SHANI RIVERS-JOSE

PYTHON FOR DOERS

BUILD, TEST, DEPLOY!

Python for Doers

Build, Test, Deploy

Shani Rivers-Jose

This book is available at https://leanpub.com/pythonfordoers

This version was published on 2025-07-05

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my family, friends and tutees.

Contents

Chapter 0 - The 100-Day Self-Study Sprint

A Guide to Mastering Code (or anything that you want to learn)

Hey there, friend! Are you ready to dive into learning Python? I bet you are, otherwise you wouldn't have picked up this book. I wanted to write this chapter on how to self-study successfully, because it is a skill that will take you far in your career and life.

If you are already a well-versed autodidact, please feel free to skim or skip this chapter.

I want stress that if you were (or are) traditionally taught and are not sure how to study coding on your own, I want to give you a process. This is what I've developed over the years to learn web development and programming without having to spend thousands of dollars.

It is a different why of learning that is the polar opposite of formal education, and if you aren't use to it, there are some pitfalls you will most likely encounter and my hope is that I can help you mitigate them, so you don't have to suffer as much as I have.

But first let's see what the benefits are for becoming an autodidact are in general.

What Is Autodidactism?

It may sound like an illness, but really it's about taking charge of your knowledge and relying on yourself to master what it is that you want to be versed in.

There is a freedom to learn what you want to and to go as deep as you desire, if the topic tickles your intellectual fancy and improving your value in this job marketplace is not just the only benefit of autodidactism.

The Core Advantages of Self-Directed Learning

Freedom and Flexibility

You Can Study at Your Own Pace: You can go as fast or as slow as you please. If time is tight for you during a week, you can pause your studies. If you have a vacation, you can study many hours per day if you want. You'll never fall behind or be held back by others, so any concepts that come easily to you can be mastered in days, plus you never have to rush through complex topic if you want to really master them so you don't have to miss out on any valuable insights.

You Can Follow Your Curiosity: You have the option to tweak your path at any point. If you want to linger on some subtopic, you're absolutely free to do so. As an autodidact, you are free to stop and take heed to that inner voice that tells you to explore deeper.

You Can Learn Anything You Want: When you're a self-learner, there are no restrictions on what you can learn. You don't have to pass up that machine learning course you really want to take, because you have to get your credits in for some required class that's not remotely related to your interest. The world of knowledge is yours to explore with no detours.

Personalized Learning Experience

You Can Cater to Your Learning Style: We all have different learning styles or a combination of them, so you should formulate a curriculum that works for you. There are no other students to take into account. If you are a verbal learner, you can just focus on listen to lectures or podcasts. Likewise, if you're a visual learner, just do interactive tutorials.

You Get to Use Learning Materials That Work for You: Hate video tutorials because your mind wanders? Use documentation and books instead. Prefer hands-on learning? Jump straight into building projects. As an autodidact, you can develop a self-education plan that works for your learning style and needs.

You Avoid Learning Methods That Clash with Your Preferences: Despise theoretical computer science without practical application? Focus on project-based learning. Can't stand multiple choice quizzes? Use coding challenges from LeetCode as your form of practice. The best study methods are the ones you can stick with for the long term.

Economic and Practical Benefits

You Can Learn New Things Affordably: You can decide if you want to buy your programming books new or used or if you just want to check them out at your local library. You can decide if you want to take an online course, even those given by top universities, because can be found for free or at affordable prices. A lot of programming education is actually free – like documentation, tutorials, open source code, even YouTube channels.

You Save Thousands to Hundreds of Thousands of Dollars: The costs of degree programs keep going up and up, but when you decide to craft a DIY curriculum, you can make it affordable.

You Can Be Nimble, Learning Only What You Need: This is especially true if you're building real projects. If you're creating a web application, you're busy, so you learn enough to solve whatever problems get in your way as you come across them. No users? Learn a bit marketing. Are you amazing at user engagement? Great, focus on scaling instead.

Personal Development and Skills

You Learn Self-Motivation and Self-Discipline: In self-directed learning, there's usually no one holding you accountable, but yourself. A skilled autodidact can stay the course without losing steam, who can code even when they don't feel like it. Self-learning forces you to improve your powers of self-motivation and discipline, which apply to all areas of work and life.

You Learn How to Teach Yourself: Being able to teach yourself is a skill that will repay you a thousand times over. Like any skill, it requires practice, and that's exactly what you do whenever you learn something new independently. You learn what learning methods work, how to plan a course of study, how to identify your learning needs, and more.

You Become Great at Planning: Scott Young, famous for learning the MIT computer science curriculum on his own, recommends spending at least 10% of your total study time planning out your learning strategy. This planning practice will improve your project management skills and take you from a poor planner to a skilled one.

Creative and Intellectual Growth

You Form an Original Point of View: By forming your own curriculum and choosing your own resources, you develop an original perspective that will help you solve problems differently than your conventionally-educated peers. This unique viewpoint becomes a competitive advantage in your field.

You Can Move On If Something's Not Worth Your Time: If a concept or technique isn't helpful to your end goal, or just plain uninteresting, then you can skip over it rather than having to study it for the upcoming quiz. This keeps you focused on what matters most for your goals.

You Can Experiment as Much as You Want: With no self-consciousness that a teacher or classmate is going to judge your code, you can try new approaches and fail in private while you learn. This freedom to experiment leads to deeper understanding and innovation.

Quality of Life Improvements

You Can Have a Fulfilling Intellectual Life: By dedicating an hour or two a day to your own self-improvement through coding is just as fulfilling as doing yoga or meditation. Programming becomes a form of mental exercise that enriches your daily life.

You Can Learn Skills That Improve Your Quality of Life: When you teach yourself to build web applications, you can create tools for your family. When you learn data analysis, you become more resourceful at work by uncovering insights that others miss. When you master automation, you can eliminate tedious tasks from your life.

Learning Never Feels Like a Chore: In any formal program, you run the risk of coding starting to feel like work. Self-directed learning helps you ensure programming always feels like play, so you do it more often and with the spirit of adventure.

Long-Term Impact

You Gain Self-Confidence: It's empowering to learn a new programming language or build a complex system on your own. You prove to yourself that you are capable of learning. You alter your identity from someone who needs to be taught to someone who can teach themselves, and then you go learn something even harder.

You Develop Resilience and Adaptability: The technology world changes rapidly, therefore self-directed learners are better equipped to adapt to new frameworks, languages, and methodologies, because they've already developed the meta-skill of learning how to learn.

In a field like programming where technologies evolve constantly, the ability to teach yourself new skills isn't just an advantage, but is essential for long-term success.

When you take control of your own education, you're not just learning to code,are also learning how to think, problem solve, and adapt. These meta-skills will serve you regardless of which specific technologies you work with or how the industry changes over time.

Autodidactism is about developing a sustainable, enjoyable relationship with learning that will serve you throughout your entire career and life.

What's a Study Sprint?

This chapter is meant to be your roadmap to mastering Python in 100 day sprint or a couple of them.

I am not going to take credit for it, and it was actually inspired by a viral coding challenge called #100DaysOfCode that I participated in a couple of times on X/Twitter several years ago, as well as, working in IT where we have sprints.

I have also have taken advise from blog articles and books that I have read over the years, like Ultralearning by Scott Young, among others, but really Cal Newport says it best in his book Deep Work:

> "...in our economy, if you can pick up new skills or ideas fast, you have a massive competitive advantage." - Cal Newport

So with this in mind, what is a "study sprint". If you are familiar with

The study strategy I want to share with you is what I like to refer to is a "study sprint". It make your goal completion to just over 90 days. This is reminiscent of how university's split up the the school year by term, but all without the stress of missing classes or doing tests.

The 100 days is broken down further to smaller sprints are actually every 7 days, where you reassess what's working or not weekly - this allows you to pivot if need be, because there's no need to read a book or finish a course if you find that it isn't working for you so you don't waste time.

The reason I did this is because of having worked at a software company and this is how they work on it, Agile or Scrum sprints work like this.

It's just enough time to feel that you have made great progress and gives you the ability to make your goals time-bound without feeling overwhelming. Plus it allows you to see how far you've got to go, because you are focusing on the total days of effort built out of habit.

Honestly, this study strategy of using sprints can be applied to any other passion you wish to learn, you'll just have to make some modifications depending on what it is, but once you work through it, you'll can make it your own.

Define Your Goal

Before you begin, you've got to know where you're going.

Set a clear, specific goal then:

Do your research. Look into what you plan to accomplish. Is it feasible to complete it in 100 days? For example, if you want to build a machine learning model that predicts stock prices, research what that actually involves. What libraries will you need? What math concepts? How complex are the datasets?

Scale appropriately. If your initial goal is too ambitious, scale it down to something manageable or that you can break up over a couple of sprints. Instead of "building the next Netflix," maybe start with "create a movie recommendation app using Streamlit that suggests films based on user ratings."

Define success. What would success look like at the end of 100 days? Be specific: "I will have built a data analysis dashboard that visualizes sales trends and can generate weekly reports" or "I will have created three micro SaaS applications using Streamlit and deployed them live." Write it down using a real pen and your physical notebook.

Gather Materials to Your Learning Style

If you know what your learning style is, go ahead and gather your learning material.

Otherwise, take the VARK Questionaire to figure out how best you learn.

Get the books, online courses, sign up for an platforms you need access to, download the programs you plan to use, and any other materials you need, so you don't waste time searching later or trying to source them later. For example:

- Books (like "Python Crash Course" or "Hands-On Machine Learning")
- Video tutorials that directly apply to your project for the sprint (specific YouTube channels or course platforms)
- Online courses (with exact URLs and module names)
- Documentation (official Python docs, pandas documentation, etc.)
- IDEs (e.g. Visual Studio Code, Anaconda, etc.)

Set Up Your Workspace

Create a dedicated directory structure on your computer for your sprint. This keeps all your files, code, resources and projects in one organized place.

```
100-day-sprint/
├── daily-notes/
├── projects/
├── code-examples/
├── resources/
└── blog-posts/
```

Your Daily Routine

The Action Imperative

Here's the uncomfortable truth:

Binging videos doesn't count as taking action. Buying and consuming a course doesn't count as taking action. Joining a mentorship doesn't count as taking action. Reading that coding book doesn't count as taking action.

Actually doing stuff—THAT is taking action.

When you watch a video about building APIs with FastAPI, for example, you need to immediately build your own API.

When you read about machine learning algorithms, you need to implement one from scratch.

When you learn about data visualization, you need to create your own charts with real data.

You will feel overwhelmed, lost, and confused if all you do is consume information—because consumption is only a small part of the puzzle. The other parts come from actual experience.

What "Doing" or Action Looks Like

Every study session must include implementation:

For Python fundamentals:

- Write your own examples of the concepts you're reading about
- Solve a couple of coding challenges using the new techniques you read about on CodeWars or Leetcode
- Refactor your old code with your new knowledge

For data analysis:

- Apply new pandas methods to your own datasets
- Create visualizations that tell a story about data you care about and use different options or colors
- Build a few mini-analysis projects that answer real questions

For AI/ML:

- Implement algorithms from scratch before using libraries
- Train models on datasets that interest you
- Create simple AI-powered tools or scripts

For micro SaaS development:

- Build small applications that solve problems you actually have
- Deploy your projects on Github or Gitlab so others can use them
- Write about your development process in a blog post and share specific aspects of your code

The Schedule That Works

To be successful during your sprints, you should try to study every day, preferably at the same time of the day and especially when you're most alert.

Even if you can't do it every day or have to adjust the timing, the idea is to try - you have to make this a habit—just like working out, because self-study requires self-discipline.

Duration: An hour or two works well, but adjust based on your schedule and how you learn best. The key is consistency over intensity.

The 50/50 Rule

You should spend half your time consuming content and the other half teaching it and using it in your projects.

This is a non-negotiable.

Here are a couple of examples of how to split up your learning every day if you have an hour or a couple of them:

Example 1: If you are learning Data Analysis with Python

- 30 minutes: Watch a tutorial on pandas GroupBy operations
- 30 minutes: Apply what you learned to your own dataset, write notes about the process, and create a blog post draft explaining how to use GroupBy for sales analysis

Example 2: If you want to build a Streamlit App

- 60 minutes: Read documentation on Streamlit about how to set up authentication
- 60 minutes: Implement user login functionality in your micro SaaS project, documenting the challenges you faced and how you solved them as a how-to tutorial.

Weekly Sprint Reviews: Your North Star

At the end of every 7 days, you'll conduct a self-review of your progress:

Identify your wins. What worked well? What concepts clicked for you? What projects made progress?

Acknowledge what didn't go well. Where did you get stuck? What concepts remain fuzzy? What projects are stalled?

Reassess and adjust. What do you need to do about what didn't go well? Do you need to spend more time on fundamentals? Should you break down a complex project into smaller pieces?

Plan the next week. Based on your review, what are your priorities for the coming week? What specific actions will you take? If everything is still working for you, then there isn't a reason to adjust and just keep the course.

Your Study Journal

Track your progress using a dedicated study journal, because writing by hand slows down your thinking process and helps you put what you're learning into your own words, which dramatically improves retention.

For a structured approach designed specifically for this method, check out this 100-day study journal that includes weekly review pages and space for final progress recaps, but you can also achieve the same result using a plain notebook.

When You Finish Early

What if you complete your goal before the end of 100 day?

That's fantastic!

Do a little victory dance, because you deserve it.

This is your victory lap — so you can take it easy and learn whatever might take you to the end of the sprint. This could be:

- Refactoring what you've built to make it more professional

- Learning another feature that could take your project(s) to the next level
- Turn your blog posts into a comprehensive tutorial series or record one
- Contributing to open source projects related to your learning

FAQs (Frequently Asked Questions)

How does self-study improve academic performance? Self-study enables you to learn the subject at your own pace. It enhances understanding, develops critical thinking abilities, and boosts retention. It betters academic performance and independent learning that focuses on weaknesses and improves problem-solving abilities.

What are the best tips for effective self-study? Active reading, note-taking, practice, steps, spaced repetition, and summarization are the best and most effective self-study methods. These focus on enhancing retention and study efficiency by setting goals, managing time, and maintaining focus.

How can you stay motivated while studying independently? By applying effective self-study techniques and tracking progress, you can stay motivated while studying independently. Set clear goals, use your notebook to report your progress, maintain a consistent study schedule, and take effective breaks to sustain motivation and focus.

Is self-study better than traditional classroom learning? Self-study offers deeper understanding, personalized learning, and flexibility while classroom education provides a standard interaction and authoritarian guidance, but can lead to apathy regarding actual mastery.

What are some common mistakes you can make while self-studying? Passive reading, skipping revision, getting distracted, lack of proper study plan, and poor time management are some common mistakes you can make while studying. One should understand and recognize these mistakes and overcome them by using solutions for proper learning outcomes.

The Path Forward

Remember: when we look at everything at once, it's easy to get overwhelmed and do nothing. But when you say, "Okay, I don't fully understand this. I don't

know how it's going to unfold, but I'll take one small step today," that's when momentum builds.

That step informs the next one. You repeat that process. Eventually, you'll have a jigsaw puzzle with a fair amount of the pieces in place.

The takeaway here is: you probably don't need more information. You need to implement the teachings from the information you've already consumed. If information alone were the solution, there would be no problems in the world.

There's no shortage of information—yet people are still stuck. Why? Because they haven't implemented for long enough to start solving their own problems.

So commit to your 100-day sprint. Show up every day. Split your time between learning and doing. Review your progress weekly.

And most importantly—take action on what you learn.

Your future self will thank you for starting today.

References:

https://knowledgelust.com/what-is-an-autodidact-the-ultimate-guide/

https://www.ivywise.com/ivywise-knowledgebase/self-studying-whats-the-benefit-and-how-to-do-it/

https://knowledgelust.com/15-benefits-of-being-an-autodidact-the-self-learners-competitive-edge/

Chapter 1: Introduction: Setting the Stage

> Give someone a program; you frustrate them for a day; teach them how to program, and you frustrate them for a lifetime – David Leinweber

Welcome to a different kind of programming book, if you are reading this, you are probably looking to learn Python because there is something that you want to do with it. This book wasn't written to memorize its syntax, but to provide you with a game plan to build good habits, and lay the foundation, as well as, the practices that will serve you well in any coding endeavor.

If you want to learn programming syntax and theory that you will find in any college course, then you've come to the wrong place. Or if you want to use some online programming provider that has you learning in their controlled environments and rewards you with arbitrary badges to give you that false sense of progress and accomplishment, doesn't teach you those crucial skills like setting up your coding environment, how to debug, test, and deploy your code in real-world - you won't find it in these pages.

Why this book is different

The goal of this book is to help you think and work like a problem solver, leveraging Python to create real-world solutions.

It aims to bridge that gap by showing you not just how to code, but how to apply programming logic effectively, whether you come from a traditional development background or, like me, have experience in different areas, such as data analysis, software implementation, or building applications with frameworks and just want to use Python to accomplish or build something.

The goal is to help you think and work like a problem solver, leveraging Python to solve problems or to make your life easier, so you don't have to waste time filling in the holes in your knowledge and forgoing those controlled environments that give you a false sense of progress.

This is why this book will introduce you to the importance of reading the documentation, start implementing version control immediately, and get you in the practice of debugging by testing your code, as well as, setting you up to deploy your code.

Learning Beyond the Code

Programming isn't just about learning a language—it's about developing a mindset and a workflow that allows you to build, iterate, and improve. So to get the most out of this book, you should embrace the following:

Tracking Your Progress: When learning to code, especially if you're going the self-taught route, it can be daunting if you have no way to track your progress. It's highly recommended that you use a coding journal to track your progress over a 100-day sprint. You can either use a simple notebook or consider purchasing one of the coding and study journals I have published on Amazon KDP, which are specifically designed to help you document what you have studied, write down things you want to remember or reflect on your learning journey. By reflecting on what you learn each day, you'll be able to see your growth over time and identify areas where you might need extra practice.

Sharing Your Knowledge: It's important to write about what you learn and to put what you are learning in your own words. You can do this with either blog posts, tweets, or short notes. It's important to share what you are learning because it not only solidifies your understanding, but it could help someone else, plus you'll be contributing to the broader developer community. These posts can also serve as a portfolio of your knowledge which can be invaluable when networking or job hunting should you want to pursue that route.

Learning Git Early: Version control is a crucial skill for any programmer. Learning Git from the start will help you track changes, collaborate with others, and avoid the frustration of losing work due to accidental errors.

Why Git? Git isn't just another tool; it's what every programmer uses to track their code changes. While it might seem excessive for a beginner to start using when writing simple programs, I believe learning Git now will make learning how to program more enjoyable especially as a beginner.

Plus using it alongside learning Python, creates the muscle memory for tracking your code changes regularly. It will also set you up to eventually collaborate with others and gracefully recover from your mistakes. By the

time you're off working on larger projects, where version control becomes an absolute necessity, using Git will be second nature to you.

Testing Your Code: By learning to write tests, this skill will help you build the essential habits to ensure that your code works as you expect it to. By incorporating unit testing and being exposed to test-driven development (TDD), you'll learn to break down problems systematically and write more reliable software.

Testing your code is usually brought up much later and it's usually when you have an application to build, but I believe that at least learning to unit test should be learned early so that you can design better solutions for your little programs as well.

It might seem a bit counterintuitive at first, but writing tests to make sure your code is working as you intended and makes debugging it easier. It is an approach that will fundamentally change how you think about the problems you are solving.

The last chapter of this book will introduce test-driven development (TDD) so that you can be aware of this development process. TDD forces you to clarify what your code should actually output or do, it makes you stop and think through the problem before you dive into coding it. It will result in more robust, maintainable solutions. TDD isn't just about testing; it's about designing code that solves clearly defined problems and can also become the de facto documentation for your project or application.

Your Coding Journey Starts Now

This book isn't just about teaching you Python—it's about equipping you with the skills and mindset to approach programming with confidence. By actively tracking your progress, sharing your learning, and practicing essential development habits, you'll set yourself up for success not just in Python, but in any programming path you choose to follow.

What to Expect

This isn't your typical programming book. I won't hold your hand through every detail of Python syntax and this is intentional. I will point out some nuances and things to be aware of, but I want you to be in the official documentation and read what the folks who built Python and Git have to say about them, as they can do a better job than I could ever do at explaining it.

Instead, I'll guide you on how to:

1. Learn directly from primary sources–the documentation
2. Track your progress and experiments–version control with Git
3. Break down problems systematically–unit testing and test-driven development

Overview of Python, Git, and TDD

Why Python is a Great First Language

Python is Readable: Python's syntax is clean and resembles plain English. But more importantly, it doesn't drown you in syntax.

Python is Versatile: You can use it to build web applications, analyze data for a variety of industries, use it in scientific research, for automating tasks, tackling machine learning and even controlling robots!

Python has Community: It has a massive community with a lot of support and so many libraries that if you are looking for a particular functionality, you might be surprised that you'll probably find one already exists so that you spend less time developing and more time building!

Python is Marketable: There is a consistent, high demand across many industries for workers who know this language.

Python removes the barriers that would traditionally frustrate beginners, like having to figure out complex compilation processes or manage memory, it lets you focus on what's important–solving problems!

Importance of Version Control with Git

Most programming courses treat Git as an afterthought or something optional, but I'm putting it front and center because that's where it belongs.

Git is not optional.

I don't care if you're writing a three-line script to rename files or building the next unicorn startup – you must learn Git now!

What is Git? Git is a time machine or think of it as save point(s) for your code or what is called, version control.

If you made a catastrophic mistake and you can't figure out what you broke? Git will allow you to roll your project back to a point where you know your code was working correctly.

Do you want to experiment with a wild idea or try out a new functionality without risking your functioning code? Branch it off and experiment to your heart's content.

Do you need to figure out why something that worked yesterday is broken today? Git can tell you exactly what changed and you can pinpoint what might have caused the failure.

Do you want to share your code with the world or work on an interesting project you found on GitHub/Gitlab? Git is used for collaborative coding. So even if you're working solo now, learning it will allow you to work with others or to share your code.

Learning Git alongside Python means you'll never develop the bad habit of working without version control.

Trust me – the future you will be grateful.

Benefits of Testing for Problem-Solving

Testing is often presented as something that only "real" engineers practice. This is nonsense. It's a practical approach to problem-solving that helped me think through problems. I would have progressed faster had I learned it from day one.

The more complicated your code becomes, the harder it is to make sure that you are covering every edge case. I learned this the hard way. So introducing unit testing early will help you make sure that the solutions you do code will work the way you intended. You will code your solution first and then create unit tests to help you debug your solution. This will help you when you get to the last chapter where we dive into test-driven development (TDD).

The premise of TDD is simple: before you write code to solve a problem, you will first write a test that defines what the output should be. You focus on the result and any edge cases. Then, you can write the simplest code so that it

passes that test. Finally, you refine your solution to make it more efficient, in what is known as refactoring.

TDD might seem advanced, but I believe that getting exposed to it early will only give you an edge, for example, being able to do:

Problem decomposition: This is just a fancy computer science term that means you can break up large problems into testable units.

Create specifications: You define exactly what your code should do before writing it. In the real world, all software has specs that are created before the programmers build it out. It's like a roadmap of sorts.

Build Confidence: You'll know immediately if you've broken some existing functionality instead of finding out later because your tests will be your beacon of truth.

Create Documentation and maintainable code: Tests detail how your code is meant to be used and behave, so if you take a break from it and come back to it months later, they will make it a lot easier to understand what you intended the code to do.

Is it more work upfront? Perhaps, but it'll save you more time especially as your projects grow in complexity.

Getting Started

Let's set up your development environment and get all the things installed so we can move on to the more interesting parts.

Install Python and Git

1. **Python Installation**:

    - Visit python.org and download the latest stable version (3.12+ recommended)
    - During installation on Windows, check "Add Python to PATH"
    - On macOS or Linux, the installer usually handles this for you
    - Verify your installation by opening a terminal (Command Prompt on Windows, Terminal on macOS/Linux) and typing:

```
python --version
```

2. **Git Installation**:

   Visit git-scm.com and download the appropriate version for your operating system.

   For Windows users: Accept the default options during installation unless you have a specific reason not to.

   For macOS users: If you've installed Xcode, you already have Git. You might be prompted to download Xcode Command Line Tools if you don't have Xcode downloaded, go ahead and install those tools. Otherwise, the Git installer will work fine. You can also install via Homebrew with `brew install git`

   - Visit git-scm.com and download the appropriate version
   - For Windows users: The default installation options are fine
   - For macOS: You can also install via Homebrew with `brew install git`
   - For Linux users: You probably already know how to install packages, but just in case:

```
# For Debian/Ubuntu
sudo apt install git

# For Fedora
sudo dnf install git
```

Verify that your installation was successful by running this command:

```
git --version
```

After installing Git from git-scm.com, you will need to configure your identity. Replace the placeholders with your actual information:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Install an IDE

An Integrated Development Environment (IDE) is where you'll spend most of your programming time. It's similar to a word processor, but specifically designed for code.

While you can write Python code in any text editor, a proper IDE will make your life easier by highlighting the syntax and providing code completion:

1. **PyCharm Community Edition**: Specifically designed for Python, with more built-in features, but it has a steeper learning curve.
    - Download from jetbrains.com/pycharm
    - Excellent for larger projects and has integrated Git support
2. **Visual Studio Code**: Lightweight, extensible, and works with nearly any programming language and has excellent Python support via extensions
    - Download from code.visualstudio.com
    - Install the Python extension by Microsoft

Choose one based on your preferences, although both are excellent options. For beginners, I recommend Visual Studio Code with the Python extension. It's has more than enough power without overwhelming you with too many options.

Creating Your First Git Repository

Let's create a repository to track all the exercises in this book.

Here are a few commands that can help you navigate your file system. You will need to think about where you want to store your files. When you first open your terminal, you will be in the root directory.

1. Open your terminal/command prompt, and type in the list command at the prompt, it could look like this `%`:

```
ls
```

- `ls` This command will list all the folders and files in your root directory. You'll most likely see your Applications, Documents, Desktop, and other directories.

I usually save my coding projects in my Documents folder, so I am going to assume you will too (but if you don't plan to, just navigate to whatever directory suits your fancy).

2. Navigate to a directory:

```
cd Documents
```

And if you type in the list `ls` again, you should see all the files and folders in your Documents directory.

3. Let's create a new folder using the "make directory" command, `mkdir`:

```
mkdir code_smart
cd code_smart
```

4. Initialize your Git repository:

```
git init
```

To see if your directory has Git, you can use the `ls -a` with the `-a` flag to see all the files, even the hidden ones. You should have a `.git` directory.

REMEMBER: You only need to use `git init` once to create a repository for your project directory, but if you want to add it to another directory, you will navigate to that project folder and you will need to run `git init` for it.

Wow! You've just created your first Git repository! Now, this folder will track all your code.

Tracking Changes

The basic Git workflow for tracking changes involves these 3 main areas:

1. Working directory - the files in your directory.
2. Staging area - files that have been added or edited and are ready to be committed
3. Repository - the committed history of your files

Let's test it out.

1. Create a basic README file in the new directory:

```
echo "# My Python Learning Journey" > README.md
```

This command will create a file named `README.md` and write the text `# My Python Learning Journey` into it. If you run this command in your terminal, you'll have created your README file and added content to it in one line!

2. Check the status of your files:

```
git status
```

3. Add files to the staging area

```
git add README.md  # You can add a specific file
git add .  # OR you can add all the files
```

4. Make your first commit, the `-m` tag is for the message.

```
git commit -m "Initial commit with README"
```

5. View commit history

```
git log
```

Congratulations –- you've just entered your first Git commit!

Documentation Assignment

Now for your first taste of technical documentation.

The Python and the Git documentation are very well organized and you need to learn how to extract information from them, as it will serve you better than any tutorial could provide.

Read these sections thoroughly:

1. Read Python Tutorial §1 & §2 - "Whetting Your Appetite" and "Using the Python Interpreter".
2. Read Git Basics Chapter 1

Don't just skim these – read them.

Yes, documentation can be dry at times. Yes, it sometimes uses language that seems deliberately obscure. Get used to it.

The ability to extract information you need to solve your problem or to build out a feature from documentation is what separates programmers who can solve their own problems from those who post the same questions on StackOverflow every day.

Coding Exercises

Time to get your hands dirty with some actual code:

Exercise 1: Hello World in the Interpreter

1. Open your terminal/command prompt
2. Start the Python interpreter by typing `python` or `python3`
3. At the `>>>` prompt, type:

```
print("Hello, World!")
```

Press Enter. If "Hello, World!" appears, you've successfully run your first Python command.

4. Exit the interpreter with `exit()` or Ctrl+Z (Windows) / Ctrl+D (Mac/Linux)

Exercise 2: Creating and Running a Python Script

1. Open your IDE
2. Create a new file named `hello.py`
3. Add the following code:

```
print("Hello, World!")
```

4. Save the file in your `python-learning` directory
5. Run the script:

- In VS Code: Right-click and select "Run Python File in Terminal"
- In PyCharm: Right-click and select "Run 'hello'"
- From terminal: Navigate to the directory and type `python hello.py`

If "Hello, World!" appears, your script works!

Exercise 3: Commit Your Script

Time to save this monumental achievement with Git: 0. Check the status:

```
git status
```

1. Add your new file to Git:

```
git add hello.py
```

And check the status again.

2. Commit it:

```
git commit -m "Add my first Python script, Hello World"
```

The `-m` flag lets you add a commit message directly. Always write meaningful commit messages–future you will need to understand what changes you made and why.

Exercise 4: Modify and Commit Changes

1. Open `hello.py` and change it to:

```
name = "Your Name"  # Replace with your name
print(f"Hello, {name}!")
```

2. Save the file
3. Run it to verify it works
4. Check the status, add the file, and commit the change:

```
git status
git add hello.py
git status
git commit -m "Update hello.py script to print my name"
```

5. Viewing your commit history:

```
git log
```

Notice how Git has tracked every change and used your commit messages. It will show you who authored the commit, what date and time it occurred as well as the message.

This is why it is important to write descriptive commit messages because you'll use those in the future. If you want to get out of the log, just type `q` and it will exit out of it.

What You've Learned

In this chapter, you've:

- Set up your development environment with Python, Git, and an IDE - The tools of modern development!
- Created and ran your first Python code
- Written and modified a simple Python script
- Started tracking your learning journey with Git
- Been introduced to the concept of reading documentation

That's a solid foundation!

In the next chapter, we'll dive deeper into Python's core data types, work with variables, learning about lists and start writing more interesting programs –- all while continuing to build good habits with version control and laying groundwork for learning test-driven development.

Remember: programming isn't about memorizing syntax or following recipes. It's about solving problems methodically and knowing where to find the information to do so.

Don't forget to write your notes in your coding journal!

Chapter 2: Variables and Basic Operations

> "The best time to plant a tree was 20 years ago. The second best time is now. The same applies to learning how to code." – Modified Chinese Proverb

Introduction

Welcome to Chapter 1. If you're reading this, you've already made the decision to learn Python and have downloaded all the things.

Good for you!

Seriously, just getting your development environment up and running is half the battle.

But let's be clear: programming is not magic. It's a skill and a way of thinking that requires practice, patience, and persistence.

This book assumes you're a grown adult who values your time, so we'll focus on practical skills, introduce good habits early (like version control with Git), and get you comfortable with reading documentation –- because that's how real programmers solve problems.

It is never to late to learn how to program and with the quote in mind, let's begin...

Python Basics

The first thing we need to work on is how we store data and the next couple of chapters we will figure it out. Data Python's way (and most languages anyway) to store data, is in variables and that values or data have a type.

So What are Variables?

Variables are like labeled containers that hold data, similarly to how a container will hold food.

Unlike some other languages, Python doesn't require you to declare the data type of a variable – it can figure it out on its own, based on what you put in it.

You create a variable by assigning a value to a name, for example:

```
# Integers (whole numbers)
age = 34
print(age)
```

```
34
```

```
# Floats (decimal numbers)
pi = 3.14159
print(pi)
```

```
3.14159
```

```
# Strings (text) with double quotes
name = "Python"
name
```

```
'Python'
```

```
# Strings (text) with single quotes
also_a_string = 'Single quotes work too'
also_a_string
```

```
'Single quotes work too'
```

```
# Booleans (True/False)
is_learning = True
is_learning
```

```
True
```

What are Data Types

Python has several basic built-in data types and different types can do different things:

1. **Integers (`int`)**: Whole numbers, so no decimal points.

```
count = 10
negative_number = -5
print(count, negative_number)
```

```
10 -5
```

2. **Floating Point Numbers (`float`)**: Numbers with decimal points.

```
temperature = 98.6
gravity = 9.8
```

3. **Strings (`str`)**: Text enclosed in quotes (they can be single or double quotes).

```
greeting = "Hello, world!"
message = 'Python is practical.'
print(greeting + " " + message)
```

```
Hello, world! Python is practical.
```

4. **Boolean (`bool`)**: True or False values.

```
is_active = True
has_permission = False
print("Are they active? ",is_active, "\nDo they have permissiont: ",
↪  has_permission)
```

```
Are they active?  True
Do they have permissiont:  False
```

You can also check a variable's type using the `type()` function, just in case you need to doublecheck:

```
print("age: ", type(age))  # <class 'int'>
print("pi: ", type(pi))   # <class 'float'>
print("name: ", type(name)) # <class 'str'>
```

```
age:  <class 'int'>
pi:  <class 'float'>
name:  <class 'str'>
```

Go ahead and check the type of the `also_a_string` and `is_learning` variables.

Dynamic Typing

Unlike many other programming languages, Python uses dynamic typing. This means you don't have to declare a variable's type beforehand, Python will automatically infer or guess what its data type is from the value you assign it:

```
x = 10  # Python infers this is an integer
y = "hello"  # Python infers this is a string
z = 3.14  # Python infers this is a float
```

```
print("x: ", type(x), "\ny: ", type(y), "\nz: ", type(z))
```

```
x:  <class 'int'>
y:  <class 'str'>
z:  <class 'float'>
```

You can change a variable's type simply by assigning it a new value:

```
x = 10.0  # x is now a float
z = "now I'm a string"  # x is now a string
z = False # z is now a boolean
```

```
print("x: ", type(x), "\ny: ", type(y), "\nz: ", type(z))
```

```
x:  <class 'float'>
y:  <class 'str'>
z:  <class 'bool'>
```

The ability for Python to guess the data type is very convenient, but that means that you will have to track what values go into your variables, because if you don't you, this could lead to bugs if you're not careful. You will always want be aware of what the data type your variables are and what values are being passes in, and if you are not sure, check it using the `type()` function.

Type Conversion

You can convert data types by using conversion functions like `int()`, `float()` and `str()`:

Convert to integer

```
int_value = int(3.14)  # 3 (truncates decimal)
int_from_string = int("42")  # 42
```

Convert to float

```
float_value = float(42)  # 42.0
float_from_string = float("3.14")  # 3.14
```

Convert to string

```
str_from_int = str(42)  # "42"
str_from_float = str(3.14)  # "3.14"
```

These conversion functions are helpful, especially when you are working with user's input or when you are working with calculations which required a specific data type in order to run.

Basic Operations

Python also supports all the arithmetic operations you'd expect:

Arithmetic Operations

Addition

```
sum_result = 5 + 3  # 8
sum_result
```

```
8
```

Subtraction

```
difference = 10 - 4  # 6
difference
```

```
6
```

Multiplication

```
product = 7 * 6  # 42
product
```

```
42
```

Division (returns a float)

```
quotient = 20 / 4  # 5.0
quotient
```

```
5.0
```

Integer Division (returns an int)

```
integer_quotient = 20 // 4  # 5
integer_quotient
```

```
5
```

Modulo (remainder)

```
remainder = 20 % 7  # 6
remainder
```

```
6
```

Exponentiation

```
power = 2 ** 3  # 8
power
```

```
8
```

String operations:

Python can also do operations on strings. You can use the + to concatenate two strings, which is extremely helpful.

You can also multiply a string to have it output multiply times–I'm honestly not sure why this is useful, but maybe for you it could be.

```
full_name = "John" + " " + "Doe"  # "John Doe"
full_name
```

```
'John Doe'
```

```
first_name = "Ada"
last_name = "Lovelace"
full_name = first_name + " " + last_name
full_name
```

```
'Ada Lovelace'
```

Repetition

```
print("Echo " * 3)  # "Echo Echo Echo "
```

```
Echo Echo Echo
```

Length

This built-in function will return the length of the string.

```
length = len(full_name)  # Ada Lovelace
length
```

```
12
```

Indexing (0-based)

```
message = "Hi!"
first_char = message[0]  # "H"
last_char = message[-1]  # "!"
print(first_char, last_char)
```

```
H !
```

Slicing

```
message = "Hello, world!"
substring = message[0:5]  # "Hello"
from_index_7 = message[7:]  # "world!"
to_index_5 = message[:5]  # "Hello"

print(substring, from_index_7, to_index_5)
```

```
Hello world! Hello
```

Type Coercion in Arithmetic

Python will also convert data types if the numbers you input are of two different data types, e.g. one is an integer and the other a float.

Here are some examples of mixed operations that follow these rules:

1. If either operand is a float, the result will be a float:

```
result = 5 + 3.0  # 8.0 (float)
result
```

```
8.0
```

2. Division with `/` always returns a float, even with integer operands:

```
result = 10 / 5  # 2.0 (float)
result
```

```
2.0
```

3. Integer division with `//` returns an integer, if both operands are integers:

```
result = 10 // 3  # 3 (int)
result
```

```
3
```

4. Operations between integers and floats will convert to float:

```
result = 5 * 2.5  # 12.5 (float)
result
```

```
12.5
```

This automatic conversion is very convenient, but if you are not careful, it can lead to unexpected results. When in doubt, use explicit type conversion.

Data Type Coercion in Arithmetic

There are some data types that Python will not automatically convert between, for example, these cases will demonstrate the behavior:

```
# This will cause an error
result = "5" + 5  # TypeError: can only concatenate str (not "int") to str
result
```

* * *

```
TypeError                                 Traceback (most recent call last)

Cell In[285], line 2
      1 # This will cause an error
----> 2 result = "5" + 5  # TypeError: can only concatenate str (not "int") to
    ↪  str
      3 result


TypeError: can only concatenate str (not "int") to str
```

You need to explicitly convert the string “5” into an integer or the integer 5 using to a string with the int() or str() functions respectively, in order to get the behavior you want:

```
result = "5" + str(5)  # "55"
result
```

```
'55'
```

```
result = int("5") + 5  # 10
result
```

```
10
```

Here are the type conversion functions you can use:

```python
# String to integer
x = int("42")  # 42

# String to float
y = float("3.14")  # 3.14

# Number to string
z = str(42)  # "42"

# Float to integer (truncates the decimal part)
a = int(3.99)  # 3

print(x,y,z,a)
```

```
42 3.14 42 3
```

Print Function

The `print()` function displays output. You can output a value, strings with variable separated by commas or you can create formatted strings using the (f) before the string and the varialbes in curly brackets `{}` :

```python
print("Hello, World!")
```

```
Hello, World!
```

`\n` is the escape sequence for a new line:

```python
# Print multiple items
print("Age:", age, "\nName:", name)
```

```
Age: 34
Name: Python
```

Format strings (f-strings):

This allows you to insert variables using curly brackets.

```
# Format strings (f-strings)
print(f"My name is {name} and I am {age} years old.")
print(2025-1991)
```

```
My name is Python and I am 34 years old.
34
```

Lists

Lists are ordered collections that can hold data, and with Python, it can even hold data of different types.

A list is also considered a type in Python, a sequence type.

If you are familiar with arrays, this is Python's flavor of that:

Create a list

To create a list you can assign a variable with empty square brackets or assign values in the brackets:

```
numbers = [1, 2, 3, 4, 5]
mixed = [1, "two", 3.0, [4, 5]]
fruits = ["apple", "banana", "cherry"]
empty_list = []

print('numbers:', numbers, '\nmixed:', mixed, '\nfruits:',
↪  fruits,'\nempty_list:', empty_list)
```

```
numbers: [1, 2, 3, 4, 5]
mixed: [1, 'two', 3.0, [4, 5]]
fruits: ['apple', 'banana', 'cherry']
empty_list: []
```

Adding to a list

You can `append()` one item to add to an existing list:

```
fruits.append("date")  # ["apple", "banana", "cherry", "date"]
fruits
```

```
['apple', 'banana', 'cherry', 'date']
```

Inserting into a list

```
fruits.insert(2, "strawberry")
fruits
```

```
['apple', 'banana', 'strawberry', 'cherry', 'date']
```

Removing from a list

You can use remove() and enter the value you want to remove.

```
fruits.remove("cherry")
fruits
```

```
['apple', 'banana', 'strawberry', 'date']
```

You can also use pop() to remove the last time of the list.

```
fruits.pop()
fruits
```

```
['apple', 'banana', 'strawberry']
```

Indexing

You can use indexing, which means you can use the index of the value you want. Indexing starts a 0, so for the first value of the index of `fruits` list, it will be `"apple"`.

```
first_fruit = fruits[0]  # "apple"
second_fruit = fruits[1]  # "banana"
last_fruit = fruits[-1]  # "date"

print('first_fruit:', first_fruit, '\nsecond_fruit:',second_fruit,
↪  '\nlast_fruit:' ,last_fruit)
```

```
first_fruit: apple
second_fruit: banana
last_fruit: strawberry
```

Splicing

We aren't splicing genomes, but rather lists. You can pull out a subset of multiple values using their index `[n: n+1]`.

For example, if you wanted to only pull out `banana` and `cherry` from the `fruit` list, you will take the index of the first and the index of the second plus 1.

```
# ["apple", "banana", "cherry", "date"]

subset = fruits[1:3]  # ["banana", "cherry"]
subset
```

```
['banana', 'cherry']
```

You can also figure out if a list contains a value, with `index()` and it will output the index of where that value is stored.

```
False if fruits.index("apple") else True
```

```
True
```

```
position = fruits.index("banana")  # 1
position
```

```
1
```

```
position = fruits.index("orange")  # ValueError
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

Cell In[174], line 1
----> 1 position = fruits.index("orange")  # ValueError


ValueError: 'orange' is not in list
```

More on Lists

Lists are very versatile data structures and you will be using them a lot.

Here are some advanced operations you can do with them:

Finding items

```
index_of_3 = numbers.index(3)  # 2
```

Checking if an item exists

Use the `in` keyword to see if your value exists. It doesn't fail spectacularly like using `index()`.

```
3 in numbers  # True
```

```
True
```

```
contains_orange = "orange" in fruits
print(contains_orange)
```

```
False
```

Nested lists

It's kind of like that movie where the characters have a dream within a dream - you can have lists within a list:

```
inception = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
middle_value = inception[1][1]  # 5
middle_value
```

```
5
```

A better way to look at this, is to organize it like a matrix or kind of like a spreadsheet:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Accessing elements in a nested list
element_0 = matrix[0][0] # want the value 1
print('(row 0, column 0):', element_0)
element_2 = matrix[1][2] # want the value 6
print('(row 1, column 2):', element_2)
```

```
(row 0, column 0): 1
(row 1, column 2): 6
```

List Methods

Sorting

The `sort()` function sorts a list in ascending order.

```
fruits = ["banana", "apple", "cherry"]
fruits.sort()  # ["apple", "banana", "cherry"]
fruits
```

```
['apple', 'banana', 'cherry']
```

Reverse

It does just what you think it, it sorts in descending order.

```
fruits.reverse()  # ["cherry", "banana", "apple"]
fruits
```

```
['cherry', 'banana', 'apple']
```

Counting occurrences

Just give the `count()` function what to look for in the list and it will count how many times they occur.

```
counts = [1, 2, 3, 1, 2, 1]
count_of_1 = counts.count(1)  # 3
count_of_1
```

```
3
```

We will cover more about list in Chapter 4 Data Structures.

Git Basics

Now that we have set up our Git, have created a repository for our directory, and some tracked changes, let's learn to create a new branch for the edits we want to make and track.

Tracking Changes

Let's review the basic Git workflow again, which involves three main areas:

1. Going into your working directory (your files) in Terminal or Command Line
2. Check the Staging area to see what files ready to be committed and to see if you have any that haven't been committed.
3. Check the log on committed history, if we need to.

Checking Status

Remember:

- `git status` shows what files are staged, unstaged, or untracked
- `git add` stages files for commit
    - You can use the period `.` to add all the files OR
    - You can specify which files you want to commit by adding the file name to the end: `git add some_file.py`
- `git commit -m "Action verb and something that you worked on` saves a snapshot of the staged changes. Always include a meaningful commit message that explains what you've changed, use a verb in the present tense.

Add files to the staging area

You can add an individual file or you can add all the files that you have created, editted and/or deleted.

```
1  git add filename.py  # Add specific file
2  git add .  # Add all files
```

Commit changes with a message

You have to commit your files with a message using the `-m` flag.

```
git commit -m "Add initial code for calculator"
```

View commit history

This command will list all of your commit history.

```
git log
```

Branching

Branches let you work on features or fixes without affecting the main codebase:

```
# Create and switch to a new branch
git checkout -b feature-user-input

# Check which branch you're on
git branch

# Switch between branches
git checkout main
```

When you're ready to merge changes from a branch back to the main branch:

```
git checkout main
git merge feature-user-input
```

Documentation Assignment

1. Read Python Tutorial §3: https://docs.python.org/3/tutorial/introduction.html
2. Open a Python interactive console (run `python` or `python3` in your terminal)
3. Experiment with the `help()` function in the Python interpreter:

```
# Start Python in interactive mode
# Type the following:
help()  # Enters help mode
modules  # Lists available modules
quit  # Exits help mode

# Or get help directly on a specific function
help(print)

# Get help on a data type
help(str)

# Get help on a specific method
help(str.upper)
help(list.append)

# Get help on a built-in function
help(len)
```

Learning to read documentation is a critical skill. Don't skip this assignment—it will save you hours of frustration later.

Remember: Documentation is your friend. The sooner you get comfortable reading it, the faster you'll progress.

Now dive into the documentation:

1. Read Python Tutorial §3.1.1- "Whetting Your Appetite" and "Using the Python Interpreter".
2. More on Lists: Stacks, Queues, Comprehensions, Nested Python Tutorial §5.1
3. Read Git Basics Chapter 2

Coding Exercises

Let's apply what you've learned.

Exercise 1: Basic Arithmetic Program

Create a program that performs basic arithmetic operations.

Create a file named `arithmetic.py` with operations for addition, subtraction, multiplication, and division:

```python
# calculator.py

# Basic arithmetic operations
a = 10
b = 5

# Addition
sum_result = a + b
print(f"{a} + {b} = {sum_result}")

# Subtraction
difference = a - b
print(f"{a} - {b} = {difference}")

# Multiplication
product = a * b
print(f"{a} * {b} = {product}")

# Division
quotient = a / b
print(f"{a} / {b} = {quotient}")  # Note: This is a float (5.0)

# Integer division
int_quotient = a // b
print(f"{a} // {b} = {int_quotient}")  # Integer result (5)

# Mixed type arithmetic
c = 10
d = 2.5
mixed_result = c + d
print(f"{c} + {d} = {mixed_result}")  # 12.5 (float)
```

Git commands: git add arithmetic.py git commit -m “Create arithmetic.py with basic operations”

Exercise 2: Working with Lists

Extend your program to work with lists. Let’s create a git branch so that we can make sure that if we mess up, we still have a save point to go back to that we know will work. git checkout -b add-lists-operations Output the list of branches you have: git branch It should output your new branch `add-lists-operations` and `main`. The asterisk next to the branch name indicates that it is the branch you are currently in.

Add the code to your arithmetic.py file:

```python
# Lists practice
numbers = [1, 2, 3, 4, 5]
print(f"Original list: {numbers}")

# Append a value
numbers.append(6)
print(f"After append: {numbers}")

# List comprehension to create a list of squares
squares = [num**2 for num in numbers]
print(f"Squares: {squares}")

# Nested list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(f"Nested list: {matrix}")

# Access an element in the nested list
print(f"Element at position [1][2]: {matrix[1][2]}")
```

```
Original list: [1, 2, 3, 4, 5]
After append: [1, 2, 3, 4, 5, 6]
Squares: [1, 4, 9, 16, 25, 36]
Nested list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Element at position [1][2]: 6
```

Now, save your file. And let's commit this change. git add arithmetic.py git commit -m "Add list operations" Now, we need to switch back to the main branch and merge our changes that we made in the branch `add-lists-operations`: git checkout main git merge add-lists-operations Go ahead and check your log `git log` and see how it tracked your changes.

Exercise 3: User Input

Now let's modify the program again to now accept user input.

Let's create another branch. git checkout -b user-input Add this code to the arithmetic file:

```
# Get input from user
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

# Perform calculations
print(f"Addition: {num1} + {num2} = {num1 + num2}")
print(f"Subtraction: {num1} - {num2} = {num1 - num2}")
print(f"Multiplication: {num1} * {num2} = {num1 * num2}")

# Check for division by zero
if num2 != 0:
    print(f"Division: {num1} / {num2} = {num1 / num2}")
else:
    print("Division by zero is not allowed")
```

To test the functionality, have Python run it by typing this in the command line: python arithmetic.py Now if it works, it will ask you for 2 numbers and output the result.

Let's go ahead and commit this.

Git commands:

```
git add arithmetic.py
git commit -m "Modify program to accept user input"
```

Git commands:

```
git add arithmetic.py
git commit -m "Modify program to accept user input"
```

Check your commit history, this will output your log history in one line:

```
git log --oneline
```

Go ahead and save your file. Then add a bo git add arithmetic.py git commit -m "Create arithmetic.py with basic operations" Now you've practiced how to make changes to your files and track them using Git!

Common Pitfalls

0. **Variable naming**: Variables can't start with numbers or contain spaces.
1. **Forgetting that Python is case-sensitive**: `variable` and `Variable` are different.
2. **Mixing tabs and spaces for indentation**: Pick one (preferably spaces) and stick with it.
3. **Not converting user input**: `input()` returns a string, so use `int()` or `float()` for numbers.
4. **Zero-based indexing confusion**: The first element is at index 0, not 1.
5. **Division by zero**: Always check before dividing by a variable that might be zero.
6. **Type confusion**: Forgetting that `5/2` returns a float (`2.5`) while `5//2` returns an integer (2).
7. **String conversion errors**: Trying to convert invalid strings like `int("3.14")` (use `float()` first).
8. **Forgetting to commit changes**: Commit early and often with meaningful messages. So small, frequent commits rather than large, infrequent ones.
9. **Indentation matters**: Python uses indentation to define blocks of code.
10. **Mutability**: Some objects (like lists) are mutable, others (like strings) are not.
11. **Integer division**: In Python 3, `5 / 2` returns `2.5`, not 2. Use `//` for integer division.
12. **Assign a List to Another List**: You can't assign a list to another list, because all you will be doing is copying the memory address for the first list. So if you need to have your new list be a copy of another, use list comprehension to effectively "copy" the values of the first list.

Conclusion

You've taken your first steps into programming with Python. You've now learned the basics of Python variables, data types, type conversion, and dabbled with lists, as well as how to use Git for version control. These fundamentals will serve as the foundation for everything else you'll learn.

Don't worry if everything isn't crystal clear yet—programming is a skill that improves with practice. Keep experimenting, break things, and learn from your mistakes.

Remember: consistent practice is more effective than occasional cramming. Write code every day, even if it's just for 10 or 15 minutes.

In the next chapter, we'll explore control flow in Python—how to make decisions and repeat actions using conditionals and loops.

Additional Resources

- Python's official documentation: https://docs.python.org/3/
- Git's official documentation: https://git-scm.com/doc
- Try Git interactive tutorial: https://try.github.io

Exercises to Try

1. Create a list of your favorite foods and write a program that prints each one with its index.
2. Write a program that converts temperatures from Fahrenheit to Celsius (formula: C = (F - 32) * 5/9).
3. Create a simple contact list using nested lists or lists of strings, then write code to search for a specific contact.
4. Write a program that calculates the area and perimeter of a rectangle based on user input.
5. Create a program that demonstrates type conversion between integers, floats, and strings.

Remember: Every time you make a meaningful change to your code, commit it to Git!