FIRST PRINCIPLES

A Beginner's Journey Into Computing and Python

PART ONE

What Computing Actually Is

-By Deepanshu

Contents

Before We Begin

Most books that teach programming open with an editor already on the screen, a blinking cursor, and an instruction to type something. This one does not, and the delay is deliberate.

There is a particular kind of confusion that comes from learning to type commands a machine will obey before understanding what the machine is, or why it obeys anything at all. It is entirely possible to memorize the shape of a loop, an if-statement, or a function without ever understanding what a program actually is, or why these particular pieces exist rather than some others. Many people who consider themselves "bad at programming" are not bad at logic, and are not bad at precise thinking. They were simply never shown the ground the subject stands on, and were left to guess at it while trying to hold a dozen unfamiliar words in mind at the same time.

This book tries something different. It borrows a method more often associated with physics and engineering than with teaching programming: reasoning from *first principles*. The idea is old. Aristotle described something close to it more than two thousand years ago, and centuries later Descartes built an entire philosophy on the refusal to accept anything that could not be reasoned upward from more basic truths. In recent years the method has resurfaced in public conversation about how to design machines and companies, in part through the engineer and entrepreneur Elon Musk, who has spoken publicly about breaking a problem down into its most fundamental, verifiable pieces and reasoning up from there — rather than reasoning by analogy, which is the far more common habit of assuming something is true simply because it resembles something else that already is.

This book applies that same discipline to computing. Rather than asking the reader to accept that a variable behaves a certain way, or that a

loop repeats for some built-in reason, because that is simply how programming works, it starts underneath all of it — with what a computer fundamentally is, what a problem must look like before a machine can do anything with it, and how a small number of electrical switches gave rise to everything from a pocket calculator to software that can hold a conversation. Only once that foundation is in place does Python enter the story: not as a set of rules to be memorized, but as one particular, carefully considered answer to problems the reader will by then already understand.

The first part of this book, accordingly, contains no code at all. Readers eager to start typing are welcome to skip ahead; the book will still be here when they come back. But for anyone who has tried to learn to program before and quietly given up, the suggestion offered here is that the trouble was rarely the syntax itself. It was that there was nowhere to put what was being learned — no underlying picture for the rules to attach to, so they stayed loose, arbitrary, and easy to forget.

What follows is meant to be read in order, at least the first time. Everything after the first few chapters depends on them.

PART ONE

What Computing Actually Is

Four chapters, and no code at all. An answer to a question most programming books never stop to ask: before anything else, what is actually happening inside the machine — and why does any of it work in the first place?

CHAPTER ONE

The Nature of a Problem

LONG BEFORE THERE WERE COMPUTERS, there were procedures. A Babylonian scribe measuring out a field according to a fixed method. A merchant converting currencies by a rule passed down rather than reasoned out fresh each time. A cook following not a memory of a dish but a repeatable path toward one. None of these people were programming, in any sense they would have recognized. But each of them was doing something a computer would later be built to do: following a finite sequence of unambiguous steps to get reliably from a starting point to a destination.

The word for this kind of procedure — *algorithm* — comes from a real person, not an abstract root invented for the occasion. Muhammad ibn Musa al-Khwarizmi was a mathematician working in ninth-century Baghdad, at a research institution called the House of Wisdom. His textbook on solving equations gave the world the word *algebra*. His name, Latinized by European translators centuries later as "algorismus," gave the world *algorithm*. It is a small piece of history worth sitting with: the most basic term in all of computing was coined for a human being solving problems with a reed pen, roughly a thousand years before the first electronic switch was built.

Al-Khwarizmi did not invent the idea of a fixed procedure — that idea is far older than his name for it. More than a thousand years earlier, Euclid had already written down a method for finding the greatest common divisor of two numbers: divide the larger by the smaller, note the remainder, then repeat the process on the smaller number and that remainder, until nothing is left over. Whatever number the process stopped on was the answer. Euclid's algorithm, as it is still called today, needed no judgment calls, no case-by-case thinking, no expertise beyond

arithmetic. Two different people, or two hundred, following the same steps on the same starting numbers, would always arrive at the same result by the same route. That reliability — not cleverness, not intelligence — is the entire point of an algorithm, and it is the property that would eventually make it possible to hand such procedures to a machine instead of a person.

That reliability — not cleverness, not intelligence — is the entire point of an algorithm.

It helps to notice what this rules out. “Cook the onions until they look right” is an instruction a competent cook can follow, but it is not an algorithm, because “look right” asks for judgment that has to live inside the cook, not on the page. “Cook the onions over medium heat for eight minutes, stirring every two minutes” is closer: almost anyone, with no special judgment at all, could follow it and land in roughly the same place. The difference has nothing to do with cooking and everything to do with precision — with whether every step is spelled out clearly enough that following it requires no private interpretation. A computer, as it happens, is an extreme case of a follower that supplies none of its own interpretation whatsoever. It will do exactly what it is told, in exactly the order it is told, and if a step is missing or ambiguous, it will not guess charitably on anyone’s behalf. It will simply fail, or do something no one intended.

This is worth pausing on, because it reframes what learning to program actually is. It is not, at bottom, learning to talk to a computer in some special dialect. It is learning to think in procedures precise enough that no judgment is required to carry them out — and then, separately, learning the particular notation a given language expects that procedure to be written in. The second part is what most beginners spend all their

effort on. The first part is what actually determines whether the resulting program does what its author intended, and it is a skill that exists independently of any programming language at all.

The first procedure ever written down specifically for a machine, rather than for a person, is usually credited to Ada Lovelace, an English mathematician working in the 1840s alongside Charles Babbage on a design for a mechanical computer called the Analytical Engine — a machine built from gears and levers that was never fully constructed in her lifetime. In a set of notes on the Engine published in 1843, Lovelace worked out, step by step, how the machine could compute a sequence of numbers known as Bernoulli numbers. Historians still debate exactly how much of that algorithm was hers alone versus developed jointly with Babbage, but the notes themselves are not in dispute, and they are frequently pointed to as the first published example of something recognizable as a computer program: a precise sequence of operations, written for a machine that did not yet exist in working form, describing how to get from a question to an answer.

None of this — Euclid, al-Khwarizmi, Lovelace — required electricity. A procedure is a procedure whether it is carried out by a scribe, a cook, a mathematician with a pen, or a machine. What a computer adds is not the idea of a procedure. It adds speed, and an almost total absence of complaint. But before any of that speed can be put to use, a machine built of physical components has to be capable of doing something with a procedure at all — has to be able to hold a step in its attention, follow it, and move to the next one. That requirement, and how it came to be satisfied by nothing more exotic than switches turning on and off, is where the story goes next.

KEY IDEAS

- An algorithm is a finite, unambiguous procedure — precise enough that no personal judgment is needed to carry it out.
- The word comes from al-Khwarizmi, a 9th-century Baghdad mathematician; the idea itself is far older (Euclid, c. 300 BCE).
- Learning to program is really learning to think in precise procedures first — a language's syntax is the smaller, second half of the skill.
- Ada Lovelace's 1843 notes on the Analytical Engine are widely credited as the first algorithm written specifically for a machine.

CHAPTER TWO

The Machine Beneath Everything

A COMPUTER, AT THE VERY BOTTOM of everything it does, does not calculate, does not remember, and does not decide. It switches. Every operation a computer will ever perform — running a spreadsheet, rendering a film, holding a conversation — reduces, at the hardware level, to enormous numbers of tiny electrical switches turning on and off in coordinated patterns. Understanding that single fact removes most of the mystery from what a computer is.

The switch itself is called a *transistor*, and its invention is usually dated to 1947, at Bell Labs in New Jersey, where physicists John Bardeen, Walter Brattain, and William Shockley built the first working example — work that earned the three of them the Nobel Prize in Physics in 1956. A transistor can do what a light switch does, letting current through or blocking it, but it can be switched billions of times a second, has no moving parts to wear out, and can be manufactured at a scale a light switch never could be: a modern processor holds many billions of transistors on a piece of silicon smaller than a fingernail.

Why two states — on and off — rather than, say, ten, to match the number system most humans grew up using? The honest answer is reliability rather than elegance. Building a component that can distinguish ten different, evenly spaced voltage levels with the reliability required to run a computer turns out to be a genuinely hard engineering problem, since electrical noise constantly threatens to blur those levels into one another. Building a component that reliably distinguishes only "on" from "off" is a far easier problem, and it is the one electrical engineering was able to solve well enough, early enough, to build a machine around. It is not the only choice ever tried — a Soviet computer called Setun, built in 1958, used three states instead of two — but

two-state, *binary* switching is the choice that won, and it is the one every mainstream computer since has been built on.

Once a device can reliably be on or off, it can represent one binary digit, or *bit*: a 0 for off, a 1 for on. A single bit is not very expressive on its own, but the same trick used in ordinary counting — where the position of a digit determines its value, so that the "1" in 10 means something different from the "1" in 100 — works identically in base two. In the everyday decimal system, the digits after the ones place stand for tens, hundreds, thousands: each position ten times the last, because there are ten symbols to work with. In binary, with only two symbols, each position simply doubles instead: ones, twos, fours, eights, sixteens. The bit pattern 1011, read the same way, comes to eight plus zero plus two plus one, or eleven. Nothing about this requires new mathematics — it is the same place-value idea taught in primary school, worked out with two symbols instead of ten, because two is the number of states a transistor can reliably hold.

A computer, at the very bottom of everything it does, does not calculate, does not remember, and does not decide. It switches.

Numbers explain themselves quickly enough this way, but a computer plainly handles more than numbers — it displays letters, plays sound, shows photographs of a beach. The resolution to this is that none of those things are ever actually stored as letters, sounds, or photographs. They are stored as numbers, and a separate agreement, called an *encoding*, defines which number stands for which letter or which shade of color. An early and highly influential encoding called ASCII, developed in the early 1960s, assigned each English letter, digit, and punctuation mark a number between 0 and 127 — the capital letter A, for instance, is by convention the number 65. Modern software mostly uses a much larger

successor called Unicode, which extends the same basic idea to cover essentially every writing system in current and historical use, using more bits per character to make room for the extra symbols. A photograph, by the same logic, is simply a very long list of numbers, one small group per pixel, each group recording how much red, green, and blue light that pixel should show. There is no special material a computer uses to store an image or a sound file that is somehow different from the material it uses to store a number. There is only ever numbers, interpreted differently depending on what a program has been told to expect.

The remaining piece is memory, and here the decisive idea belongs to the mathematician John von Neumann, who described it clearly in a 1945 report on the design of a computer called the EDVAC — building, as von Neumann was careful to acknowledge, on engineering work already underway by J. Presper Eckert and John Mauchly, and on ideas Alan Turing had explored theoretically several years earlier. Before this design took hold, the "program" a machine was to run — the sequence of steps — was often physically separate from the data it worked on, sometimes wired in by hand for each new task. Von Neumann's report described something different: a single memory holding both the instructions and the data those instructions operated on, with nothing to distinguish one from the other except how the machine chose, at a given moment, to treat them. A computer built this way can be handed an entirely new set of instructions — a new program — simply by loading different numbers into the same memory it always uses, rather than being physically rewired. This is the idea, more than any single invention of hardware, that turned a calculating machine into a general-purpose one: the same box of switches, doing arithmetic on Tuesday and rendering a video on Wednesday, because both tasks are, underneath, only ever numbers in memory being followed as instructions.

None of this, so far, explains how anyone writes software, because no one writes software as literal sequences of ones and zeros — or almost no one, and not for very long. Binary is what the machine ultimately

requires, but it is a poor language for a human being to think or write in, for reasons that have nothing to do with intelligence and everything to do with working memory: it is simply too easy to lose track of the two-thousandth digit in a row of them. What happened next in the history of computing is a series of inventions aimed at a single goal — letting people describe what they want done in a form closer to how they think, while some other piece of software handles the tedious, error-prone work of turning that description into the ones and zeros the machine actually runs. That series of inventions is the subject of the next chapter.

KEY IDEAS

- Every computer operation reduces to transistors — tiny electrical switches — turning on and off.
- Binary won out because reliably telling two states (on/off) apart is far easier than reliably telling ten apart.
- Letters, images, and sound are never stored directly — only ever as numbers, read through an agreed encoding like ASCII or Unicode.
- The stored-program (von Neumann) design — instructions and data sharing one memory — is what makes a computer general-purpose rather than single-task.

CHAPTER THREE

Climbing the Ladder of Abstraction

NO ONE WRITES SOFTWARE in binary anymore, and almost no one ever did for very long. In the earliest electronic computers of the 1940s, programming genuinely did mean setting switches and rewiring panels by hand, or at best entering raw sequences of ones and zeros that corresponded directly to the machine's built-in instructions. It worked, in the sense that it produced running programs, but it was slow, exhausting, and desperately easy to get wrong — a single misplaced digit in a long binary sequence could send the machine somewhere entirely unintended, with no clue as to why.

The first rung up from raw binary was *assembly language*, which replaced numeric instruction codes with short, memorable abbreviations a person could actually read: an instruction to add two numbers might be written ADD instead of some binary pattern standing for it. Assembly did not change what the machine ran — every assembly instruction still corresponds directly to one particular binary instruction understood by the underlying processor — but it changed what the human being had to hold in mind while writing it. A separate program called an assembler took care of the mechanical translation from the readable abbreviation back into the binary the machine actually required. Kathleen Booth, working at Birkbeck College in London in the late 1940s, is credited with some of the earliest work on assembly language and the assemblers that translated it, developed for a machine she helped design there.

Assembly was a real improvement, but it still required thinking in terms the machine cared about — individual instructions, specific memory locations, one register at a time — rather than in terms of the problem being solved. The next leap was the *high-level language*: a way

of writing a program that looked much closer to ordinary mathematical or logical notation than to any particular machine's instruction set, paired with software sophisticated enough to translate that notation down into working machine code automatically. Grace Hopper, a mathematician and naval officer working at Remington Rand in the early 1950s, built one of the first tools to do this translation — a program she called a *compiler* — and spent much of the following decade arguing, against considerable early skepticism, that programs could and should be written in something closer to English, rather than requiring every programmer to think like the machine. Her advocacy fed directly into COBOL, one of the earliest high-level languages to see wide business use.

The word compiler names one of two broad strategies for getting from human-readable source code to something a processor can execute, and the distinction matters for understanding Python later on. A compiler translates an entire program, in advance, into machine code, producing a separate file that the computer can then run directly and repeatedly without needing the compiler again. An *interpreter*, by contrast, reads the source code and carries out its instructions more or less as it goes, without producing a separate standalone machine-code file first. Compiling generally produces faster-running programs, since all the translation work happens once, ahead of time. Interpreting tends to be slower to run but faster to experiment with, since there is no separate translation step standing between changing a line of code and seeing what it does.

No one writes software in binary anymore, and almost no one ever did for very long.

Python sits in an interesting middle position that is worth getting right, since it is common to hear Python described simply as "an interpreted language," and that description is not quite complete. When a

Python program runs, it is first translated into an intermediate form called *bytecode* — not the final binary a processor understands, but not the original source code either, something in between — and that bytecode is what a piece of software called the Python interpreter actually executes, translating it into machine instructions as it goes. This detail rarely matters day to day, but it explains, among other things, why Python code often runs a little slower than a fully compiled language like C: some of the translation work Python does is happening while the program runs, rather than entirely beforehand.

Stepping back, all of this — assembly, compilers, interpreters — is really one idea wearing different clothes, and the idea is *abstraction*: building a layer that hides the tedious or error-prone detail of the layer beneath it, so that the person working at the new layer can think about the problem instead of the machinery. A person writing Python does not think about registers or binary opcodes, in the same way a person driving a car does not think about fuel injection timing. The detail has not gone away — it is still happening, several layers down, exactly as it always did — it has simply been made someone else's problem: the compiler's, the interpreter's, the processor's. Nearly the entire history of software development since the 1950s can be read as the steady construction of more such layers, each one built to let people reason a little further from the raw switches and a little closer to whatever they actually wanted the machine to do.

Python is one particular point near the top of that ladder — a deliberate, carefully argued-for design, built by a specific person for specific reasons, rather than an inevitable or obvious way for a language to look. What those reasons were, and why they produced a language that looks the way Python does, is where the first part of this book finishes.

KEY IDEAS

- Assembly language replaced raw binary codes with readable abbreviations, mechanically translated by an assembler.
- Grace Hopper's early compilers made it possible to write programs in something closer to English than to machine instructions.
- A compiler translates a whole program in advance; an interpreter carries it out as it goes — Python actually does a version of both, via compiled bytecode.
- Assembly, compilers, and interpreters are all the same idea: abstraction — hiding detail so people can reason at a higher level.

CHAPTER FOUR

Why Python Exists

IN DECEMBER 1989, a computer scientist in Amsterdam named Guido van Rossum was looking for a project to occupy the week between Christmas and New Year, with his usual office closed for the holiday. He had spent several years working on a teaching language called ABC, well regarded for its readability but limited in ways that kept it from wider use, and he decided to build a successor: something that kept ABC's emphasis on clear, readable code, while fixing the design choices that had held it back. He had recently been reading scripts from the British comedy series *Monty Python's Flying Circus*, and rather than name his new language after himself or describe it technically, he named it after the show. The choice had nothing to do with snakes, whatever the logo on the official website might suggest today.

The language was not built inside a large company, and it was not commissioned by anyone. Van Rossum released it publicly in 1991, and for years afterward continued to guide its design personally and informally, earning the semi-official title "Benevolent Dictator for Life" from the community that grew up around it — a title he held until stepping back from the role in 2018. That origin as one person's deliberately-considered side project, rather than a committee's output or a corporation's product, shows up throughout Python's design in a preference for a single, clear way of doing something over several competing ones.

That preference was written down explicitly in 1999 by Tim Peters, a member of the early Python community, in a short list of guiding principles that came to be known as "The Zen of Python." Its lines argue, among other things, for readability over cleverness, for explicitness over implicit magic, and for resisting the temptation to solve a problem in

more ways than the language strictly needs to offer. Those principles are still built into Python today, both formally — as an actual, hidden text a Python installation carries with it — and informally, in the instinct experienced Python programmers develop for what "looks like Python" and what does not.

The clearest way to see what any of this actually bought is to look at the same small task rendered at different rungs of the ladder described in the previous chapter. Adding two numbers and displaying the result, at the level of raw assembly language, involves naming specific registers, issuing an explicit instruction to add their contents, and issuing a further explicit instruction to hand the result to whatever routine prints output — several distinct lines, none of which mention "add these numbers and show me the answer" in anything resembling those words. Written in C, a widely used compiled language from the 1970s still fundamental to modern computing, the same task shrinks to a handful of lines, but still requires declaring the type of every value in advance and wrapping the output in a specific formatting instruction. Written in Python, it is one line: `print(2 + 2)`. Every layer of ceremony the earlier approaches required — naming registers, declaring types, formatting output — has been pushed down into the interpreter, which handles it invisibly. Nothing about the underlying computer changed to make this possible. What changed is how much of the machine's detail a human being is required to look at before getting an answer.

This is not a claim that Python is simply "better," full stop — a claim any honest account of the language has to resist. Every design choice that buys readability and ease of use costs something elsewhere, and Python's costs are the ones already touched on: a program written in it will typically run slower than the same logic written in C, and the interpreter has to be present on whatever machine eventually runs the code. Which trade-off is the right one depends entirely on the job — a video game engine rendering graphics sixty times a second makes a different bargain than a script analyzing a spreadsheet once a week, and mature software

projects frequently use several languages together for exactly this reason, reaching for a compiled language where raw speed matters and a language like Python where a person's time and clarity of thought matter more.

Every design choice that buys readability and ease of use costs something elsewhere.

For a great deal of the work software is actually asked to do, however, the second bargain is the more valuable one, which goes some way toward explaining Python's current reach: it sits underneath a large share of present-day data analysis and machine learning work, runs a substantial portion of the backend behind ordinary websites, and automates everything from spreadsheets to laboratory instruments — all fields where a person's ability to read, reason about, and modify code matters at least as much as raw execution speed. None of that reach was the plan in December 1989. It was a byproduct of a language designed, from its very first week, around the needs of the person reading the code rather than the machine running it.

That is the point this book has been building toward since its first page, and it is worth stating plainly before Part One ends. A computer is, underneath everything, a very large number of switches. Those switches can represent numbers, and numbers can represent anything else a program needs to work with. Getting a machine to do something useful with those numbers requires an unambiguous procedure — an algorithm — and getting that procedure into the machine at all requires passing it down through several layers of translation, each one built to spare a human being some portion of the machine's underlying detail. Python is one particular, carefully reasoned stopping point on that ladder: high enough that a beginner can write something true and useful within their first hour of study, low enough that what they write still corresponds,

however many layers down, to real switches actually turning on and off. Nothing about the chapters that follow will require taking any of that on faith. It has, at this point, already been shown.

KEY IDEAS

- Guido van Rossum built Python as a December 1989 side project — a successor to a teaching language called ABC, named after Monty Python's Flying Circus.
- Python's guiding philosophy, later written down as "The Zen of Python," favors readability, explicitness, and one obvious way of doing things.
- The same task takes many lines in assembly, fewer (but type-declared) lines in C, and one plain line in Python — nothing about the machine changed; what changed is how much detail a person has to look at.
- Readability isn't free: Python typically trades raw execution speed for a person's time and clarity, which is why compiled languages still matter for other jobs.

PART TWO

The Grammar of a Program

Nine chapters that finally put Part One's ideas to work. Nothing here is an arbitrary rule to memorize — every piece of Python syntax that follows is a specific, deliberate answer to something the first four chapters already explained the need for.

CHAPTER FIVE

Preparing the Ground

BY THE END OF THE LAST CHAPTER, Python existed only as an idea — a set of design decisions made by one person in Amsterdam, and the reasons behind them. Turning that idea into something a reader can actually use requires one more, entirely unglamorous step: getting a copy of the Python interpreter running on an actual machine. This chapter is about what that step really involves, and about the two fundamentally different ways of using the interpreter once it is there. The literal, click-by-click instructions for installing it on a particular operating system are saved for an appendix at the back of this book — they change more often than anything else here, and they do not require the same kind of explanation as everything else in this chapter.

Installing Python means placing two things on a computer: the interpreter itself — the program that actually reads Python code and carries it out, doing for real what the last four chapters described in principle — and the standard library, a large, pre-written collection of tools for common tasks (reading files, doing math beyond basic arithmetic, working with dates, and a great deal else) that ships alongside the interpreter so that not every program has to build them from nothing. Together, interpreter and library are usually just called "Python," as in "I have Python installed," and that shorthand is harmless as long as it is clear that both pieces are included.

Once the interpreter exists on a machine, there are two entirely different ways to put it to work, and the distinction matters more than it might first appear. The first is to run it interactively, in what is usually called the interpreter's interactive shell, or REPL — a slightly awkward acronym for read, evaluate, print, loop, which is a fair description of what it does. Typing a line and pressing Enter hands that line to the

interpreter immediately; the interpreter evaluates it and prints back whatever the result is, right away, and then waits for the next line. A short exchange with it looks like this:

Typed at the interactive prompt

```
>>> 2 + 2
4
>>> 9 * 6
54
```

The three greater-than signs are the prompt: the interpreter's way of announcing that it is waiting, and every line starting with them in this book is something typed into that live exchange, not part of a saved program. Nothing is remembered here for later — close the shell, and the numbers 2, 4, 9, 6, and 54 are gone as if the exchange never happened. That is exactly what makes the shell good for one particular purpose: trying a small idea immediately, seeing exactly what it does, and discarding it just as quickly. It is a poor tool for anything meant to be run twice.

The second way of using the interpreter is to write instructions into a plain text file, save it with a `.py` extension, and hand the whole file to the interpreter at once, rather than one line at a time. Such a file is called a script, and running one means the interpreter reads it from the first line to the last, carrying out each instruction in order, with no live back-and-forth and no one watching until it either finishes or fails. The script itself, though, persists — sitting on disk exactly as written, ready to be handed to the interpreter again tomorrow, or given to somebody else's machine entirely.

The shell is for trying an idea out in ten seconds. The script is for anything meant to last longer than that.

This is the more important of the two habits to build, because almost everything worth writing is worth running more than once. A script named `convert.py`, once saved, can be run from a command line — a text-based way of giving instructions directly to the operating system — by typing `python convert.py` (or, on some systems, `python3 convert.py`) and pressing Enter. The interpreter then does exactly what it did in the live shell, only now it reads its instructions from the file instead of from a person typing in real time.

Both habits stay useful for as long as anyone writes Python: the shell for trying an idea out immediately, the script for anything meant to last. What either one actually does, though, is still an open question at this point, because nothing has been given to the interpreter yet except arithmetic a pocket calculator could have handled. The next chapter changes that, by giving the interpreter something new to hold onto: a name for a piece of data, and everything that follows from being able to name things at all.

KEY IDEAS

- Installing Python puts two things on a machine: the interpreter itself, and a large, pre-written standard library that ships alongside it.
- The interactive shell (REPL) runs one line at a time and remembers nothing once closed — good for trying an idea immediately.
- A script is a saved .py file the interpreter reads start to finish in one pass — good for anything meant to run more than once.
- Running a script from the command line means typing something like python filename.py and letting the interpreter read the whole file.

CHAPTER SIX

Naming and Holding Data

A VARIABLE IS NOT A BOX, though it is almost always taught as one. The box metaphor suggests a container: some fixed slot in memory with a label on it, holding a value the way a shoebox holds whatever gets put inside. It is a comforting picture, and it is not quite right. A variable is closer to a name tag placed on something that already exists — a sticky label reading age stuck onto the number 25, which was going to exist as a piece of data in the computer's memory whether or not anyone chose to call it anything at all. The difference sounds pedantic. It is not, and later chapters will occasionally depend on getting it right.

Giving something a name uses a single symbol, and Python calls the act itself assignment:

```
age = 25
```

read from right to left as much as left to right: first the value 25 comes into being, and only then does the name age get attached to it. Once this line has run, writing age anywhere else in the program means exactly what writing 25 would have meant — the interpreter looks up what age currently points to and uses that, silently, every time the name appears.

A name can also be moved. Nothing stops the same tag from being peeled off one value and stuck onto another:

```
age = 25
age = 26
```

A variable is closer to a name tag placed on something that already exists.

which is what the word “variable” actually refers to — not that some underlying value changes shape, but that which value a name currently points to can change over time. The number 25 has not been edited into a 26; it has simply been set back down, unlabeled, while age walks over and attaches itself to a different number entirely. Once no name points to a piece of data anymore, Python quietly clears it out of memory to make room for other things — a piece of routine housekeeping the reader will never have to manage by hand.

Not every piece of data is the same kind of thing, and the interpreter needs to know which kind it is dealing with before it can decide what operations even make sense. Adding two numbers together is straightforward; “adding” two pieces of text means something else entirely — joining them — and adding a number to a photograph does not mean anything at all. Python sorts data into types, and four of the most common ones can be seen simply by writing them down:

```
age = 25           # int — a whole number
price = 19.99      # float — has a decimal point
name = "Alice"     # str — text in quotes
is_open = True     # bool — True or False
```

Python decides which type a value belongs to from nothing more than how it is written down, with no separate declaration required anywhere. A number written with no quotation marks and no decimal point is an int; the same number with a decimal point added is a float instead, even if that decimal part is zero. Anything wrapped in quotation marks is a str — text — regardless of what is inside the quotes, which matters more than it sounds like it should:

Typed at the interactive prompt

```
>>> type(25)
<class 'int'>
>>> type("25")
<class 'str'>
```

Both of those are called 25, and a person glancing at them might not think twice about the difference. The interpreter, asked directly with the built-in `type()` function, is not fooled: the first is a number that could be used in arithmetic, and the second is two characters of text that, as far as addition and subtraction are concerned, might as well be the word "hello." The quotation marks are not decoration. They are the entire difference.

This is also a good place to notice what Python is not asking for, compared with the assembly and C examples from the previous chapter. Neither of those languages would accept `age = 25` on its own — C, in particular, requires the type to be declared up front, as in `int age = 25;`, before the name can be used at all. Python infers the type from the value instead, and does not even permanently bind a name to one type: the same name age could, later in the same program, be reassigned to a string or a float with no complaint from the interpreter. That flexibility is convenient, and it is not free — a name that quietly changes what kind of thing it refers to partway through a program is a common source of confusing bugs, and disciplined Python code tends to avoid doing so without a good reason.

One further value is worth introducing here, precisely because it represents the absence of one. Python has a special, singular value called `None`, used whenever there is deliberately nothing to point to — not zero, not an empty string, not False, but a distinct marker meaning that no meaningful value applies yet. A later chapter on functions will lean on it heavily, for exactly the cases where a function has nothing useful to hand back.

Names and values on their own do not accomplish very much, though — every example so far has been a single, static fact handed to the interpreter and left alone. The next chapter is about combining values into something more than the sum of their parts: expressions, the operators that build them, and the rules the interpreter follows to decide what to compute first.

KEY IDEAS

- A variable is a name attached to a value, not a container holding it — reassignment moves the name, not the underlying data.
- Python decides a value's type from how it is written: quotation marks make it a str regardless of content; a decimal point makes it a float instead of an int.
- The built-in type() function asks the interpreter directly what type a given value is.
- None is a distinct value meaning "nothing here" — not zero, not empty text, not False.

What Comes Next

This installment carries Part Two two chapters deep — enough to install Python, understand the shell-versus-script distinction, and give the interpreter its first real vocabulary: names, values, and the handful of basic types every later chapter builds on. Seven chapters of Part Two remain: combining values into real expressions, branching with conditionals, repeating work with loops, the collection types that hold more than one value at a time, strings treated as sequences rather than solid blocks, functions as the first real tool for abstraction, and learning to read an error instead of fearing it.

Beyond that, Part Three turns to larger programs: organizing code across multiple files, saving information to disk so a program can remember something after it stops running, and modeling more complicated ideas using classes and objects. Part Four steps back from syntax entirely, to the habits of mind that separate someone who can write a program from someone who can reliably find and fix the one that is broken, and break a large, intimidating problem into pieces small enough to solve one at a time. Part Five closes the book by building several complete, small programs from everything that came before, and by pointing toward where the study of Python usually leads next — web development, data and artificial intelligence, and the automation of ordinary, tedious work.

All of that is still ahead. What exists so far — four chapters of foundations and two of real Python — already stands on its own; everything still to come simply keeps building on the same ground.