "ten to the thirds" place, "ten squareds" place, "ten to the firsts" place, and "ten to the zeroth" place

We can write decimal numbers like this because there are 10 possible characters for us to choose from for each "place." But what if there were only two possible characters, a 1 and a 0? In that case,

we would have to use the binary or base 2 system. The number `10101` has a 1 in the $2^4$'s place, a 0 in the $2^3$'s place, a 1 in the $2^2$'s place, a 0 in the $2^1$'s place, and a 1 in the $2^0$'s place (see Figure 1.3). Therefore:

```
10101 (binary)  = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 2 * 2^1 + 1 * 2^0 = 21 (decimal)
```

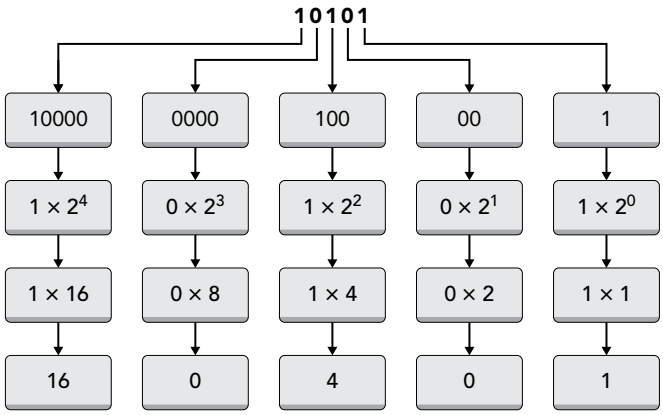


**FIGURE 1.3:** The binary number 10101 broken down into the "two to the fourths" place, "two to the thirds" place, "two squareds" place, "two to the firsts" place, and "two to the zeroth" place

In the same way that humans can write down any possible number with only the 10 digits 0 through 9, computers can store any possible number with just 2 bits (binary digits): 0 and 1. The bits just take up a little more space.

Very rarely do programmers need to actually do any of this binary math. But understanding exactly how the data is represented is invaluable. For example, the word `four` as text will take more room to store on disk than the number 4. The word `four` has four ASCII characters in it, where each character requires 8 bits, for a total of 32 bits:

```
01100110 01101111 01110101 01110010
   f        o        u        r
```

The number 4 can be represented simply as 100, for a total of 3 bits.

Theoretically, this is correct and important to understand. However, in practice, it's a little bit disingenuous. Although the number 4 only needs 3 bits, the number 1,000,000 needs 19 bits. And the computer doesn't know how big the number is we're trying to store or how to separate the numbers in memory. For instance, the sequence `100101` could be read separately as `100` and `101` (4 and 5 in decimal), or it could be read as a single number, `100101` (37 in decimal), or some other combination.

To solve this problem, integers in many programming languages are declared to be 4 bytes, or 32 bits, in length. This allows integer numbers up to $2^{32}$ or 4,294,967,296 to be stored.[4] Therefore, the integer number 4 would be:

```
00000000 00000000 00000000 00000100
```

This makes it just as long, in practice, as the ASCII word representation `four`. However, when dealing with larger numbers, you'll find that storing them as integers is far more efficient. Consider, for example, that 1,234,567 written out as "`one million two hundred thirty four thousand five hundred sixty seven`" is 69 bytes long!

# A BRIEF HISTORY OF MODERN COMPUTING

There is a long and rich history of programming before 1970. However, the development of the languages and software that still impact us today began in the early 1970s with the release of the operating system Unix and the programming language C.

In fact, the date January 1, 1970, itself has important significance in modern computing. When computers store dates and times, they often store them as "the number of seconds since midnight on January 1st 1970." The time 0 is 01/01/1970 00:00:00, the time 3600 is 01/01/1970 01:00:00, and the time 1,000,000,000 is 09/09/2001 01:46:40. This is called *Unix time* or *epoch time* and it is a standard recognized by every computing system today.

So, for all intents and purposes, I like to think of the world as beginning at time 0 on midnight January 1, 1970. Back then, operating systems weren't much more than a command-line interface. You get a prompt, give it a command to execute, and the operating system interprets the command and executes it.

Operating systems themselves were varied and were often developed by computer manufacturers for their specific hardware. Therefore, the commands that were interpreted by these operating systems were also extremely varied, as were the programming languages they used and understood.

## The Unix Operating System

Unix was originally developed by three programmers at Bell Labs in a relatively small-scale and modest project. Fortunately for the history of computing, an antitrust case in 1956 brought against AT&T by the Eisenhower administration meant that any non-telecommunications patents and documentation had to be licensed to all applicants at reasonable royalty rates. Because Bell Labs was a subsidiary of AT&T and Unix was definitely not a piece of telecommunications software, this meant that Unix had to be licensed and made open source and could not be kept as proprietary and expensive like many other operating systems of the time.

Unix quickly spread throughout academia, which welcomed a cheap and universal operating system. Academics, after graduating, spread it to the companies they worked at, where it gained popularity in the corporate world as well.

[4] This is an oversimplification and ignores the existence of negative numbers. In Python, the size of integers is complicated and not something you generally need to worry about. However, the concept of numbers being a relatively large default size (often 4 bytes) does hold true across programming languages.

A large part of Unix's appeal was that it was written in the C programming language, which was also gaining popularity. Although C was not the first high-level[5] modern programming language, or even the first high-level famous language (that would be FORTRAN), C has outlasted its predecessors and is still in common use today. C is also what most popular modern languages, such as C#, C++, Go, Haskell, Java, JavaScript, Perl, PHP, and Python, are derived from today.

Although modern operating systems are not derived from, or "based on" Unix in the sense that they directly use Unix, many of their principles of operation and the commands they use are based on Unix commands. In fact, many of the commands you'll learn and work with later in this book such as `cd`, `ls`, and `mkdir`, were developed for Unix and later adopted by many other operating systems.

## Modern Programming

When you write a Python program, the Python code that you write gets translated by the computer into C code before being executed. In fact, much of the Python language itself is written in C. You can see an example of some of the C code that runs Python on the Python GitHub page: `https://github.com/python/cpython/blob/main/Python/getversion.c`

If the computer has to translate Python into C code before it can be run, why don't we simply write all programs in C? If written correctly and competently, a program written in C will usually be faster than the same program written in Python. However, C's syntax is much more complex. It also relies on the programmer to perform low-level tasks, such as allocating and releasing memory, that are taken care of in Python.

Prior to 1970, programming was very hardware-specific. A programmer might have to learn new languages or new ways to write those languages as new computers were released. The wide adoption and portability of C led to a flourishing of new programming languages that focused on style and architecture of the code, rather than the underlying hardware. Programming languages could be designed for specific types of applications and specific types of programming philosophies.

In the mid-1980s, higher-level languages like Objective-C and C++ were released. These languages used a programming style that focused on relationships between, and attributes of, objects or entities.[6] This style is called *object-oriented programming*, and it grew quickly in popularity over the next decades, led by languages like Java and C#, which came on the scene in the mid-1990s.

In 1995 JavaScript was introduced, which revolutionized a completely different area of computing: the internet. JavaScript allowed web pages to become dynamic and have programmable content for the first time.

[5] "High-level" and "low-level" languages will be discussed in more detail later; for now, know that a high-level programming language is, essentially, one that is written more like plain English. It has many abstractions and features that hide details of the underlying operating system and hardware, making it easier to write programs. Python is considered a high-level language, whereas assembly is considered a low-level language.

[6] Objects are values in computer science that have a complex structure to them. Unlike simple values like numbers or words, objects might have an elaborate schema with many attributes and functionality associated with them. For example, you might have a "user" object or a "blog post" object which contains all the information associated with that user or blog post. Don't worry about objects too much for now; we'll be revisiting them shortly and throughout the rest of the book!

Up until now, programs had to be executed by specialized software that the user purposefully installed and ran. Or it was compiled into an executable file that could only be run by a very specific operating system. Now, no special software or specific operating system was needed, the web browser itself executed JavaScript. If you had a web browser, you could run JavaScript on a web page.

This required cooperation from every company that had a web browser. The JavaScript code had to execute in the same way whether users visited a website with Internet Explorer or Netscape. However, it provided tremendous benefit to users, who could now use dynamic web applications powered by JavaScript.

Although the name "JavaScript" is similar to "Java," the two languages have nothing to do with each other. A popular saying in programming is that "Java is similar to JavaScript in the same way that a car is similar to a carpet." This isn't completely true, after all, Java and JavaScript are at least both programming languages. However, JavaScript was named after Java for marketing reasons, rather than any meaningful connection between the two languages.

Although several other languages attempted to compete with, or at least complement, JavaScript on the web (notably Java Applets and Flash Player), none succeeded in the long run. Today, JavaScript is the only in-browser language available for programming websites.

In 2009, Node JS was released; which allowed JavaScript to be run outside a web browser and power any code running on a computer. Now, it is one of the most popular general-purpose programming languages, alongside Python, Java, and C#.

## TALKING ABOUT PROGRAMMING LANGUAGES

With so many programming languages in the world, a set of jargon was developed to help quickly describe and differentiate programming languages. If one programmer is trying to explain to another why they should learn Python, they might say, "It's a dynamically typed, garbage-collected multiparadigm language."

But what does this mean? Understanding these words doesn't just help us understand Python—it helps us understand how programmers think about programming languages in general.

This section, at first glance, may look more like a glossary to be used as needed than something to be seriously perused. However, I encourage you to read each term and try to understand it, at least in an abstract way. I've put the terms into logical groupings for ease of comparisons.

**Open Source** An *open source* programming language, like any piece of open source software, is one where you can read the source code. It's a popular misconception that open source necessarily means "free." This is usually the case; however, you must still check the software's *license*, which will dictate its terms of use.

If you cannot read the source code, the software is called *closed-source*. Java, .NET, and C++ are all closed-source programming languages.

**Garbage-Collected** Previously, we looked at how computers store values in memory. Before the values can be written in memory, that memory must be *allocated* for the values that will be stored. This allocation process makes sure that no other programs are using the memory (the memory is free) and then reserves the memory so that no other programs can use it.