

THE

AUTOMATION YEAR

Build dependable workflow automations
and turn them into client-ready systems.

PYTHON / APIs / EVALUATION / RETRIEVAL / AGENTS / SAFETY



INCLUDES THE KINETIVY FIELD GUIDE TO WINNING
WORKFLOW AUTOMATION PROJECTS

ANUSTUP MUKHERJEE

REVISED EDITION / BOOK ONE OF THREE

The Automation Year

BOOK ONE OF THREE

The Automation Year

Python, APIs, workflow automation, language models,
evaluation, retrieval, agents and safety

Anustup Mukherjee

REVISED EDITION
September 2026

The Automation Year
Book One of Three
Revised edition, September 2026

Copyright © 2026 Anustup Mukherjee. All rights reserved.

Examples, sample companies and numerical scenarios are instructional unless identified otherwise. Product names belong to their respective owners. No endorsement by the named vendors is implied.

Code listings teach design patterns and require adaptation, testing and current documentation before use with client systems. Legal, tax and commercial examples require advice appropriate to the engagement and jurisdiction.

Typeset for a 210 × 297 mm, monochrome, no-bleed interior. Edition references and web checks are recorded in the back matter.

Contents

Before you begin	xii
Python you own	1
1.1 What a program is, and what Python does with it	2
1.2 Values and types: every piece of data has a kind	6
1.3 Variables are names, not boxes	8
1.4 Making decisions	9
1.5 Repeating work with loops	11
1.6 Functions: named, reusable recipes	12
1.7 Collections: lists, dictionaries, tuples and sets	14
1.8 Reading files, and building this chapter's artifact	19
1.9 Testing: proving the script does what you say it does	24
1.10 Reading a traceback	29
1.11 Git: save points, parallel versions and undo	32
Chapter 1 in one page	37
Practice	38
Where to go deeper	48
APIs and the shape of integration	49
How long this chapter takes	49
2.1 What an API is	50
2.2 HTTP: the language every web API speaks	51
2.3 JSON: the shape of the data coming back	56
2.4 REST, and how to read documentation you have never seen	58
2.5 Authentication: proving who you are	59

2.6 Making requests from Python	64
2.7 Pagination: getting all of the results	65
2.8 Rate limits	67
2.9 Retries with exponential backoff and jitter	69
2.10 Idempotency: making repeats safe	75
2.11 Webhooks: being told instead of asking	77
2.12 Building the artifact: pull_insights.py	83
Chapter 2 in one page	86
Practice	87
Where to go deeper	96
Automation platforms and self-hosting	97
3.1 What a workflow platform is	98
3.2 How data flows in n8n	99
3.3 The core nodes	101
3.4 Self-hosting n8n on your own VPS	103
3.5 Credentials and secrets hygiene	112
3.6 Scheduling and triggers	113
3.7 Error handling: the part clients pay for	114
3.8 Queues and scaling	117
3.9 Make and Zapier: enough to talk to their users	118
3.10 Building the artifact: the reporting workflow	120
Practice	122
LLM mechanics	133
4.1 What a language model actually does	134
4.2 Tokens: what actually costs money	137
4.3 How generation works: probabilities, temperature and sampling	142

4.4 Messages, roles and the system prompt	144
4.5 Structured output: a schema that never breaks downstream	146
4.6 Reaching a model: SDKs, gateways and lock-in	152
4.7 Embeddings: meaning as coordinates	157
4.8 Prompt caching and its breakeven point	160
4.9 Building the artifact: the lead classifier workflow	162
Practice	165
Evaluation, the skill that gates everything	179
5.1 Why evaluation gates everything	180
5.2 Start by looking at the data	181
5.3 Defining a metric precisely	184
5.4 Does the confidence number mean anything?	186
5.5 Putting a price on each error	190
5.6 Building a test set, and justifying it	193
5.7 Freshness and drift	196
5.8 A failure taxonomy for LLM outputs	197
5.9 LLM-as-judge: when you cannot check with code	200
5.10 Regression testing when prompts change	200
5.11 Cost per successful outcome	203
5.12 Building the artifact: the evaluation suite	206
Practice	211
AI-powered automation at scale	225
6.1 From one call to a pipeline	226
6.2 Getting text out of documents	229
6.3 Extraction: unstructured documents into structured rows	233
6.4 Confidence scoring	235

6.5 Routing and human escalation	238
6.6 Summarisation with quality controls	240
6.7 Routing, triage and cascades	245
6.8 Content pipelines with human review gates	247
6.9 Cost control	248
6.10 When not to use an LLM	251
6.11 Building the artifact: the document pipeline	255
Practice	257
Data and reliability engineering	269
7.1 Why workflows need memory	270
7.2 Databases and PostgreSQL	271
7.3 Schema design for a reporting workflow	274
7.4 Changing the schema on a live system	278
7.5 SQL for real questions	281
7.6 Indexes	288
7.7 Idempotency in the database	291
7.8 Queues and background jobs	294
7.9 Backups, and the restore you have actually run	298
7.10 Observability: seeing inside the system	302
7.11 Cost monitoring and runaway protection	308
7.12 Data protection: the DPDP Act as a working checklist	311
7.13 Building the artifact	321
Practice	322
Retrieval and RAG	337
8.1 Why retrieval	338
8.2 Before you reach for a vector store	341

8.3 The indexing pipeline	347
8.4 Chunking, and the three ways it goes wrong	348
8.5 Vector stores and indexing	353
8.6 Keyword search and BM25	357
8.7 Hybrid search and reciprocal rank fusion	361
8.8 Measuring retrieval	364
8.9 Reranking, and whether it earns its latency	368
8.10 Context assembly within a prompt budget	370
8.11 Changing the embedding model: migration and re-indexing	372
8.12 Building the artifact: a research assistant with measured recall	377
Practice	380
Productising	401
9.1 From gigs to products	402
9.2 The three offers	403
9.3 Pricing: the floor, the ceiling, and the number between them	406
9.4 When value pricing is the wrong approach	411
9.5 The discovery call	417
9.6 Scoping a fixed outcome	419
9.7 Statements of work and acceptance criteria	420
9.8 Service levels and what happens at 06:10	422
9.9 The commercial mechanics: contracts, and how the money legally arrives	426
9.10 Handover, and the retainer that follows it	435
9.11 Templates: getting faster every time	436
9.12 Building the artifact	438
Practice	439

Agentic systems **453**

10.1 What an agent is, and what it is not	454
10.2 Five workflow patterns	457
10.3 Tools and function calling	460
10.4 The loop, in code — and how to test it without a model	467
10.5 Multi-step loops and where they diverge	476
10.6 State and context across steps	479
10.7 Failure recovery and compensating actions	482
10.8 When an agent is the wrong architecture	488
10.9 Evaluating agents	491
10.10 Building the artifact: an anomaly investigator	494
Practice	497

MCP and safety **513**

11.1 Why a protocol	514
11.2 Architecture: hosts, clients and servers	516
11.3 Messages and lifecycle	517
11.4 Building a server	521
11.5 The OWASP LLM risk list	524
11.6 Prompt injection	529
11.7 Guardrails for irreversible actions	535
11.8 Supply-chain hygiene	548
11.9 Ordinary server hardening	555
11.10 Building the artifact	558
Practice	560

Consolidation and positioning	587
12.1 Harvest time	588
12.2 Writing up an artifact	590
12.3 Product requirements documents for AI features	591
12.4 Unit economics: a cost-per-call model for a real product	597
12.5 The model nobody builds: what your own practice costs	604
12.6 The public case study: Indian growth measurement	611
12.7 A profile you can defend	614
12.8 Applications: the second path	616
12.9 The honest claim, and what Book 2 adds	617
Practice	620
Winning workflow automation projects	633
Kinetivy field guide: turn working workflows into paid projects	633
Version check register	641
Documentation and verification sources	649
Subject index	653

Before you begin

This book covers twelve stages of the plan: a foundation for becoming useful to clients and developing paid work. By the last chapter you should be able to sit across from a technical buyer and defend every claim on an AI automation profile, because you will have built, broken and repaired every system those claims describe.

The title gives the plan its shape, not a deadline. Work through twelve stages in order and move on when the checkpoint holds. The chapter estimates add up to roughly 745–770 hours. At eleven hours a week, allow about 68–70 study weeks before breaks; a twelve-month route needs closer to fifteen hours a week and some spare capacity for access, debugging and revision. Chapters 2, 7, 8 and 11 need especially generous space. Schedule client work separately from protected study time.

It is written for someone who has spent a decade inside marketing operations and knows exactly where the money goes, but whose Python is still at beginner level. So every idea arrives in the same order: the plain-words version first, then the official name, then the mechanics, then a worked example on marketing data, then the mistake that costs real money, then practice.

The plain-words box is not a simplification you grow out of. It is the version you will use in client conversations. The technical version is what you need to build the thing and to read the documentation. You need both, and most books give you only one.

	What it teaches	What you can defend afterwards
Part I · Ch 1-3	Python, APIs, self-hosted automation	"I build automations that survive a failed run at 3am"
Part II · Ch 4-6	Model mechanics, evaluation, scale	"I can tell you my error rate and what it costs"
Part III · Ch 7-9	Data, retrieval, priced offers	"Every number I report is queryable and reconciles"
Part IV · Ch 10-12	Agents, safety, positioning	"I can stop an agent before it does something irreversible"

Figure 0.1 Four parts. Each one ends with a sentence you can say to a technical buyer and then prove.

How this edition differs

This is a revised edition, and the revisions came from a review of the first one. Three findings drove almost all of the changes.

The first was that roughly two thirds of the exercises had no answers, and the unanswered ones were systematically the harder ones. For a reader studying alone, with a rule that says the code block must be unaided, an unanswered exercise is a dead end — the one legitimate way to check your work had been removed. **The exercises include worked answers or marking guidance, including open-ended builds.** Where an exercise can be made self-verifying instead — "check it against the library's own version", "assert the totals reconcile" — it has been, because an exercise that marks itself beats an answer key.

The second was that the book defended itself well against facts going out of date and badly against choices going out of date. It was careful never to quote a model name or a token price, and then it picked one tool at every layer and presented the choice as settled. A dead model name is a five-second fix. A single blessed tool with no alternative and no switching criteria is a dead end. So this edition adds an **alternatives block** wherever the book picks a tool: what else exists, why this one, and the specific conditions under which you should switch.

The third was that the one-chapter-one-month framing was not real. Chapters 2, 7, 8 and 11 are each closer to two months at eleven hours a week. Pretending otherwise means falling behind in month two and feeling like the failure is yours. **Every chapter now states an honest hour estimate** in its opening box.

Revision area	What to use
Practice	Worked answers and marking guidance; self-checking exercises wherever the result is objectively testable.
Technical accuracy	Corrected code contracts, arithmetic, statistical interpretation and safety claims.
Tool choices	Alternatives and switching criteria, supported by chapter shelf-life notes.
Delivery	Testing, backup restoration, migrations, service operations and commercial handover.
Finding projects	A Kinetivy field guide connecting workflow demonstrations to qualification, paid pilots and proposals.

Figure 0.2 What the revised edition adds to the original learning sequence.

How the book maps to your week

	When	Length	What to use from this book
Recall drill	Every weekday, first	10 min	Tier A questions, plus any Key term box you cannot define aloud
Code block	Every weekday, middle	35 min	Tier B tasks. Unaided: no model, no autocompletable beyond the editor's own
Concept block	Every weekday, last	15 min	Tier C prompts. Explain it out loud to the named sceptic
Weekend build	Saturday or Sunday	5-6 hrs	Tier D, one increment of the month's artifact

Figure 0.3 The weekly rhythm this book is written against: about eleven hours.

A sensible pace is one or two sections per weekday session in weeks one and two, then practice-heavy sessions in weeks three and four. Solutions sit at the end of each chapter. Look at them only after an honest attempt; the attempt is where the learning happens.

WATCH OUT

The one rule The 35-minute code block is unaided. Vibe-coding during study hours produces shipped output and flat understanding. Every Tier B exercise assumes you are typing it yourself, reading your own error messages and fixing them yourself.

This rule gets more important as models get better, not less. The skills in this book are exactly the skills that let you judge a model's output instead of trusting it, and you cannot build the judgment you use to check the machine by asking the machine.

Conventions

Shelf life. Every chapter opens with a box saying what in it is permanent, what to re-verify before relying on it, and where to check. Read it first and again when you return to the chapter.

The [v] marker. A [v] after a claim means it is version-coupled: a flag name, a menu path, a parameter, a default. It was true when this was written and is exactly the kind of thing that moves. The consolidated list at the back collects every one of them.

Code. Python 3.12 or later unless stated [v]. Lines are kept short enough to type comfortably. When a line in a listing is a comment starting with #, read it — comments carry half of the explanation.

Models. No specific model is ever named. Code uses a MODEL constant you set once. Named models and their capabilities can change; verify availability and record the version you actually use.

Numbers. Rupee amounts and audience figures use lakhs and crores. Where a number is invented to make the arithmetic clear it is labelled illustrative. Prices are given as a formula plus a dated example, never as a fact — a cost model is a formula plus current numbers, and only the formula is durable.

Reading lists. Use the current edition of each named resource, and record its edition or revision date when you rely on it. Every recommendation names two alternatives and the search terms to find more.

Python you own

Enough Python to read code you have never seen, predict what it will do, and write a script a client can rely on.

SHELF LIFE

Permanent. Almost all of it. Sequence, types, variables, conditionals, loops, functions, collections, files, tests and version control work the same way in five years as they do today, and mostly the same way in every other language you will ever touch. The habits — return rather than print, catch the specific error, commit early — outlast every tool in this book. **Re-verify before you rely on it.** The Python version this book assumes, how you install Python and make an environment, the exact pytest command-line flags, and `git switch`, which is newer than `git checkout` and missing from older installations. **Where to check.** docs.python.org for the language and the standard library, docs.pytest.org for testing, git-scm.com/docs for Git. All three are written by the people who ship the thing, are free, and are the version of the truth your machine is actually running.

Python is not the goal of this year. It is the ground everything else stands on. Every automation platform you will sell has a "run this code" node in it, every API client you write is Python, and every time a workflow behaves strangely at 11 p.m. on a Friday, the difference between a ten-minute fix and a lost client is whether you can read the code in front of you.

By the end of this chapter you can

Explain what the Python interpreter does with a file, step by step, and say which mistakes it catches before running and which it cannot.

Choose the right type for campaign data and convert between types without silently corrupting a number.

Write functions that return values rather than print them, and explain to a non-programmer why that matters.

Use lists, dictionaries and comprehensions to group and summarise rows.

Write a test with `assert`, run a file of tests with `pytest`, and tell a useful test from one that cannot fail.

Read a traceback from the bottom up and locate the real error.

Use Git to branch, compare, undo and recover work.

This chapter's artifact. A command-line script that reads a CSV of campaign data and prints spend, conversions and cost per acquisition by channel, with a small suite of tests beside it. Written entirely by hand.

The checkpoint. Take 40 lines of Python you have never seen. Predict the output on paper, then run it. Fewer than three correct out of five attempts means you repeat the chapter rather than moving on.

How long this takes. At eleven hours a week, budget four weeks — roughly 44 hours — and expect the second half of the chapter to be slower than the first. Sections 1.1 to 1.6 are readable in a week. Sections 1.7 to 1.9, where collections, files and tests come together into a working script, are where the time actually goes. This is one of only three chapters in the book that genuinely fits inside a month; several later ones do not, and the book will say so when they arrive.

1.1 What a program is, and what Python does with it

IN PLAIN WORDS

Imagine a kitchen assistant who is extremely fast, never gets tired and never argues, but who takes every instruction completely literally. If your recipe says "add salt" and you forgot to say how much, the assistant stops and tells you it cannot continue. If your recipe says "stir for 10 minutes" but you meant 10 seconds, the assistant stirs for 10 minutes without comment. A program is that recipe. Python is that assistant. It starts at the top of your file, does exactly what each line says, in order, and either finishes or stops at the first line it cannot carry out. It never guesses what you meant.

That last sentence explains most of the frustration beginners feel. Python does not do what you meant. It does what you wrote. Learning to program is mostly learning to write what you mean with no gaps.

KEY TERM

Interpreter The program that reads your Python file and carries out its instructions one statement at a time. When you type `python3 summary.py`, you are asking the interpreter to run the file called `summary.py`.

KEY TERM

Script A plain text file of Python instructions, saved with the `.py` extension, meant to be run from start to finish. The artifact for this chapter is a script.

KEY TERM

REPL Short for read, evaluate, print, loop. The interactive prompt you get when you type `python3` on its own. It shows `>>>`, waits for one line, runs it, shows the result, and waits again. Ideal for quick experiments, useless for anything you want to keep.

What actually happens when you run a file

Four things happen between pressing Enter and seeing output. The interpreter reads the whole file; it checks the grammar of the whole file; it translates the file into an internal form called bytecode, which you never see and never need to think about; and only then does it start carrying out statements, top to bottom, one at a time.

The order matters more than it looks. Because grammar is checked for the whole file first, a missing bracket on line 40 means nothing on lines 1 to 39 runs either — you get a `SyntaxError` and an empty screen. But once running starts, every other kind of error is discovered only when the interpreter actually reaches the line that causes it. A mistake inside a function that nothing ever calls will sit there quietly for months. And a line that is grammatical, well-typed and simply wrong will never announce itself at all.

	When Python notices	How much of your file ran	Where it usually is
SyntaxError	Before the first line runs; the whole file is parsed first	Nothing at all, not even line 1	A missing bracket, colon or quote, often on the line ABOVE the one reported
IndentationError	Before the first line runs	Nothing at all	A block not lined up under its colon, or tabs mixed with spaces
NameError, TypeError, ValueError, KeyError	Only when that exact line is reached	Everything above it, including anything already written to a file	A typo or a bad value on a path no earlier run happened to take
No error at all, just a wrong number	Never	All of it, successfully	Correct grammar, correct types, wrong logic. Only a test or a reader finds this

Figure 1.1 Grammar errors stop the file before anything runs. Everything else waits for its own line to be reached, and a wrong answer never stops anything at all.

That last row of the figure is the reason Section 1.9 exists. Python protects you from grammar, and partially from types. Nothing in the language protects you from a report that adds the wrong column. Only a test does.

WORKED EXAMPLE

1.1 Your first script, and the REPL Create a file called `hello.py` containing these three lines:

```
brand = "Himalaya Baby Care"
spend = 125000
print(brand, "spent", spend, "rupees this week")
```

Run it from a terminal in the same folder:

```
$ python3 hello.py
Himalaya Baby Care spent 125000 rupees this week
```

Now type `python3` on its own to open the REPL, enter `spend = 125000`, and try `spend * 4`. The REPL prints `500000` without you writing `print`, because echoing the result is part of what the REPL does. A script does not do this. In a script, a line that just says `spend * 4` calculates the answer and throws it away. This difference catches almost everyone once.

WATCH OUT

Which Python, and where your packages land The old advice was that `python` means Python 2. It does not, any more. Python 2 has been end of life since 2020, and on a current Mac or most Linux distributions a bare `python` command does not exist at all — only `python3`. On Windows the launcher is `py`. So the modern version of this problem is not "wrong major version", it is **too many Pythons**: the one your operating system ships, one from Homebrew or your package manager, one from a `python.org` installer, and one inside every project folder. When `pip` installs a package that `python3` cannot find, mismatched environments are a common cause.

Two commands settle it. `python3 --version [v]` tells you which version you are about to run; which `python3 (` where `python` on Windows) tells you which file that actually is. This book assumes Python 3.12 or later [v].

Then, one rule, from your very first project: never install packages into the system Python. Make a virtual environment per project — a private folder holding that project's own Python and its own packages:

```
python3 -m venv .venv          # create it, once per project
source .venv/bin/activate      # macOS and Linux
.venv\Scripts\activate.bat     # Windows Command Prompt
# PowerShell: .venv\Scripts\Activate.ps1
python3 -m pip install pytest  # now this is project-local
```

Add `.venv/` to your `.gitignore` in Section 1.11. Environments are rebuildable; they do not belong in version control.

ALTERNATIVES, AND WHEN TO SWITCH

Why plain venv here, and when something else wins The alternatives. venv plus pip (in the standard library, nothing to install), uv [v] (a much faster installer and environment manager that also installs Python versions for you), pyenv for juggling several Python versions, Poetry or PDM for library packaging and lock files, and Conda where you need non-Python scientific libraries. **Why this book picks venv and pip .** They ship with Python, so they are on every machine you will ever be asked to debug on, including a client's server at 11 p.m. Every alternative is a thing that has to be installed first, and the failure mode of "my tool is not installed here" is exactly the failure mode you are learning to avoid. **When to switch.** Move to uv the moment environment creation and installs become a noticeable part of your day, or when you start rebuilding environments in CI where seconds cost money. Move to Poetry or PDM when you start publishing a package rather than running scripts. Move to pyenv when a client pins you to a Python version your machine does not have. **What stays true either way.** One isolated environment per project, its contents reproducible from a file you commit, and the environment folder itself never committed.

1.2 Values and types: every piece of data has a kind

IN PLAIN WORDS

In a spreadsheet, a cell showing 1,200 might be a number or it might be text that happens to look like a number. You find out which when you try to sum the column and get zero. The cell's format, not its appearance, decides what you can do with it. Python works the same way. Every value has a kind, and the kind decides which operations make sense. You can add two numbers. You can join two pieces of text. You cannot add a number to a piece of text, and Python refuses rather than guessing.

KEY TERM

Type The kind of a value, which determines what operations are allowed on it. You can ask any value its type with the built-in `type()` function: `type(42)` gives `int`, `type("42")` gives `str`.

The five types you will use every single day:

Table 1.1 Core Python types and their uses in campaign data

TYPE	PLAIN MEANING	EXAMPLES	TYPICAL USE IN CAMPAIGN DATA
int	Whole number	0 , 42 , -7 , 1_00_000	Impressions, clicks, conversions
float	Number with a decimal point	0.5 , 3.14 , 1250.75	Spend, CTR, CPA, ROAS
str	Text, in quotes	"Meta" , '2026-10-01'	Channel names, campaign names, dates as read from a file
bool	Yes or no	True , False	Is this ad set active? Did the row pass validation?
NoneType	Nothing, deliberately	None	A CPA that cannot be calculated because there were zero conversions

Notice `1_00_000` in that table. Python lets you put underscores inside numbers purely for readability, and ignores them, so you can group digits the Indian way in your own code. The value is exactly one lakh.

Converting between types

With the CSV reader used here, field values arrive as text. A CSV cell containing 1250 is the string `"1250"`, not the number 1,250. Before you can do arithmetic you must convert it, using the type's name as a function:

```
int("42")          # 42
float("1250.75")   # 1250.75
str(42)            # "42"
int(3.99)          # 3   (cuts off the decimal, does not round)
round(3.99)        # 4
bool("")           # False (empty text counts as false)
bool("0")          # True  (any non-empty text counts as true, even "0")
```

The conversions fail loudly when the text is not a clean number. `float("1,250")` raises a `ValueError` — but the reason is worth getting right, because the usual explanation is wrong. It is not that Python cannot tell a thousands separator from a decimal point and refuses to guess. `float` has no concept of grouping at all. For ordinary decimal numbers it accepts an optional sign, digits, one decimal point, an optional exponent and surrounding whitespace. It also accepts special values such as `nan` and `inf`. To `float`, a comma is simply a character it cannot parse, exactly like a letter; `float("1 250")` fails for the same reason `float("1x250")` does.

That distinction has a sharp edge. A European-formatted export writes one thousand two hundred and fifty as `1.250`, and `float("1.250")` does **not** fail. It succeeds and gives you 1.25. The failure you can see is the comma; the failure you cannot see is the dot, and it understates the number by a factor of a thousand. Underscores, oddly, are accepted: `float("1_250")` gives 1250.0, because that is Python's own literal syntax. Real exports from ad platforms and Excel are full of commas, currency symbols, blank cells and dashes meaning zero, so cleaning text before converting it is a skill you will use every week.

WORKED EXAMPLE

1.2 Cleaning a rupee amount from an export A finance export gives you spend as text like `"₹1,23,456.50"`, and sometimes as `"-"` when there was no spend. This function turns any of these into a float:

```
def parse_number(text):
    """Turn an exported amount into a float. '-' and '' mean zero."""
    cleaned = text.replace("₹", "").replace(",", "").strip()
    if cleaned in ("", "-"):
        return 0.0
    return float(cleaned)

parse_number("₹1,23,456.50") # 123456.5
parse_number(" 4,500 ")     # 4500.0
parse_number("-")           # 0.0
```

Read it line by line. `replace` swaps one piece of text for another and returns a new string; chaining two of them removes both the symbol and every comma. `strip` removes spaces from both ends. The `if` catches the two ways an export says "nothing". Only then is `float` asked to convert, and by that point the text is clean.

Note what this function deliberately does not do: it does not turn unparseable text into zero. `parse_number("42000 INR")` still raises. Silently turning an unreadable value into 0.0 would understate spend without telling anyone, which is the worst kind of bug in a client deliverable — one that produces a plausible number. Section 1.9 tests exactly this behaviour, and Exercise B7 makes the script count and report the rows it could not read.

WATCH OUT

Floats and money Floats are stored in binary, and some decimal fractions cannot be represented exactly. Type `0.1 + 0.2` into the REPL and you get `0.30000000000000004`. For reporting, round at the end with `round(value, 2)`. For anything that must reconcile to the paisa, such as an invoice, use the `decimal` module or store amounts as whole paise in an `int`. When rounding error is possible, compare floats with a deliberate tolerance rather than exact equality. Section 1.9 shows the one-line way to do that in a test.

1.3 Variables are names, not boxes

IN PLAIN WORDS

Many courses describe a variable as a box you put a value into. A better picture is a sticky note. The value exists on its own; the variable is a label you have stuck onto it so you can find it again. You can move the sticky note to a different value at any time. You can also stick two notes on the same value, and this is where surprises come from.

KEY TERM

Variable and assignment A variable is a name that refers to a value. Assignment, written with a single equals sign, attaches the name to the value: `spend = 125000` means "from now on, the name `spend` refers to 125000". It is an instruction, not a statement of equality.

```
spend = 125000
spend = spend + 5000      # right side first (130000), then move the label

channels = ["Meta", "Google"]
backup = channels          # a second label on the SAME list
backup.append("YouTube")
print(channels)            # ['Meta', 'Google', 'YouTube'] (surprise!)
```

The second line reads strangely as mathematics but perfectly as an instruction: compute the right-hand side using the current value, then point the name at the result.

The last four lines show the sticky-note effect. `backup = channels` does not copy the list. It puts a second label on the same list, so changing it through one name changes what you see through the other. When you genuinely want a copy, say so: `backup = channels.copy()`. Numbers and strings cannot be changed in place, so this effect never bites with them; it only matters with things that can be modified, like lists and dictionaries. You will meet it again in Section 1.6, where it causes one of the classic bugs in Python functions.

Name variables for what they hold, in lowercase words joined by underscores: `total_spend`, `conversions_by_channel`, `cpa_target`. This style is called snake case and it is the Python convention. A reader should never need to guess what `x` or `temp2` contains.

1.4 Making decisions

IN PLAIN WORDS

If you have ever set an automated rule in Ads Manager — "if cost per result is above ₹400 for three days, pause the ad set" — you have already written a conditional. Python's `if` statement is the same idea. It checks a condition that is either true or false, and runs a block of instructions only when the condition is true.

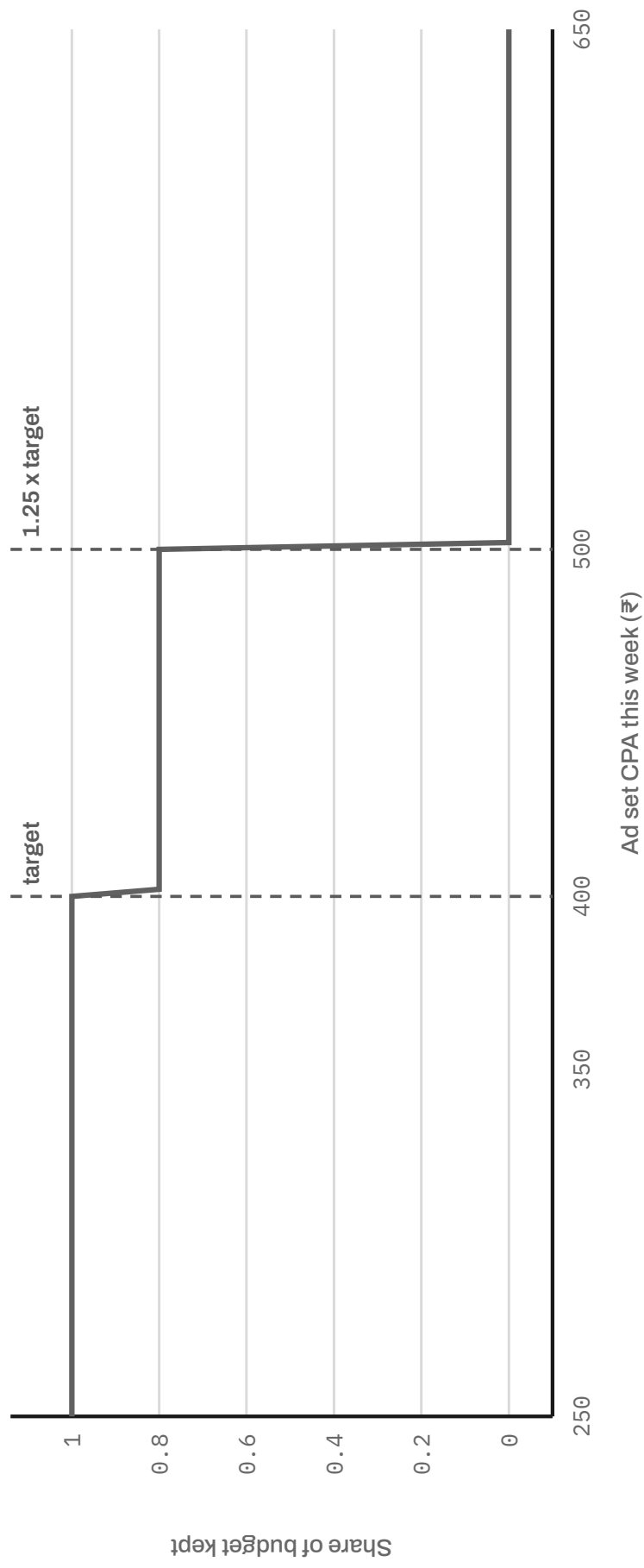
```
cpa = 460
target = 400

if cpa > target * 1.25:
    action = "pause"
elif cpa > target:
    action = "reduce budget"
else:
    action = "keep"

print(action)  # reduce budget
```

Three things to notice. The colon at the end of each condition line is required. The indented lines underneath (four spaces, by convention) are the block that belongs to that condition; indentation is not decoration in Python, it is grammar. And the branches are checked top to bottom, stopping at the first one that is true, so order matters.

That last point is worth seeing rather than being told. Drawing the rule across the whole range of possible CPAs shows why: the "pause" band is entirely inside the "above target" band. The two conditions overlap, and the only thing keeping the pause branch reachable is that it is written first.



The pause band sits inside the above-target band, which is why the elif order decides the answer.

Figure 1.2 The three branches of the CPA rule drawn as one step. Budget drops to 80% above the target and to zero above 1.25x the target. Target of ₹400 is illustrative.

Conditions are built from comparisons (`==` equal, `!=` not equal, `<`, `<=`, `>`, `>=`) and combined with the words `and`, `or` and `not`:

```
if channel == "Meta" and spend > 50000 and not is_paused:
    print("Large active Meta line, check frequency")
```

KEY TERM

Truthiness Python treats some values as false when used in a condition even though they are not literally False: the number 0, the empty string "", the empty list [], the empty dictionary {}, and None. Everything else counts as true. So `if rows:` means "if the list of rows is not empty".

WATCH OUT

Three classic mistakes One equals sign or two. `=` assigns, `==` compares. Writing `if spend = 0:` is a syntax error, which is lucky, because in some languages it silently does the wrong thing.

Comparing text that looks like numbers. `"10" > "9"` is False, because text is compared character by character and "1" comes before "9". Convert to numbers before comparing.

Testing for None. Write `if value is None:`, not `if not value:`. The second form also treats a CPA of zero, an empty channel name and an empty list as missing, which will one day hide a real zero.

1.5 Repeating work with loops

IN PLAIN WORDS

When you drag a formula down a spreadsheet column, you are asking the software to do the same calculation for every row. A loop is that drag, written out. You describe what to do with one item, and Python repeats it for every item in a collection.

```
spends = [42000, 38500, 51200, 12000]

total = 0
for s in spends:
    total = total + s

print(total)  # 143700
```

This is the accumulator pattern, and it is worth recognising on sight: create a variable before the loop (here `total = 0`), update it inside the loop, use it after the loop. Summing, counting, finding the maximum and grouping by channel are all versions of it. The two mistakes to watch for are creating the variable inside the loop, which resets it every time round, and using it inside the loop, which reports a partial answer.

`range` gives you a sequence of numbers to loop over. `range(5)` produces 0, 1, 2, 3, 4 — five numbers, starting at zero, stopping before five. `range(2, 8)` produces 2 through 7. When you need both the position and the item, use `enumerate`: