Python

for Network Engineers

From Zero to Automated Networks

JOZEF BAROŠ

FREE SAMPLE

Python for Network Engineers

From Zero to Automated Networks

These are the opening pages of a 632-page book written for network engineers who have never written a line of Python, and who want to stop doing by hand what a script could do in seconds.

WHAT IS IN THIS SAMPLE

- The complete front matter, including the full table of contents of all seventeen chapters and three appendices.
- Chapter 1 in full — what automation actually costs you today, what Python changes, and what it will not fix.
- Chapter 2 up to and including the Windows installation, so you can have a working Python before you decide anything.

WHAT IS IN THE FULL BOOK

- Python from variables to comprehensions, taught with network data rather than shopping lists.
- Netmiko, Paramiko, NAPALM, Nornir, pyATS, TextFSM, NETCONF, RESTCONF and Jinja2, each with runnable code.
- Off-box automation: inventory, IP address maths, Excel, KeePass.
- Using AI properly — the APIs, local models, and the prompt patterns that produce network scripts you can actually trust.
- Worked solutions to every exercise, a glossary and a full index.

---

leanpub.com/python-for-network-engineer

Copyright and Disclaimer

Python for Network Engineers — From Zero to Automated Networks

Trademarks. Cisco, IOS, IOS-XE, NX-OS, Catalyst, Nexus, DNA Center and Meraki are trademarks or registered trademarks of Cisco Systems, Inc. Juniper and Junos are trademarks of Juniper Networks, Inc. Fortinet and FortiOS are trademarks of Fortinet, Inc. Python is a trademark of the Python Software Foundation. Windows is a trademark of Microsoft Corporation. macOS is a trademark of Apple Inc. All other trademarks are the property of their respective owners. This book is an independent publication and is not affiliated with, authorised by, sponsored by or otherwise approved by any of these organisations.

Disclaimer. The information in this book is provided on an as-is basis, without warranty of any kind. Every script, command and configuration example has been written with care, but network environments differ enormously and the author accepts no liability for any loss, outage or damage arising directly or indirectly from the use of the material in this book. Never run a configuration-changing script against production infrastructure without testing it first, understanding exactly what it does, and having a rollback plan.

Edition. First edition, 2026. Examples are written against Python 3.11 and later, Netmiko 4.x, NAPALM 5.x, Nornir 3.x and Jinja2 3.x.

About This Book

This is a Python course written specifically for network engineers, by a network engineer. It assumes you know networks and assumes nothing at all about programming.

Who this book is for

You are a network engineer — CCNA, CCNP, or years of hands-on experience without the certificate. You configure switches, routers, firewalls and wireless controllers. You have repeatedly thought that there must be a better way than doing the same change on eighty devices, and you have possibly tried to learn Python before from a general-purpose tutorial that spent three chapters on shopping carts and lost you.

This book takes the opposite approach. There is no chapter on shopping carts. The very first data structure you meet is a device inventory. Loops are introduced by iterating over switches. String formatting is taught by building configuration lines. Every concept in the language arrives attached to a network problem, because that is what makes it stick.

What you will be able to do at the end

- Read, understand and modify any beginner-to-intermediate Python script you encounter in a networking context.
- Connect to Cisco, Juniper and Fortinet devices with Netmiko, NAPALM and Nornir, gather information reliably, and push configuration safely.
- Consume REST, RESTCONF and NETCONF APIs on controllers such as DNA Center, Meraki and FortiManager.
- Turn unstructured `show` command output into structured data using regular expressions, TextFSM and Genie parsers.
- Generate configuration for an entire site from a YAML data file and a Jinja2 template, and validate the result.
- Build the operational scripts that make a team faster: configuration backup, compliance auditing, interface error reports, topology discovery, and Excel deliverables.

- Automate work that never touches a device — inventory reconciliation, IP address maths, KeePass credential retrieval, scheduled reporting and notifications.
- Use AI models and APIs from Python to accelerate script writing and log analysis, with the judgement to validate what they produce.

What you need

A computer running Windows, macOS or Linux, and roughly two hours per chapter. No lab, no licences and no budget are required for the majority of the book — most examples run entirely offline against sample data files that you create as you go. Chapters that genuinely need a device explain how to use the free Cisco DevNet Sandbox.

How to work through it

In order, with a keyboard, typing the shorter examples rather than copying them. Every section ends with a Summary and a Key Takeaways box, and every chapter closes with a set of hands-on exercises that build directly on what you have just read. Do the exercises — they are where the reading turns into skill, and several of them produce files you will keep using for the rest of the book.

Worked solutions to every exercise are in Appendix A. Use them to check your answer once yours runs, or to unblock yourself when you have been stuck for a while — but attempt the exercise first, because a solution read too early replaces learning with recognition.

> ✔ **Pro Tip — If you are short on time**
>
> Read Chapter 1, install Python from Chapter 2, set up a virtual environment from section 3.1, then go straight to Part II and work forward. Come back to sections 3.3 to 3.6 when you need an editor, version control or a production jump host. Do not skip the virtual environment; it is the one piece of setup that causes problems later if it is missing.

The Complete Book at a Glance

Seventeen chapters in five parts. Each part builds directly on the one before it.

Part I — Getting Set Up

Ch	Title	What you walk away with
1	Why Python Belongs in Your Network Toolkit	The economics and the mental model
2	Installing Python on Windows, macOS and Linux	A working interpreter on your machine
3	Your Working Environment	venv, pip, VS Code, Git, jump hosts

Part II — Python Fundamentals, Taught on Network Data

Ch	Title	What you walk away with
4	Variables, Types and Strings	f-strings, slicing, the string methods that parse output
5	Lists, Dictionaries, Sets and Tuples	Modelling an inventory as data
6	Conditionals and Loops	Iterating a fleet, filtering, comprehensions
7	Functions, Modules and Error Handling	Reusable code, try/except, logging
8	Files, CSV, JSON and YAML	Reading inventories, writing reports, config as data
9	Regular Expressions and Parsing show Output	re, TextFSM, ntc-templates, Genie parsers

Part III — Talking to the Network

Ch	Title	What you walk away with
10	Paramiko and Netmiko	SSH to devices, multi-device loops, concurrency, secrets
11	REST, RESTCONF and NETCONF	requests, tokens, ncclient, DNA Center, Meraki, FortiOS
12	NAPALM, Nornir and pyATS	Multi-vendor getters, inventory-driven runs, testing

Part IV — Generating Configuration

Ch	Title	What you walk away with
13	Jinja2 Templates and Data-Driven Config	One template, fifty switches, and a compliance diff

Part V — Use Cases, Off-Box Tooling and AI

Ch	Title	What you walk away with
14	Operational Use Cases	Backups, audits, error reports, topology maps, Excel
15	Off-Box Use Cases	Inventory search, KeePass, ipaddress, email and chat alerts
16	AI-Assisted Network Automation	Claude and OpenAI APIs, local models, prompt patterns
17	Putting It Together: Your First Project	Repository structure, testing, staged rollout, handover

Appendix

	Title	What you walk away with
A	Solutions to the Hands-On Practice	Every exercise, worked and explained

> ◆ **From the Field — Why the order is what it is**
>
> Parts II and III are deliberately separated. Most network engineers who fail to learn Python fail because they jumped straight to Netmiko, got a working script by copying, and then could not modify it when the requirement changed. Six chapters of fundamentals first feels slow for about a week and then pays back permanently — because from Chapter 10 onward you are reading library documentation rather than searching for a snippet that happens to fit.

Table of Contents

Page numbers refer to this document.

Chapter 1 — Why Python Belongs in Your Network Toolkit

Every network engineer eventually hits the same wall. You know exactly what needs to happen — the change is simple, you have done it a hundred times — but it needs to happen on two hundred devices, tonight, without a single typo. The knowledge is not the bottleneck. Your hands are.

This chapter is about that wall, and about the specific ways Python removes it. We are not going to write much code yet. Instead we are going to build the mental model that makes everything in the rest of the book click: what automation actually is, what it costs, where it pays back fastest, and — just as importantly — where it does not pay back at all.

If you have picked up a Python book before and bounced off it because the examples were about shopping carts and student grade averages, this one will feel different. Every single example in this book is a network engineering example. The first program you write reads a device inventory. The last one generates configuration for an entire site and validates the result afterwards.

Section	What you will learn	Style
1.1	The real cost of manual network work	Concept
1.2	What Python actually changes in your day	Concept
1.3	What Python is not, and its honest limitations	Concept
1.4	How to use this book, and your lab options	Practical

> ✔ **Pro Tip — Read this chapter, do not skip it**
>
> It is tempting to jump straight to Chapter 2 and start installing things. Resist for twenty minutes. Engineers who understand **why** a script exists write better scripts than engineers who only know **how** to write one. The mental model in this chapter is what stops you from automating the wrong thing — which is the most expensive mistake in this field.

1.1 The Real Cost of Doing It By Hand

Let us start with something concrete. Suppose your organisation runs 180 access switches, and you need to add a new VLAN for a printer network. On one switch the work is trivial: SSH in, enter configuration mode, three lines, write memory, done. Two minutes if you are slow.

The change, on one device, exactly as you would type it today

```
configure terminal
 vlan 340
  name PRINTERS
 exit
 interface range GigabitEthernet1/0/1 - 24
  switchport trunk allowed vlan add 340
 end
write memory
```

Two minutes multiplied by 180 devices is six hours. That arithmetic is wrong, and it is wrong in a way that matters. The real cost is considerably higher, and it is worth understanding exactly where the extra hours come from.

The hidden multipliers

Manual work does not scale linearly. Every additional device you touch brings costs that never appear in the two-minute estimate:

- **Context switching.** You are not performing 180 identical operations in a clean room. You are being interrupted by tickets, calls and colleagues, and every interruption means re-checking which switches you already finished. The spreadsheet with 180 rows and a colour-coded 'done' column is itself a piece of work that nobody accounts for.
- **Error rate.** Human accuracy on repetitive typing tasks is roughly 99 percent per operation at best, and it degrades sharply with fatigue. At 180 devices that is one or two devices configured incorrectly. You will not know which ones until something breaks, possibly weeks later, and by then nobody will connect the outage to that Tuesday evening.
- **Verification.** Making the change is half the job. Proving it was applied correctly across all 180 devices is the other half, and done manually it takes just as long as the change itself.

- **Documentation.** Someone has to record what changed, when, and on which devices. Manually this is a separate task, which is why it is usually the first thing to be skipped when the window runs late.
- **Rollback.** If the change goes wrong at 02:00, you need to undo it on 180 devices quickly. Manually, that is another six hours — and you do not have six hours at 02:00.

Add all of that together and a 'six hour' change is realistically a one to two day task with an ongoing tail of cleanup. And the moment somebody asks for a second VLAN next month, you pay the entire bill again.

> ◆ **From the Field — Why the same change keeps coming back**
>
> In enterprise networks no change is ever done once. Networks grow, sites open, acquisitions land on your desk, and the security team requests a new segment every quarter. The engineer who scripted the VLAN change in 2023 spends four minutes on the 2024 request. The engineer who did it by hand spends another two days. Over a career, that gap compounds into entire months of your life.

The asymmetry that makes automation worth it

Here is the key insight, and it is the one thing to take away from this chapter. Writing a script has a fixed cost. Running a script has a cost that is effectively zero. Manual work has no preparation cost at all, but it charges you in full on every single execution.

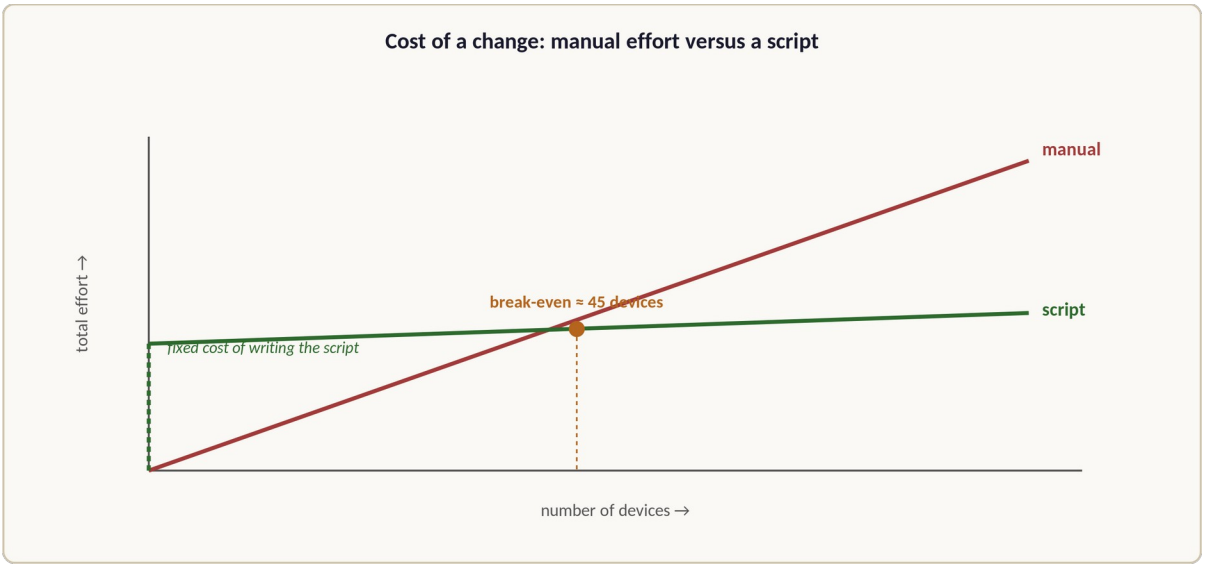

Figure 1.1 — Manual effort rises with every device. A script costs more up front and almost nothing thereafter.

The break-even point in this example arrives at around 45 devices. Anything above that and the script wins on its very first run. Below it, the script still wins on the second run. This is why experienced automation engineers say the question is never 'is this worth automating' but rather 'how many times will this happen'.

Approach	Cost to prepare	Cost per device	Cost of a second run
Manual	0 minutes	2 minutes	6 hours
Script	90 minutes	~0.5 seconds	2 minutes

> **★ Did you know? — The three-times rule**
>
> A common heuristic in automation circles: if you have done something manually three times, script it on the third occasion. The first time you do not yet understand the shape of the problem. The second time you confirm that it recurs. By the third you have enough understanding to write a script that actually works — and you are almost certainly about to face a fourth.

The cost automation does not remove

Honesty matters more than enthusiasm here. Automation moves work; it does not delete it. A script that pushes configuration to 180 devices can also break 180 devices in eight seconds. That is a genuinely new failure mode which manual work does not have, and it is why later chapters spend serious time on dry runs, pre-change snapshots, error handling and rollback.

There is also a maintenance cost. Scripts rot. A parser that worked perfectly against IOS 15.2 output may break when a site is upgraded to IOS-XE 17.9 and the column headings shift by two characters. Part of adopting automation is accepting that you now own a small piece of software, with everything that implies.

> ⚠ **Common Pitfall — Do not automate a process you do not understand**
>
> The worst possible automation outcome is a script that faithfully and rapidly does the wrong thing everywhere. If you cannot describe the change precisely enough to explain it to a junior colleague, you cannot describe it precisely enough to write a script. Understand first, automate second.

Summary

Manual network work looks cheap per device and is expensive in aggregate, because the two-minute estimate ignores context switching, error rate, verification, documentation and rollback. Automation converts a repeated per-device cost into a one-time fixed cost, which is why the break-even point arrives far sooner than most engineers expect — often on the very first run. The trade-offs are real: automation introduces a new failure mode in which mistakes propagate at machine speed, and it makes you the owner of software that will need maintenance as device software changes. The correct response is not to avoid automation but to build verification, staged rollout and rollback into every script from the beginning.

Key Takeaways

Manual work does not scale linearly — hidden costs multiply with device count. **Human error rate is roughly 1 percent** per repetitive operation, which is unacceptable at scale. **Scripts have a fixed cost and near-zero marginal cost**, so break-even usually arrives within a few dozen devices. **The three-times rule**: if you have done it manually three times, script it. **Automation propagates mistakes as fast as it propagates fixes** — verification is not optional. **Scripts rot** as device software changes; you now own software. **Never automate a process you cannot explain precisely.**

1.2 What Python Actually Changes

So far we have argued that automation is worth doing. That is not the same as arguing that Python specifically is the right tool. Network engineers already automate: with TCL scripts on IOS, with EEM applets, with Excel macros that generate configuration text, with SecureCRT scripting, with Ansible playbooks. Why add Python to that list?

Python sits at the centre of the network automation ecosystem

The honest answer is not that Python is the most elegant language ever designed. It is that Python is where the network automation libraries live. Netmiko, NAPALM, Nornir, ncclient, Scrapli, pyATS, Genie, TextFSM — all of them are Python. Cisco's own DevNet learning material is Python. Ansible itself is written in Python, and its network modules are Python modules. When a vendor ships an SDK for their controller, it arrives as a Python package on PyPI.

This creates a self-reinforcing cycle. Because the libraries are in Python, engineers learn Python; because engineers know Python, the next library is also written in Python. As a practical matter, learning Python gives you access to more than a decade of accumulated networking tooling that simply has no equivalent in any other language.

Task	Python library	What it saves you from doing yourself
SSH to network devices	Netmiko	Vendor-specific prompt and paging handling
Low-level SSH control	Paramiko	Implementing the SSH protocol yourself
Multi-vendor abstraction	NAPALM	Learning every vendor's CLI quirks
Parallel execution + inventory	Nornir	Writing your own threading and inventory code
REST APIs (controllers, cloud)	requests	Hand-rolling HTTP and JSON plumbing

Task	Python library	What it saves you from doing yourself
NETCONF	ncclient	Raw XML over an SSH subsystem
Parsing CLI output into data	TextFSM / Genie	Fragile hand-written regular expressions
Config generation from templates	Jinja2	String concatenation nightmares
Reading and writing Excel	openpyxl	Manual copy-paste into reports
IP address and subnet maths	ipaddress	Binary arithmetic by hand
Reading a KeePass password vault	pykeepass	Hard-coding credentials in your scripts

Every one of those libraries is covered somewhere in this book. You do not need to memorise the table — it is here so you can see the shape of the road ahead.

From typing commands to describing intent

The deeper change Python brings is not speed. It is a shift in how you think about the network. Today you probably think in terms of commands: the things you type into a device. Automation pushes you toward thinking in terms of data — the facts about your network, stored somewhere structured, from which configuration is then generated.

Consider the difference. Here is the command-oriented view, which is how a CLI engineer naturally thinks:

Command-oriented — the configuration itself is the source of truth

```
interface GigabitEthernet1/0/5
 description AP-FLOOR2-EAST
 switchport mode access
 switchport access vlan 220
```

```
 spanning-tree portfast
```

And here is the data-oriented view, which is how an automation engineer thinks. Exactly the same information, expressed as facts rather than as commands:

Data-oriented — the data is the source of truth, configuration is generated from it

```
port: GigabitEthernet1/0/5
role: access-point
description: AP-FLOOR2-EAST
vlan: 220
```

Why does this matter? Because data can be queried, validated, compared and reused, while a wall of configuration text cannot. Once your network is described as data, you can answer a question like 'which ports across all 180 switches are in VLAN 220' with a two-line script instead of a two-day audit. You can also regenerate the configuration for any device at any time, which means a failed switch can be replaced in minutes rather than reconstructed from memory.

> **★ Did you know? — This is what 'infrastructure as code' really means**
> The phrase gets used loosely, but the substance is exactly the shift above: your network's desired state lives in structured, version-controlled data files, and the running configuration is a **derived artefact** produced from them. If the two ever disagree, the data wins and you push the configuration again. Chapter 13 on Jinja2 templating is where this stops being a slogan and becomes something you can run.

The four things Python does for a network engineer

Strip away the buzzwords and Python does exactly four useful things in a network context. Everything in this book is one of these four, or a combination of them:

1. **Collect.** Connect to devices or APIs and gather information — configurations, interface states, routing tables, serial numbers, software versions, licence entitlements.
2. **Transform.** Turn messy text into structured data, filter it, join it against other sources, and calculate things from it.

3. **Generate.** Produce output — configuration files, reports, spreadsheets, diagrams, tickets, emails, chat messages.
4. **Push.** Send configuration or API calls back to devices and controllers, then verify that the result is what you intended.

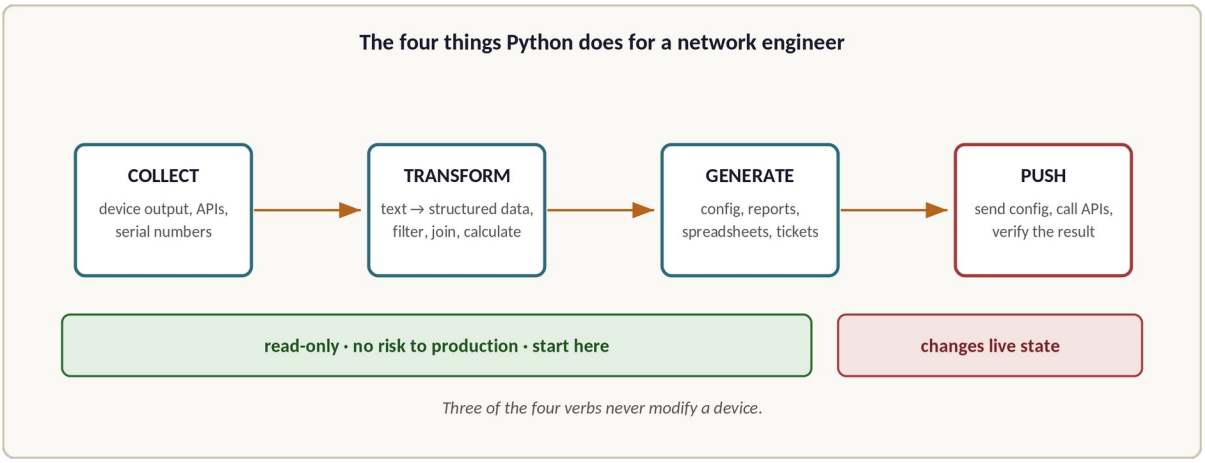

Figure 1.2 — Collect, transform and generate are read-only. Only push changes production state.

Notice that three of the four never touch a production device in a way that can break anything. This is important and frequently overlooked: the majority of valuable network automation is read-only. Collecting an inventory, auditing compliance, generating a report — these carry almost no operational risk, and they are exactly where you should start.

> ◆ **From the Field — Where automation programmes actually begin**
>
> In practice, teams that open with 'let us push configuration automatically' get blocked by change management for months. Teams that open with 'let us produce a daily report of every interface with CRC errors across the estate' ship something useful in a week, build credibility with operations and management, and are trusted with write access later. Start read-only. Earn write access.

⚠ Common Pitfall — Python is not a replacement for networking knowledge

A script that configures OSPF is written by someone who understands OSPF. Python will not tell you that your area design is wrong, that an MTU mismatch will break adjacency, or that your change window collides with a business-critical batch job. Automation amplifies expertise; it does not supply it. Your years of networking experience are precisely the reason you will write better network scripts than a professional software developer would.

Summary

Python matters for network engineers not because it is the most elegant language but because it is where the entire network automation ecosystem lives — Netmiko, NAPALM, Nornir, ncclient, pyATS, Jinja2 and the vendor SDKs are all Python packages. Beyond tooling, Python encourages a shift from thinking in commands to thinking in data, which is the essence of infrastructure as code: the desired state lives in structured files and the running configuration is generated from it. Practically, Python does four things — collect, transform, generate and push — and three of those four are read-only, low-risk, and the right place to begin building both skill and organisational credibility.

Key Takeaways

The ecosystem is the reason — every major network automation library is Python. **Shift from commands to data**: data can be queried, validated and reused; configuration text cannot. **Infrastructure as code** means configuration is a derived artefact generated from version-controlled data. **The four verbs are collect, transform, generate, push. Start read-only** — reporting and auditing deliver value with almost no operational risk. **Python amplifies networking expertise, it does not replace it.**

1.3 What Python Is Not

Every technology book has a chapter that oversells its subject. This is the section that does the opposite. Knowing the limits of a tool is what separates an engineer from an enthusiast, and there are several things Python is genuinely not good at, or simply not the right choice for, in a network context.

Python is not fast, and that is usually fine

Python is an interpreted language. The code you write is executed line by line by the Python interpreter rather than compiled to machine code ahead of time. That makes it perhaps thirty to fifty times slower than C for raw computation.

In network automation this almost never matters, because your scripts are not CPU-bound — they are network-bound. A script that connects to 180 switches spends 99.9 percent of its wall-clock time waiting for SSH sessions to establish and for devices to respond. The Python interpreter is idle during that wait. Making the interpreter faster would change essentially nothing.

> **★ Did you know? — Where the time actually goes**
> A typical Netmiko session to a Cisco IOS switch takes roughly 3 to 8 seconds to establish, and each `show` command adds 0.5 to 2 seconds on top. Executing the Python that issues those commands takes microseconds. This is why the technique that genuinely speeds up network scripts is **concurrency** — talking to many devices at the same time — rather than optimising the Python itself. Chapter 10 covers exactly this.

Python is not a configuration management system

Python is a general-purpose programming language. Ansible, Terraform and Cisco NSO are configuration management and orchestration systems. They provide things that a bare Python script does not: idempotency, an inventory model, state tracking, a declarative syntax, and an established plugin ecosystem for handling credentials and secrets.

You can build all of that in Python, and Nornir in particular gets you a long way down that road. But if your requirement is 'apply this standard configuration to every device and keep it that way indefinitely', a purpose-built tool is often a better answer than a bespoke script that you alone maintain.

Requirement	Better answer	Why
Ad-hoc data gathering	Python	Total flexibility, no framework overhead
Custom report or spreadsheet	Python	Nothing handles arbitrary output as well
Enforce standard config on a fleet	Ansible / NSO	Idempotency and state tracking built in
Provision cloud network resources	Terraform	Declarative state file and drift detection
Complex conditional logic	Python	Real branching beats YAML gymnastics
Glue between systems and APIs	Python	Best-in-class library support
One-off migration with many exceptions	Python	Exceptions are what a language is for

◆ From the Field — The pragmatic hybrid

Most mature teams do not choose one or the other. They use Ansible or Terraform for the repeatable, declarative parts of the estate, and Python for everything that does not fit that model — custom reports, one-off migrations, reconciling an IPAM against a monitoring system, and the scripts that generate the Ansible inventory in the first place. Learning Python also makes you better at Ansible, because Ansible's harder features are Python underneath.

Python is not a substitute for a change process

A script does not exempt you from peer review, change tickets, maintenance windows or rollback plans. If anything the opposite is true: because a script acts on many devices simultaneously, it deserves more scrutiny than a manual change, not less. Chapter 17 covers structuring a script so it can be reviewed, dry-run and staged before anyone trusts it with production.

> ⚠ **Common Pitfall — The most dangerous script is the one nobody reviewed**
> Never run a configuration-push script for the first time against production. Run it against one lab device, then one non-critical production device, then a small batch, then the fleet. Almost every serious automation incident in the industry follows the same pattern: a script that worked in testing met a device that was slightly different, and nobody had built in a check for it.

Python will not make your network data correct

This one catches almost everyone. Automation depends on knowing what devices exist, what they are called, what their management addresses are, and who owns them. Most organisations discover — at the exact moment they first try to automate — that their inventory spreadsheet is out of date, their DNS entries are stale, and three devices in the list were decommissioned in 2021.

This is not a Python problem, but it will very quickly become your problem. The good news is that Python is an excellent tool for fixing it: a script that walks the network via CDP and LLDP and compares what it finds against the inventory spreadsheet is one of the highest-value first projects you can possibly build. We build exactly that in Chapter 15.

Python is not always allowed

A final practical limit worth naming early. In some organisations, installing Python on a corporate workstation requires approval, package installation from PyPI is blocked by a proxy, and jump servers are locked down. None of these are technical problems, but they are real and they will shape how you work. Chapter 3 covers offline installation from a local wheel repository and

running scripts from a restricted jump host, because pretending these constraints do not exist would not serve you.

Summary

Python is interpreted and therefore slow at raw computation, but network scripts are bound by device response time rather than CPU, so this rarely matters — concurrency is the meaningful optimisation. Python is a general-purpose language rather than a configuration management system, so for enforcing declarative standard configuration across a fleet, Ansible, Terraform or NSO may fit better; mature teams use both and let Python handle everything that does not fit a declarative model. A script does not replace change process or peer review — it raises the stakes — and it cannot compensate for inaccurate inventory data, though it is the best available tool for discovering and correcting that data. Finally, corporate restrictions on installation and package access are a real constraint that Chapter 3 addresses directly.

Key Takeaways

Python is slow at computation and it does not matter — network scripts wait on devices, not on the CPU. **Concurrency, not micro-optimisation, is what speeds up network scripts. Ansible and Terraform beat bespoke Python for declarative, idempotent fleet configuration. Python excels at glue, custom reporting, and anything with real conditional logic. Automation increases the need for review and staged rollout, it does not reduce it. Bad inventory data breaks automation** — but Python is the best tool for fixing bad inventory data. **Corporate lockdown is a real constraint**; plan for offline installs.

1.4 How to Use This Book

This book is built to be worked through in order, with your hands on a keyboard. Reading it like a novel will teach you vocabulary but not skill. Every chapter contains code that you are expected to type, run, break and fix.

The structure

Part	Chapters	What you get out of it
I — Setup	1 – 3	Python installed and a working environment on your OS
II — Fundamentals	4 – 9	The Python language, taught entirely on network data
III — The Network	10 – 12	Netmiko, Paramiko, REST, NETCONF, NAPALM, Nornir, pyATS
IV — Templates	13	Jinja2 and data-driven configuration generation
V — Use Cases & AI	14 – 17	Operational scripts, off-box tooling, AI-assisted workflows

Parts II onwards assume you have completed the part before. If you already know Python fundamentals you may skim Part II, but do at least read Chapter 9 on parsing — regular expressions and TextFSM are where most experienced network engineers have their largest gap, because parsing is a networking-specific application rather than general Python.

What you need in order to follow along

Deliberately, almost nothing. The majority of examples in this book run entirely on your own machine using sample data files that you create as you go — saved command output, CSV inventories, YAML variable files. This is a design decision rather than a compromise: it means you

can practise on a train, in a hotel, or at 23:00 without VPN access, which is the difference between finishing a book and abandoning it.

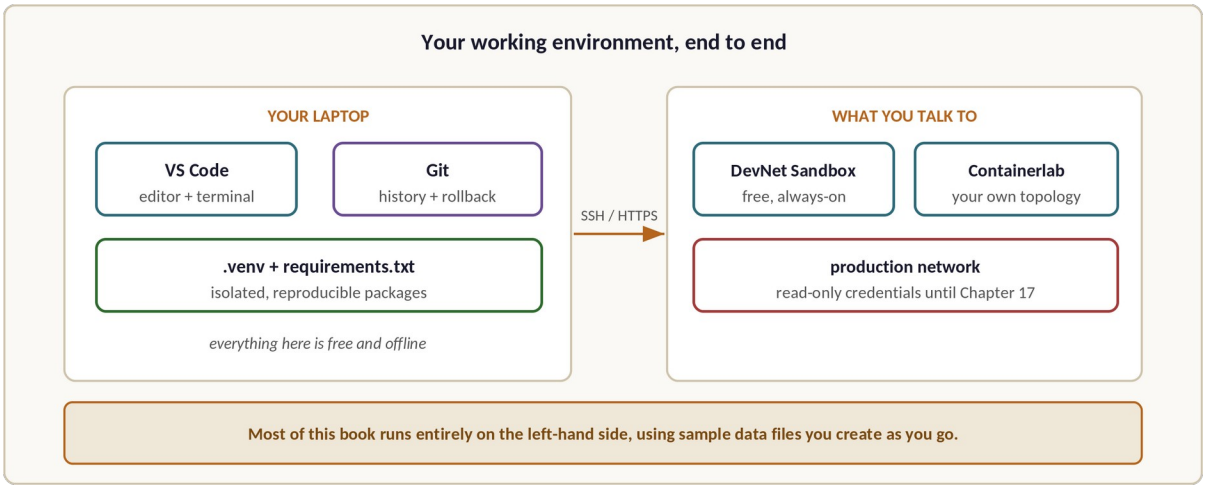

Figure 1.3 — Most of this book runs on the left-hand side. Real devices are optional until Part III.

For the chapters that genuinely need a device, you have three options, in increasing order of effort:

5. **Cisco DevNet Sandbox.** Free, always-on virtual devices reachable over the internet with published credentials. Requires only a free Cisco account. This is the best starting point for the Netmiko, RESTCONF and NETCONF chapters.
6. **Containerlab, GNS3 or EVE-NG.** Run virtual routers and switches on your own machine or on a lab server. More setup effort, but you get full control and you can break things freely, which is exactly what you want while learning.
7. **Your own lab, or read-only production access.** If you have real gear, use it — but with read-only credentials only, until you reach Chapter 17 and understand staged rollout.

> **✔ Pro Tip — Create a dedicated practice folder now**
>
> Make a folder called `net-automation` somewhere sensible on your machine — not on your desktop, and not inside a synced cloud folder that will fight you over file locks while a script is writing. Every script in this book goes there. By Chapter 17 you will have a genuinely useful personal toolkit, and you will want it under version control.

How each section is laid out

Every numbered section follows the same rhythm, so you always know where you are:

- **Explanation from first principles.** Jargon is defined the first time it appears. No prior programming knowledge is assumed anywhere in the book.
- **Runnable code with a caption.** Every code block is complete and works as printed. Where a block depends on a data file, that file's contents are shown too, so you never have to guess.
- **Value boxes.** Did you know (a deeper mechanism), Pro Tip (a practical technique), Common Pitfall (a mistake to avoid), From the Field (how it plays out in production), Try This (a small exercise).
- **Summary and Key Takeaways.** A prose recap and a dense bolded list, designed for revision rather than first reading.
- **Hands-On Practice.** Every chapter closes with exercises that build on what you have just read. Worked solutions to all of them are in Appendix A.

How to read the code in this book

Four conventions appear in every code block from here on. None of them is Python you type; they are marks that tell you what you are looking at.

The four things you will see, and what each means

```
# This line is a comment. Everything after a # is ignored by Python.
# Comments explain what the code is doing. You never have to type them,
# and in this book there are a great many of them on purpose.

hostname = "lab-sw-01"      # a comment can also sit at the end of a line



>>> hostname                # >>> is the interactive prompt. It is NOT typed —
'lab-sw-01'                 # it marks a line you type at the >>> prompt, and
                            # the line under it is what Python answered.


>>> commands = [
...     "vlan 340",         # ... means "this is a continuation of the line
... ]                       # above", again shown by Python, not typed by you
```

```
print("a very long line that does not fit the width of this printed page and
    ↳ therefore continues here")   # ↳ means the line was too wide for the
                                   # page and was wrapped. In your editor it
                                   # is one single line.
```

So: lines beginning with `>>>` or `...` are the interactive interpreter, and you type only what follows the prompt. Everything after a `#` is a comment, written for you rather than for Python. And a `↳` at t he start of a line means the previous line was too long for the page and continues — in a file it would be one line.

> **✔ Pro Tip — Blocks with no >>> are files, not the prompt**
>
> When a code block has no `>>>` anywhere in it, it is the contents of a file. Save it with the name given in the small italic caption above the block — `scripts/check_env.py`, for example — and run it with `python scripts/check_env.py`. That distinction between 'type this at the prompt' and 'save this as a file' is the one thing that confuses people in their first week.

A note on typing versus copying

You will learn substantially more by typing the code examples than by copying them. Typing forces you to read every character, and the errors you make while typing — a missing colon, a mismatched bracket, wrong indentation — are precisely the errors you need to learn to recognise and fix quickly. Copy-paste teaches you nothing about debugging, and debugging is most of the job.

> **▶ Try This — Your commitment for this book**
>
> Type every code block under twenty lines; copy the longer ones. When a script errors, read the error message properly before doing anything else. Python's error messages are unusually good, and learning to read them fluently is genuinely half of learning to program.

◆ From the Field — Why this book ends with AI

Chapter 16 covers calling AI models and APIs from Python, and writing prompts that produce useful network scripts. It is placed at the end deliberately. AI is an extraordinary accelerator for an engineer who can read and validate the code it produces, and a serious liability for one who cannot. By Chapter 16 you will be the former, and the chapter will make you considerably faster rather than dependent.

Summary

This book is a hands-on course in five parts: setup, Python fundamentals taught on network data, the libraries that talk to devices, templating, and real-world use cases including AI-assisted workflows. Almost every example runs offline on your own machine using sample data, with Cisco DevNet Sandbox, Containerlab or your own lab available for the chapters that need real devices. Each section follows a consistent rhythm — first-principles explanation, runnable code, value boxes, a summary and a five-question knowledge check — and you will learn far more by typing the examples and debugging your own mistakes than by copying them. The AI chapter comes last, once you are able to validate what a model produces.

Key Takeaways

Work through the book in order, with a keyboard. **Most examples run entirely offline** on sample data you create as you go. **Cisco DevNet Sandbox is the lowest-effort route** to real devices when you need them. **Create a dedicated practice folder now** — it becomes your personal toolkit. **Type short code blocks rather than copying them** — the typos you make are the debugging skills you need. **Read Python error messages properly** before doing anything else. **AI comes last** because you must be able to validate what it produces.

Chapter 1 — Hands-On Practice

There is no code in this chapter, so the practice is analytical. Twenty minutes here will shape which scripts you build first, and picking the right first project matters far more than picking the right first library.

Exercise 1 — Find your own break-even

Open a blank document and list the three tasks you have personally performed most often in the last six months. For each one, write down four numbers:

8. How many devices it touches on a typical run.
9. How many minutes it takes per device, honestly, including the checking.
10. How many times you have done it in the last year.
11. How long you think a script would take to write — then double that figure, because everyone underestimates.

Multiply the first three numbers. Compare the result with the fourth. In almost every case one of your three tasks will be dramatically over the line, and that task is your first automation project.

Exercise 2 — Classify them by risk

Take the same three tasks and sort each into one of the four verbs from section 1.2 — collect, transform, generate or push. If your highest-value task is a push, look for a collect or transform task in your list instead and start there. You want your first script to be one that cannot possibly cause an outage.

Exercise 3 — Write the specification

For the task you selected, write a plain-English description precise enough that a colleague who has never done it could follow it exactly. Include what happens when a device does not respond, what happens when the output looks unexpected, and how you would verify the result afterwards.

If you cannot write those three cases down, you are not ready to automate that task yet — and discovering that now, on paper, is considerably cheaper than discovering it at 02:00. This document becomes the comment block at the top of your script in Chapter 7.

> ✔ **Pro Tip — Keep the answers**
>
> Come back to this page after Chapter 14. By then you will have the tools to build the task you chose, and you will have a written specification waiting for you. A surprising number of engineers finish a Python book without ever building anything for their own network; this exercise is designed to make sure you are not one of them.

Chapter 2 — Installing Python on Windows, macOS and Linux

Installing Python is genuinely easy, and it is also where a surprising number of people give up. Not because the installer is difficult, but because of one checkbox, one version confusion, and one error message that appears the first time you open a terminal. This chapter deals with all three properly, on all three operating systems.

By the end of this chapter you will have a working Python interpreter, you will know which version you have and why it matters, you will be able to run a Python file from a terminal, and — most importantly — you will understand what actually happens when you type `python` at a prompt. That last piece of understanding is what lets you fix your own environment problems for the rest of your career, instead of searching for the same Stack Overflow answer every six months.

Section	What you will do	Applies to
2.1	Choose the right Python version and understand the numbering	Everyone
2.2	Install and verify Python on Windows	Windows
2.3	Install and verify Python on macOS	macOS
2.4	Install and verify Python on Linux	Linux
2.5	Confirm the install works, and fix it when it does not	Everyone

> ✔ **Pro Tip — Read 2.1 and 2.5, then only your own operating system**
> Sections 2.2, 2.3 and 2.4 are alternatives, not a sequence. Read the one that matches your machine. If you use Windows at work and a Mac at home — a very common combination for network engineers — read both, because the differences will otherwise bite you when you copy a script between them.

2.1 Choosing Your Python Version

Before installing anything, five minutes on versions will save you hours later. Python's version numbering is simple once explained, and almost every 'this script does not work on my machine' problem in a networking team traces back to a version mismatch that nobody documented.

Python 2 versus Python 3

You may still encounter references to Python 2 in older documentation, on older network devices, and in scripts a colleague wrote in 2016. Python 2 reached end of life on 1 January 2020. It receives no security updates, most libraries have dropped support for it, and none of the tooling in this book works with it. There is no scenario in which you should learn Python 2 today.

The practical consequence is a naming quirk you will meet on macOS and Linux: because both operating systems historically shipped Python 2 as `python`, the version 3 interpreter is usually called `python3`. On Windows the command is simply `python`, or the launcher `py`. This inconsistency is the single most common source of confusion for beginners, and it explains why examples you find online sometimes say one and sometimes the other.

Command	Windows	macOS	Linux
`python`	Python 3 (correct)	may not exist, or is Python 2	may not exist
`python3`	usually works too	Python 3 (correct)	Python 3 (correct)
`py`	the launcher (recommended)	not available	not available
`pip`	works	prefer `pip3`	prefer `pip3`

> ⚠ **Common Pitfall — Do not assume python means Python 3**
>
> On a Mac or a Linux server, typing `python` may give you a Python 2 interpreter, an error, or nothing at all depending on the distribution. Throughout this book, commands are shown for both. When you write scripts for colleagues, always specify which interpreter you mean — in documentation, in the shebang line, and in your README.

How Python version numbers work

A Python version looks like `3.13.2`. The three parts are not decoration:

- **3** is the major version. Changing it means the language itself changed incompatibly. This has happened once, from 2 to 3, and there is no Python 4 planned.
- **13** is the minor version, and this is the number that matters to you. A new minor version arrives every October, adds features, and is supported for five years. Libraries state their compatibility in terms of the minor version — for example 'requires Python 3.9 or later'.
- **2** is the patch version — bug fixes and security patches only. Take the latest patch you can get; it never breaks anything.

Which version should you actually install?

The rule that serves network engineers best is: install the latest stable minor release, minus one. At the time of writing that means if 3.13 is the newest, install 3.12 or 3.13, but do not rush to a version released last month.

The reason is library lag. Netmiko, NAPALM, Nornir and especially pyATS depend on compiled components and on each other. When a brand-new Python minor version appears, it typically takes two to four months before every library in the network automation stack publishes a compatible build. Being one version behind costs you nothing and avoids the problem entirely.

Situation	Recommended version	Reason
Learning from this book	Latest, or latest minus one	Everything in the book works on

Situation	Recommended version	Reason
		3.9+
Corporate workstation	Whatever IT approves, 3.9 or newer	Approval matters more than novelty
Shared automation server	Match what colleagues run	Consistency prevents subtle bugs
Using pyATS or Genie	Check the pyATS docs first	pyATS is the strictest about versions

> ★ **Did you know? — Why October matters**
>
> Python follows an annual release cadence defined in PEP 602: a new minor version every October, with five years of support — roughly eighteen months of bug fixes followed by three and a half years of security-only patches. This predictability is genuinely useful for planning: you can tell your management exactly when the interpreter on your automation server will fall out of support, which is the kind of answer that gets budget approved.

A word about distributions

You will encounter alternatives to the official Python installer. Two are worth knowing about:

- **Anaconda / Miniconda.** A scientific Python distribution with its own package manager, `conda`. Excellent for data science, and genuinely useful if you plan to do heavy analysis of network telemetry. For everyday network automation it adds complexity you do not need, and it can conflict with the standard tooling on a corporate machine.
- **System Python.** The interpreter that ships with macOS and every Linux distribution. It exists to run operating system tooling, not your scripts. You should install packages into it as rarely as possible — Chapter 3 explains exactly why, and what to do instead.

For this book: use the official installer from python.org on Windows and macOS, and your distribution's package manager on Linux. That is the path with the fewest surprises and the most documentation.

Summary

Python 2 is dead and irrelevant; everything in this book requires Python 3. The lasting consequence of that transition is a command-name inconsistency — `python` on Windows, `python3` on macOS and Linux — which is the most common beginner confusion and the reason online examples disagree. Version numbers follow major.minor.patch, and the minor version is the one libraries care about; a new one arrives every October with five years of support. The safest choice for network automation is the latest stable minor version or the one before it, because the automation library ecosystem takes a few months to catch up with each new release. Use the official python.org installer on Windows and macOS, and your package manager on Linux; avoid Anaconda unless you have a specific data-science reason for it, and leave the operating system's own Python alone.

Key Takeaways

Python 2 is end-of-life — never learn or write it. **`python` on Windows, `python3` on macOS and Linux** — the single biggest source of beginner confusion. **The minor version is what libraries care about** (3.12, 3.13), and a new one lands every October. **Install the latest stable minor version or the one before it** — library support lags new releases by two to four months. **Use python.org on Windows and macOS, your package manager on Linux. Do not install your packages into the system Python.**

2.2 Installing Python on Windows

Windows is the most common workstation operating system in enterprise network teams, and it is also the platform where the installation goes wrong most often — because of a single unticked checkbox. Follow these steps exactly and you will not have that problem.

Step by step

1. Go to **python.org/downloads**. The site detects Windows and offers the current stable release. Download the 64-bit installer. If your organisation blocks the download, section 2.2 later in this chapter covers the Microsoft Store and offline options.
2. Run the downloaded `.exe`. The first screen is the important one and it is easy to click past.
3. **Tick 'Add python.exe to PATH' at the bottom of the first screen.** This is the checkbox. It is unticked by default, it is easy to miss, and forgetting it causes the single most common error in this chapter.
4. Tick 'Use admin privileges when installing py.exe' if you have local administrator rights. If you do not, leave it unticked — Python installs perfectly well into your user profile without admin rights, which matters on locked-down corporate machines.
5. Click **Install Now** for the default installation, which includes pip, the IDLE editor, the `py` launcher and the standard library. This is what you want.
6. When the installer finishes, if it offers **'Disable path length limit'**, click it. Windows has a legacy 260-character path limit that occasionally breaks deeply nested package installations. Removing it costs nothing.

> ⚠ **Common Pitfall — The checkbox is the whole chapter**
> If you forget 'Add python.exe to PATH', Python installs correctly but your terminal cannot find it, and you get `'python' is not recognized as an internal or external command`. You do not need to reinstall: re-run the same installer, choose **Modify**, and on the Advanced Options page tick 'Add Python to environment variables'. Then close and reopen every terminal window — an already-open terminal keeps the old PATH.

What PATH actually is

Since this single concept causes most installation failures on every operating system, it is worth understanding rather than memorising. PATH is an environment variable holding an ordered list of folders. When you type a command, the shell looks in each of those folders in order and runs the first matching program it finds. It does not search your whole disk.

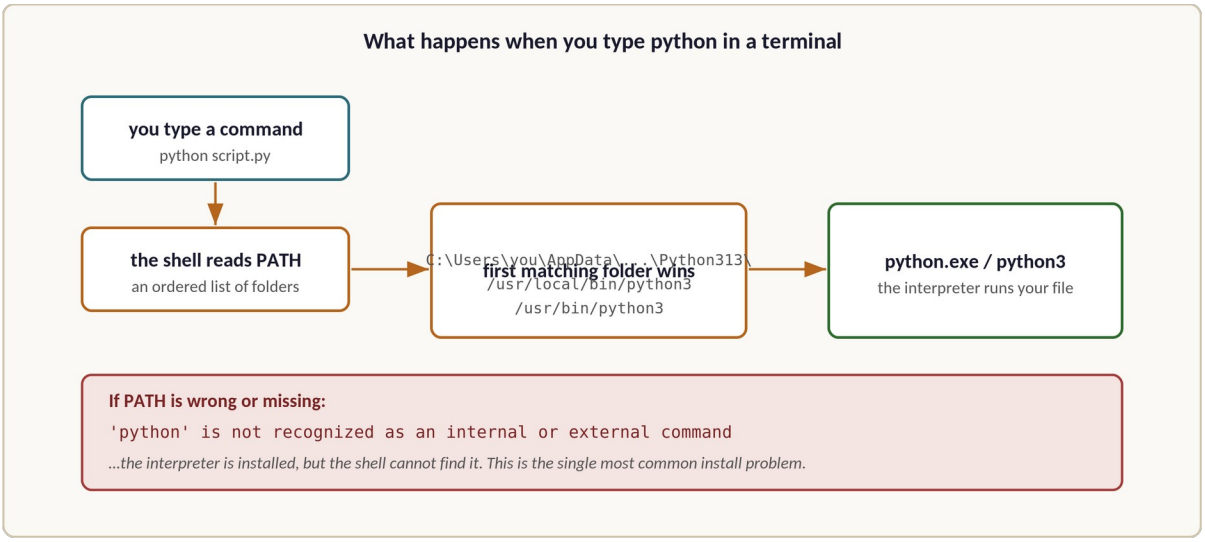

Figure 2.1 — Typing `python` starts a search through the folders listed in PATH. If the interpreter's folder is not in that list, the command fails even though Python is installed.

This is why the error message says the command is not recognised rather than saying Python is missing: from the shell's point of view, those are the same thing. It also explains why the order matters — if two Python versions are both on PATH, the one whose folder appears first wins.

Verifying the installation

Open a **new** PowerShell or Command Prompt window — new, because an already-open window still holds the PATH from before the install. Then run these three commands.

PowerShell — confirm the interpreter, the launcher and pip are all reachable

```
python --version
py --version
pip --version
```

Expected output (your version numbers may differ, that is fine)

```
Python 3.13.2
Python 3.13.2
pip 25.0 from C:\Users\you\AppData\Local\Programs\Python\Python313\Lib\site-packag
    ↳ es\pip (python 3.13)
```

If all three respond, you are done and you can move to section 2.5. If any of them produce an error, section 2.5 has a diagnosis table that covers every common case.

The py launcher, and why Windows users should use it

Windows installs a small program called the Python launcher, invoked as `py`. It is genuinely useful and most tutorials never mention it. The launcher knows about every Python version installed on the machine and lets you choose between them without touching PATH at all.

PowerShell — the launcher's most useful invocations

```
py -0                   # list every Python version installed on this machine
py                      # run the newest installed version
py -3.12                # run a specific version
py -3.12 -m pip install netmiko    # install a package into that specific version
py backup_configs.py    # run a script with the default version
```

> **★ Did you know? — Why py -m pip is safer than pip**
>
> Running `pip install netmiko` uses whichever `pip` appears first on PATH — which, on a machine with several Python versions, may not belong to the interpreter you think you are using. Writing `py -m pip install netmiko` asks a **specific interpreter** to run its own pip module, so the package definitely lands where you expect. The equivalent on macOS and Linux is `python3 -m pip install netmiko`. Get into this habit now; it eliminates an entire category of confusing bug.

Alternatives when python.org is blocked

Corporate environments frequently block installer downloads. There are three realistic routes, and it is worth knowing all three before you raise a ticket.

Route	How	Trade-off
Microsoft Store	Search 'Python 3.x' in the Store app	Sandboxed; PATH handled for you, but some tools struggle with it
Winget	`winget install Python.Python.3.13`	Command line, no browser needed; may itself be blocked
Offline installer	IT downloads the `.exe` once and hosts it internally	Needs a ticket, but is the most defensible option

> ◆ **From the Field — How to phrase the request to IT**
>
> Asking for 'Python' sometimes triggers a security review. Asking for 'the standard Python interpreter from python.org, to run internal network operations scripts, installed into my user profile without administrator rights' usually does not — because it is specific, it names the source, and it makes clear you are not requesting elevated privileges. Mention that Cisco, Juniper and Fortinet all ship Python-based tooling; it reframes the request as vendor-aligned rather than personal.

Where things actually get installed

Knowing the layout helps enormously when something goes wrong. A default per-user Windows install looks like this:

Typical layout of a per-user Windows installation

```
C:\Users\you\AppData\Local\Programs\Python\Python313\
    python.exe              <- the interpreter itself
    Lib\
        site-packages\      <- where pip installs third-party libraries
    Scripts\
        pip.exe
        netmiko-related command line tools appear here too

C:\Windows\py.exe           <- the launcher, available system-wide
```

▶ Try This — Find your own installation

Run `py -c "import sys; print(sys.executable)"` in PowerShell. It prints the full path of the interpreter that just ran. Then run `py -c "import sys; print(sys.path)"` to see every folder Python searches for libraries. Five seconds of output here explains most import errors you will ever hit.

Summary

On Windows, download the 64-bit installer from python.org and — the critical step — tick 'Add python.exe to PATH' on the first screen before clicking Install Now. Forgetting it produces the classic 'python is not recognized' error, fixed by re-running the installer in Modify mode and reopening your terminal. PATH is simply an ordered list of folders the shell searches, which explains both the error message and why version order matters. Verify with `python --version`, `py --version` and `pip --version` in a newly opened terminal. Windows users should adopt the `py` launcher and the `py -m pip install` form, because it targets a specific interpreter and eliminates a whole class of 'installed but not found' problems. When python.org is blocked, the Microsoft Store, winget, or an internally hosted offline installer are the realistic alternatives.

Key Takeaways

Tick 'Add python.exe to PATH' — this one checkbox causes most Windows install failures. **Open a new terminal after installing**; existing windows keep the old PATH. **PATH is an ordered folder list** — first match wins, and nothing outside it is searched. **Verify with `python --version`, `py --version`, `pip --version`. Use the `py` launcher**: `py -0` lists versions, `py -3.12` selects one. **Always prefer `py -m pip install X`** over bare `pip install X`. **Admin rights are not required** — a per-user install works fine.

The sample ends here

You have just installed Python and run your first command. That is the point where most people put a book down and never come back to it, because nothing they have learned yet does anything useful at work. The rest of this book exists to fix that, and it does so in the order an engineer actually needs.

WHAT COMES NEXT

Part I — Getting Set Up (chapters 1 to 3)

Finish the install, choose an editor, understand virtual environments and pip, and get a lab you can break safely.

Part II — The Python You Actually Need (chapters 4 to 9)

Variables, lists and dictionaries, loops, functions, files, CSV, JSON, YAML and error handling — every example built on interface tables, inventories and show output.

Part III — Talking to the Network (chapters 10 to 12)

Paramiko and Netmiko over SSH, REST and RESTCONF, NETCONF with ncclient, NAPALM across vendors, and Nornir when one device becomes four hundred. Concurrency, retries and credentials done properly.

Part IV — Templates and Real Use Cases (chapters 13 to 15)

Jinja2 configuration generation, golden configs and drift detection, then off-box work: inventory reconciliation, IP address maths, Excel reports and KeePass.

Part V — AI and Where to Go Next (chapters 16 and 17)

The Claude and OpenAI APIs, running a model locally, prompt patterns that generate network scripts worth keeping, and how to put your code under version control and test it.

What the full edition includes

- 632 pages, seventeen chapters in five parts, written and typeset as a book rather than a pile of blog posts.
- Every listing is complete and runnable. Nothing is left as an exercise for the reader unless it is meant to be.
- Code commented for someone who has never programmed — and only where a thing is new, so the listings never become noise.
- Hands-On Practice at the end of every chapter, with worked solutions to all of them in Appendix A.
- Appendix B: a glossary of every term used, plus a quick reference for Netmiko, NAPALM, Nornir and Jinja2.
- Appendix C: a full index, and a clickable contents in the PDF.
- Vendor coverage is Cisco-first, with Juniper and Fortinet examples where the difference matters.

WHO WROTE IT

Jozef Baroš is a network architect and team leader working in enterprise networking, with certifications including CCNP Enterprise and CCNP Security. This book is the course he wanted when he started automating his own work: no computer science, no toy examples, just the Python that removes real hours from a network engineer's week.

PYTHON FOR

NETWORK ENGINEERS

From Zero to Automated Networks

Jozef Baroš