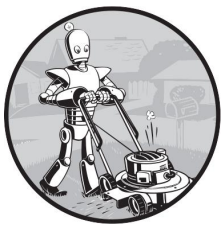
INTRODUCTION



"You've just done in two hours what it takes the three of us two days to do." My college roommate was working at a retail electronics store in the early 2000s. Occasionally, the store would receive a spreadsheet of thousands of product prices from other stores. A team of three employees would print the spreadsheet onto a thick stack of paper and split it among themselves. For each product price, they would look up their store's price and note all the products that their competitors sold for less. It usually took a couple of days.

"You know, I could write a program to do that if you have the original file for the printouts," my roommate told them when he saw them sitting on the floor with papers scattered and stacked all around.

After a couple of hours, he had a short program that read a competitor's price from a file, found the product in the store's database, and noted whether the competitor was cheaper. He was still new to programming, so he spent most of his time looking up documentation in a programming book. The actual program took only a few seconds to run. My roommate and his co-workers took an extra-long lunch that day.

This is the power of computer programming. A computer is like a Swiss Army knife with tools for countless tasks. Many people spend hours clicking and typing to perform repetitive tasks, unaware that the machine they're using could do their job in seconds if they gave it the right instructions.

Who Is This Book For?

Software is at the core of so many of the tools we use today: nearly everyone uses social networks to communicate, virtually all people have internet-connected phones in their purse or pocket, and most office jobs involve interacting with a computer to get work done. As a result, the demand for people who can code has skyrocketed. Countless books, online tutorials, and developer boot camps promise to turn ambitious beginners into software engineers with six-figure salaries.

This book is not for those people. It's for everyone else.

On its own, this book won't turn you into a professional software developer any more than a few guitar lessons will turn you into a rock star. But if you're an office worker, administrator, academic, or anyone else who uses a computer for work or fun, you will learn the basics of programming so that you can automate simple tasks such as these:

- Moving and renaming thousands of files and sorting them into folders
- Filling out online forms—no typing required
- Downloading files or copying text from a website whenever it updates
- Having your computer text custom notifications to your phone
- Updating or formatting Excel spreadsheets
- Checking your email and sending out prewritten responses
- Creating databases and querying them for information
- Extracting text from images and audio files

These tasks are simple but time-consuming for humans, and they're often so trivial or specific that there's no ready-made software to perform them. Armed with a little bit of programming knowledge, however, you can have your computer do these tasks for you.

Coding Conventions Used in This Book

This book is not designed as a reference manual; it's a guide for beginners. The coding style sometimes goes against best practices (for example, some programs use global variables), but this trade-off makes the code simpler to learn. Sophisticated programming concepts—like object-oriented programming, list comprehensions, and generators—aren't covered because of the complexity they add. Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort on your part.

What Is Programming?

Television shows and films often show programmers furiously typing cryptic streams of 1s and 0s on glowing screens, but modern programming isn't that mysterious. *Programming* is writing instructions for the computer to perform in a language the computer can understand. These instructions might crunch some numbers, modify text, look up information in files, or communicate with other computers over the internet.

All programs use basic instructions as building blocks. Here are a few of the most common ones, in English:

"Do this; then do that."

"If this condition is true, perform this action; otherwise, do that action."

"Do this action exactly 27 times."

"Keep doing that until this condition is true."

You can combine these building blocks to implement more intricate decisions too. For example, here are the programming instructions, called the *source code*, for a simple program written in the Python programming language. Starting at the top, the Python software runs lines of code (some of which are run only *if* a certain condition is true, or *else* Python runs some other line) until it reaches the bottom:

```
❶ password_file =
open('SecretPasswordFile.txt')
❷ secret_password = password_file.read()
❸ print('Enter your password.')
typed_password = input()
❹ if typed_password == secret_password:
    ❺ print('Access granted')
    ❻ if typed_password == '12345':
        ❼ print('That password is one that an
idiot puts on their luggage.')
  else:
    ❽ print('Access denied')
```

You might not know anything about programming, but you could probably make a reasonable guess at what the previous code does just by reading it. First, the file *SecretPasswordFile.txt* is opened ❶, and the

secret password in it is read ❷. Then, the user is prompted to input a password (from the keyboard) ❸. These two passwords are compared ❹, and if they're the same, the program prints *Access granted* to the screen ❺. Next, the program checks whether the password is *12345* ❻ and hints that this choice might not be the best for a password ❼. If the passwords are not the same, the program prints *Access denied* to the screen ❽.

Programming is a creative task, as are painting, writing, knitting, and constructing LEGO castles. Like a blank canvas for painting, software has many constraints but endless possibilities. The difference between programming and other creative activities is that your computer comes with all the raw materials you need to program; you don't have to buy any additional canvas, paint, film, yarn, LEGO bricks, or electronic components. A decade-old laptop is more than powerful enough to write programs. Once you've written your program, you can copy it perfectly an infinite number of times. A knit sweater can be worn by only one person at a time, but a useful program can easily be shared online with the entire world.

What Is Python?

Python refers to both a programming language (with syntax rules for writing what is considered valid Python code) and the interpreter software that reads source code (written in the Python language) and performs its instructions. You can download Windows, macOS, and Linux versions of the Python interpreter for free at *https://python.org*.

There are several programming languages, each with their strengths and weaknesses. Debating which is best often leads to pointless arguments over matters of opinion. But in my opinion, Python is the best *first* language to learn if you are new to programming. Python has a gentle learning curve and readable syntax. It doesn't require learning dense concepts to do simple tasks. And if you want to go further into programming, learning a second language is easier when you first understand Python.

The name Python comes from the surreal British comedy group Monty Python, not from the snake. Python programmers are affectionately called Pythonistas, and both Monty Python and serpentine references usually pepper Python tutorials and documentation.

Common Myths About Programming

Programming has an intimidating reputation, and billion-dollar software companies are household names. Even the English word *code* has an association with secrecy and cryptic connotations. This leads many people to think that only a select few can program. But coding is a skill

anyone can learn, and I'd like to address some of the more common myths directly.

Programmers Don't Need to Know Much Math

The most common anxiety I hear about learning to program is the notion that it requires a lot of math. Actually, most programming doesn't require math beyond basic arithmetic. Programming requires deduction and paying attention to detail more than mathematics. In fact, being good at programming isn't that different from being good at solving Sudoku puzzles.

To solve a Sudoku puzzle, you must fill in the numbers 1 through 9 for each row, each column, and each 3×3 interior square of the full 9×9 board. The puzzle provides you with some numbers to start, and you can find a solution by making deductions based on these numbers. In the puzzle shown in Figure 1, a 5 appears in the first and second rows, so it can't show up in these rows again. Therefore, in the upper-right grid, it must appear in the third row. Because the last column also already has a 5 in it, the 5 can't go to the right of the 6, so it must go to the left of the 6. Solving one row, column, or square will provide more clues for solving the rest of the puzzle, and as you fill in one group of numbers 1 to 9 and then another, you'll soon solve the entire grid.

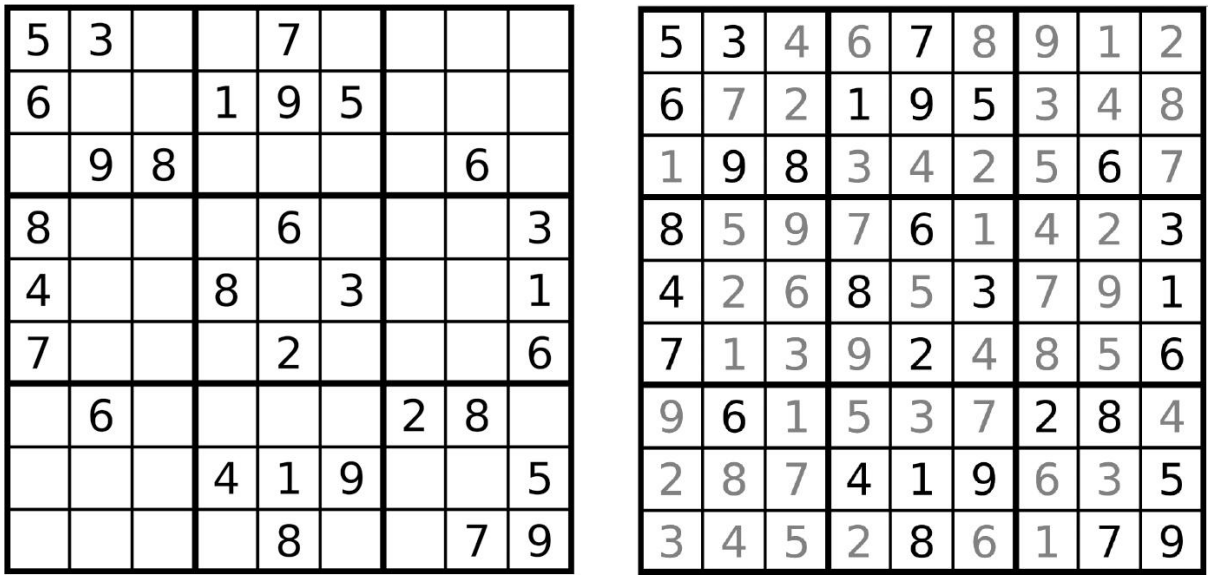


Figure 1: A new Sudoku puzzle (left) and its solution (right). Despite using numbers, Sudoku doesn't involve much math. (Images © Wikimedia Commons)

The fact that Sudoku involves numbers doesn't mean you have to be good at math to figure out the solution. The same is true of programming. Like solving a Sudoku puzzle, writing programs involves paying attention to details and breaking down a problem into individual steps. Similarly, when *debugging* programs (that is, finding and fixing

errors), you'll patiently observe what the program is doing and find the cause of the bugs. And like all skills, the more you program, the better you'll become.

You Are Not Too Old to Learn Programming

The second most common anxiety I hear about learning to program is that people think they're too old to learn it. I read many internet comments from folks who think it's too late for them because they are already (gasp!) 23 years old. This is clearly not "too old" to learn to program; many people learn much later in life.

You don't need to have started as a child to become a capable programmer. But the image of programmers as whiz kids is a persistent one. Unfortunately, I contribute to this myth when I tell others that I was in grade school when I started programming.

However, programming is much easier to learn today than it was in the 1990s. Today, there are more books, better search engines, and many more online question-and-answer websites. On top of that, the programming languages themselves are far more user-friendly. For these reasons, *everything I learned about programming in the years between grade school and high school graduation could be learned today in about a dozen weekends*. My head start wasn't really much of a head start.

It's important to have a "growth mindset" about programming—in other words, understand that people develop programming skills through practice. They aren't just born as programmers, and being unskilled at programming now is not an indication that you can never become an expert.

AI Won't Replace Programmers

In the 1990s, nanotechnology promised to change everything. New manufacturing processes and scientific innovation would revolutionize society, the thinking went. Carbon nanotubes, buckyballs, and diamonds the price of pencil lead, all assembled one atom at a time out of plentiful carbon by germ-size nanobots, would pave the way for materials 10 times stronger than steel at a fraction of the weight and cost. This would lead to space elevators, medical miracles, and home appliances that could create anything, just like the replicators on *Star Trek*! It would mean an end to economic scarcity, world hunger, and war. It would bring on a new age of enlightenment—as long as the nanobots didn't turn on their human creators in a tiny robot uprising. And the technology was only 10 years away!

Of course, the hype never happened. Real innovations certainly occurred at the nanoscale (the smartphone in your pocket uses a number

of them). But the Star Trek replicators and other grandiose promises didn't arrive, and the excitement over nanotechnology deflated to more realistic proportions.

Let's talk about AI.

Personal computers changed everything. The internet changed everything. Social media changed everything. Smartphones changed everything. Cryptocurrency did not change everything, but it did reveal which of your cousins and co-workers were susceptible to get-rich-quick scams. Today, AI is the latest marvel to emerge from the tech industry. People use the term *AI* to mean everything from chess-playing computers to chatbots, so-called expert systems, and machine learning. In this book, I'll use the term *large language model (LLM)*, which is the conceptual category behind OpenAI's ChatGPT, Google's Gemini, Facebook's LLaMA, and other generative text systems.

LLMs have caused many breathless claims and questions. Is AI going to take all of our jobs? Is it still worth learning to code? Are the AIs alive, and can I survive the AI-robot uprising?

LLM technology is exciting, but to make it useful, we need to set realistic expectations so that we can avoid falling prey to sensationalist journalism and questionable "investment opportunities." I hope I can deflate the hype and give you a more realistic view of how LLMs can help you learn to code. Let's get some of these misconceptions out of the way right now:

- LLMs are not conscious or sentient.
- LLMs will not replace human software engineers.
- LLMs do not alleviate the need to learn programming.
- LLMs will not replace most human jobs (though this won't prevent your manager from thinking they can and laying you off anyway).
- LLMs are far from perfect, and are even *often* wrong.

Those who insist on these misconceptions get their information from science fiction movies and internet videos, not from experience with actual LLMs. I highly recommend Simon Willison, Python Software Foundation board member and co-creator of the Django Web Framework, for his writing about AI at *https://simonwillison.net/about*. You can watch his sober and illuminating PyCon 2024 keynote speech on LLMs at *https://autbor.com/pycon2024keynote*.

How could LLMs help you as a programmer? What is obvious at this early stage is that learning to communicate with LLMs (so-called prompt engineering) is a skill, just like learning to effectively use a search engine is a skill. LLMs are not people, and you need to learn how to phrase your questions to get relevant and reliable answers. When LLMs confidently make up incorrect answers, we say that they are *hallucinating*. But this is just another way of anthropomorphizing an

algorithm. LLMs don't think; they generate text. That is to say, LLMs are *always* hallucinating, even when their answers happen to be correct.

LLMs make large, obvious mistakes and simple, subtle mistakes, however. If you use them as a learning aid, you must vigilantly check everything the LLM tells you, big or small. It's entirely valid to choose to forgo learning with LLMs altogether. At this point, the effectiveness of using LLMs in education is unproven. We don't know for sure in what situations LLMs are useful as learning tools, or even if their benefits outweigh their costs.

It's no wonder that so many buy into the hype of LLMs. We carry devices in our pockets that scientists of the last century would consider supercomputers. They connect us to a global network of information (and misinformation). They can identify their position anywhere on Earth by listening to satellites in outer space. Software can already do seemingly magical things, but software isn't magic. And by learning to program, you'll get a far more grounded idea of what computers are capable of versus what is just hype.

About This Book

The first part of this book teaches you how to program in Python. The second part covers various software libraries for automating different kinds of tasks. I recommend reading the chapters of Part I in order, then skipping to the chapters in Part II that interest you. Here's a brief rundown of what you'll find in each chapter.

Part I: Programming Fundamentals

Chapter 1: Python Basics Covers expressions, the most basic type of Python instruction, and how to use the Python interactive shell software to experiment with code.

Chapter 2: if-else and Flow Control Explains how to make programs decide which instructions to execute so that your code can intelligently respond to different conditions.

Chapter 3: Loops Explains how to make programs repeat instructions a set number of times, or for as long as a certain condition holds.

Chapter 4: Functions Instructs you on how to define your own functions so that you can organize your code into more manageable chunks.

Chapter 5: Debugging Shows how to use Python's various bug-finding and bug-fixing tools.

Chapter 6: Lists Introduces the list data type and explains how to organize data.

Chapter 7: Dictionaries and Structuring Data Introduces the dictionary data type and shows you more powerful ways to organize data.

Chapter 8: Strings and Text Editing Covers working with text data (called *strings* in Python).

Part II: Automating Tasks

Chapter 9: Text Pattern Matching with Regular Expressions Covers how Python can manipulate strings and search for text patterns with regular expressions.

Chapter 10: Reading and Writing Files Explains how your program can read the contents of text files and save information to files on your hard drive.

Chapter 11: Organizing Files Shows how Python can copy, move, rename, and delete large numbers of files much faster than a human user can. Also explains compressing and decompressing files.

Chapter 12: Designing and Deploying Command Line Programs Explains how you can package your Python programs to easily run them either on your own computer or on co-workers' computers.

Chapter 13: Web Scraping Shows how to write programs that can automatically download web pages and parse them for information. This is called *web scraping*.

Chapter 14: Excel Spreadsheets Covers programmatically manipulating Excel spreadsheets so that you don't have to read them. This is helpful when the number of documents you have to analyze is in the hundreds or thousands.

Chapter 15: Google Sheets Covers how to read and update Google Sheets, a popular web-based spreadsheet application, using Python.

Chapter 16: SQLite Databases Explains how to use relational databases with SQLite, the tiny but powerful open source database that comes with Python.

Chapter 17: PDF and Word Documents Covers programmatically reading Word and PDF documents.

Chapter 18: CSV, JSON, and XML Files Continues to explain how to programmatically manipulate documents, now discussing the data serialization formats CSV, JSON, and XML.

Chapter 19: Keeping Time, Scheduling Tasks, and Launching Programs Explains how Python programs handle time and dates and how to schedule your computer to perform tasks at certain

times. Also shows how your Python programs can launch non-Python programs.

Chapter 20: Sending Email, Texts, and Push Notifications Explains how to write programs that can notify you via email or mobile communications, or send these messages to others.

Chapter 21: Making Graphs and Manipulating Images Explains how to programmatically manipulate images, such as JPEG or PNG files, and work with the Matplotlib graph-making library.

Chapter 22: Recognizing Text in Images Covers how to extract text from images and scanned documents for further processing with the PyTesseract package.

Chapter 23: Controlling the Keyboard and Mouse Explains how to programmatically control the mouse and keyboard to automate clicks and key presses.

Chapter 24: Text-to-Speech and Speech Recognition Engines Covers how to use advanced computer science packages to not only generate spoken audio from text, but also convert spoken audio to text.

You can download source code and other resources for the examples in this book at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*. To see many of this book's programs in action, visit *https://autbor.com/3*.

Downloading and Installing Python

You can download Python for Windows, macOS, and Ubuntu Linux for free at *https://python.org/downloads*.

The download page detects your operating system and recommends the download package for your computer. There are unofficial Python installers for Android and iOS mobile operating systems, but those are beyond the scope of this book. Windows, macOS, and Linux have their own installation options as well. Download the installer for your operating system and run the program to install the Python interpreter software.

New versions of Python or your operating system may change the steps needed to install Python. If you encounter difficulties, you can consult *https://autbor.com/install/* for up-to-date instructions.

Downloading and Installing Mu

While the *Python interpreter* is the software that runs your Python programs, the *Mu editor software* is where you'll enter your programs,

much the way you enter text in a word processor. You can download Mu from *https://codewith.mu*.

On Windows and macOS, download the installer for your operating system, then run it by double-clicking the installer file. If you are on macOS, running the installer opens a window where you must drag the Mu icon to the Applications folder icon to continue the installation. If you are on Ubuntu, you'll need to install Mu as a Python package. In that case, click the **Instructions** button in the Python Package section of the download page.

Starting Mu

Once it's installed, you can start Mu:

- On Windows, click the Start icon in the lower-left corner of your screen, enter **Mu Editor** in the search box, and select it.
- On macOS, open the Finder window, click **Applications**, and then click **mu-editor**. You can also run **Mu Editor** from Spotlight.
- On Ubuntu, select **Applications ▸ Accessories ▸ Terminal** and then enter **`python3 -m mu`**.

The first time Mu runs, a Select Mode window will appear with options for Adafruit CircuitPython, BBC micro:bit, Pygame Zero, Python 3, and others. Select **Python 3**. You can always change the mode later by clicking the **Mode** button at the top of the editor window.

Starting IDLE

This book uses Mu as an editor and interactive shell. However, you can use any number of editors for writing Python code. The *interactive development environment (IDLE)* software installs along with Python, and it can serve as a second editor if for some reason you can't get Mu installed or working. Let's start IDLE now (assuming Python 3.13 is the version you installed):

- On Windows, click the Start icon in the lower-left corner of your screen, enter **IDLE** in the search box, and select **IDLE (Python 3.13 64-bit)**.
- On macOS, open the Finder window, click **Applications**, click **Python 3.13**, and then click the IDLE icon. You can also run IDLE from Spotlight.
- On Ubuntu, select **Applications ▸ Accessories ▸ Terminal** and then enter **`idle`**. (You may also be able to click **Show Apps** at the bottom of the Ubuntu sidebar and then click **IDLE**.)

The Interactive Shell

When you run Mu, the window that appears is called the *file editor* window. You can open the *interactive shell* by clicking the REPL button. A shell is a program that lets you enter instructions into the computer, much like the Terminal or Command Prompt on macOS and Windows, respectively. Python's interpreter software will immediately run any instructions you enter into the Python interactive shell.

In Mu, the interactive shell is a pane in the lower half of the window with something like the following text:

```
Jupyter QtConsole
Python 3
Type 'copyright', 'credits' or 'license' for
more information
IPython -- An enhanced Interactive Python.
Type '?' for help.

In [1]:
```

If you run IDLE, the interactive shell is the window that first appears. It should be mostly blank except for text that looks something like this:

```
>>>
```

`In [1]:` and >>> are called *prompts*. The examples in this book will use the >>> prompt to represent the interactive shell, since it's more common. If you run Python from the Terminal or Command Prompt, they'll use the >>> prompt as well. The `In [1]:` prompt was invented by Jupyter Notebook, another popular Python editor.

For example, enter the following into the interactive shell next to the prompt:

```
>>> print('Hello, world!')
```

After you write the line and press ENTER, the interactive shell should display this in response:

```
>>> print('Hello, world!')
Hello, world!
```

You've just given the computer an instruction, and it did what you told it to do!

How to Find Help

Programmers tend to learn by searching the internet for answers to their questions. This is quite different from the way many people are accustomed to learning—through an in-person teacher who lectures and can answer questions. What's great about using the internet as a schoolroom is that there are whole communities of folks who can help you solve your problems. Indeed, your questions have probably already been answered, and the answers are waiting online for you to find them. If you encounter an error message or have trouble making your code work, you won't be the first person to have your problem, and finding a solution is easier than you might think.

For example, let's cause an error on purpose: enter `'42' + 3` into the interactive shell. You don't need to know what this instruction means right now, but the result should look like this:

```
>>> '42' + 3
❶ Traceback (most recent call last):
    File "<pyshell#0>", line 1, in <module>
      '42' + 3
❷ TypeError: can only concatenate str (not
"int") to str
>>>
```

The error message ❷ appears because Python couldn't understand your instruction. The traceback part ❶ of the error message shows the specific instruction and line number that Python had trouble with. If you're not sure what to make of a particular error message, search for it online. Enter **"TypeError: can only concatenate str (not "int") to str"** (including the quotation marks) into your favorite search engine, and you should see tons of links explaining what the error message means and what causes it, as shown in Figure 2.

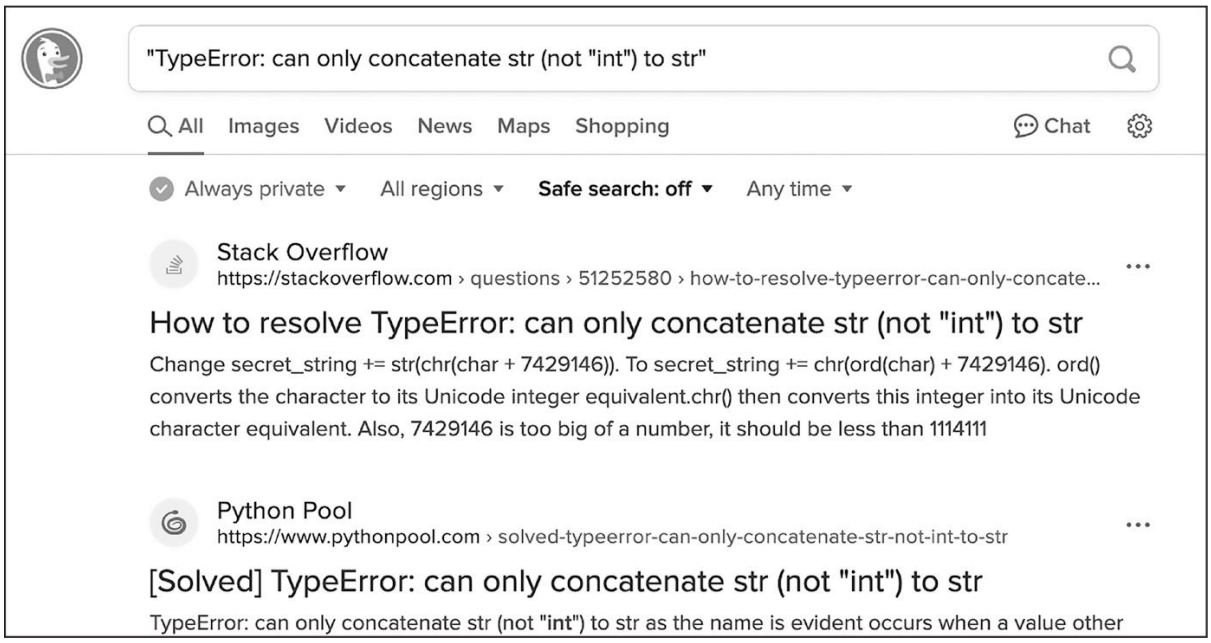


Figure 2: The Google results for an error message can be very helpful.

You'll often find that someone else had the same question as you and that some other helpful person has already answered it. No one person can know everything about programming, so an everyday part of any software developer's job is looking up answers to technical questions.

Asking Smart Programming Questions

If you can't find the answer by searching online, try asking people in a web forum such as Stack Overflow (*https://stackoverflow.com*) or the "learn programming" subreddit at *https://reddit.com/r/learnprogramming*. But keep in mind there are smart ways to ask programming questions that help others help you. To begin with, be sure to read the Frequently Asked Questions sections at these websites about the proper way to post questions.

When asking programming questions, remember to do the following:

- Explain what you are trying to do, not just what you did. This lets your helper know if you are on the wrong track.
- Specify the point at which the error happens. Does it occur at the very start of the program or only after you do a certain action?
- Copy and paste the *entire* error message and your code to *https://pastebin.com* or *https://gist.github.com*. These websites make it easy to share large amounts of code with people online, without losing any text formatting. You can then put the URL of the posted code in your email or forum post. For example, here are the locations of some pieces of code I've posted: *https://pastebin.com/2k3LqDsd* and *https://gist.github.com/asweigart/6912168*.

- Explain what you've already tried to do to solve your problem. This tells people you've already put in some work to figure things out on your own.
- List the version of Python you're using. Also, say which operating system and version you're running.
- If the error came up after you made a change to your code, explain exactly what you changed.
- Say whether you're able to reproduce the error every time you run the program or whether it happens only after you perform certain actions. In the latter case, also explain what those actions are.

Always follow good online etiquette as well. For example, don't post your questions in all caps or make unreasonable demands of the people trying to help you.

You can find more information on how to ask for programming help in the blog post at *https://autbor.com/help*. I love helping people discover Python. I write programming tutorials on my blog at *https://inventwithpython.com/blog*, and you can contact me with questions at *al@inventwithpython.com*, although you may get a faster response by posting your questions to *https://reddit.com/r/inventwithpython*.

New to the Third Edition

This fully revised and updated edition includes 16 new programming projects so you can practice the skills you'll learn. You'll find a complete introduction to SQLite and relational databases with Python's sqlite3 module. You'll explore how to compile Python scripts into executable programs on Windows, macOS, and Linux; and create command line programs and run them. You'll also learn how to perform PDF operations with PyPDF and PdfMiner, use Playwright in addition to Selenium to control web browsers, make your programs talk with text-to-speech libraries, use OpenAI's Whisper library to create text transcriptions from audio and video files, extract text from images with PyTesseract, create graphs with matplotlib, and more.

Summary

For most people, their computer is just an appliance instead of a tool. But by learning how to program, you'll gain access to one of the most powerful tools of the modern world, and you'll have fun along the way. Programming isn't brain surgery—it's fine for amateurs to experiment and make mistakes.

This book assumes you have zero programming knowledge and will teach you quite a bit, but you may have questions beyond its scope.

Remember that asking effective questions and knowing how to find answers are invaluable tools on your programming journey.

Let’s begin!