```python
# your first program
def greet(name):
    return f"Hello, {name}!"

print(greet("World"))
```

```
Hello, World!
```

THE COMPLETE PRACTICAL GUIDE

Python Programming

from Zero — your first line of code to real applications

BASICS · FUNCTIONS · OOP · FILES · TESTING · DATA · WEB

THENEXBOOKS

Python Programming from Zero

The Complete Practical Guide from Your First Line of Code to Real Applications

First Edition

THENEXBOOKS

Python Programming from Zero
The Complete Practical Guide from Your First Line of Code to Real Applications

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Python and its libraries evolve; always consult the official documentation for the versions you use. The publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

"Python" is a registered trademark of the Python Software Foundation. pandas, matplotlib, Flask, FastAPI, pytest, VS Code, and other names are trademarks of their respective owners. This book is independent and is not affiliated with or endorsed by any of these organizations.

Preface

Learning to program can feel like standing at the foot of a mountain. There are unfamiliar words, strict rules, and error messages that seem to shout at you. Yet programming is one of the most useful skills you can learn today — and Python is one of the friendliest places to start. Its code reads almost like English, it runs everywhere, and it powers everything from simple scripts that save you an hour a week to websites, data analysis, automation, and artificial intelligence.

This book assumes you know **nothing** about programming. We start with what a program is and how to install Python, and we explain every new idea the first time we use it. Step by step, you will learn to store and transform data, make decisions, repeat work, organize code into functions and classes, read and write files, handle errors gracefully, and test your work. Then we go further, into the tools real developers use every day: virtual environments, pytest, pandas, web APIs, Flask and FastAPI, and automation.

Throughout the book, you will not just read code — you will build something. By the final chapters you will have written a complete command-line store application and a sales dashboard with its own API.

Who This Book Is For

- **Complete beginners** who have never written a line of code.
- **Students** who want a clear, structured companion to their courses.
- **Career changers** moving toward software development, data, or automation.
- **Analysts, marketers, and office professionals** who want to automate repetitive work.
- **Developers from other languages** who want a thorough, practical tour of Python.

What You Will Learn

- How to install Python, use an editor, and run your programs.
- Variables, numbers, strings, lists, dictionaries, sets, and tuples.
- Decisions and loops, functions, modules, and the standard library.
- Error handling, debugging, and working with text, CSV, and JSON files.
- Object-oriented programming, comprehensions, generators, type hints, and dataclasses.
- Professional habits: virtual environments, packages, testing with pytest, and clean code.
- Real-world skills: data analysis with pandas, charts, web APIs, Flask and FastAPI, and automation.

How to Use This Book

Type the examples yourself rather than copying them. Every time you type code, run it, and see the result — or an error — you learn more than from reading alone. Each chapter includes **Try It Yourself** exercises with hints and solutions; do them before moving on. If something does not work, read the error message slowly: it usually tells you exactly what went wrong and where.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) sets up your tools and teaches the basic building blocks. **Part II — Control Flow and Data Structures** (Chapters 6–10) teaches decisions, loops, and Python's core collections. **Part III — Functions and Program Structure** (Chapters 11–15) shows how to organize code, handle errors, and work with files. **Part IV — Object-Oriented and Pythonic Code** (Chapters 16–19) covers classes and idiomatic Python. **Part V — Real-World Python** (Chapters 20–26) introduces professional tools, data, the web, and automation. **Part VI — Projects** (Chapters 27–28) brings everything together in two complete applications. The appendices provide a cheat sheet, a guide to common errors, a code-quality checklist, and a glossary.

The Running Example: Orbit Store

Throughout the book we build **Orbit Store**, a small fictional online shop. We start by printing a product label, then calculate prices and discounts, manage a shopping cart, save products to files, model products and orders with classes, analyze sales, and finally serve data through a web API. Every concept you learn is used to make Orbit Store a little more real.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake that can cause bugs, crashes, or security problems.

Code appears in `monospace` blocks. When a program prints something, its output appears in a separate block labeled **Output**. Examples target Python 3.12 or newer.

Contents

Preface

PART I — GETTING STARTED

1 Welcome to Programming and Python

2 Setting Up: Installing Python, Editors, and Running Code

3 Values, Variables, and Types

4 Numbers, Math, and Strings

5 Input, Output, and Your First Programs

PART II — CONTROL FLOW AND DATA STRUCTURES

6 Making Decisions: Booleans, Comparisons, and if Statements

7 Loops: while, for, and range

8 Lists and Tuples

9 Dictionaries and Sets

10 Working with Text: String Methods and Formatting

PART III — FUNCTIONS AND PROGRAM STRUCTURE

11 Functions: Defining, Calling, and Returning Values

12 Scope, Arguments, and Lambda Functions

13 Modules, Packages, and the Standard Library

14 Handling Errors: Exceptions and Debugging

15 Files and Data: Text, CSV, and JSON

PART IV — OBJECT-ORIENTED AND PYTHONIC CODE

16 Classes and Objects

17 Inheritance, Composition, and Special Methods

18 Comprehensions, Iterators, and Generators

19 Type Hints, Dataclasses, and Clean Code

PART V — REAL-WORLD PYTHON

20 Virtual Environments, pip, and Project Structure

21 Testing Your Code with pytest

22 Working with Data: pandas and Visualization

23 Talking to the Web: HTTP Requests and APIs

24 Building a Web App or API with Flask and FastAPI

25 Automation and Scripting: Files, Folders, and Tasks

26 Where to Go Next: Async, Databases, and More

PART VI — PROJECTS

27 Project: The Orbit Store Command-Line App

28 Project: Orbit Store Sales Dashboard and API

APPENDICES

A Python Cheat Sheet

B Common Errors Reference

C Code-Quality Checklist

D Glossary

PART I

Getting Started

Install Python, write and run your first programs, and learn the basic building blocks: values, variables, numbers, strings, input, and output.

1 · Welcome to Programming and Python

2 · Setting Up

3 · Values, Variables, and Types

4 · Numbers, Math, and Strings

5 · Input, Output, and Your First Programs

CHAPTER 1 · PART I — GETTING STARTED

Welcome to Programming and Python

Before you write a single line of code, it helps to know what programming really is, why Python is such a friendly place to start, and how to learn in a way that sticks. By the end of this chapter you will have read, understood, and run your very first Python program.

What Is Programming?

A **program** is a list of instructions that tells a computer what to do. **Programming** is the craft of writing those instructions. That is the whole idea. Everything else in this book — variables, loops, functions, classes, web apps — is a set of tools for writing instructions that are clearer, shorter, more reliable, and more useful.

You already follow programs every day. A recipe is a program for a cook: "Preheat the oven to 180 °C. Mix the flour and sugar. Add two eggs. Bake for 25 minutes." A set of driving directions is a program for a driver. Assembly instructions for a bookshelf are a program for whoever holds the screwdriver. Each is a sequence of steps, performed in order, that turns a starting situation into a desired result.

The difference with a computer is the kind of reader you are writing for.

The computer as a very literal assistant

Imagine you hire an assistant who is astonishingly fast, never gets tired, never gets bored, and never forgets anything — but who has absolutely no common sense. This assistant does *exactly* what you say, not what you mean.

If a recipe says "add two eggs," a human cook knows to crack them first and throw away the shells. Your literal assistant might drop two whole eggs, shells and all, into the bowl. If you write "bake until golden," a human understands; the literal assistant asks, "What is golden? How will I measure it?" and then stands still, waiting.

That is what a computer is like. It can perform billions of simple operations every second, but it cannot guess your intentions. This has two important consequences for you as a new programmer:

- **Precision matters.** A missing bracket, a misspelled word, or a quotation mark in the wrong place can stop a program from running. This is not the computer being difficult; it simply has no way to fill in the gap.
- **The computer is honest.** When something goes wrong, the computer did what you told it. That sounds harsh, but it is actually good news: it means every problem has a cause you can find and fix.

Learning to program is largely learning to think precisely: breaking a task into small, unambiguous steps and expressing each step in a form the computer understands.

Programming languages

Computers do not understand English. Deep down, the processor inside your machine understands only very simple numeric instructions, often called **machine code**. Writing machine code by hand is slow and painful, so people invented **programming languages**: languages that are easier for humans to read and write, but still strict enough that a computer can translate them into machine code without any guessing.

There are hundreds of programming languages. Some you may have heard of are JavaScript, Java, C, C++, C#, Go, Rust, and of course Python. They differ in style and in the jobs they are best suited for, but they share the same core ideas. Once you learn to program well in one language, picking up a second one is much easier. The thinking transfers.

What Is Python?

Python is a general-purpose programming language. It was created by Guido van Rossum and first released in 1991. The name comes from the British comedy group Monty Python, not from the snake — which is why you will sometimes find playful references to their sketches in Python tutorials and documentation.

Python was designed with a clear goal: code should be easy to read. Compare a line of Python with its meaning:

```
if stock == 0:
    print("Sold out")
```

Even without any training, you can probably guess what this does: if the stock is zero, show the message "Sold out." That readability is not an accident. Python uses plain English words such as `if`, `for`, `in`, `and`, `or`, and `not`, and it uses indentation (spaces at the start of a line) to show structure instead of lots of curly braces and semicolons.

Python is also **free and open source**. You can download it, use it for personal or commercial projects, and even read its source code, all without paying anything. It is developed by a large community and overseen by the Python Software Foundation, a non-profit organization.

Where Python is used

Python is one of the most widely used programming languages in the world, and it shows up in a remarkable range of places. Here are some of the main areas you will meet it:

Area	What Python does there

Web development	Runs the server side of websites and APIs
Data analysis	Cleans, explores, and charts data
Automation	Renames files, fills forms, sends reports
AI and machine learning	Trains and uses models
Scripting	Glues tools and systems together
Education	A first language in many schools

Let us look at each of these a little more closely.

- **Web development.** Frameworks such as Django, Flask, and FastAPI let you build websites and web APIs in Python. In Chapter 24 you will build a small web API for the Orbit Store.
- **Data analysis and science.** Libraries such as pandas and matplotlib make it practical to load a spreadsheet of sales, calculate totals, and draw a chart in a handful of lines. You will do exactly this in Chapter 22.
- **Automation.** Python is excellent for the boring, repetitive jobs that eat up an afternoon: renaming hundreds of files, combining reports, checking a folder every morning. Chapter 25 is all about this.
- **Artificial intelligence and machine learning.** Many of the popular tools for machine learning and AI research offer Python as their main interface. You will not build an AI model in this book, but the skills you learn here are the foundation those tools assume.
- **Scripting and system administration.** Small Python scripts connect other programs together, process logs, and manage servers.
- **Education.** Because it is readable and forgiving, Python is widely used to teach programming in schools and universities.

The practical upshot is this: the time you invest in Python is not locked into one narrow niche. Whatever direction you eventually take — web, data, automation, or something else — the fundamentals in this book will carry over.

How Python Runs Your Code

We said earlier that computers only truly understand machine code. So how does a line like `print("Hello")` ever get executed? Something has to translate it. There are two broad approaches to that translation, and it is worth knowing about both at a beginner level.

Compiled languages

In a **compiled** language such as C or Rust, you write your program and then run a tool called a **compiler**. The compiler reads your whole program, checks it, and translates it into a separate file of machine code — an executable. You then run that executable. If you change your source code, you

compile again.

Think of it like translating an entire book from English into Japanese before handing it to a Japanese reader. The translation step takes time up front, but afterwards the reader can read the translated book quickly.

Interpreted languages

Python is usually described as an **interpreted** language. Instead of producing a separate executable for you to run, you hand your code to a program called the **Python interpreter**, and the interpreter runs it directly, statement by statement.

Think of an interpreter at a meeting: someone speaks a sentence, and the interpreter immediately conveys it to the listener, one sentence at a time. There is no separate "translate the whole book first" step for you to perform.

```
Compiled (e.g. C):

  your code  --compiler-->  executable file  --run-->  result

Interpreted (Python):

  your code  --python interpreter-->  result
```

> **NOTE**
>
> The full story is a little more detailed. Behind the scenes, the standard Python interpreter first converts your code into an intermediate form called *bytecode* and then runs that bytecode. You do not need to think about this as a beginner; from your point of view, you write code and Python runs it.

What does this mean for you in practice?

- **Fast feedback.** You can type a line of Python and see the result immediately. This makes experimenting and learning much quicker.
- **Some mistakes appear only when a line runs.** Python checks basic grammar before starting, but many errors — for example, trying to add a number to a piece of text — are only discovered when Python reaches that line. You will learn to read these error messages in this chapter and in Chapter 14.
- **You need Python installed to run Python code.** Anyone who wants to run your program needs a Python interpreter on their machine. Chapter 2 shows you how to install one.

Interpreted languages are often somewhat slower than compiled ones for heavy number crunching, but for the vast majority of everyday tasks the difference simply does not matter, and the time Python saves *you* as a programmer usually matters far more.

How to Learn Programming Effectively

Programming is a practical skill, much more like learning to cook or play an instrument than learning a list of historical dates. You cannot learn to swim by reading about water. Here are the habits that make the biggest difference.

1. Type the code yourself

It is tempting to read an example, nod, and move on. Resist that temptation. Type every example into your own computer and run it. Typing forces you to notice details — where the brackets go, which quotes are used, how lines are indented. Those details are exactly what trips beginners up, and your fingers will learn them faster than your eyes will.

Avoid copying and pasting from the book, at least in the early chapters. The small effort of typing pays off many times over.

2. Change things and see what happens

After you get an example working, poke at it. Change a number. Change a word. Delete a line. Predict what will happen *before* you run it, then check whether you were right. When your prediction is wrong, you have just found something worth understanding. This "predict, run, compare" loop is one of the most powerful learning tools you have.

3. Expect mistakes — and welcome them

Every programmer, including those with decades of experience, writes code that does not work the first time. Errors are not a sign that you are bad at this; they are simply part of the process. Professional programming is, to a large extent, the art of finding and fixing mistakes calmly.

You cannot break your computer by writing a wrong Python program in the exercises of this book. The worst that normally happens is an error message. So experiment freely.

4. Read error messages carefully

When Python cannot run your code, it prints an error message. Beginners often see a wall of red text, panic, and stop reading. Instead, treat the error message as a helpful note from Python explaining what went wrong. Here is a typical example. Suppose you accidentally type `prnt` instead of `print`:

oops.py

```
prnt("Hello")
```

Output

```
Traceback (most recent call last):
  File "oops.py", line 1, in <module>
    prnt("Hello")
    ^^^^
```

```
NameError: name 'prnt' is not defined. Did you mean: 'print'?
```

Let us read this from the bottom up, which is usually the best way:

- **The last line** tells you the kind of error (`NameError`) and a description: Python does not know anything called `prnt`. Recent Python versions even suggest what you probably meant.
- **The lines above it** show where the problem happened: the file name, the line number (`line 1`), and the offending line of code itself, with carets (`^^^^`) underlining the exact part Python is complaining about.
- **"Traceback"** is simply Python's name for this report. It traces back through the steps that led to the error.

> **NOTE**
>
> The exact wording of error messages, and the file path shown, can vary a little between Python versions and depending on how you run the file. The structure — where it happened, then what kind of error — stays the same.

Once you get into the habit of reading error messages, many problems will take seconds to fix instead of minutes.

5. Practice little and often

Thirty minutes a day for a week is usually more effective than one exhausting four-hour session. Your brain needs time to consolidate new ideas. Short, regular practice also keeps the details fresh, so you spend less time relearning what you forgot.

6. Build things you care about

Knowledge sticks best when it is attached to a goal. That is why this book follows a single running project from start to finish, which we introduce next. When you finish the book, try building something of your own: a tool for your hobby, your job, or your household.

> **TIP**
>
> Keep a simple "learning log" — a text file or notebook where you jot down each new concept and one example of it. Writing things in your own words is a surprisingly effective way to remember them.

Meet the Orbit Store

Throughout this book, you will build software for a fictional small online shop called the **Orbit Store**. The Orbit Store sells a modest range of space-themed products: star maps, telescope accessories, astronaut mugs, and similar items. It is small enough to understand completely but realistic enough to

teach you the skills real programs need.

Using one project across many chapters has two big benefits. First, you see how individual concepts fit together into something useful, rather than learning them as isolated tricks. Second, each chapter builds on what you did before, so your skills compound.

What you will build

Here is a taste of what you will create for the Orbit Store, roughly in the order you will build it:

1. **A product label** printed to the screen, including a name and price.
2. **Price calculations**: totals, tax, and discounts.
3. **Interactive programs** that ask the user questions and respond.
4. **A product list** and a **shopping cart** that can hold many items.
5. **Reusable functions** for common store tasks, such as formatting a price.
6. **Reading and saving product data** in CSV and JSON files.
7. **Classes** that model a `Product`, a `Cart`, and an `Order`.
8. **A command-line store app** that ties everything together.
9. **A sales report** using data analysis and charts.
10. **Fetching data from a web API**, and building a **small web API** of your own.
11. **Automated tests** that check your code keeps working as it grows.

Do not worry if many of those terms mean nothing to you yet. Each one will be explained from scratch when you reach it.

A Roadmap of This Book

The book is divided into six parts. Each part builds on the one before, so we recommend reading them in order, at least the first time through.

Part	Chapters	Focus
I — Getting Started	1–5	Setup, values, math, text, input
II — Control Flow and Data	6–10	Decisions, loops, collections
III — Functions and Structure	11–15	Functions, modules, errors, files
IV — Object-Oriented Code	16–19	Classes, iterators, clean code
V — Real-World Python	20–26	Tools, tests, data, web, automation
VI — Projects	27–28	Two complete Orbit Store projects

Part I gets you set up and writing simple programs. In Chapter 2 you will install Python and learn to run code. Chapters 3 and 4 cover the basic building blocks — values, variables, numbers, and text — and Chapter 5 shows you how to read input from the user and produce nicely formatted output.

Part II teaches your programs to make decisions and repeat work, and introduces Python's core data structures: lists, tuples, dictionaries, and sets. These are the tools that let one program handle a whole catalogue of products instead of just one.

Part III is about structure. You will learn to package code into functions and modules, handle errors gracefully, and read and write files.

Part IV introduces object-oriented programming and the "Pythonic" style that experienced Python developers use: comprehensions, generators, type hints, and dataclasses.

Part V moves into the real world: managing projects and packages, testing, data analysis, working with the web, and automating everyday tasks.

Part VI brings everything together in two larger projects: a complete command-line Orbit Store app, and a sales dashboard with an API.

At the back of the book you will find appendices with a Python cheat sheet, a reference of common errors, a code-quality checklist, and a glossary. Keep them handy.

How each chapter is organized

Every chapter follows a similar pattern so you always know where you are:

- Explanations with many **small, runnable examples**.
- An **Output** block after each example showing exactly what it prints.
- **Tip**, **Note**, and **Warning** boxes highlighting shortcuts, extra details, and common traps.
- A **Try It Yourself** section with exercises, followed by solutions and hints.
- A **Key Takeaways** box summarizing what matters most.

Try the exercises before looking at the solutions. Even a failed attempt teaches you more than reading an answer you have not struggled with.

Your Very First Program

It is a long-standing tradition that a programmer's first program in a new language simply displays a greeting. We will follow that tradition, with an Orbit Store twist. Here is the entire program:

hello.py

```
print("Hello, Orbit Store!")
```

Output

```
Hello, Orbit Store!
```

One line of code. When Python runs it, the text `Hello, Orbit Store!` appears on the screen. You will install Python and actually run this in Chapter 2; for now, let us take it apart piece by piece so that nothing about it feels like magic.

Piece 1: `print`

`print` is the name of a **function** that comes built into Python. A function is a named, reusable piece of behavior — a small machine that does one job. The job of `print` is to display things on the screen. You did not have to write `print` yourself; Python provides it ready to use, along with dozens of other built-in functions you will meet later.

Piece 2: the parentheses `( )`

Writing a function's name on its own does not run it. To actually use, or **call**, a function, you put parentheses after its name. Whatever you place inside the parentheses is handed to the function to work on. The value you hand over is called an **argument**. Here, the argument is the text we want printed.

If you are used to maths, this is a lot like writing $f(x)$: the function name, then the input in brackets.

Piece 3: the quotation marks `" "`

The quotation marks tell Python that `Hello, Orbit Store!` is a piece of **text** — literally these characters, exactly as written — rather than instructions or names that Python should try to understand. In programming, a piece of text is called a **string** (short for "a string of characters"). You will learn much more about strings in Chapters 3 and 4.

Notice that the quotation marks themselves are not printed. They mark where the string starts and ends, like the covers of a book, but they are not part of the content.

You can use either double quotes or single quotes in Python, as long as they match:

```
print("Hello, Orbit Store!")
print('Hello, Orbit Store!')
```

Output

```
Hello, Orbit Store!
Hello, Orbit Store!
```

Piece 4: the line as a statement

The whole line is a **statement**: one complete instruction to Python. A Python program is simply a sequence of statements, run from top to bottom, one after another. Our first program happens to contain just one statement. Add more lines and Python runs them in order:

label.py

```
print("Hello, Orbit Store!")
print("Today's featured product: Star Map Poster")
print("Price: $12.50")
```

Output

```
Hello, Orbit Store!
Today's featured product: Star Map Poster
Price: $12.50
```

Look at the second line carefully. It contains an apostrophe in `Today's`. Because the string is wrapped in double quotes, the apostrophe (a single quote) is treated as ordinary text. If we had used single quotes around this string, Python would think the string ended at the apostrophe, and we would get an error. Choosing the other kind of quote is the simplest fix.

Each call to `print` starts a new line of output. That is why the three messages appear on three separate lines.

Printing numbers and calculations

`print` can display numbers too, and Python will happily do arithmetic for you first:

```
print(12.5)
print(3 + 4)
print("3 + 4")
```

Output

```
12.5
7
3 + 4
```

This small example teaches an important lesson. In the second line, `3 + 4` has no quotes, so Python treats it as a calculation and prints the result, `7`. In the third line, the same characters are inside quotes, so Python treats them as plain text and prints them exactly as written. Quotes change meaning.

You can also give `print` several arguments separated by commas. It prints them all on one line with a space between each:

```
print("Mugs in stock:", 24)
print("Total for 3 mugs:", 3 * 8)
```

Output

```
Mugs in stock: 24
Total for 3 mugs: 24
```

In Python, the asterisk `*` means multiplication. Chapter 4 covers all the arithmetic operators in detail.

Common Beginner Mistakes

Even a one-line program leaves room for mistakes. Here are the ones we see most often, so you can recognize them when they happen to you.

Capital letters

Python is **case-sensitive**: it treats uppercase and lowercase letters as different. `print` works; `Print` and `PRINT` do not.

```
Print("Hello, Orbit Store!")
```

Output

```
Traceback (most recent call last):
  File "hello.py", line 1, in <module>
    Print("Hello, Orbit Store!")
    ^^^^^
NameError: name 'Print' is not defined. Did you mean: 'print'?
```

Missing or mismatched quotes

Every string needs an opening quote and a matching closing quote. Forgetting one gives a `SyntaxError`, which means Python could not even understand the grammar of the line:

```
print("Hello, Orbit Store!)
```

Output

```
  File "hello.py", line 1
    print("Hello, Orbit Store!)
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

The little caret `^` points to where Python noticed the problem — here, the start of a string that never ends. "Unterminated string literal" is Python's way of saying "this text has no closing quote."

Missing parentheses

A function needs its parentheses to be called. Both the opening and the closing bracket must be present:

```
print("Hello, Orbit Store!"
```

Output

```
  File "hello.py", line 1
    print("Hello, Orbit Store!"
         ^
SyntaxError: '(' was never closed
```

Curly "smart" quotes

If you copy code from a word processor, a web page, or a chat app, straight quotes (`"`) are sometimes replaced by curly typographic quotes (“ and ”). Python does not accept curly quotes in code and will report an "invalid character" error. Always write code in a proper code editor, which we set up in Chapter 2, and type the quotes yourself.

Unexpected spaces at the start of a line

In Python, spaces at the start of a line (indentation) have meaning. A top-level statement must start at the very left edge. An accidental leading space causes an error:

```
 print("Hello, Orbit Store!")
```

Output

```
  File "hello.py", line 1
    print("Hello, Orbit Store!")
IndentationError: unexpected indent
```

You will learn when indentation *is* required in Chapter 6. For now, start every line at the left margin.

> **WARNING**
>
> When you see an error, fix only the first problem Python reports, then run the program again. One mistake can sometimes cause several confusing messages, and fixing the first one often makes the rest disappear.

Try It Yourself

You will be able to run these once Python is installed in Chapter 2. For now, write your answers on paper or in a text file and predict the output. Then come back and check them after the next chapter.