QUICK PYTHON PROJECTS

A Beginner's Guide to 25 Useful Mini Scripts

Learn by building one small program at a time

RUTH BARNHART

25 Tiny Python Scripts You Can Build in One Hour

Disclaimer

The code in this book is provided for educational purposes only. The scripts are designed for Python 3.10 or newer. Software can behave differently from one computer to another. Test any script in a safe practice folder before relying on it. Run the scripts in a separate practice folder, and always keep backups of any files that matter to you. The author accepts no responsibility for any loss or damage arising from the use of the code or instructions in this book. Use it at your own discretion.

Contents

Foreword

I still remember how it felt to write my first program that actually worked. It was tiny, just a few lines, and it did something almost silly. But it ran, and it did exactly what I had told it to, and something clicked. That moment is what this whole book is built around.

Many beginners do not get stuck because Python is too hard. They get stuck because the gap between a beginner lesson and a real, finished program feels enormous. You learn what a loop is, you nod along, and then you sit in front of a blank screen with no idea what to actually build. This book exists to fill that gap with the smallest possible steps.

Inside you will find twenty-five little scripts. Not one of them is complicated. Each solves a small, real problem, and each is small enough for a short practice session, often under an hour. They are short on purpose, because short scripts get finished, and finishing is what builds confidence. One after another, these small wins add up into something that will surprise you.

You do not need to be good at math. You do not need an expensive computer. You only need Python and a simple editor. You only need a little curiosity and a willingness to type things out and see what happens. Mistakes are part of the deal, and you will make plenty. So do I, still, every week. The trick is not to avoid errors but to stop fearing them.

My advice is simple. Do not just read this book. Open your editor, type each script with your own hands, and run it. Change something and run it again. Break it on purpose, then fix it. The reading takes minutes, but the doing is where the learning actually lives.

Whenever you feel ready, turn the page, pick a script that catches your eye, and build it. I think you are going to enjoy this more than you expect.

Ruth Barnhart

How to Use This Mini Script Book

Welcome. You are about to build twenty-five small Python scripts, and most of them will take you less than an hour each. This is not a textbook full of theory. It is a hands-on script library you can dip into whenever you want a quick win.

You do not have to read it in order. Pick a script that looks useful or fun, build it, and move on. Every script stands on its own. Some take ten minutes, others closer to an hour, and a few you will want to come back to and improve later.

Why Tiny Scripts Are the Fastest Way to Learn Python

Big projects can feel overwhelming when you are new. A tiny script does not. A program of just a few lines feels doable, runs fast, and shows you a result right away. That small hit of success is what keeps you going.

Each script also quietly drills the core skills you already met: input, variables, if statements, loops, lists, functions, and working with files. You will practice them again and again without it ever feeling like a chore.

What You Need Before You Start

You only need a few things, and you probably have them already:

1. Python 3 installed on your computer (3.10 or newer is ideal)
2. VS Code or another code editor you like
3. One folder where you save all your scripts
4. A basic feel for print, input, variables, if and else, loops, lists, and functions

If any of those last ideas feel a little shaky, that is fine. The scripts will remind you how they work as you go.

The Structure of Every Script

To keep things simple, every script in this book follows the same five-part shape:

1. What This Script Does, so you know the goal before you type anything
2. What You'll Practice, so you can see which skills you are strengthening
3. The Code, ready to type and run
4. How It Works, a plain walkthrough of the important lines
5. Try This Upgrade, a small challenge to make the script your own

Once you get used to this rhythm, every new script will feel familiar before you even read it.

How to Learn Without Getting Stuck

Here is the most important thing to remember: error messages are normal. Every coder sees error messages regularly. An error is not a wall. It usually points you to a good place to start looking.

So type the code yourself instead of copying and pasting. You will catch small mistakes and learn faster that way. Change one thing at a time and run it again to see what happens. And save every script in your folder, even the messy ones. That folder slowly becomes proof of how much you can already do. None of this needs to be perfect. Working beats pretty, especially while you are still learning.

Ready? Pick a script that sparks your interest, and let us build something that works.

Quick Input and Output Scripts

These are the simplest scripts in the book, and they are the perfect place to begin. Each one takes a small piece of information from the person using it and prints something back. That loop of ask, then answer, sits at the heart of almost every program you will ever write. None of these will take more than a few minutes, and every one of them gives you an instant result right there on screen. If you are nervous about typing code, start here. Get a few of these running, watch them respond to you, and the rest of the book will feel far less intimidating. A quick tip before you dive in: type each script by hand rather than copying it. Your fingers learn the patterns, and you will spot the small typos that teach you the most.

Script 1: Personalized Welcome Message

What This Script Does. It asks the user for their name and then prints a warm greeting that includes it. It is tiny, but it already feels like real software, because it reacts to whoever is using it. The same screen says something different to every person, and that is the first small spark of programming.

What You'll Practice. Collecting input with input(), storing it in a variable, joining text together, and showing the result with print().

The Code.

```
name = input("What is your name? ")
print("Hello, " + name + "! Welcome aboard.")
```

How It Works. The first line shows your question on screen and then waits. Whatever the user types is handed back and stored in the variable called name. You can picture a variable as a labeled box: the label is name, and the box now holds the text that was typed. The second line builds one longer message by joining pieces of text with the plus sign, which programmers call concatenation, then prints it. Because name holds whatever was typed, the greeting is different for every person who runs the script. Run it twice with two different names to see that for yourself.

Try This Upgrade. Ask a second question, such as a favorite color or hobby, store it in its own variable, and weave that answer into the greeting so it feels even more personal.

Script 2: Simple Age Calculator

What This Script Does. It asks for the year you were born and then works out roughly how old you are. It is a nice first taste of doing real math with numbers the user provides, and it shows why the type of a value matters so much.

What You'll Practice. Turning text into whole numbers, getting the current year automatically, and subtracting one number from another.

The Code.

```
from datetime import date

birth_year = int(input("What year were you born? "))
current_year = date.today().year
age = current_year - birth_year
print("You are about", age, "years old.")
```

How It Works. Everything that comes back from input() is text, even when it looks like a number. If you tried to subtract one piece of text from another, Python would complain, so the int() function converts that text into a whole number first. Instead of typing this year by hand, the script asks your computer for the current year through the date tool. That one small choice means the script stays correct next year and every year after, with no edits from you. The last line subtracts the two numbers and prints the result. The word about is there because someone may not have had their birthday yet this year.

Try This Upgrade. Add one more line that prints your age in months by multiplying the result by twelve, then print both numbers in a single friendly sentence.

Script 3: Favorite Things Sentence Builder

What This Script Does. It asks a few quick questions and then stitches the answers into one playful sentence. It is a great way to see how neatly Python can drop your values into a piece of text without any messy plus signs.

What You'll Practice. Collecting several inputs in a row and building a sentence with an f-string, which is the cleanest way to mix text and variables.

The Code.

```
name = input("What is your name? ")
food = input("What is your favorite food? ")
place = input("What is your favorite place? ")
print(f"{name} loves {food} and dreams about {place}.")
```

How It Works. Each input() call asks one question and stores the answer in its own variable, so you end up with three labeled boxes ready to use. The last line uses an f-string, which is any string with the letter f placed just before the opening quote. Inside that string, anything you put between curly braces is swapped out for the value of that variable. It reads almost like a normal sentence, which is exactly why beginners reach for f-strings again and again. Once you are comfortable with them, you will rarely build text any other way.

Try This Upgrade. Add three more questions, such as a hobby, a dream job, and a favorite season, then extend the sentence to include them all.

Script 4: Temperature Comment Script

What This Script Does. It asks for the current temperature and prints a friendly comment that matches it. This is your first script that makes a decision instead of just repeating what it is told, and that is a big step forward.

What You'll Practice. Converting input to a number and using if, elif, and else to choose between different responses.

The Code.

```
temp = int(input("What is the temperature in degrees? "))

if temp < 10:
    print("That is cold. Bring a jacket.")
elif temp < 25:
    print("That is warm. Enjoy the day.")
else:
    print("That is hot. Drink some water.")
```

How It Works. Python reads the conditions from the top down. If the temperature is below ten, it prints the first message and skips everything else. If that test fails, it tries the elif, which is just short for else if and only runs when the line above it did not match. The else at the bottom is the catch-all for anything that slipped past the earlier checks. Notice that the lines under each condition are indented by four spaces. That spacing is not decoration. It is how Python knows which lines belong inside each branch, so keep it tidy. Many beginners hit their first error here by mixing tabs and spaces, so let your editor insert real spaces and you will sidestep a lot of frustration.

Try This Upgrade. Add more advice to each branch, such as wearing sunscreen on a hot day or carrying an umbrella when it is cold and wet.

Script 5: Daily Motivation Generator

What This Script Does. It greets the user by name and then prints a random encouraging line from a list you create. It is a small feel-good script that is fun to run and easy to share with a friend who needs a lift.

What You'll Practice. Storing several pieces of text in a list and letting Python pick one of them at random.

The Code.

```
import random

quotes = [
    "You are doing better than you think.",
    "Small steps still move you forward.",
    "Every expert was once a beginner.",
]

name = input("What is your name? ")
boost = random.choice(quotes)
print(name + ", here is your boost: " + boost)
```

How It Works. The square brackets create a list, which is simply an ordered collection of items kept together under one name. Here the items are short pieces of text, separated by commas. The random module, which comes with Python for free, includes a handy tool called choice that reaches into a list and pulls out one item for you. Because the pick changes each time, the script feels alive even though it is only a handful of lines. Lists like this will show up in almost everything you build from here on, so it is worth getting friendly with them now.

Try This Upgrade. Add ten of your own quotes to the list so the script has plenty of variety, then run it a few times and watch them take turns.

That is five working scripts already. Each one was short, yet together they cover input, variables, text, numbers, decisions, and lists. If you typed them out and ran them, you are no longer guessing at how Python works. You are using it. Keep these in your folder, because you will reuse pieces of them again and again in the chapters ahead. Before you move on, try changing one line in each script and running it again. Breaking something on purpose, then fixing it, is one of the fastest ways to learn.

Tiny Math and Decision Scripts

Now that you can take input and print output, let us put Python to work on small everyday problems. The five scripts in this chapter do quick calculations and make simple decisions, the kind of thing you might otherwise reach for a calculator or a quick web search to handle. Along the way you will get comfortable with decimal numbers, percentages, the comparison operators, and the way Python chooses between options. Each one is short, useful, and easy to adapt to your own needs. As before, type each one out yourself and run it before reading the explanation, because seeing the result first makes the walkthrough click into place much faster. None of these scripts need anything beyond what already comes with Python, so there is nothing extra to install.

Script 6: Tip Calculator

What This Script Does. It takes a bill amount and a tip percentage, then works out the tip and the new total. It is a small tool you might actually use at a restaurant, and it shows how to handle money values cleanly.

What You'll Practice. Working with decimal numbers, doing multiplication and division, and formatting a result so it shows exactly two decimal places.

The Code.

```
bill = float(input("What was the bill amount? "))
tip_percent = float(input("Tip percentage? "))

tip = bill * tip_percent / 100
total = bill + tip

print(f"Tip: {tip:.2f}")
print(f"Total: {total:.2f}")
```

How It Works. Money usually needs decimals, so this script uses float() instead of int() to keep the cents. It multiplies the bill by the percentage and divides by one hundred to find the tip, then adds that to the bill for the total. The part that reads colon point two f inside the curly braces is a formatting instruction. It tells Python to show the number with exactly two digits after the decimal point, which is just what you want for a price. Without that formatting, a tip might print as something like 7.5000000001, which looks messy and is not how you want to show money.

Try This Upgrade. Ask how many people are splitting the bill, then divide the total by that number and show what each person owes. Round the per person amount to two decimals as well.

Script 7: Discount Price Checker

What This Script Does. It takes an original price and a discount percentage, then shows how much you save and what the final price will be. It is handy for quick sanity checks while you shop.

What You'll Practice. Turning a percentage into a real amount and doing arithmetic with decimal numbers.

The Code.

```
price = float(input("What is the original price? "))
discount = float(input("Discount percentage? "))

savings = price * discount / 100
new_price = price - savings

print(f"You save {savings:.2f}")
print(f"New price: {new_price:.2f}")
```

How It Works. The pattern here is almost the same as the tip calculator, which is the point. Once you learn a small piece of logic, you can reuse it in many places. The script finds the savings by taking the discount percentage of the price, then subtracts that from the original to get the new price. Both results are printed with two decimal places so they read like real prices. Treat the numbers as a quick estimate rather than a final bill, since real stores may round differently. Reusing a pattern you already understand, with only the names changed, is exactly how experienced programmers move so quickly.

Try This Upgrade. Add a sales tax of your choice. Work out the tax on the new price and show the final amount the shopper would pay, then compare the totals with and without tax so the difference is clear.

Script 8: Even or Odd Checker

What This Script Does. It asks for a whole number and tells you whether it is even or odd. It looks simple, but it teaches one of the most useful little tricks in all of programming.

What You'll Practice. Using the modulo operator and making a yes or no decision with if and else.

The Code.

```
number = int(input("Enter a whole number: "))

if number % 2 == 0:
    print(f"{number} is even.")
else:
    print(f"{number} is odd.")
```

How It Works. The star of this script is the percent sign, which is the modulo operator. It gives you the remainder after division rather than the result. Any number divided by two leaves a remainder of either zero or one. If the remainder is zero, the number divides evenly, so it is even. The double equals sign asks a question, is the left side equal to the right side, and Python answers with true or false. That answer decides which message prints. You will see modulo turn up again whenever you need to check for patterns, such as every third item in a list. It is one of those small tools that feels minor at first and then shows up almost everywhere.

Try This Upgrade. After the even or odd check, add another decision that says whether the number is positive, negative, or zero.

Script 9: Grade Result Calculator

What This Script Does. It takes a score from zero to one hundred and prints the matching letter grade. It is a clear example of how a chain of conditions can sort a number into the right bucket.

What You'll Practice. Using a chain of if and elif tests with the greater than or equal comparison.

The Code.

```
score = int(input("Enter a score from 0 to 100: "))

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

How It Works. Order is everything here. Python checks the conditions from the top down and stops at the first one that is true. A score of ninety-five passes the very first test and prints an A, so none of the lower tests even run. This is why the highest grade sits at the top. If you flipped the order and put sixty first, almost every score would

match it and everyone would get a D. The else at the bottom catches anything below sixty. A good habit is to read a chain like this out loud, from top to bottom, as a series of if this then that statements. It makes the logic obvious and helps you spot a gap before you ever run the code.

Try This Upgrade. Add a short line of feedback after each grade, such as praise for an A or a gentle nudge to review for an F.

Script 10: Password Strength Checker

What This Script Does. It looks at a password you type and gives a simple comment on whether it seems strong enough. This is a learning exercise to practice checking text, not a real security tool, so never rely on it to protect a real account.

What You'll Practice. Measuring the length of text, checking whether it contains a digit, and combining two true or false tests.

The Code.

```
password = input("Type a password to test: ")

length_ok = len(password) >= 8
has_digit = any(char.isdigit() for char in password)

if length_ok and has_digit:
    print("That passes this simple check.")
else:
    print("Try making it longer and adding a number.")
```

How It Works. The len() function counts how many characters the text contains, and the comparison stores a simple true or false in length_ok. The next line checks whether any character is a digit. The any() tool returns true the moment it finds one match, which saves you from writing a longer check by hand. The word and means both tests must be true for the password to pass. Real security is far more involved than this, but the script is a friendly way to see how length and content checks work together. If you are curious, print length_ok and has_digit on their own lines to see the true and false values your tests produce.

Try This Upgrade. Add a check for an uppercase letter using a similar line, then require all three tests to pass before calling the password strong, and print a different message for each level of strength.

Five more scripts done, and every one of them solved a small real problem. You have now used decimals, percentages, the modulo trick, and longer chains of decisions. Notice how often the same shapes return: take some input, do a little math, then decide what to print. That

rhythm is most of programming. In the next chapter you will hand the repetitive work over to loops and watch Python do the same task many times without complaint. That single idea, letting the computer repeat work for you, is where Python starts to feel genuinely powerful.

List and Loop Productivity Scripts

This is the chapter where Python starts to feel powerful. Up to now your scripts handled one piece of information at a time. Here you will work with lists, which hold many items under a single name, and loops, which repeat a task as many times as you need without you writing the same line over and over. That combination is the engine behind almost every useful program. The five scripts below are small, but they show you how to store a collection, walk through it, and act on each item in turn. Type them out, run them, and change the contents to make them your own. If a loop ever seems confusing, slow down and trace it on paper. Write out what each variable holds on the first trip around, then the second, then the third. Seeing those snapshots side by side makes the whole idea click.

Script 11: Simple To-Do List Printer

What This Script Does. It stores a few tasks in a list and prints them out as a tidy numbered list. It is your first taste of working with a whole collection instead of a single value.

What You'll Practice. Creating a list, looping through it, and numbering each item automatically.

The Code.

```
tasks = [
    "Reply to emails",
    "Buy groceries",
    "Call the dentist",
]

for index, task in enumerate(tasks, start=1):
    print(f"{index}. {task}")
```

How It Works. The square brackets hold three tasks in a list. The for loop visits each task in order, one trip around the loop per item. The clever part is enumerate, which hands you both the item and a running number at the same time. Starting that number at one gives you a clean human-friendly list instead of one that begins at zero. Whatever you put in the list, the loop prints it in the same neat format, so you could have three tasks or thirty without changing the printing code at all. That

reusability is the real lesson here. You write the loop once, and it scales to any size of list for free.

Try This Upgrade. Let the user add their own tasks first, then print the finished list. The next script shows you exactly how to collect items one by one. For now, simply changing the three tasks in the list and rerunning the script is a good warm up.

Script 12: Grocery List Builder

What This Script Does. It lets the user type items one at a time and keeps adding them to a list until they decide to stop. Then it prints the whole list back. This is a genuinely useful little tool.

What You'll Practice. Building a list as you go with append, and using a while loop that runs until the user is finished.

The Code.

```
groceries = []

while True:
    item = input("Add an item, or type done to finish: ")
    if item == "done":
        break
    groceries.append(item)

print("Your grocery list:")
for item in groceries:
    print("- " + item)
```

How It Works. The script starts with an empty list, shown by the empty brackets. The while True loop keeps going around forever until something stops it. Each time around, it asks for an item. If the user types done, the break statement jumps straight out of the loop. Otherwise, append tacks the new item onto the end of the list. When the loop ends, a second loop prints every item that was collected. This pattern, keep asking until the user signals they are finished, appears in countless real programs. It also explains why the empty list comes first: the loop needs something to append to before it can grow, so starting with the empty brackets is a habit worth keeping.

Try This Upgrade. Before printing, sort the list into alphabetical order so it is easier to read in the store.

Script 13: Number Countdown Timer

What This Script Does. It counts down from a number you choose all the way to zero, then prints a cheerful finish. It is a simple, satisfying way to watch a loop in motion.

What You'll Practice. Running a while loop that ends on its own and lowering a number step by step.

The Code.

```
start = int(input("Count down from what number? "))

while start > 0:
    print(start)
    start = start - 1

print("Liftoff!")
```

How It Works. This while loop has a condition instead of running forever. As long as start is greater than zero, the loop keeps going. Inside, it prints the current number and then lowers start by one. Because the number shrinks every time, it eventually reaches zero, the condition becomes false, and the loop stops on its own. Forgetting to lower the number is one of the most common beginner mistakes, since it traps the program in a loop that never ends. Once the countdown finishes, the last line prints the celebration. A loop that never stops is called an infinite loop, and at some point every programmer creates one by accident. If it happens to you, you can stop a runaway program by pressing Control and C together in most editors.

Try This Upgrade. Add the time module at the top and place time.sleep(1) inside the loop so it pauses one second between numbers, like a real timer.

Script 14: Name List Cleaner

What This Script Does. It takes a messy list of names with stray spaces and odd capitals, then prints a clean, neatly formatted version. This is a small example of something programmers do constantly, which is tidying up untidy data.

What You'll Practice. Looping through a list and using string methods to trim spaces and fix capitalization.

The Code.

```
messy_names = ["  alice ", "BOB", "Charlie  ", "  dana"]

clean_names = []
for name in messy_names:
    clean_names.append(name.strip().title())

print(clean_names)
```

How It Works. The loop visits each messy name in turn. The strip method removes any spaces from the front and back, and the title method puts a capital at the start of each word and lowercases the rest. Notice how the two methods are chained together with a dot, so the output of strip is passed straight into title. Each cleaned name is appended to a fresh list, which leaves the original untouched. Building a new list rather than editing the old one is a safe habit that saves you from surprises later. Chaining methods like this is common in Python, and you read it from left to right: take the name, strip it, then title it. Each step hands its result to the next.

Try This Upgrade. After cleaning the names, remove any duplicates so each name appears only once.

Script 15: Simple Quiz Game

What This Script Does. It asks a series of questions, checks each answer, and keeps score. By the end it tells the player how many they got right. It is a small project that pulls together everything in this chapter.

What You'll Practice. Storing question and answer pairs, looping through them, comparing input, and counting with a score variable.

The Code.

```
questions = [
    ("Which keyword repeats while true?", "while"),
    ("Which function prints text?", "print"),
    ("Which symbol gives a remainder?", "%"),
]

score = 0
for question, answer in questions:
    reply = input(question + " ")
    if reply.strip().lower() == answer:
        print("Correct!")
        score = score + 1
    else:
        print(f"Not quite. The answer is {answer}.")

print(f"You scored {score} out of {len(questions)}.")
```

How It Works. Each item in the questions list is a pair: the question and its correct answer. The loop unpacks each pair into two variables at once, which is why you see two names after the word for. The score starts at zero and goes up by one every time the reply matches. Before comparing, the script trims spaces and lowers the case of the reply so that a perfectly good answer is not marked wrong over a stray capital. At the end, len counts how many questions there were, so the final score always matches the real total even if you add more questions. That small detail, using len instead of typing the number three, is the difference between code that breaks the moment you change it and code that quietly keeps working. Whenever you are about to type a count by hand, ask whether Python can work it out for you instead.

Try This Upgrade. Turn it into a multiple choice quiz by printing a few options with each question and accepting a letter as the answer.

Look at what you can do now. You can store a collection, loop through it, build it up while the program runs, clean it, and search it for matches. Loops are the moment most beginners feel the gears finally catch, because a few lines of code can now handle work that would take ages by hand. Keep that quiz script in particular, since it is a tiny project you can grow into something much bigger. Next we will let Python reach beyond the screen and start working with real files. Take a moment to appreciate the jump you just made. Storing many things, repeating work, and making choices are three of the four ideas that almost every program is built from. The fourth, saving your results so they last, is exactly where the next chapter begins.

Text and File Scripts

So far your scripts have forgotten everything the moment they finish. This chapter changes that. You will learn to read and write real files on your computer, which is how a program remembers things between runs. You will also sharpen your text handling skills, since most files are made of text. A small safety note before you begin: keep these scripts in their own practice folder, and use the safe modes shown here. None of the scripts in this chapter delete anything. They only add to files or read from them, which means your work stays intact even if you run them many times. Working in a separate folder also keeps these test files away from anything important, so you can experiment freely without a moment of worry.

Script 16: Word Counter

What This Script Does. It takes a sentence or paragraph and tells you how many words it contains. It is genuinely useful for writers, students, and anyone working within a word limit.

What You'll Practice. Splitting text into separate words and counting the items in a list.

The Code.

```
text = input("Type a sentence or paragraph: ")

words = text.split()
print(f"That text has {len(words)} words.")
```

How It Works. The split method does the heavy lifting. With no instructions inside its brackets, it breaks the text apart wherever it finds spaces and hands you back a list of the individual words. Once you have that list, len simply counts how many items are in it. It is a lovely example of how two small built-in tools combine to solve a real task in a single line. Because split is smart about runs of spaces, a little extra whitespace will not throw the count off. This is the same idea you met when looping through a list, only here the list is created for you on the spot rather than typed out by hand. Many of the most useful one-liners in Python work exactly like this, by chaining a method that produces a list into a function that measures it.

Try This Upgrade. Add a second line that counts characters as well, using len on the original text instead of the list of words, then print both numbers in one friendly sentence.

Script 17: Text Case Converter

What This Script Does. It takes some text and prints it three ways: all lowercase, all uppercase, and title case with each word capitalized. It is a quick tool for fixing the look of a heading or a name.

What You'll Practice. Calling several string methods on the same piece of text.

The Code.

```
text = input("Type some text: ")

print("Lowercase:", text.lower())
print("Uppercase:", text.upper())
print("Title case:", text.title())
```

How It Works. A string method is just an action you can ask a piece of text to perform, written with a dot after the text. Here lower, upper, and title each return a new version of the text in a different style. An important detail is that none of these change the original. They hand back a fresh copy, which is why you can call all three on the same text without one affecting the next. If you ever want to keep a result, store it in a variable rather than only printing it. This idea, that methods give you a new value instead of quietly altering the original, is worth holding on to. It comes up constantly, and beginners who miss it often wonder why their text looks unchanged after calling a method but forgetting to capture what it returned.

Try This Upgrade. Turn it into a menu. Ask the user which style they want, then print only that one using an if and elif chain. It is a neat way to combine the decision making from earlier chapters with the string skills from this one.

Script 18: Simple Journal Saver

What This Script Does. It asks for a short journal entry and saves it to a text file. Run it whenever you like and your entries pile up safely in one place. This is your first script that creates something lasting on your computer.

What You'll Practice. Opening a file in append mode and writing a line of text to it.

The Code.

```
entry = input("Write a short journal entry: ")

with open("journal.txt", "a") as file:
    file.write(entry + "\n")

print("Saved to journal.txt")
```

How It Works. The open function reaches for a file called journal.txt, and the letter a stands for append, which means add to the end. Append adds to the end of the file instead of replacing what is already there. If the file does not exist yet, Python creates it for you. The with line is a tidy way to work with files, since it closes the file automatically when the block finishes, even if something goes wrong. The little piece that reads backslash n is a newline, which drops each entry onto its own line so your journal does not turn into one long run-on sentence. It helps to picture the difference between the two main writing modes. Append, the mode used here, is like adding a new page to the back of a notebook. The other common mode, write, would be like tearing out every page first and starting fresh, which is why this script deliberately avoids it. When in doubt while you are learning, reach for append so you never lose earlier work by accident.

Try This Upgrade. Add the date and time to each entry. Import datetime, fetch the current moment, and write it on the line before the entry so each note is stamped with when you wrote it.

Script 19: Read Your Notes Script

What This Script Does. It opens the journal file from the previous script and prints everything you have saved. Together, the two scripts make a tiny notebook you actually control.

What You'll Practice. Opening a file for reading and handling the case where the file is not there yet.

The Code.

```
try:
    with open("journal.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("No journal yet. Write an entry first.")
```

How It Works. This time the letter r means read, so the script looks at the file rather than changing it. The read method pulls the entire contents back as one piece of text, which is then printed. The try and except lines are the real lesson. If the file is missing, Python would normally stop with an error, but the except block catches that specific

problem and prints a friendly message instead. Anticipating what could go wrong and handling it gently is what separates a script that crashes from one that feels finished. Notice that the except line names a specific problem, FileNotFoundError, rather than catching every possible error. That precision is good practice. It means a genuine bug somewhere else will still show up loudly instead of being swallowed and hidden, which would make it far harder to find later.

Try This Upgrade. Show only the most recent entry. Read the lines into a list with readlines and print just the last one, which is a handy way to glance at what you wrote most recently.

Script 20: Simple Text Cleaner

What This Script Does. It takes messy text and tidies it up by trimming the ends and squeezing out doubled spaces. It is a small helper you will reach for whenever text arrives in a less-than-perfect state.

What You'll Practice. Combining strip and replace to clean up a piece of text.

The Code.

```
messy = input("Paste some messy text: ")

clean = messy.strip()
# two spaces become one space
clean = clean.replace("  ", " ")

print("Cleaned text:")
print(clean)
```

How It Works. First strip trims any spaces hanging off the front and back of the text. Then replace looks for any place where two spaces sit together and swaps them for a single space. The result is stored back into the same clean variable, so each line builds on the one before. This is a gentle introduction to cleaning data, which is a surprisingly large part of real programming. Text rarely arrives perfectly formatted, and a few lines like these can save a lot of manual tidying. One small caveat to notice: replacing two spaces with one will not catch three or four spaces in a single pass, since it works on pairs. For now that is fine, and later you will meet tools that scrub every kind of stray spacing at once.

Try This Upgrade. Extend the cleaner to remove an unwanted symbol of your choice, such as an extra exclamation mark, using another replace chained after the first.

You have crossed an important line in this chapter. Your scripts can now remember things, because they can write to files and read them back. That is the foundation under journals, logs, settings, save files, and almost every app that keeps your data between visits. You also practiced the calm, professional habit of expecting the occasional problem and handling it kindly. In the next chapter you will point these skills outward and let Python tackle small real-world chores. The jump from a program that talks only to the screen to one that works with files is a big one, and you just made it. Everything from here builds on the simple, safe habits you practiced in this chapter, so take a moment to feel good about how far you have come.

Tiny Automation and Real-Life Utility Scripts

This is the chapter that makes people say, I did not know Python could do that. The scripts here reach out from your editor and do small jobs in the real world: picking from a list, peeking inside a folder, previewing changes, and opening websites for you. They stay short and friendly, but they feel useful and a little bit magical. One promise runs through this chapter: nothing here deletes or overwrites your files. Most of these scripts only look at things or open them, and one script adds a line to a practice log file, so keep that one in your practice folder. That keeps everything safe to run. As with every chapter, type the scripts yourself and run them. Automation only clicks once you watch a small one actually work, and these are about as small and safe as automation gets.

Script 21: Random Lunch Picker

What This Script Does. It picks a random lunch from a list, so you never have to make that tiny decision again. It is lighthearted, but it is also a perfect, low-pressure way to see a real module in action.

What You'll Practice. Storing options in a list and letting the random module choose one for you.

The Code.

```
import random

lunches = [
    "Sandwich",
    "Salad",
    "Soup",
    "Pasta",
    "Leftovers",
]
print("Today's lunch is:", random.choice(lunches))
```

How It Works. You met random.choice earlier, and here it is again doing the same simple job: reaching into a list and pulling out one item at random. The word import at the top is the part to focus on. It tells Python to load a module, which is a bundle of ready-made tools someone else has already written and tested. The random module comes with Python, so there is nothing to install. Once it is imported, all of its tools become available with a dot, the same way string methods worked. Modules are how a few lines of your code can stand on a

mountain of work done by others. That is the quiet superpower behind Python. You rarely build everything from scratch, because so much of what you need is already a single import away.

Try This Upgrade. Let the user add their own lunch ideas to the list before the choice is made, so the picker fits your own tastes.

Script 22: Folder File Counter

What This Script Does. It looks inside a folder you choose and tells you how many files are in it. This is your first real automation, since the script is now working with your computer rather than just the screen.

What You'll Practice. Using the os module to list a folder and looping through the results to count the files.

The Code.

```
import os

folder = input("Type the path to a folder: ")
items = os.listdir(folder)

count = 0
for item in items:
    if os.path.isfile(os.path.join(folder, item)):
        count = count + 1

print(f"That folder has {count} files.")
```

How It Works. The os module is your link to the operating system, the part of your computer that manages files and folders. Its listdir tool returns the names of everything inside the folder you name. The loop then checks each name and, with isfile, asks whether it is a file rather than a folder, counting only the files. The join tool builds a full path safely, which matters because Windows and macOS handle paths a little differently. Paste a real folder path when you run it, since pointing at a folder that does not exist will produce an error rather than a count. If that happens, read the error closely. It usually names the exact path it could not find, which is often just a small typo away from the right one.

Try This Upgrade. Count only files of one type, such as those ending in .txt or .py, by checking the end of each name before adding to the count. The endswith method is perfect for this.

Script 23: Simple File Renamer Preview

What This Script Does. It shows you how a batch of files would be renamed, without actually changing a single one. It is the safe, sensible first step before any automation that touches your files.

What You'll Practice. Listing a folder and building a tidied up version of each file name, all in preview form.

The Code.

```
import os

folder = input("Type the path to a folder: ")

for name in os.listdir(folder):
    new_name = name.lower().replace(" ", "_")
    print(name, "would become", new_name)
```

How It Works. For every name in the folder, the script builds a cleaned-up version, lowercased and with spaces turned into underscores, then prints the old name next to what it would become. Here is the important part: the word would is honest. This script only prints. It never renames anything, so you can run it as many times as you like and your files stay exactly as they are. Previewing first is how careful programmers work. You look at the planned changes, confirm they are right, and only then consider acting. Building this habit early will save you from painful mistakes. Many professional tools offer a preview or dry run mode for exactly this reason. Seeing the plan before it runs turns a scary, irreversible action into a calm, checkable one.

Try This Upgrade. As an advanced challenge, you could make the rename real with the os module. Only attempt it after copying the folder somewhere safe and testing on files you would not mind losing, since real renames cannot be undone with a button.

Script 24: Website Opener Script

What This Script Does. It opens a list of websites in your browser, one after another. It is a tiny taste of automating your daily routine, and it feels great the first time it works.

What You'll Practice. Using the webbrowser module and a loop to open several pages in turn.

The Code.

```
import webbrowser

sites = [
    "https://www.python.org",
    "https://docs.python.org/3/",
]

for site in sites:
    webbrowser.open_new_tab(site)
```

How It Works. The webbrowser module, included with Python, knows how to hand a web address to your default browser. The loop walks through the list of addresses and tries to open each one in a new browser tab. That is the whole trick. Notice that this script only opens pages, the same thing you would do by clicking a bookmark. It does not log in, read, or collect anything from those pages. Keeping a script to a single, clear job like this makes it easy to understand and easy to trust. Be ready for a handful of tabs to spring open the moment you run it. If that feels like too many, trim the list down to a single site while you are testing, then add more once you are happy with how it behaves.

Try This Upgrade. Build your own morning routine by replacing the list with the sites you check each day, such as your email, a news page, and your calendar. Save it somewhere you can double click, and your whole morning setup becomes a single step.

Script 25: Daily Log Timestamp Script

What This Script Does. It saves whatever you type to a log file, with the current date and time added automatically. It is a quiet, dependable way to keep a running record of anything you want to track.

What You'll Practice. Getting the current date and time, formatting it, and appending a stamped line to a file.

The Code.

```
from datetime import datetime

note = input("What would you like to log? ")
stamp = datetime.now().strftime("%Y-%m-%d %H:%M")

with open("daily_log.txt", "a") as file:
    file.write(stamp + " " + note + "\n")

print("Logged.")
```

How It Works. This script brings together two skills you already have and adds one new one. The new piece is datetime.now, which asks your computer for the exact current moment. On its own that moment is a

bit unwieldy, so strftime formats it into a clean, readable stamp using a simple pattern of codes for the year, month, day, hour, and minute. From there it is familiar ground: open the file in safe append mode, write the stamped note on its own line, and confirm. Because it appends, every run adds to the record without ever erasing what came before. The pattern of codes inside strftime is worth a quick look in the documentation, since the same idea lets you format dates and times in dozens of ways.

Try This Upgrade. Add a category to each entry, such as work, health, or learning, by asking for it and writing it alongside the note.

You have now written scripts that pick, count, preview, open, and log, all working with the real world instead of just the screen. Just as important, you did it safely. You saw how to look before you leap with a preview, how to keep a script focused on one honest job, and how to add to your records without ever destroying them. That careful mindset matters far more than any single trick, because it is what lets you automate with confidence. One short chapter remains, and it is about where you go from here. Before you turn the page, pick the one script from this chapter that felt most useful to you and keep it somewhere handy. A tool you actually reach for is worth far more than one you only read about.

What to Build Next

You made it. Twenty-five scripts across five project chapters, and a folder full of working code that you typed and understood. This last chapter is short on purpose. It is not about learning something new. It is about pointing you toward what comes next, so the momentum you built does not quietly fade once you close the book. Read it now, then come back to it whenever you are not sure what to try next. There is no test at the end and no right order to follow. The only rule that matters from here is to keep building.

Combine Tiny Scripts into Bigger Tools

The fastest way to grow is to glue together scripts you already have. Each one is a building block, and they fit together far more easily than you might expect. Your word counter and your text cleaner can become a small writing helper that tidies text and then reports on it. Your journal saver and your timestamp logger can join into a proper diary app that dates every entry. Your to-do list printer, paired with file saving, becomes a task manager that actually remembers your tasks between runs. Your folder counter and your renamer preview can grow into a gentle file organizer. None of these need new ideas. They only need you to connect the pieces you have already built. When you combine two scripts, start by copying both into one file, get them running side by side, and only then weave them together. Small, working steps beat one giant leap every single time.

How to Upgrade Every Script

When you want to improve a script but are not sure how, there is a reliable route you can follow. First, add input so the script asks the user for values instead of using fixed ones. Next, add error handling with try and except so it fails gently when something is off. Then, save the results to a file so they survive after the program closes. After that, take any chunk of code you keep repeating and turn it into a function you can call by name. Once that feels comfortable, add a simple menu so the user can choose what to do, and then offer a few more options to make it flexible. The final step is the one from the section above: combine it with another script. You do not have to do all of these at once. Pick the next single step, make it work, and run the script again. Each pass leaves you with a script that is a little more capable and a little more

yours. That steady, one change at a time rhythm is exactly how large programs get built, only repeated many more times.

Your Beginner Python Portfolio

Those files in your folder are more than practice. Together they are the start of a portfolio, real proof that you can take an idea and make a computer carry it out. Treat them with a little care. Give each script a clear name, and add a short comment at the very top of each one explaining what it does, so the future you knows at a glance. Keep them in one tidy place. As the collection grows, you might write a simple list of what each script does, and one day you may even share the whole folder online. For now, the goal is humble and powerful at the same time: a growing pile of things you built yourself. Looking back over it in a month will surprise you, because progress is hard to feel day to day but obvious across a stack of finished work.

Final Encouragement

Here is the truth that took with twenty-five scripts to earn: you are not someone who is trying to learn Python anymore. You are someone who writes it. The gap between those two is enormous, and you have already crossed it. From here, the single best thing you can do is build something you actually care about, however small. A tool for your hobby, a helper for a boring chore, a tiny game for a friend. Motivation comes far more easily when the result matters to you.

Expect errors, because they never stop, not even for experts. The difference is that you now know an error is just a signpost, not a stop sign. Read it, change one thing, and run again. That simple loop will carry you through problems far bigger than anything in this book. Keep your scripts, keep your curiosity, and keep going. You have everything you need to take the next step, and the next one after that. Whatever you build next, you will not be starting from zero. You will be starting from here, with twenty-five scripts behind you and a way of thinking that turns big problems into small, solvable steps. That is what it means to be a programmer, and it is yours to keep.