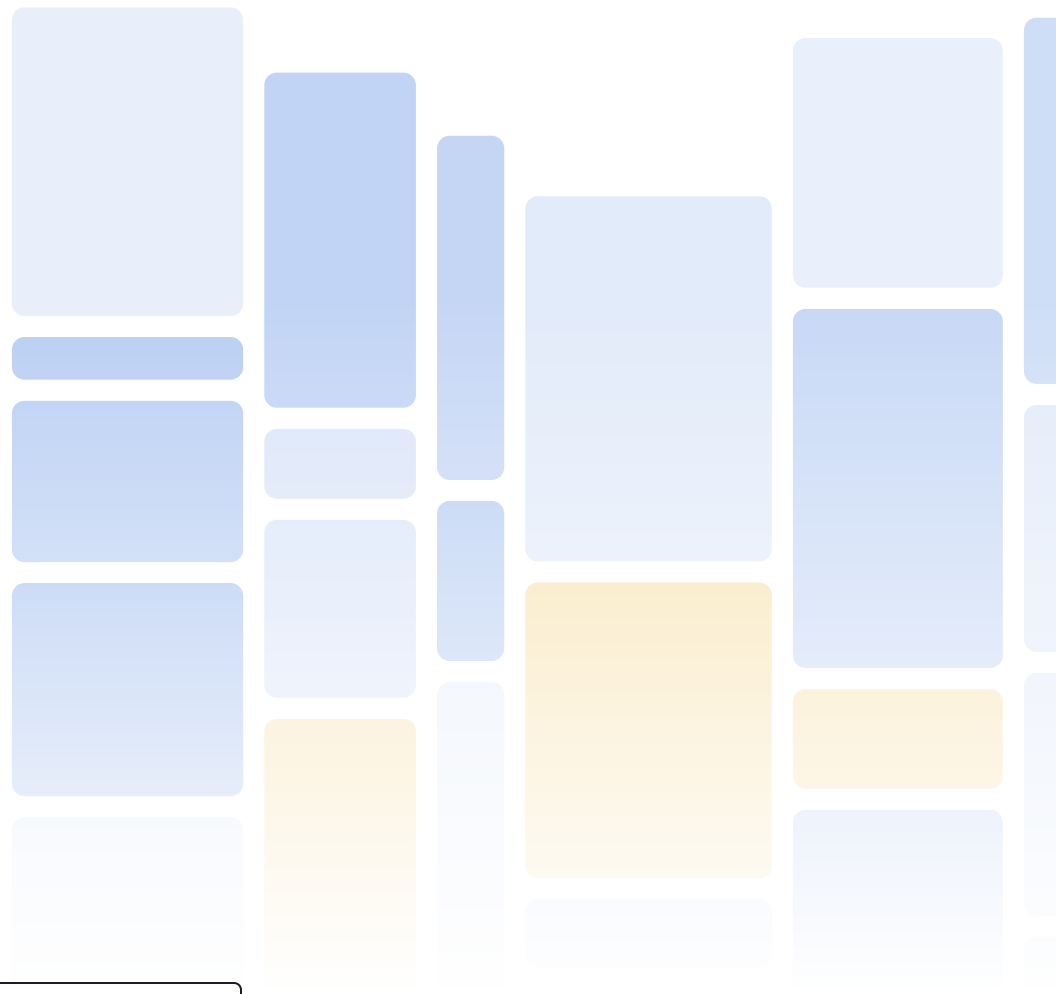




A BEGINNER'S MANUAL

GROK EDITION

Python for AI Beginners

From installing Python to scripts that call ChatGPT,
Claude, Gemini, Grok and local models, step by step

Python for AI Beginners

From installing Python to scripts that call ChatGPT, Claude, Gemini, Grok and local models, step by step

GROK EDITION

Written by Grok Bot, with facts checked against official sources in October 2026.

This manual teaches just enough Python to make AI useful in your own scripts. You'll install Python, learn the language one small idea at a time, manage packages and secret keys safely, and then build real projects. Those projects include a document summarizer, a spreadsheet categorizer, a terminal chatbot, and a batch processor that works with cloud AI APIs and with free local models.

Every code sample is short and complete. The beginner examples were run on Python 3.13 and 3.15 in October 2026, and the API examples use the current official SDKs. Libraries change, so each chapter tells you where to check the official documentation.

HOW TO READ THIS MANUAL

Tip — a shortcut or good habit.

Warning — easy to get wrong.

\$ `command` — text you type, exactly as shown.

Note — useful background.

Danger — can lose data, money, or access.

`Settings` > `Privacy` — a menu path to click.

New words appear in **bold** the first time and are collected in the Glossary at the back.

Contents

01	Why Python, and What You'll Build	6
1.1	What a program actually is	6
1.2	What you'll build	7
1.3	How to use this manual	7
02	Install Python	8
2.1	Windows	8
2.2	macOS	9
2.3	Linux	9
2.4	Alternative: let uv manage Python for you	9
2.5	Install a code editor	10
2.6	Terminal survival kit	10
03	Your First Programs	11
3.1	The REPL	11
3.2	Your first script	12
3.3	Reading an error message	12
04	Values, Variables, and Text	13
4.1	The core types	13
4.2	Converting types	14
4.3	Working with text	14
05	Lists, Dictionaries, and Loops	15
5.1	Lists	15
5.2	Dictionaries	16
5.3	Loops	16
5.4	Decisions with <code>if</code>	17
5.5	<code>while</code> loops	17
5.6	List comprehensions (a shortcut you'll see everywhere)	17
06	Functions, Modules, and Errors	18
6.1	Defining a function	18
6.2	Importing modules	19

6.3	Handling errors	19
07	Files, Folders, CSV, and JSON	21
7.1	Text files with pathlib	21
7.2	Every file in a folder	22
7.3	CSV spreadsheets	22
7.4	JSON	23
08	Virtual Environments and Packages	24
8.1	Why you need one	24
8.2	Create and use a venv	24
8.3	Packages used in this book	26
8.4	The same thing with uv	26
09	Keeping Secret Keys Secret	27
9.1	Get a key	27
9.2	Store keys in a .env file	27
9.3	Set a spending limit before you start	28
10	Calling AI from Python	29
10.1	OpenAI	30
10.2	Anthropic (Claude)	30
10.3	Google (Gemini)	31
10.4	xAI (Grok)	31
10.5	A free local model with Ollama	32
10.6	Streaming: show words as they arrive	32
10.7	One function, any provider	33
11	Getting Reliable Structured Output	35
11.1	Define the shape	35
11.2	OpenAI	36
11.3	Anthropic	36
11.4	Gemini	36
11.5	Local models (Ollama)	37
12	Working with Data in pandas	38
12.1	Load and look	38
12.2	Select, filter, sort	39
12.3	Clean common problems	39

12.4	Group and total	39
12.5	Add an AI column	39
13	Projects	41
13.1	Project 1: Folder summarizer	42
13.2	Project 2: Expense categorizer	42
13.3	Project 3: Terminal chatbot with memory and a cost meter	44
13.4	Project 4: Batch document Q&A to a spreadsheet	45
13.5	Project 5: Morning briefing from RSS feeds	46
14	Debugging, and Using AI to Help You Code	47
14.1	The errors you'll see most	48
14.2	Print debugging	48
14.3	The VS Code debugger	48
14.4	Asking AI for coding help	49
15	Troubleshooting and FAQ	50
	Quick Reference	51
	Glossary	53

Why Python, and What You'll Build

Python is the language most AI tools are built around. The official SDKs (software development kits) from OpenAI, Anthropic, Google and xAI all come in Python. So do the local-model tools Ollama and llama.cpp, and the data tools that turn messy files into tables. Python reads almost like English, which makes it the friendliest place to start.

IN THIS CHAPTER

- What Python is and what “running a script” means
- What you’ll be able to do by the end
- How to use this manual

1.1 What a program actually is

A **program** (or **script**) is a text file of instructions. The **Python interpreter** reads the file from top to bottom and carries out each instruction. Python files end in `.py`. You write them in a text editor and run them from a **terminal** (also called the command line or shell).

```
HELLO.PY
```

```
name = "Dylan"  
print(f"Hello, {name}! Python is working.")
```

Running `python hello.py` prints `Hello, Dylan! Python is working.` That’s the whole loop you’ll use for the rest of the book: **write, run, read the output, fix, run again.**

1.2 What you'll build

PROJECT	WHAT IT DOES	CHAPTER
Folder summarizer	Reads every <code>.txt</code> file in a folder and writes a one-paragraph AI summary of each	<i>Projects</i>
Expense categorizer	Reads a bank CSV and asks an AI to assign each line a category, then totals them	<i>Projects</i>
Terminal chatbot	A chat loop that remembers the conversation, with a cost counter	<i>Projects</i>
Local, private assistant	The same chatbot running on a free local model through Ollama, so nothing leaves your PC	<i>Calling AI from Python</i>
Batch document Q&A	Asks the same questions of 50 documents and saves the answers to a spreadsheet	<i>Projects</i>

1.3 How to use this manual

- 1 Type the examples yourself.**
Copying and pasting works, but typing teaches your fingers and your eyes. Typos are good practice at reading error messages.
- 2 Run every example.**
Don't move on until you've seen it work.
- 3 Change one thing.**
After each example works, change a value and predict the result before you run it.
- 4 Use AI as a tutor, not a crutch.**
When stuck, paste the *exact* error and your code into a chat assistant and ask "Explain what this error means and how to fix it." Then type the fix yourself.

TIP

You don't need to memorize anything. Professional programmers look things up all day. The *Quick Reference* at the back collects everything in this book on a few pages.

Install Python

You need Python 3.12 or newer for this book. As of October 2026 the newest release is **Python 3.15** (released October 9, 2026), and **3.14** and **3.13** are still fully supported. Any of them works. If a library you need doesn't support 3.15 yet, 3.14 is the safe choice.

IN THIS CHAPTER

- Installing on Windows, macOS and Linux
- Checking that it worked
- The two commands you'll use most: `python` and `pip`

2.1 Windows

1

Download.

Go to **python.org/downloads** and download the installer for the latest Python 3 release.

2

Tick the PATH box.

On the first installer screen, tick **Add python.exe to PATH**. This lets you type `python` in any terminal. It's the step people most often miss.

3

Install.

Click **Install Now** and wait for it to finish.

4

Open a terminal.

Press `Win`, type **Terminal** (or **PowerShell**), and open it.

5

Check.

Type `python --version` and press `Enter`. You should see something like `Python 3.15.0`.

NOTE

Windows also installs the **Python launcher**, `py`. If `python` opens the Microsoft Store or says “not found”, try `py --version`. You can use `py` instead of `python` everywhere in this book, e.g. `py hello.py`.

2.2 macOS

1 Download.

Go to python.org/downloads and download the macOS installer.

2 Install.

Open the `.pkg` file and click through the installer.

3 Open Terminal.

Press `Cmd` + `Space`, type **Terminal**, and press `Enter`.

4 Check.

Type `python3 --version`. On a Mac the command is usually `python3` rather than `python`, and `pip3` rather than `pip`.

2.3 Linux

Most Linux distributions ship with Python 3. Check with `python3 --version`. If it's older than 3.12, install a newer one with your package manager, or use **uv** (below), which can download any Python version for you.

2.4 Alternative: let uv manage Python for you

uv is a fast, free tool from Astral that installs Python versions, creates virtual environments and installs packages, all with one command. Many AI projects now use it. Install instructions are at docs.astral.sh/uv. After installing:

TERMINAL

```
uv python install 3.14      # download a Python version
uv init my-project          # create a project folder with a pyproject.toml
cd my-project
uv add requests             # install a package into the project's environment
uv run main.py              # run a script inside that environment
```

You can follow this whole book with either plain `python` + `pip` or with `uv`. Each chapter shows the plain version, and the *Packages* chapter shows the `uv` equivalents.

2.5 Install a code editor

You can write Python in Notepad, but a code editor colors your code, catches typos and runs scripts with one click.

1 Install VS Code.

It's free from **code.visualstudio.com**. Cursor and other VS Code-based editors work the same way.

2 Add the Python extension.

Open the Extensions panel (**Ctrl**/**Cmd** + **Shift** + **X**), search **Python**, and install the one published by Microsoft.

3 Open a folder.

Use **File** > **Open Folder** and make a new folder called `python-ai`. Keep every file from this book in it.

4 Open the built-in terminal.

Use **View** > **Terminal**. It opens in your folder, ready to run scripts.

2.6 Terminal survival kit

YOU WANT TO	WINDOWS (POWERSHELL)	MACOS / LINUX
See where you are	<code>pwd</code>	<code>pwd</code>
List files	<code>ls</code> (or <code>dir</code>)	<code>ls</code>
Go into a folder	<code>cd python-ai</code>	<code>cd python-ai</code>
Go up one folder	<code>cd ..</code>	<code>cd ..</code>
Make a folder	<code>mkdir projects</code>	<code>mkdir projects</code>
Run a script	<code>python hello.py</code>	<code>python3 hello.py</code>
Stop a running program	Ctrl + C	Ctrl + C
Repeat the last command	↑ then Enter	↑ then Enter

TIP

Press **Tab** while typing a file or folder name, and the terminal completes it for you. This prevents most “file not found” typos.

Your First Programs

Now you'll write and run code. Start in the **interactive shell** (the REPL), where Python runs each line as soon as you press Enter.

IN THIS CHAPTER

- The REPL: a calculator that speaks Python
- Running a `.py` file
- `print()`, `input()` and comments

3.1 The REPL

Type `python` (or `python3`) in the terminal and press Enter. You'll see a `>>>` prompt.

PYTHON

```
>>> 2 + 3
5
>>> 10 / 4
2.5
>>> "AI " * 3
'AI AI AI '
>>> exit()
```

The REPL is perfect for trying one small idea. For anything longer than a few lines, use a file.

3.2 Your first script

Create `first.py` in your folder:

FIRST.PY

```
# Lines starting with # are comments. Python ignores them.
print("What's your name?")
name = input("> ")
print(f"Nice to meet you, {name}.")
print(f"Your name has {len(name)} letters.")
```

Run it with `python first.py`. Type your name and press Enter.

PIECE	WHAT IT DOES
<code>print(...)</code>	Shows text on the screen
<code>input("> ")</code>	Shows a prompt, waits for you to type, and gives back what you typed as text
<code>name = ...</code>	Stores a value in a variable called <code>name</code>
<code>f"...{name}..."</code>	An f-string : anything in <code>{}</code> is replaced with its value
<code>len(name)</code>	The length of the text

3.3 Reading an error message

Change the last line to `print(f"Your name has {len(nam)} letters.")` (a typo) and run it again:

TEXT

```
Traceback (most recent call last):
  File "first.py", line 5, in <module>
    print(f"Your name has {len(nam)} letters.")
                                ^^^
NameError: name 'nam' is not defined. Did you mean: 'name'?
```

Read errors **from the bottom up**. The last line says *what* went wrong (`NameError`) and often suggests a fix. The line above it shows *where* (file, line 5). Modern Python is very good at pointing to the exact spot.

Values, Variables, and Text

Every value in Python has a **type**. Most of the time you'll work with the five types below.

IN THIS CHAPTER

- Numbers, text, true/false, and “nothing”
- Converting between types
- Working with text, the thing AI tools deal in most

4.1 The core types

TYPE	EXAMPLE	USED FOR
<code>int</code>	<code>42</code> , <code>-7</code>	Whole numbers, counts
<code>float</code>	<code>3.14</code> , <code>0.002</code>	Decimals, prices, scores
<code>str</code>	<code>"hello"</code> , <code>'hi'</code>	Text. Single or double quotes both work
<code>bool</code>	<code>True</code> , <code>False</code>	Yes/no answers
<code>None</code>	<code>None</code>	“No value yet”

PYTHON

```
price_per_million = 2.00          # float
tokens_used = 15_000              # int; underscores make big numbers readable
cost = tokens_used / 1_000_000 * price_per_million
print(f"Cost: ${cost:.4f}")      # :.4f = 4 decimal places → Cost: $0.0300
print(type(cost))                # <class 'float'>
```

4.2 Converting types

`input()` always gives you text. To do math with it, convert it:

PYTHON

```
age_text = input("Age? ")
age = int(age_text)          # "42" → 42
print(age + 1)

str(3.5)    # "3.5"
float("2")  # 2.0
int("abc")  # ValueError: invalid literal for int()
```

4.3 Working with text

AI work is mostly text work: building prompts, cleaning responses, and splitting documents into chunks.

PYTHON

```
text = " The Quick Brown Fox "
text.strip()          # "The Quick Brown Fox" (trim spaces)
text.lower()          # " the quick brown fox "
text.replace("Fox", "Dog") # swap words
"quick" in text.lower() # True
words = text.split()   # ['The', 'Quick', 'Brown', 'Fox']
"| ".join(words)       # "The | Quick | Brown | Fox"
text.strip()[:9]       # "The Quick" (first 9 characters)
```

Multi-line text uses triple quotes. This is perfect for prompts:

PYTHON

```
topic = "solar panels"
prompt = f"""You are a helpful explainer.
Explain {topic} to a 12-year-old in 3 short bullet points.
End with one question they could ask a teacher."""
print(prompt)
```

💡 TIP

Keep prompts in variables (or separate text files) instead of buried in your code. You'll tweak prompts far more often than code.

Lists, Dictionaries, and Loops

Real data comes in groups: a list of files, a table of expenses, a conversation of messages. Python's two workhorse containers are the **list** and the **dictionary**.

IN THIS CHAPTER

- Lists: ordered collections
- Dictionaries: named fields, the shape of almost every AI API message
- `for` loops, `if` decisions and `while` loops

5.1 Lists

PYTHON

```
models = ["small", "medium", "large"]
models[0]          # "small" – counting starts at 0
models[-1]         # "large" – negative counts from the end
models.append("xl") # add to the end
len(models)        # 4
"medium" in models  # True
models[1:3]         # ["medium", "large"] – a slice
sorted([3, 1, 2])   # [1, 2, 3]
```

5.2 Dictionaries

A **dictionary** (`dict`) stores values under names called **keys**. Every chat API sends messages as dictionaries, so get comfortable with them.

PYTHON

```
message = {"role": "user", "content": "Write a haiku about tea."}
message["role"]           # "user"
message["content"] = "Write a limerick instead."
message.get("model", "none") # "none" - .get() avoids errors for missing keys

conversation = [
    {"role": "system", "content": "You are concise."},
    {"role": "user", "content": "What is Python?"},
]
conversation.append({"role": "assistant", "content": "A programming language."})
print(conversation[-1]["content"])
```

5.3 Loops

A `for` loop does something once for each item:

PYTHON

```
expenses = [12.50, 8.00, 45.99, 3.25]
total = 0
for amount in expenses:
    total += amount           # same as total = total + amount
print(f"Total: ${total:.2f}")

for i, model in enumerate(["small", "large"], start=1):
    print(i, model)          # 1 small, then 2 large
```

☆ IMPORTANT

Indentation matters. The lines “inside” the loop must be indented (4 spaces is standard; your editor does it when you press `Tab`). Un-indenting ends the loop. Mixing tabs and spaces causes `IndentationError`.

5.4 Decisions with `if`

PYTHON

```
cost = 0.42
budget = 0.50
if cost > budget:
    print("Over budget. Stopping.")
elif cost > budget * 0.8:
    print("Warning: 80% of budget used.")
else:
    print("Fine.")
```

Comparison operators: `==` (equal), `!=` (not equal), `<`, `>`, `<=`, `>=`. Combine them with `and`, `or`, `not`.

5.5 `while` loops

A `while` loop repeats until its condition becomes false. Chatbots use one to keep asking for input:

PYTHON

```
while True:
    line = input("You (or 'quit'): ")
    if line.lower() == "quit":
        break          # leave the loop
    print(f"You said {len(line.split())} words.")
```

5.6 List comprehensions (a shortcut you'll see everywhere)

PYTHON

```
prices = [10, 25, 3]
with_tax = [p * 1.07 for p in prices]           # transform each
big = [p for p in prices if p > 5]              # filter
names = [m["content"] for m in conversation]    # pull a field out of each dict
```

Functions, Modules, and Errors

A **function** is a named, reusable block of code. As soon as you copy and paste the same three lines twice, make a function.

IN THIS CHAPTER

- Defining functions with inputs and outputs
- Importing modules from the standard library
- Handling errors with `try / except`, which matters a lot when calling the internet

6.1 Defining a function

PYTHON

```
def estimate_cost(input_tokens, output_tokens, in_price, out_price):
    """Return the cost in dollars. Prices are per 1 million tokens."""
    return (input_tokens * in_price + output_tokens * out_price) / 1_000_000

print(estimate_cost(20_000, 2_000, in_price=2.00, out_price=10.00)) # 0.06
```

PART	MEANING
<code>def name(...):</code>	Starts a function definition
Parameters	The names in the parentheses, which are the inputs
<code>"""docstring"""</code>	A description that editors show when you hover
<code>return</code>	Sends a value back to whoever called the function
Keyword arguments	<code>in_price=2.00</code> names the argument, which makes calls easy to read

Default values make parameters optional:

PYTHON

```
def make_prompt(text, style="plain English", max_words=100):  
    return f"Summarize in {style}, under {max_words} words:\n\n{text}"  
  
make_prompt("Long article...") # uses the defaults  
make_prompt("Long article...", style="bullet points")
```

6.2 Importing modules

Python comes with a large **standard library**. You `import` what you need:

PYTHON

```
import json # read/write JSON  
import csv # read/write spreadsheets as CSV  
import time # pause, measure time  
from pathlib import Path # work with files and folders  
from datetime import date  
  
print(date.today().isoformat()) # e.g. 2026-10-17  
time.sleep(1) # wait one second
```

You can import your own files too. If you have `helpers.py` with a function `clean()`, then `from helpers import clean` works from another file in the same folder.

6.3 Handling errors

Networks fail, files go missing, and AI services hit rate limits. `try / except` lets your program recover instead of crashing:

PYTHON

```
try:  
    number = int(input("A number: "))  
    print(100 / number)  
except ValueError:  
    print("That wasn't a whole number.")  
except ZeroDivisionError:  
    print("Can't divide by zero.")
```

A **retry with back-off** pattern is essential for AI APIs, which sometimes return “rate limited” or “overloaded”:

RETRY.PY

```
import time

def with_retries(func, attempts=4):
    for attempt in range(attempts):
        try:
            return func()
        except Exception as error:      # catch any error
            if attempt == attempts - 1:
                raise                   # give up: re-raise the last error
            wait = 2 ** attempt         # 1, 2, 4, 8 seconds
            print(f"Error: {error}. Retrying in {wait}s...")
            time.sleep(wait)
```

 NOTE

The official AI SDKs already retry some errors automatically (typically twice). Your own retry wrapper is for longer outages, or for libraries that don't retry.

Files, Folders, CSV, and JSON

Most useful AI scripts read files in and write results out. Python's `pathlib`, `csv` and `json` modules cover nearly everything.

IN THIS CHAPTER

- Reading and writing text files
- Looping over every file in a folder
- Spreadsheets as CSV, and structured data as JSON

7.1 Text files with pathlib

PYTHON

```
from pathlib import Path

notes = Path("notes.txt")
notes.write_text("First line\nSecond line\n", encoding="utf-8")
content = notes.read_text(encoding="utf-8")
print(content.splitlines())      # ['First line', 'Second line']

# Append instead of overwrite:
with notes.open("a", encoding="utf-8") as f:
    f.write("Third line\n")
```

TIP

Always pass `encoding="utf-8"`. Without it, Windows may use an older encoding and garble accented letters, emoji and curly quotes.

7.2 Every file in a folder

PYTHON

```
from pathlib import Path

folder = Path("documents")
folder.mkdir(exist_ok=True)          # create it if missing
for path in sorted(folder.glob("*.txt")): # every .txt file
    text = path.read_text(encoding="utf-8")
    print(f"{path.name}: {len(text.split())} words")

# Every .md file in all subfolders:
for path in folder.rglob("*.md"):
    print(path)
```

Useful `Path` parts: `path.name` (`report.txt`), `path.stem` (`report`), `path.suffix` (`.txt`), `path.parent` (the folder), and `path.with_suffix(".summary.txt")` for a new file name.

7.3 CSV spreadsheets

Every spreadsheet app can save or export **CSV** (comma-separated values). `csv.DictReader` turns each row into a dictionary keyed by the header row:

READ_CSV.PY

```
import csv

with open("expenses.csv", newline="", encoding="utf-8") as f:
    rows = list(csv.DictReader(f))

for row in rows:
    print(row["Date"], row["Description"], row["Amount"])

total = sum(float(r["Amount"]) for r in rows)
print(f"Total: {total:.2f}")
```

Writing a CSV:

PYTHON

```
import csv

results = [{"file": "a.txt", "summary": "..."}, {"file": "b.txt", "summary": "..."}]
with open("summaries.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["file", "summary"])
    writer.writeheader()
    writer.writerows(results)
```

7.4 JSON

JSON is the format AI APIs speak. It looks almost exactly like Python dictionaries and lists.

PYTHON

```
import json
from pathlib import Path

data = {"model": "example", "tags": ["fast", "cheap"], "score": 0.92}
text = json.dumps(data, indent=2)          # Python → JSON text
back = json.loads(text)                   # JSON text → Python
Path("data.json").write_text(text, encoding="utf-8")

# Read a JSON file:
loaded = json.loads(Path("data.json").read_text(encoding="utf-8"))
```

A common AI pattern is to ask the model to reply **only in JSON**, then parse its reply with `json.loads`. Always wrap that in `try / except json.JSONDecodeError`, because models occasionally add extra words. Better still, use the provider's **structured output** feature (see *Getting Reliable Structured Output*).

Virtual Environments and Packages

Python's real power is its **packages**: free libraries that other people wrote, installed with one command from **PyPI** (the Python Package Index, pypi.org). To keep each project's packages separate, you use a **virtual environment**.

IN THIS CHAPTER

- Why virtual environments exist
- Creating one with `venv`, and installing with `pip`
- Recording dependencies in `requirements.txt`
- The same workflow with `uv`

8.1 Why you need one

Project A may need version 2 of a library while project B needs version 3. A **virtual environment** (`venv`) is a private folder of packages for one project, so the projects never clash. It also means you can delete the folder and start fresh without harming anything else on your computer.

8.2 Create and use a `venv`

1 Open your project folder in the terminal.

For example, `cd python-ai`.

2 Create the environment.

Run `python -m venv .venv` (on a Mac, `python3 -m venv .venv`). This makes a hidden `.venv` folder.

3 Activate it.

Windows PowerShell: `.venv\Scripts\Activate.ps1` . macOS/Linux: `source .venv/bin/activate` . Your prompt now starts with `(.venv)` .

4 Install packages.

For example, `pip install requests python-dotenv` .

5 Save the list.

`pip freeze > requirements.txt` records exact versions so you (or anyone) can rebuild the environment with `pip install -r requirements.txt` .

6 Deactivate when done.

Type `deactivate` .

⚠ WARNING

On Windows, PowerShell may refuse to run `Activate.ps1` with a message about **execution policy**. Run `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned` once, answer **Y**, and try again. This only affects your user account.

💡 TIP

In VS Code, open the Command Palette (`Ctrl` / `Cmd` + `Shift` + `P`), run **Python: Select Interpreter**, and choose the one inside `.venv` . Then the Run button and new terminals use your environment automatically.

8.3 Packages used in this book

PACKAGE	INSTALL NAME	WHAT IT'S FOR
OpenAI SDK	<code>openai</code>	GPT models through the OpenAI API
Anthropic SDK	<code>anthropic</code>	Claude models through the Claude API
Google Gen AI SDK	<code>google-genai</code>	Gemini models through the Gemini API
xAI SDK	<code>xai-sdk</code>	Grok models through the xAI API
Ollama	<code>ollama</code>	Free local models running on your own computer
<code>python-dotenv</code>	<code>python-dotenv</code>	Loads secret keys from a <code>.env</code> file
<code>pandas</code>	<code>pandas</code>	Tables and spreadsheets
<code>requests</code>	<code>requests</code>	Calling any web API
Pydantic	<code>pydantic</code>	Defining data shapes for structured AI output (installed automatically with the AI SDKs)

Install everything at once:

TERMINAL

```
pip install openai anthropic google-genai xai-sdk ollama python-dotenv pandas requests
```

These were the latest versions on PyPI on October 16, 2026: `openai` 3.33.0, `anthropic` 1.15.0, `google-genai` 2.31.0, `xai-sdk` 1.20.0, `ollama` 0.6.3 and `pandas` 3.0.6. Newer versions usually work. If an example breaks after an upgrade, check that library's changelog.

8.4 The same thing with uv

TASK	PIP + VENV	UV
New project	<code>python -m venv .venv</code>	<code>uv init</code>
Add a package	<code>pip install pandas</code>	<code>uv add pandas</code>
Install from a list	<code>pip install -r requirements.txt</code>	<code>uv sync</code> (reads <code>pyproject.toml</code>)
Run a script	activate, then <code>python app.py</code>	<code>uv run app.py</code> (no activation needed)
One-off tool	<code>pip install tool</code>	<code>uvx tool</code>

Keeping Secret Keys Secret

To call a cloud AI API, you need an **API key**, a long password that identifies your account and lets the service bill you. Anyone with your key can spend your money. This chapter is short, and it's the most important one for your wallet.

IN THIS CHAPTER

- Where keys come from
- Storing keys in a `.env` file that never gets shared
- What to do if a key leaks

9.1 Get a key

PROVIDER	WHERE TO CREATE A KEY	ENVIRONMENT VARIABLE THE SDK READS
OpenAI	platform.openai.com → API keys	<code>OPENAI_API_KEY</code>
Anthropic	platform.claude.com (Claude Console) → API keys	<code>ANTHROPIC_API_KEY</code>
Google Gemini	aistudio.google.com → Get API key	<code>GEMINI_API_KEY</code> (or <code>GOOGLE_API_KEY</code>)
xAI	console.x.ai → API keys	<code>XAI_API_KEY</code>

API usage is billed separately from consumer subscriptions such as ChatGPT Plus, Claude Pro, Google AI Pro or SuperGrok. A subscription doesn't include API credit. See the *AI APIs & Costs* manual (Vol. 15) for current prices and how to set spending limits.

9.2 Store keys in a `.env` file

1

Create the file.

In your project folder, create a file named exactly `.env`.

2 Add keys, one per line.

No quotes or spaces are needed:

3 Ignore it.

Create a `.gitignore` file containing `.env` so Git never uploads it.

4 Load it in code.

Use `python-dotenv`, as shown below.

`.ENV`

```
OPENAI_API_KEY=sk-...your key...
ANTHROPIC_API_KEY=sk-ant-...your key...
GEMINI_API_KEY=...your key...
XAI_API_KEY=xai-...your key...
```

`CHECK_KEYS.PY`

```
import os
from dotenv import load_dotenv

load_dotenv() # reads .env into environment variables
for name in ["OPENAI_API_KEY", "ANTHROPIC_API_KEY", "GEMINI_API_KEY", "XAI_API_KEY"]:
    value = os.getenv(name)
    print(f"{name}: {'set' if value else 'MISSING'}") # never print the key itself
```

⚠ DANGER

Never paste a key into your code, a screenshot, a chat with an AI assistant, or a public repository. Bots scan GitHub for leaked keys within minutes. If a key leaks, **revoke it immediately** on the provider's key page and create a new one. Then check your usage page for charges you don't recognize.

9.3 Set a spending limit before you start

Every provider lets you see usage, and most let you set budgets or alerts. Prepaid credit (Anthropic and xAI, and OpenAI for many accounts) means you can't spend more than you loaded. Do this **before** your first script runs in a loop.

Calling AI from Python

This is the chapter you’ve been building up to. Each provider’s official SDK follows the same pattern: create a **client**, send **messages**, read the **text** of the reply.

IN THIS CHAPTER

- One working example per provider: OpenAI, Anthropic, Google and xAI
- A free local model through Ollama
- Streaming, system instructions and token usage
- A small wrapper so you can switch providers with one word

NOTE

Model names change often. The names below were current on the providers’ model pages on October 16, 2026. If one is retired, the error message says so. Pick a current name from the provider’s models page, and your code otherwise stays the same.

10.1 OpenAI

OPENAI_HELLO.PY

```
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()
client = OpenAI() # reads OPENAI_API_KEY

response = client.responses.create(
    model="gpt-6-luna", # a low-cost model; see Vol. 15 for prices
    instructions="You are a concise tutor.",
    input="Explain what a Python list is in two sentences.",
)
print(response.output_text)
print(response.usage) # input and output token counts
```

The **Responses API** (`client.responses.create`) is OpenAI's main interface. The older Chat Completions API (`client.chat.completions.create`) still works and appears in many tutorials.

10.2 Anthropic (Claude)

CLAUDE_HELLO.PY

```
from dotenv import load_dotenv
import anthropic

load_dotenv()
client = anthropic.Anthropic() # reads ANTHROPIC_API_KEY

message = client.messages.create(
    model="claude-haiku-5-5",
    max_tokens=500, # required: the most tokens Claude may write
    system="You are a concise tutor.",
    messages=[{"role": "user", "content": "Explain what a Python list is in two sentences."}],
)
print(message.content[0].text)
print(message.usage.input_tokens, message.usage.output_tokens)
```

10.3 Google (Gemini)

GEMINI_HELLO.PY

```
from dotenv import load_dotenv
from google import genai
from google.genai import types

load_dotenv()
client = genai.Client()          # reads GEMINI_API_KEY

response = client.models.generate_content(
    model="gemini-3.8-flash",
    contents="Explain what a Python list is in two sentences.",
    config=types.GenerateContentConfig(system_instruction="You are a concise tutor."),
)
print(response.text)
print(response.usage_metadata)
```

TIP

Gemini's API has a **free tier** for some models, which is great for learning. The trade-off is that free-tier content may be used to improve Google's products. Don't send private data on the free tier. Check the current terms at ai.google.dev/gemini-api/docs/pricing.

10.4 xAI (Grok)

GROK_HELLO.PY

```
from dotenv import load_dotenv
from xai_sdk import Client
from xai_sdk.chat import system, user

load_dotenv()
client = Client()              # reads XAI_API_KEY

chat = client.chat.create(model="grok-4.7", messages=[system("You are a concise tutor.")])
chat.append(user("Explain what a Python list is in two sentences."))
response = chat.sample()
print(response.content)
```

xAI's API is also **compatible with the OpenAI SDK**: `OpenAI(base_url="https://api.x.ai/v1", api_key=os.getenv("XAI_API_KEY"))`. Many tools that "support OpenAI" can therefore talk to Grok, and to local models, just by changing the base URL.

10.5 A free local model with Ollama

Ollama runs open-weight models on your own computer. It's free, works offline, and nothing you type leaves your machine. Install it from **ollama.com**, then download a model in the terminal (Vol. 11, *Running AI Locally*, covers model choice and hardware):

TERMINAL

```
ollama pull llama3.2          # a small model, about 2 GB
```

LOCAL_HELLO.PY

```
import ollama

response = ollama.chat(
    model="llama3.2",
    messages=[
        {"role": "system", "content": "You are a concise tutor."},
        {"role": "user", "content": "Explain what a Python list is in two sentences."},
    ],
)
print(response.message.content)
```

Ollama also offers an OpenAI-compatible endpoint at `http://localhost:11434/v1`, so the OpenAI SDK works with it too.

10.6 Streaming: show words as they arrive

Long answers feel faster when you print them as they're generated:

STREAM_CLAUDE.PY

```
import anthropic
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic()
with client.messages.stream(
    model="claude-haiku-5-5",
    max_tokens=800,
    messages=[{"role": "user", "content": "Write a 150-word story about a robot gardener."}],
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True)
print()
```

STREAM_OPENAI.PY

```

from openai import OpenAI
from dotenv import load_dotenv

load_dotenv()
client = OpenAI()
stream = client.responses.create(model="gpt-6-luna", input="Write a 150-word story.", stream=True)
for event in stream:
    if event.type == "response.output_text.delta":
        print(event.delta, end="", flush=True)
print()

```

10.7 One function, any provider

Wrap each provider in one function, and your projects can switch with a single word:

AI.PY

```

import os
from dotenv import load_dotenv

load_dotenv()

DEFAULTS = {
    "openai": "gpt-6-luna",
    "claude": "claude-haiku-5-5",
    "gemini": "gemini-3.8-flash",
    "grok": "grok-4.7",
    "local": "llama3.2",
}

def ask(prompt, provider="local", system="You are helpful and concise.", model=None):
    model = model or DEFAULTS[provider]
    if provider == "openai":
        from openai import OpenAI
        r = OpenAI().responses.create(model=model, instructions=system, input=prompt)
        return r.output_text
    if provider == "claude":
        import anthropic
        r = anthropic.Anthropic().messages.create(
            model=model, max_tokens=2000, system=system,
            messages=[{"role": "user", "content": prompt}])
        return r.content[0].text
    if provider == "gemini":
        from google import genai
        from google.genai import types
        r = genai.Client().models.generate_content(
            model=model, contents=prompt,

```

AI.PY (CONTINUED)

```
        config=types.GenerateContentConfig(system_instruction=system))
    return r.text
if provider == "grok":
    from xai_sdk import Client
    from xai_sdk.chat import system as sys_msg, user
    chat = Client().chat.create(model=model, messages=[sys_msg(system)])
    chat.append(user(prompt))
    return chat.sample().content
if provider == "local":
    import ollama
    r = ollama.chat(model=model, messages=[
        {"role": "system", "content": system}, {"role": "user", "content": prompt}])
    return r.message.content
raise ValueError(f"Unknown provider: {provider}")

if __name__ == "__main__":
    print(ask("Say hello in five words.", provider=os.getenv("AI_PROVIDER", "local")))
```

Every project in this book imports `ask` from this file. Set `AI_PROVIDER=claude` (or another name) in your `.env` to switch everything at once.

 NOTE

`if __name__ == "__main__":` means “run this part only when the file is run directly, not when it’s imported”. It lets `ai.py` be both a library and a quick test.

Getting Reliable Structured Output

Free-form text is fine for humans. Programs need **structure**: a category, a number, a yes/no. Every major API can now return JSON that matches a shape you define with **Pydantic**.

IN THIS CHAPTER

- Describing the shape you want with a Pydantic model
- Asking each provider to fill it in
- A fallback for local models

11.1 Define the shape

PYTHON

```
from pydantic import BaseModel
from typing import Literal

class Expense(BaseModel):
    category: Literal["Groceries", "Dining", "Transport", "Bills", "Shopping", "Other"]
    confidence: float          # 0 to 1
    reason: str
```

11.2 OpenAI

PYTHON

```
from openai import OpenAI
r = OpenAI().responses.parse(
    model="gpt-6-luna",
    input="Categorize this card transaction: 'SHELL OIL 5741 SPRINGFIELD' $43.10",
    text_format=Expense,
)
expense = r.output_parsed          # an Expense object
print(expense.category, expense.confidence)
```

11.3 Anthropic

PYTHON

```
import anthropic
r = anthropic.Anthropic().messages.parse(
    model="claude-haiku-5-5",
    max_tokens=500,
    messages=[{"role": "user", "content": "Categorize: 'SHELL OIL 5741 SPRINGFIELD' $43.10"}],
    output_format=Expense,
)
print(r.parsed_output.category)
```

11.4 Gemini

PYTHON

```
from google import genai
r = genai.Client().models.generate_content(
    model="gemini-3.8-flash",
    contents="Categorize: 'SHELL OIL 5741 SPRINGFIELD' $43.10",
    config={"response_mime_type": "application/json", "response_schema": Expense},
)
print(r.parsed.category)
```

11.5 Local models (Ollama)

PYTHON

```
import ollama
r = ollama.chat(
    model="llama3.2",
    messages=[{"role": "user", "content": "Categorize: 'SHELL OIL 5741 SPRINGFIELD' $43.10"}],
    format=Expense.model_json_schema(),  # constrain output to the schema
)
expense = Expense.model_validate_json(r.message.content)
```

TIP

Even with structured output, **validate** the results. A model can return a valid category that's wrong. Spot-check a sample by hand, and keep the `reason` field. It makes mistakes easy to spot.

Working with Data in pandas

pandas turns spreadsheets into tables you can filter, group and total with a line or two of code. It's the standard tool for preparing data before AI touches it, and for summarizing what AI returns.

IN THIS CHAPTER

- Loading CSV and Excel files into a DataFrame
- Selecting, filtering, sorting and grouping
- Adding an AI-generated column, and saving the results

12.1 Load and look

PYTHON

```
import pandas as pd

df = pd.read_csv("expenses.csv")      # pd.read_excel("file.xlsx") needs: pip install openpyxl
print(df.head())                     # first 5 rows
print(df.shape)                       # (rows, columns)
print(df.columns.tolist())            # column names
print(df.describe())                  # quick statistics for number columns
```

A **DataFrame** is a table. Each column is a **Series**.

12.2 Select, filter, sort

PYTHON

```
df["Amount"]                                # one column
df[["Date", "Amount"]]                      # several columns
df[df["Amount"] > 50]                       # rows where Amount > 50
df[df["Description"].str.contains("AMAZON", case=False)]
df.sort_values("Amount", ascending=False).head(10) # 10 biggest
```

12.3 Clean common problems

PYTHON

```
df["Amount"] = (df["Amount"].astype(str)
                .str.replace("$", "", regex=False)
                .str.replace(",", "", regex=False)
                .astype(float))           # "$1,234.50" → 1234.5
df["Date"] = pd.to_datetime(df["Date"])  # text → real dates
df = df.dropna(subset=["Amount"])         # drop rows with no amount
df = df.drop_duplicates()
```

12.4 Group and total

PYTHON

```
by_month = df.groupby(df["Date"].dt.to_period("M"))["Amount"].sum()
print(by_month)

by_category = df.groupby("Category")["Amount"].agg(["count", "sum", "mean"])
print(by_category.sort_values("sum", ascending=False))
```

12.5 Add an AI column

PYTHON

```
from ai import ask

def categorize(description):
    return ask(f"One word category for this purchase: {description}. Reply with the word only.").strip()

df["Category"] = df["Description"].apply(categorize) # one API call per row
df.to_csv("expenses_categorized.csv", index=False)
```

 WARNING

`.apply` with an AI call runs **one request per row**. 1,000 rows means 1,000 calls. Test on `df.head(10)` first, estimate the cost (see Vol. 15), and prefer sending unique descriptions only: `df["Description"].unique()`.

Projects

Each project is a complete script you can run today. They all use the `ask()` helper from *One function, any provider*. With `AI_PROVIDER=local` in your `.env`, all of them run free on your own computer through Ollama.

IN THIS CHAPTER

- Project 1: summarize every document in a folder
- Project 2: categorize a bank export and total it
- Project 3: a terminal chatbot with memory and a cost meter
- Project 4: ask the same questions of many documents
- Project 5: a daily briefing from RSS feeds

13.1 Project 1: Folder summarizer

SUMMARIZE_FOLDER.PY

```
from pathlib import Path
from ai import ask

SOURCE = Path("documents")
OUT = Path("summaries")
OUT.mkdir(exist_ok=True)

PROMPT = """Summarize the document below in one paragraph of at most 80 words.
Then list up to 3 action items, if any, as bullets starting with "- ".

DOCUMENT:
{text}"""

for path in sorted(SOURCE.glob("*.txt")):
    target = OUT / f"{path.stem}.summary.txt"
    if target.exists():
        print(f"skip {path.name} (already done)")
        continue
    text = path.read_text(encoding="utf-8")[:20_000]    # cap very long files
    summary = ask(PROMPT.format(text=text))
    target.write_text(summary, encoding="utf-8")
    print(f"done {path.name}")
```

What makes it robust:

- It **skips** files that already have a summary, so you can stop and restart safely.
- It **caps** input length, so one giant file can't blow your budget or the model's context window.
- The prompt lives in one constant, so it's easy to tune.

Try next: read `.pdf` files with `pip install pypdf` (`from pypdf import PdfReader`, then join `page.extract_text()` for every page), or `.docx` with `pip install python-docx`.

13.2 Project 2: Expense categorizer

Export your transactions from your bank as CSV. **Delete account-number columns first.** Then:

CATEGORIZE_EXPENSES.PY

```

import json
from pathlib import Path
import pandas as pd
from ai import ask

CATEGORIES = ["Groceries", "Dining", "Transport", "Bills", "Shopping",
              "Health", "Entertainment", "Income", "Transfer", "Other"]

df = pd.read_csv("transactions.csv")
unique = sorted(df["Description"].dropna().unique())
print(f"{len(df)} rows, {len(unique)} unique descriptions")

mapping = {}
BATCH = 40
for i in range(0, len(unique), BATCH):
    chunk = unique[i:i + BATCH]
    prompt = (f"Assign each transaction description one category from {CATEGORIES}.\n"
              "Reply with only a JSON object mapping each description to its category.\n\n"
              + "\n".join(chunk))
    reply = ask(prompt)
    start, end = reply.find("{"), reply.rfind("}") + 1  # tolerate extra words
    try:
        mapping.update(json.loads(reply[start:end]))
    except json.JSONDecodeError:
        print(f"Couldn't parse batch {i // BATCH}; marking as Other")

df["Category"] = df["Description"].map(mapping).fillna("Other")
df.to_csv("transactions_categorized.csv", index=False)
print(df.groupby("Category")["Amount"].sum().sort_values())

```

Why batching matters: sending 40 descriptions per request instead of one cuts the number of calls by 40×, and usually the cost too, because instructions are sent once per batch instead of once per row.

 DANGER

This is for organizing your own records, not financial advice. Check the totals against your statement. Vol. 17, *AI for Personal Finance & Admin*, covers privacy rules for financial data.

13.3 Project 3: Terminal chatbot with memory and a cost meter

CHATBOT.PY

```
import anthropic
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic()
MODEL = "claude-haiku-5-5"
PRICE_IN, PRICE_OUT = 0.10, 0.50      # $ per 1M tokens, checked Oct 16, 2026 (prompts ≤100k tokens)

history = []
spent = 0.0
print("Chat with Claude. Type 'quit' to stop, 'reset' to clear memory.")
while True:
    user_text = input("\nYou: ").strip()
    if user_text.lower() == "quit":
        break
    if user_text.lower() == "reset":
        history.clear()
        print("memory cleared")
        continue
    history.append({"role": "user", "content": user_text})
    reply = client.messages.create(model=MODEL, max_tokens=1000,
                                   system="You are a friendly, concise assistant.",
                                   messages=history)

    text = reply.content[0].text
    history.append({"role": "assistant", "content": text})
    spent += (reply.usage.input_tokens * PRICE_IN + reply.usage.output_tokens * PRICE_OUT) / 1e6
    print(f"\nClaude: {text}\n    [this session: ${spent:.5f}"])
```

Key lesson: the model has no memory of its own. “Memory” is just the `history` list you resend every turn. That’s why long chats get more expensive: each new message re-sends everything before it. The `reset` command is your cost control.

13.4 Project 4: Batch document Q&A to a spreadsheet

DOC_QA.PY

```
from pathlib import Path
import pandas as pd
from ai import ask

QUESTIONS = {
    "date": "What date is this document from? Reply YYYY-MM-DD or 'unknown'.",
    "party": "Who is the other party (company or person)? Short answer.",
    "amount": "What is the main dollar amount, if any? Number only or 'none'.",
    "deadline": "Is there a deadline? Reply YYYY-MM-DD or 'none'.",
}

rows = []
for path in sorted(Path("letters").glob("*.txt")):
    text = path.read_text(encoding="utf-8")[:15_000]
    row = {"file": path.name}
    for key, question in QUESTIONS.items():
        row[key] = ask(f"{question}\n\nDocument:\n{text}").strip()
    rows.append(row)
    print(row)

pd.DataFrame(rows).to_excel("letters.xlsx", index=False) # needs: pip install openpyxl
```

Improve it: ask all four questions in **one** request, using structured output (*Getting Reliable Structured Output*). That's 4× fewer calls.

13.5 Project 5: Morning briefing from RSS feeds

BRIEFING.PY

```
import requests
import xml.etree.ElementTree as ET
from datetime import date
from ai import ask

FEEDS = [
    "https://feeds.bbc1.co.uk/news/technology/rss.xml",
    # add any RSS feed URLs you like
]

headlines = []
for url in FEEDS:
    xml = requests.get(url, timeout=20).text
    for item in ET.fromstring(xml).iter("item"):
        title = item.findtext("title")
        if title:
            headlines.append(title.strip())

brief = ask("Group these headlines into 3-5 themes and write a 150-word briefing. "
            "Do not invent details beyond the headlines.\n\n" + "\n".join(headlines[:60]))
out = f"briefing-{date.today().isoformat()}.md"
open(out, "w", encoding="utf-8").write(brief)
print(brief)
```

Run it every morning with Windows **Task Scheduler** or `cron` on macOS/Linux, or with an automation tool (see Vol. 6, *n8n*, and Vol. 16, *Zapier & Make*).

Debugging, and Using AI to Help You Code

Every programmer spends a lot of time debugging. The skill is staying calm and narrowing the problem down.

IN THIS CHAPTER

- The five most common beginner errors
- Print debugging and the VS Code debugger
- How to ask an AI assistant for help effectively

14.1 The errors you'll see most

ERROR	USUALLY MEANS	FIX
<code>NameError: name 'x' is not defined</code>	A typo, or using a variable before creating it	Check the spelling; check the order of lines
<code>IndentationError</code> / <code>unexpected indent</code>	Inconsistent spaces	Re-indent the block with 4 spaces
<code>TypeError: can only concatenate str (not "int") to str</code>	Mixing text and numbers	Use an f-string: <code>f"Total {n}"</code>
<code>KeyError: 'Amount'</code>	A dictionary or column doesn't have that key	Print <code>.keys()</code> or <code>df.columns</code> ; check capitals and spaces
<code>ModuleNotFoundError: No module named 'openai'</code>	The package isn't installed in the active environment	Activate <code>.venv</code> , then <code>pip install openai</code> ; select the right interpreter in VS Code
<code>FileNotFoundError</code>	Wrong folder or name	<code>print(Path.cwd())</code> to see where you are; use <code>Tab</code> completion
<code>AuthenticationError</code> / <code>401</code>	The API key is missing or wrong	Run <code>check_keys.py</code> ; make sure <code>load_dotenv()</code> runs before creating the client
<code>RateLimitError</code> / <code>429</code>	Too many requests, or no credit left	Slow down, add retries, check billing

14.2 Print debugging

The simplest technique is to print values right before the line that fails:

PYTHON

```
print("DEBUG row:", row)
print("DEBUG type:", type(row["Amount"]), repr(row["Amount"]))
```

`repr()` shows hidden characters, so `'12.50\n'` reveals a stray newline.

14.3 The VS Code debugger

1

Set a breakpoint.

Click in the margin to the left of a line number. A red dot appears.

2

Start debugging.

Press `F5` and choose **Python File**.

3

Inspect.

When execution pauses on your line, hover over variables to see their values, or look in the **Variables** panel.

4

Step.

Use **Step Over** (**F10**) to run one line at a time, and **Continue** (**F5**) to run to the next breakpoint.

14.4 Asking AI for coding help

PROMPT

I'm a Python beginner. Here is my code and the full error.
Explain in plain English what the error means, why it happened,
and the smallest change that fixes it. Don't rewrite the whole program.

CODE:

<paste>

ERROR (full traceback):

<paste>

- ☐ ☐ Paste the **full** traceback, not just the last line
- ☐ ☐ Remove API keys and personal data before pasting
- ☐ ☐ Ask for an explanation, not just a fix, so you learn
- ☐ ☐ Run the fix and check it actually worked
- ☐ ☐ If the AI suggests installing packages, check the name on pypi.org first. Fake look-alike packages exist

Troubleshooting and FAQ

PROBLEM	FIX
<code>python</code> opens the Microsoft Store (Windows)	Reinstall from python.org with Add python.exe to PATH ticked, or use <code>py</code> . Or turn off the store aliases in Settings > Apps > Advanced app settings > App execution aliases
<code>pip</code> isn't recognized	Use <code>python -m pip install ...</code> . It always uses the matching Python
Packages install but imports fail	You installed into a different Python. Activate <code>.venv</code> , and in VS Code select the <code>.venv</code> interpreter
Accented letters or emoji look garbled	Add <code>encoding="utf-8"</code> to every <code>open()</code> / <code>read_text()</code> / <code>write_text()</code>
Script works in the terminal but not with VS Code's Run button	VS Code is using a different interpreter. Use Python: Select Interpreter
AI replies are cut off	Raise <code>max_tokens</code> (Anthropic) or <code>max_output_tokens</code> (OpenAI)
JSON parsing fails on AI output	Use structured output, or extract the text between the first <code>{</code> and last <code>}</code>
Ollama: <code>connection refused</code>	Start the Ollama app (or run <code>ollama serve</code>), then try again
Ollama is very slow	The model is too big for your memory. Try a smaller one, and see Vol. 11
Unexpected API bill	Revoke the key if it may have leaked. Set budgets or alerts, and cap loops with a counter

Do I need to learn math? No. You can do a lot of useful AI work with the Python in this book.

Python 2 or 3? Always Python 3. Python 2 ended in 2020.

Should I use notebooks (Jupyter)? They're great for exploring data step by step. Install with `pip install notebook` and run `jupyter notebook`, or open a `.ipynb` file in VS Code. Use plain scripts for anything you'll run repeatedly.

What next? Build one tool you'll actually use every week. Then read the official tutorial at docs.python.org/3/tutorial, and the SDK docs for the AI provider you like best.

Quick Reference

Run and install

- `python file.py` (Mac: `python3`), `python -m pip install pkg`
- `python -m venv .venv` → activate → `pip install -r requirements.txt`
- uv: `uv init`, `uv add pkg`, `uv run file.py`

Text

- `f"Hi {name}"`, `s.strip()`, `s.lower()`, `s.split()`, `", ".join(list)`
- `"""multi-line"""`, `s[:100]` (first 100 characters)

Collections

- List: `x = [1, 2]`, `x.append(3)`, `x[0]`, `x[-1]`, `len(x)`
- Dict: `d = {"k": "v"}`, `d["k"]`, `d.get("k", default)`, `d.items()`
- Comprehension: `[f(i) for i in items if cond]`

Control flow

- `if / elif / else`, `for item in items:`, `while cond:`, `break`, `continue`
- `try: ... except SomeError: ...`

Files

- `Path("a.txt").read_text(encoding="utf-8"), .write_text(...)`
- `Path("folder").glob("*.txt"), .mkdir(exist_ok=True)`
- `csv.DictReader(f), json.loads(text), json.dumps(obj, indent=2)`

pandas

- `pd.read_csv(), df.head(), df[df["col"] > 5], df.groupby("c")["n"].sum(), df.to_csv(index=False)`

AI SDKs

- OpenAI: `OpenAI().responses.create(model=..., input=...).output_text`
- Claude: `Anthropic().messages.create(model=..., max_tokens=..., messages=[...]).content[0].text`
- Gemini: `genai.Client().models.generate_content(model=..., contents=...).text`
- Grok: `Client().chat.create(model=...) → .append(user(...)) → .sample().content`
- Ollama: `ollama.chat(model=..., messages=[...]).message.content`

Glossary

API	Application Programming Interface: a way for programs to talk to a service over the internet.
API key	A secret password that identifies your account to an API and lets it bill you.
Argument	A value you pass into a function when you call it.
Context window	The maximum amount of text (in tokens) a model can consider at once, including the conversation history.
CSV	Comma-separated values: a plain-text spreadsheet format.
DataFrame	A pandas table of rows and columns.
Dictionary	A Python collection of key-value pairs, like <code>{"role": "user"}</code> .
Environment variable	A named setting stored outside your code, used for secrets such as API keys.
f-string	A string starting with <code>f</code> where <code>{expressions}</code> are filled in.
Function	A named, reusable block of code defined with <code>def</code> .
Interpreter	The program that reads and runs Python code.
JSON	JavaScript Object Notation: the text format most APIs use to send data.
List	An ordered Python collection, like <code>[1, 2, 3]</code> .
Module	A Python file, or built-in library, that you can import.
Ollama	A free app that downloads and runs open-weight AI models locally.
Package	Installable code published on PyPI, such as <code>pandas</code> .
pip	Python's standard package installer.
Pydantic	A library for defining data shapes; the AI SDKs use it for structured output.
PyPI	The Python Package Index, <code>pypi.org</code> , where packages are published.

REPL	Read-Evaluate-Print Loop: the interactive <code>>>></code> prompt.
SDK	Software Development Kit: an official library that makes an API easy to call.
Streaming	Receiving a model's reply piece by piece as it's generated.
Structured output	Making a model reply in JSON that matches a schema you define.
Token	A chunk of text (roughly $\frac{3}{4}$ of an English word) that AI models read and write, and bill by.
Traceback	Python's error report, showing where an error happened.
uv	A fast tool that manages Python versions, environments and packages.
Variable	A name that holds a value.
Virtual environment	A private folder of packages for one project.