Python for Beginners: A DevOps Starter Guide

Preface: The Automator's Mindset

"I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it." — Bill Gates

Why Python for DevOps?

You typically aren't here to build the next social media giant. You are here because **infrastructure Management is demanding**:

- Logs require automated parsing.
- Backups require distinct verification.
- APIs require programmed querying.
- Configurations require dynamic generation.

While you *could* accomplish this in Bash, scripts often become unmaintainable after 50 lines. You *could* use C++, but system administration does not afford the luxury of long development cycles.

Python is the sweet spot. It is readable, powerful, and installed everywhere. It is the language of Cloud Automation (Ansible, SaltStack), Data (Pandas), and AI (PyTorch).

What actually is "DevOps Programming"?

It's not about complex algorithms for 3D graphics. It's about **Glue**. Connecting System A to System A. Taking output from a Database and formatting it for a Slack Alert.

It requires a specific mindset:

1. **Paranoia:** "What if the network fails?" (Error Handling)
2. **Laziness:** "I shouldn't have to type this twice." (Functions/Loops)
3. **Efficiency:** "I can't load a 100GB log into RAM." (Streams/Generators)

The Journey Ahead

This book is structured to take you from **Zero to DevOps Hero**:

1. **The Foundation (Part 1):** We start with the absolute basics. Variables, Loops, and Logic.
2. **The Toolkit (Part 2):** We move to Systems thinking. Files, Errors, and OOP.
3. **The Theory (Part 3):** We dive into Algorithms. Big O, Recursion, and Data Structures.
4. **The Automation (Part 4-5):** We build real tools. CLI apps, Docker controllers, and Cloud scripts.
5. **The Career (Part 6):** We ensure you are job-ready with checklists and roadmaps.

This Book's Promise

We won't treat you like a child. We won't use metaphors about "Space Ships" or "Pizza Shops" unless they help explain a technical concept. We will treat you like an Engineer who needs to solve problems.

How to read this book

1. **Don't just read.** Type the code.
2. **Break things.** Change permissions. Delete config files. See how your script reacts.
3. **Automate your life.** As you learn, look at your daily tasks. Can Python do them for you?

Let's get to work.

Chapter 1: Introduction - Automating the Infrastructure

"The computer is incredibly fast, accurate, and stupid. Humans are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination." — Albert Einstein

The Goal: No More Manual Work

Imagine you are a System Administrator. You have 100 servers to update. **Bad Way:** SSH into every single server, type apt-get update, wait, exit. Repeat 100 times.

The Efficient Approach

Write a Python script to execute the task automatically, freeing you to focus on architecture.

In the DevOps ecosystem:

- **You** act as the Architect.
- **The Servers** act as the Workers.
- **Python** serves as the command language that coordinates them.

1.1 Installing Python

Before we automate, we must install our tools.

Linux (Ubuntu/Debian)

Most Linux distros come with Python, but let's ensure we have the latest version.

```
sudo apt update
sudo apt install python3 python3-pip python3-venv
```

Verify Installation:

```
python3 --version
# Output: Python 3.10.6
```

macOS

Use brew (Homebrew).

```
brew install python
```

Windows

1. Download the installer from python.org.
2. **Crucial Step:** Check the box **"Add Python to PATH"** during installation.
3. Open PowerShell and type python --version.

1.2 Under the Hood: How Python Works

When you write a script (e.g., backup.py), it's just text. How does it control a server?

The Two-Step Translation Process

1. **The Source Code (backup.py):** Human-readable instructions.print("Starting Backup...")

2. **Lexing & Parsing:**
    a. The Python Interpreter reads your script.
    b. It checks for syntax errors (e.g., missing quotes).
    c. *DevOps Note:* This happens *before* the script runs. If you have a syntax error, the script won't even try to touch your servers.
3. **Bytecode Compilation:**
    a. Python translates your code into .pyc files (Bytecode). These are low-level instructions for the Python Virtual Machine (PVM).
4. **The Python Virtual Machine (PVM):**

   a. The PVM executes the bytecode on the CPU.
   b. *Crucial for DevOps:* The PVM creates a layer of abstraction. You can interpret the same Python script on a Linux Web Server, a Windows Database Server, or a Mac Laptop. It works everywhere.

1.2 Your First Automation: print()

The most basic way to log what your script is doing.

print("System Update Initiated")

What happens:

1. Python finds the built-in function print.
2. It takes the string "System Update Initiated".
3. It sends the characters to **Standard Output (stdout)**. In a terminal, this appears on your screen. In a pipeline, this might go to a log file.

1.3 The 4 Laws of Script Syntax

Servers are exacting; a single typo can cause a script to fail.

1. **The Law of Case Sensitivity:**
   a. print is distinct from Print.
   b. **DevOps Context:** Just as Linux distinguishes between ls and LS, Python demands precise casing.
2. **The Law of Quotes:**
   a. Text strings must be enclosed in quotes.
   b. **Incorrect:** print(Server_01) (Interpreted as a variable named Server_01).
   c. **Correct:** print("Server_01").
3. **The Law of Matching:**
   a. Don't mix quotes.
   b. **Bad:** print("Error detected')
   c. **Good:** print("Error detected")
4. **The Law of Parentheses:**
   a. Python 3 requires parentheses for functions.
   b. **Bad:** print "Done" (Old Python 2 syntax - often found in legacy sysadmin scripts. Avoid!)
   c. **Good:** print("Done")

1.4 Formatting Output (Logging)

Clear logs are essential for debugging.

Printing Multiple Metrics

```
print("CPU:", 45, "Memory:", 2048)
# Output: CPU: 45 Memory: 2048
```

Controlling the Separator (sep)

Useful for generating CSVs or formatted logs.

```
print("2023-10-27", "ERROR", "Disk Full", sep="|")
# Output: 2023-10-27|ERROR|Disk Full
```

Controlling the Ending (end)

Useful for progress bars.

```
print("Uploading", end="...")
print("Done")
# Output: Uploading...Done
```

1.5 Escape Sequences (Special Characters)

Sometimes you need to print special characters like newlines or tabs in your logs.

Sequence	Meaning	Usage	Example Output
\n	New Line	print("Error:\nDisk Full")	Error: Disk Full
\t	Tab	print("Status:\tOK")	Status: OK
\"	Double Quote	print("Process \"nginx\" died")	Process "nginx" died
\\	Backslash	print("C:\\Windows\\System32")	C:\Windows\System32

1.6 Comments: Documentation

Scripts are read more often than they are written. Document your logic. Python ignores anything after #.

```
# Check disk space before backup
# If space < 10GB, alert admin
current_space = 500  # GB
```

Theory Note: The interpreter strips comments during the parsing phase. They take up 0ms of runtime.

1.7 The Interactive Shell (REPL)

Perfect for quick checks without writing a file. Open your terminal and type python3.

```
$ python3
>>> 1024 * 1024
1048576
>>> print("Test connection")
Test connection
```

Use this to test a command before putting it into your main automation script.

[!] Common Mistakes

Error #1: Quote mismatch

```
print("Starting service)
```

Error: SyntaxError: EOL while scanning string literal **Translation:** You forgot to close the quote before the line ended.

Error #2: Misspelling Functions

```
prnt("Hello")
```

Error: NameError: name 'prnt' is not defined **Translation:** Python doesn't know a command called prnt.

Practice Problems

Problem 1: The Log Header

Print a log header row separated by pipes (|). Columns: Timestamp, Level, Message.

Problem 2: The Path

Print this Windows path correctly (handling backslashes): C:\Users\Admin\Scripts

Problem 3: JSON-like Output

Print the following JSON structure using \n and \t:

```
{
  "status": "active",
  "uptime": 99.9
}
```

Chapter Summary

- **Python for DevOps:** It acts as a glue language to control infrastructure.
- **Interpreter:** Python runs code line-by-line via the PVM (portable across OSs).
- **Syntax:** Strict rules for case, quotes, and parentheses.
- **Escape Sequences:** Use \ for special characters in logs/paths.
- **Comments:** Vital for documenting maintenance scripts.

Chapter 2: Variables - Infrastructure Resources

"If you can't measure it, you can't manage it." — Peter Drucker

The Goal: Monitoring Metrics

Imagine you are monitoring a web server. You have constant streams of data:

- CPU Usage: 45%
- Available RAM: 16 GB
- Hostname: "prod-web-01"
- Service Status: Active

You need to store these values so your script can check them (e.g., "If CPU > 90%, send alert").

In Python, **Variables** are the labels for these system resources.

2.1 What is a Variable? Labels, Not Boxes

In DevOps, we often use the container metaphor. However, in Python, variables function closer to **labels** than containers.

```
hostname = "web-01"
```

The Process:

1. Python instantiates a string object "web-01" in memory.
2. Python creates the label hostname.
3. Python binds the label to the object.

If you reassign it:

```
hostname = "db-01"
```

Python simply moves the hostname label to the new object "db-01". The original "web-01" object remains in memory, unreferenced.

2.2 Under the Hood: Memory & Garbage Collection

When you move a label, what happens to the old data?

The Garbage Collector (GC): Python's internal process that acts like a janitor.

1. It scans memory for objects that have **zero** labels attached.
2. If an object (like the old "web-01") has no labels, the GC deletes it to free up RAM.

DevOps Note: This is why Python is "memory safe." You don't have to manually free() memory like in C++, preventing memory leaks in your long-running automation scripts.

2.3 The 4 Basic Data Types (Resource Types)

Different metrics require different data types.

1. Integers (int) - Whole Counts

Used for discrete resources.

```
request_count = 5000
cpu_cores = 8
```

```
failed_logins = 0
```

Feature: Python ints can grow as large as memory allows (no 32-bit overflow limit).

2. Floats (float) - Percentages/Averages

Used for continuous metrics.

```
cpu_load = 45.5
memory_usage = 12.4
uptime_days = 99.9
```

Precision Warning: 0.1 + 0.2 might equal 0.30000000000000004 due to binary floating-point math. For precise financial calculations, use Decimal.

3. Strings (str) - Configuration / Logs

Text data.

```
server_name = "nginx-proxy"
error_msg = "Connection Refused"
api_key = "xy78-bb92"  # Even though it has numbers, it's text!
```

4. Booleans (bool) - Flags

Status checks.

```
is_active = True
maintenance_mode = False
```

2.4 Dynamic Typing: Flexibility vs Risk

Python allows a variable to switch types.

```
config = 10       # int
config = "default" # str (Allowed!)
```

DevOps Pro/Con:

- **Pro:** Great for quick scripts where parsing loose log data is messy.

- **Con:** Dangerous in large systems. Did you expect config to be a number? Now it's text.
- **Fix:** Use **Type Hinting** (modern Python standard):port: int = 8080
  host: str = "localhost"

2.5 Naming Rules: Configuration Standards

Your variables should look like standard configuration keys (Snake Case).

1. **Snake Case (snake_case):**
   a. All lowercase, underscores between words.
   b. **Good:** max_connections, retry_count, admin_email
   c. **Bad:** maxConnections (Java style), MaxConn (Go style).
2. **Descriptive Names:**
   a. **Bad:** t = 5 (What is t? Time? Timeout? Threads?)
   b. **Good:** timeout_seconds = 5
3. **No Keywords:**
   a. Don't name a variable list, str, or min. These are built-in Python tools.
   b. **Bad:** str = "Log line" (You just broke the string converter!)

2.6 The Type Trap (Casting)

Logs always come in as Strings. You cannot do math on strings.

```
# Reading from a config file usually returns text
cpu_limit = "80"
current_usage = 85

# print(current_usage > cpu_limit)
# ERROR: TypeError: '>' not supported between instances of 'int' and 'str'
```

The Fix: Cast (convert) the data first.

```
cpu_limit = int("80")  # Convert text "80" to integer 80
if current_usage > cpu_limit:
   print("ALARM: High CPU")
```

Casting Functions:

- int("404") -> 404 (HTTP Status)
- float("10.5") -> 10.5 (Load Average)
- str(200) -> "200" (Stringified Status)

[!] Common Mistakes

Error #1: The Uninitialized Variable

```
print(cluster_status)
```

Error: NameError: name 'cluster_status' is not defined. **Fix:** Assign a default value first: cluster_status = "unknown".

Error #2: The Math Trap

```
version = "3.10"
new_version = version + 1
```

Error: TypeError. Strings generally cannot be added to numbers.

Error #3: Variable Shadowing

```
id = 5
print(id(x))  # Crash!
```

Analysis: id is a built-in function that gets memory addresses. You overwrote it with the number 5. Never name a variable id. Use user_id or server_id.

Practice Problems

Problem 1: Resource Calculation

- Total RAM: 32 GB
- Used RAM: 14.5 GB Calculate free_ram and print it.

Problem 2: Config Builder

Create three variables: host, port, protocol. Add them together to form a URL string: https://api.server.com:8080 (Hint: Use str(port) if port is an integer).

Chapter Summary

- **Variables:** Labels for system resources (CPU, Memory, Config).
- **Dynamic Typing:** Variables can change type, but be careful.
- **Garbage Collection:** Automatic memory cleanup prevents leaks.
- **Casting:** Always convert textual inputs (logs/configs) to numbers before math.
- **Naming:** Use snake_case (e.g., server_status).

Chapter 3: User Interaction - The CLI Experience

"Tools that don't talk back are dangerous."

The Goal: Interactive Scripts

Most DevOps scripts run silently in the background (cron jobs). But sometimes you need to build **Interactive CLI Tools** for other developers.

- "Which database do you want to restore?"
- "Are you sure you want to delete production? (yes/no)"

3.1 Getting Input: input()

The input() function pauses the script and waits for the admin to type.

```
env = input("Target Environment (dev/prod): ")
print(f"Deploying to {env}...")
```

What happens:

1. Script prints "Target Environment (dev/prod): "
2. Script **BLOCKS** (freezes) until Enter is pressed.
3. Admin types "dev".
4. Variable env becomes "dev".

3.2 Under the Hood: stdin & stdout

In Linux, everything is a file stream.

- **stdin (Standard Input):** Keyboard -> Python
- **stdout (Standard Output):** Python -> Screen

DevOps Trick: You can Pipe data into your script!

```
echo "prod" | python deploy.py
```

Because input() reads from stdin, it will see "prod" and continue without stopping. This is how you automate interactive scripts!

3.3 The Law of Strings: All Input is Text

Even if you type a port number, it comes in as text.

```
port = input("Enter Port: ")  # User types 8080
# type(port) is <class 'str'>
```

The Crash:

```
next_port = port + 1
# TypeError: can only concatenate str (not "int") to str
```

The Fix (Casting):

```
port = int(input("Enter Port: "))
next_port = port + 1  # 8081
```

3.4 Handling Validation (Sanitizing Input)

Admins make typos. If you ask for "yes", they might type " YES ".

```
confirm = input("Delete Database? (yes/no): ")

# Bad
if confirm == "yes": ...

# Good (Sanitized)
clean_confirm = confirm.strip().lower()
if clean_confirm == "yes": ...
```

3.5 The Toolkit: 20+ Essential String Methods

In DevOps, 90% of your work is parsing text (logs, configs, outputs). Python has a massive toolkit for this.

Group 1: The Cleaners

Sanitize your input.

Method	Description	Example	Result
.strip()	Removes whitespace from both ends	" Hi ".strip()	"Hi"
.lstrip()	Removes whitespace from left	" Hi".lstrip()	"Hi"
.rstrip()	Removes whitespace from right	"Hi ".rstrip()	"Hi"
.replace(old, new)	Replaces substring	"v1.0".replace(".", "-")	"v1-0"
.lower()	Converts to lowercase	"Admin".lower()	"admin"
.upper()	Converts to uppercase	"dev".upper()	"DEV"
.title()	Capitalizes first letter of words	"john doe".title()	"John Doe"
.capitalize()	Capitalizes only first letter	"hello world".capitalize()	"Hello world"
.swapcase()	Swaps case	"Hi".swapcase()	"hI"

Group 2: The Inspectors

Check what the string contains (returns True/False).

Method	Description	Example	Result
.startswith(x)	Checks start	"error.log".startswith("err")	True
.endswith(x)	Checks end	"sys.conf".endswith(".conf")	True
.isdigit()	Checks if only numbers	"123".isdigit()	True
.isalpha()	Checks if only letters	"abc".isalpha()	True
.isalnum()	Checks if letters+numbers	"web01".isalnum()	True
.isspace()	Checks if only whitespace	" ".isspace()	True
.islower()	Checks if lowercase	"abc".islower()	True
.isupper()	Checks if uppercase	"ABC".isupper()	True

Group 3: The Searchers & Splitters

Find data inside strings.

Method	Description	Example	Result
.find(x)	Returns index of x (or -1)	"Error: Disk".find(":")	5
.count(x)	Counts occurrences	"10.0.0.1".count(".")	3
.split(x)	Splits into a list	"a,b,c".split(",")	['a', 'b', 'c']
.splitlines()	Splits by newline	"Line1\nLine2".splitlines()	['Line1', 'Line2']

.join(list)	Joins list into string	"-".join(['a', 'b'])	"a-b"
.zfill(width)	Pads with zeros	"5".zfill(3)	"005"
.center(width)	Centers text	"Menu".center(10)	" Menu "

3.6 Use F-Strings for Logging

Building log messages with + is slow and ugly. Use **f-strings** (formatted string literals).

Bad:

```
print("Deploying version " + str(ver) + " to server " + host)
```

Good:

```
print(f"Deploying version {ver} to server {host}")
```

DevOps Pro Tip: You can format numbers too!

```
success_rate = 0.9567
print(f"Success Rate: {success_rate:.1%}")
# Output: Success Rate: 95.7%
```

3.7 The 5 Laws of Input

1. **The Law of Casting:** Input is *always* a string. Cast it (int()).
2. **The Law of Hygiene:** Always .strip() input.
3. **The Law of Skepticism:** Never trust input. Validate it.
4. **The Law of Clarity:** Use f-strings.
5. **The Law of Resilience:** Expect the user to hit Enter without typing anything.

[!] Common Mistakes

Error #1: The Hanging Script

You wrote input() in a script meant for a cron job. **Result:** The script runs forever, waiting for a ghost to type something. **Fix:** Use command-line arguments (sys.argv) or environment variables for non-interactive scripts.

Error #2: The Newline Trap

When reading from a file, readline() returns "server01\n". If you don't .strip(), your code breaks:

```
if line == "server01":  # False because of \n
```

Practice Problems

Problem 1: The Log Parser

Given a log line: "[ERROR] Connection Refused - 10.0.0.5".

- Use methods to extract the IP address.
- Check if it starts with [ERROR].

Problem 2: The ID Generator

Ask for a username (e.g., " john ").

- Strip whitespace.
- Convert to lowercase.
- Replace spaces with dots.
- Pad with zeros to make it at least 8 chars long (zfill won't work perfectly on strings, try .rjust or logic!).

Chapter Summary

- **input()**: Reads from stdin (Keyboard or Pipe).
- **Sanitization:** The 20+ methods are your toolkit for cleaning messy data.
- **Casting:** Always convert numeric input (int()).
- **Streams:** Understanding stdin helps you automate automation.

Chapter 4: Logic - The Decision Engine

"A smart script is one that knows when to stop."

The Goal: Automated Decisions

Your deployment script is running.

- **IF** the tests pass -> Deploy to Prod.
- **ELSE** -> Rollback and Slack the team.

This is **Flow Control**. It turns a dumb list of commands into an intelligent agent.

4.1 Boolean Logic (Status Checks)

In DevOps, everything is a status check. Pass (True) or Fail (False).

```
build_success = True
server_online = False
```

Comparison Operators

Operator	Meaning	DevOps Example
==	Equal	status_code == 200
!=	Not Equal	error_count != 0
>	Greater	latency_ms > 500
<	Less	disk_usage < 90
>=	Greater or Equal	uptime_days >= 30
<=	Less or Equal	packet_loss <= 1

4.2 The if Statement (The Gatekeeper)

Execute code only if a condition is met.

```
http_status = 500

if http_status == 500:
    print("CRITICAL: Server Error")
    restart_service()
```

Indentation is Law: Python needs those 4 spaces to know what code belongs to the "Critical" block.

4.3 The else Clause (The Backup Plan)

```
ping_result = "OK"

if ping_result == "OK":
    print("Host is up")
else:
    print("Host is down! Paging on-call...")
```

4.4 The elif Ladder (Multiple Statuses)

```
code = 404

if code == 200:
    print("Success")
elif code == 404:
    print("Not Found")
elif code == 500:
    print("Server Error")
else:
    print(f"Unknown Status: {code}")
```

The One-Path Rule: Only ONE block runs. Even if code matches multiple conditions, Python stops after the first True.

4.5 Advanced Operators (The Power Tools)

Beyond and/or, typical DevOps scripts leverage these advanced tools.

Logical Operators (and, or, not)

These combine boolean conditions.

- **Why not && and ||?** Python prioritizes readability (English words).

- **Short-Circuiting:**
    - A and B: If A is False, B is never checked. (Safety mechanism).
    - A or B: If A is True, B is never checked.

Bitwise Operators (&, |, ^, ~)

These work on **bits** (1s and 0s). Rarely used for logic, but CRITICAL for **File Permissions (chmod)**.

Operator	Name	DevOps Use Case
&	Bitwise AND	Checking permission masking (e.g., mode & 0o777).
`	`	Bitwise OR
^	Bitwise XOR	Toggling bits.
~	Bitwise NOT	Inverting bits.
<<	Left Shift	Multiplying by 2 (binary shift).
>>	Right Shift	Dividing by 2.

Example: Checking Permissions

```
READ = 4   # 100
WRITE = 2  # 010
EXEC = 1   # 001

current_perm = 6  # 110 (Read + Write)

# Check if Write is allowed (Bitwise AND)
if current_perm & WRITE:
   print("Writable!")
```

The Identity Operator (is)

Checks if two variables point to the **same object in memory**.

```
a = [1, 2]
b = [1, 2]

print(a == b)  # True (Values are equal)
print(a is b)  # False (Different memory addresses)
```

Usage: Only use is for None. (if x is None:).

The Walrus Operator (:=)

Python 3.8+ feature. Assigns a value *inside* an expression.

```
# Old way
data = get_data()
if data:
   process(data)

# Walrus way
if (data := get_data()):
   process(data)
```

Benefit: Saves lines of code in looping through file chunks.

4.6 The Rules of Logic

1. **Indentation is Law:** 4 spaces. No tabs.
2. **Comparison (==) vs Assignment (=):**
    a. if cpu = 100: (SyntaxError)
    b. if cpu == 100: (Correct)
3. **The Colon:** Don't forget the : at the end.
4. **Boolean Simplicity:** Don't check if x == True. Just use if x.

[!] Common Mistakes

Error #1: Confusing & and and

```
if user_exists & password_correct:
```

Issue: & is bitwise. It works on numbers/sets. If you use it on Booleans, it works but it is NOT short-circuited. **Fix:** Always use and for logic.

Error #2: Indentation

```
if True:
print("Run")
```

Fix: Indent.

Practice Problems

Problem 1: The Permission Checker

Given a permission code 7 (RWX), use bitwise & to check if EXEC (1) is enabled.

Problem 2: Load Balancer Logic

Write an if/elif/else chain:

- Requests < 100: "Low"
- 100-500: "Normal"
- 500: "High"
- **Bonus:** Use the Walrus operator to simulate reading the request count.

Chapter Summary

- **Decisions:** Scripts adapt using if/elif/else.
- **Operators:** and/or for logic, &/| for permissions (bitwise).
- **is vs ==:** Memory identity vs Value equality.
- **DevOps Context:** Essential for monitoring thresholds and permission management.

Chapter 5: Loops - The Automation Engine

"Never repeat yourself in code."

The Goal: Batch Processing

You need to analyze 1,000 log files. **Bad Way:** Write 1,000 commands. **Good Way:** Write one loop.

5.1 The for Loop (Foreach)

In DevOps, we iterate over Lists (servers, files, logs).

```
servers = ["web-01", "web-02", "db-01"]

for server in servers:
   print(f"Deploying to {server}")
   restart_service(server)
```

What happens:

1. Python takes servers.
2. Assigns "web-01" to server.
3. Runs the block.
4. Repeats for "web-02", "db-01".

5.2 The range() Function (Counters)

Need to retry a connection 3 times? Use range().

```
for retries in range(3):
   print(f"Connecting... Attempt {retries + 1}")
   if connect():
      break
```

Theory Note: range(3) creates [0, 1, 2].

5.3 Under the Hood: Iterators

How does Python know which item is "next"?

- **Iterable:** The container (List of Servers).
- **Iterator:** The cursor pointing to current server.

The for loop asks the cursor "Next?" until it hits StopIteration.

5.4 The while Loop (Wait Until)

Typically used for polling or waiting for a service.

```
attempts = 0
max_attempts = 5

while attempts < max_attempts:
  if check_health() == "OK":
    print("Service UP!")
    break
  attempts += 1
  print("Waiting...")
```

Warning: Infinite Loops (while True) are common in daemons, but dangerous in one-off scripts. Always have an exit condition (break).

5.5 Controlling the Flow

break (Emergency Stop)

If the database connection fails, **stop** the entire batch.

```
for log_file in logs:
  if "FATAL ERROR" in log_file:
    print("Stopping deployment!")
    break
```

continue (Skip)

Ignore harmless warnings.

```
for user in users:
  if user == "root":
    continue  # Don't delete root!
  delete_user(user)
```

5.6 Nested Loops (Matrix Configs)

Deploying across multiple regions and environments.

```
regions = ["us-east", "eu-west"]
envs = ["dev", "prod"]

for region in regions:
   for env in envs:
      deploy(region, env)
```

Performance: Beware deeply nested loops (O(n^2)).

5.7 The Laws of Loops

1. **Guard Your Exits:** Every while needs a state change.
2. **Don't Touch the List:** Never .remove() while iterating.
3. **Off-by-One Law:** range(5) stops at 4.
4. **Break Early:** Don't waste CPU cycles.

[!] Common Mistakes

Error #1: Infinite Wait

```
while service_down:
   print("Waiting...")
   # Forgot to check status again!
```

Fix: Update service_down inside the loop.

Error #2: Modifying List

```
servers = ["a", "b", "c"]
for s in servers:
   servers.remove(s)
# Result: Only "a" and "c" removed. "b" skipped.
```

Fix: for s in servers[:]: (Iterate over copy).

Practice Problems

Problem 1: Log Filter

Filter a list of log lines. Print only lines containing "ERROR".

Problem 2: Retry with Backoff

Write a loop that retries a connection 5 times, waiting longer each time (1s, 2s, 4s...).

Chapter Summary

- **Automation:** Loops replace manual repetition.
- **for:** For collections (servers, logs).
- **while:** For conditions (waiting for UP).
- **Control:** break / continue manage flow.

Chapter 6: Functions - Building Tools

"The Linux philosophy: Do one thing and do it well."

The Goal: Modular Scripts

You copy-pasted the backup code 10 times. Now you need to change the path. **Pain:** You have to edit 10 places. **Solution:** Wrap it in a function.

A function is a **Reusable Tool**.

6.1 Defining a Function (def)

```
def check_disk_space():
  percent = get_usage("/")
  print(f"Disk Usage: {percent}%")
```

Calling it:

```
check_disk_space()
```

6.2 Parameters (Inputs)

Make the tool flexible.

```
def check_disk_space(mount_point):
    percent = get_usage(mount_point)
    print(f"Usage for {mount_point}: {percent}%")

check_disk_space("/var/log")
check_disk_space("/home")
```

6.3 Return Values (Outputs)

Scripts need Data, not Print statements.

```
def get_uptime(server):
    uptime_seconds = fetch_metric(server, "uptime")
    return uptime_seconds / 3600  # Return hours

hours = get_uptime("web-01")
if hours < 1:
    print("Server recently rebooted")
```

Why Return? Because you can't do math on print().

6.4 UNDER THE HOOD: The Call Stack

When you run deploy(), Python creates a workspace (Stack Frame).

- Variables (server_ip, status) live here.
- When deploy() finishes, the Frame is destroyed.

Recursion: A function calling itself (e.g., traversing directories). Too deep? RecursionError (Stack Overflow).

6.5 Variable Scope (Global vs Local)

```
api_key = "12345"  # Global (Read-only usually)

def connect():
    socket = create_connection() # Local (Destroyed after function)
```

Rule: Passed parameters are better than Globals.

6.6 Default Parameters (Configuration)

```
def connection(host, port=22, timeout=10):
    print(f"SSH to {host}:{port} (timeout {timeout}s)")

connection("web-01")       # Uses 22, 10
connection("db-01", 5432)   # Overrides port
```

Warning: Mutable Defaults (list=[]) persist between calls. Avoid them!

6.7 Function Design Laws

1. **Rule of Singularity:** One job per function (parse_log, not parse_and_upload).
2. **Rule of Descriptiveness:** backup_db() is better than run().
3. **Rule of Return:** Return data for the caller to handle.
4. **Rule of Scope:** Don't rely on global state.

[!] Common Mistakes

Error #1: The Phantom Return

```
def add(a, b):
    res = a + b
    # Forgot return!

total = add(10, 20)
print(total) # None
```

Fix: return res.

Error #2: Global Modification

```
count = 0
def inc():
    count = 1  # Creates a NEW local variable 'count'

inc()
```

print(count) # 0

Fix: return count + 1.

Practice Problems

Problem 1: URL Builder

Write build_url(host, path, ssl=True). Returns https://host/path or http://host/path.

Problem 2: Log Formatter

Write log(level, message) that prints [TIMESTAMP] [LEVEL] Message.

Chapter Summary

- **Functions:** Wrap logic into named blocks.
- **Parameters:** Inputs make functions flexible.
- **Return:** Outputs allow chaining logic.
- **Scope:** Local variables keep memory clean.
- **Design:** Keep functions small (Unix Philosophy).

Chapter 7: Data Structures - The System Inventory

"Organizing data is half the battle."

The Goal: Managing Collections

You manage 1,000 servers. **Bad Way:** server1 = "web01", server2 = "web02"... **Good Way:** Put them in a **List**.

You need to know the IP, OS, and Uptime for each. **Good Way:** Put them in a **Dictionary**.

7.1 Lists (Ordered Collections)

A list is like a rack of servers. Ordered and indexable.

servers = ["web-01", "db-01", "cache-01"]

Accessing Items

Zero-indexed (like rack units starting at 0).

```
print(servers[0])  # "web-01"
print(servers[-1]) # "cache-01" (Last one)
```

Modifying Lists (Mutable)

You can add or remove hardware.

```
servers.append("web-02")     # Add to end
servers.insert(0, "lb-01")   # Add to front
servers.remove("db-01")      # Decommission
```

7.2 Tuples (Immutable Lists)

Sometimes you want a list that **cannot** be changed. Example: Database Credentials, Geographic Coordinates.

```
db_config = ("192.168.1.5", 5432)
# db_config[0] = "10.0.0.1"  # ERROR: TypeError
```

DevOps Use: Return multiple values from a function.

```
def get_metrics():
   return (0.5, 2048)  # Load, RAM

load, ram = get_metrics()  # Unpacking
```

7.3 Dictionaries (Key-Value Stores)

The most useful structure for DevOps. It maps **Keys** (Names) to **Values** (Data). Think of it like a JSON configuration file.

```
server_config = {
   "hostname": "web-01",
   "ip": "10.0.0.5",
   "active": True,
   "roles": ["web", "api"]
}
```

Accessing Data

```
print(server_config["ip"])  # "10.0.0.5"
```

Adding/Updating

```
server_config["region"] = "us-east"  # Add new key
server_config["active"] = False      # Update existing
```

7.4 Sets (Unique Collections)

A bag of unique items. No duplicates allowed. Great for combining lists of IPs from different logs.

```
error_ips = {"10.0.0.1", "10.0.0.2", "10.0.0.1"}
print(error_ips)
# Output: {"10.0.0.1", "10.0.0.2"} (Duplicate removed)
```

Set Math (Venn Diagrams):

```
group_a = {"web01", "web02"}
group_b = {"web02", "db01"}

# Intersection (Who is in BOTH?)
print(group_a & group_b)  # {"web02"}

# Union (Combine both)
print(group_a | group_b)  # {"web01", "web02", "db01"}
```

7.5 Under the Hood: Hash Maps (Dictionaries)

Why are Dictionaries so fast? If you have a list of 1,000,000 items, searching it takes time (O(n)). If you have a dictionary, looking up a key is instant (O(1)).

How? Python runs the key through a **Hash Function** (e.g., hash("ip") -> 12345). It uses that number to jump directly to the memory address. No searching required.

7.6 Nested Data Structures (Complex Configs)

Real-world configs are rarely flat. You put lists inside dictionaries, and dictionaries inside lists.

List of Dictionaries (Server Inventory)

Commonly returned by cloud APIs (AWS/Azure).

```
inventory = [
  {"hostname": "web-01", "ip": "10.0.0.1", "role": "web"},
  {"hostname": "db-01", "ip": "10.0.0.5", "role": "db"},
  {"hostname": "cache-01", "ip": "10.0.0.9", "role": "cache"}
]

# Accessing the IP of the second server
print(inventory[1]["ip"])  # "10.0.0.5"

# Looping through them
for server in inventory:
  print(f"Pinging {server['hostname']}...")
```

Dictionary of Lists (Grouping)

```
load_balancers = {
  "us-east": ["lb-01", "lb-02"],
  "eu-west": ["lb-eu-01", "lb-eu-02"]
}

# Accessing the first LB in eu-west
print(load_balancers["eu-west"][0])  # "lb-eu-01"
```

7.7 The Laws of Data Structures

1. **The Law of Order:** Lists and Tuples preserve order. Sets and Dicts (mostly) do not.

2. **The Law of Uniqueness:** Sets and Dictionary Keys must be unique.
3. **The Law of Mutability:**
    a. Lists/Dicts/Sets -> Mutable (Changeable).
    b. Tuples/Strings -> Immutable (Frozen).

[!] Common Mistakes

Error #1: Key Error

```
print(config["password"])
```

Error: KeyError: 'password' **Fix:** Use .get() for safe access.

```
print(config.get("password", "default"))
```

Error #2: Modifying Tuples

```
coords = (10, 20)
coords[0] = 5
```

Error: TypeError. Tuples are read-only.

Practice Problems

Problem 1: The Inventory Manager

Create a dictionary representing a server (hostname, ip, os). Add a new key "uptime". Print the OS.

Problem 2: The Log Deduper

Given a list of IPs: ["1.1.1.1", "2.2.2.2", "1.1.1.1"]. Use a Set to remove duplicates and print the unique count.

Problem 3: The Nested Config

Create a list of 3 dictionaries (each representing a user with name and role). Loop through the list and print the name ONLY if the role is "admin".

Chapter Summary

- **Lists:** Ordered, mutable sequences (Server Racks).
- **Tuples:** Immutable sequences (Coordinates).
- **Dicts:** Key-Value stores (Configurations).
- **Sets:** Unique bags of items (IP Whitelists).
- **Nesting:** Combining these structures allows needed complexity.

Chapter 8: File I/O - Persistence & Reports

"Data in memory is a dream. Data on disk is reality."

The Goal: Logs, Configs, and Reports

Scripts die when they finish. To keep data, you must write it to a **File**.

- **Input:** Reading config.ini, access.log, or inventory.csv.
- **Output:** Writing backup_report.txt or audit.csv.

In DevOps, you aren't just reading text files. You are parsing **Paths**, manipulating **Directories**, and generating **Reports**.

8.1 Reading Files (open and read)

The open() function is your portal to the filesystem.

```
# 'r' means Read Mode (default)
with open("server.conf", "r") as file:
   content = file.read()
   print(content)
```

The with Statement (Context Manager): You *could* do file = open(...) and file.close(). but if your script crashes in the middle, the file stays open (Resource Leak). **With with**, Python auto-closes the file even if errors occur. **Always use with!**

8.2 Reading Line-by-Line (Log Parsing)

Don't use .read() on a 10GB log file. You will blow up your RAM. Iterate line by line instead.

```
with open("access.log", "r") as log_file:
   for line in log_file:
      if "ERROR" in line:
        print(f"Alert: {line.strip()}")
```

DevOps Note: This is "Lazy Loading". Python reads one line, processes it, forgets it, then reads the next. Memory usage stays low.

8.3 Writing Files (w and a)

- "w": **Write**. Overwrites the entire file! (Dangerous).
- "a": **Append**. Adds to the end of the file (Safe for logs).

```
# Overwrite configuration
with open("config.txt", "w") as f:
    f.write("host=localhost\n")
    f.write("port=8080\n")

# Append to log
with open("deploy.log", "a") as f:
    f.write("[INFO] Deployment started...\n")
```

8.4 Modern Paths with pathlib

Old Python used os.path.join(). New Python uses pathlib. It treats paths as **Objects**, not Strings.

```
from pathlib import Path

# Create a path object (Works on Windows AND Linux)
log_dir = Path("/var/log/nginx")
log_file = log_dir / "access.log"  # Use / to join paths!

print(log_file)  # /var/log/nginx/access.log

# Check existence
if not log_file.exists():
    print("Log file missing!")

# Create directory if missing
```

```
log_dir.mkdir(parents=True, exist_ok=True)
```

Why use Pathlib?

- **Cross-Platform:** Handles \ vs / automatically.
- **Methods:** .stem (filename), .suffix (extension), .parent (folder).

```
filename = Path("data/backup.tar.gz")
print(filename.stem)   # backup.tar (Smart!)
print(filename.suffix) # .gz
```

8.5 Working with CSV (Spreadsheets)

DevOps Engineers live in Excel/Google Sheets. Python's csv module lets you read/write them easily.

Reading CSVs

```
import csv

with open("inventory.csv", "r") as f:
    reader = csv.DictReader(f)  # Treating first row as Headers
    for row in reader:
        print(f"Server: {row['Hostname']}, IP: {row['IP']}")
```

Writing CSVs

```
data = [
    {"Hostname": "web01", "IP": "10.0.0.1"},
    {"Hostname": "db01", "IP": "10.0.0.5"}
]

with open("report.csv", "w", newline="") as f:
    writer = csv.DictWriter(f, fieldnames=["Hostname", "IP"])
    writer.writeheader()
    writer.writerows(data)
```

Newline Note: Always use newline="" when writing CSVs to work correctly on Windows.

8.6 File Operations (shutil)

Need to **Copy**, **Move**, or **Delete** files? Use shutil (Shell Utilities).

```
import shutil

# Copy
shutil.copy("source.txt", "dest.txt")

# Move (Rename)
shutil.move("old.txt", "new.txt")

# Delete Directory (Recursively - Dangerous!)
# shutil.rmtree("temp_dir")
```

8.7 Working with JSON (Configs)

Text files are hard to parse. JSON is structured.

```
import json

# Writing JSON
data = {"host": "web01", "roles": ["api", "db"]}
with open("config.json", "w") as f:
  json.dump(data, f, indent=4)

# Reading JSON
with open("config.json", "r") as f:
  config = json.load(f)
  print(config["host"])
```

8.8 Under the Hood: File Descriptors & Buffers

When you open() a file, the OS gives you a number (File Descriptor). Use lsof in Linux to see them. **Buffering:** Python doesn't write to disk immediately. It waits for the buffer (4KB or

8KB) to fill up to save CPU. **Force Write:** file.flush() forces data to disk immediately (crucial for crash logs).

8.9 The Laws of File I/O

1. **The Law of Closing:** Always use with open(...) to ensure files close.
2. **The Law of Modes:**
    a. r = Read (Safe).
    b. w = Wipe & Write (Dangerous).
    c. a = Append (Safe).
3. **The Law of Newlines:** .write() does NOT add \n. You must add it manually.
4. **The Law of Paths:** Use pathlib, not string concatenation for file paths.

[!] Common Mistakes

Error #1: The Missing File

open("ghost.txt", "r")

Error: FileNotFoundError. **Fix:** Check if Path("ghost.txt").exists(): before opening.

Error #2: RAM Explosion

open("massive.log").read()

Error: MemoryError. **Fix:** Iterate line-by-line: for line in file:.

Error #3: Windows Backslashes

open("C:\Users\name")

Error: \n is Newline, \u is Unicode. **Fix:** Use Raw Strings r"C:\Users\name" or Path("C:/Users/name").

Practice Problems

Problem 1: The Cleaner

Write a script that reads "dirty_log.txt", removes empty lines, and saves it to "clean_log.txt".

Problem 2: The CSV Converter

Read a CSV "users.csv" (Name, Role). Write a JSON file "users.json" containing the same data as a list of dictionaries.

Problem 3: The Backup Script

Use shutil to copy all .conf files from /etc/myapp to /backup/conf. (Hint: Use pathlib.Path("/etc/myapp").glob("*.conf") to find them).

Chapter Summary

- **open():** Opens a file handle.
- **with:** Auto-closes the handle (Safety).
- **pathlib:** Object-oriented path handling (Modern Standard).
- **csv:** Handling spreadsheets.
- **shutil:** Copying/Moving files like a shell script.
- **JSON:** The standard format for DevOps configs.

Chapter 9: Error Handling - Reliability Engineering

"Failures are inevitable. Crashing is optional."

The Goal: Resilience

In DevOps, networks flake, disks fill up, and APIs time out. **Bad Script:** Crashes with a traceback when curl fails. **Good Script:** Catches the error, retries, or alerts the on-call engineer.

This is **Exception Handling**.

9.1 The try-except Block (The Safety Net)

Wrap dangerous code in a try block.

```
try:
   # Dangerous operation
   conn = connect_to_db("10.0.0.5")
   print("Combined!")
except:
   # Safety net
   print("Connection Failed. Check VPN.")
```

Result: The script doesn't crash. It prints the error message and *continues running*.

9.2 Catching Specific Errors

Catching *everything* (except:) is bad practice. You might hide real bugs (like a syntax error). Catch only what you expect.

```
try:
  with open("config.json", "r") as f:
    data = f.read()
except FileNotFoundError:
  print("Config missing! Using defaults.")
except PermissionError:
  print("Access Denied! Run as sudo.")
  exit(1) # Stop script with error code
```

Common DevOps Exceptions:

- TimeoutError: Server didn't respond.
- ConnectionRefusedError: Port closed.
- KeyError: Missing config key.
- ValueError: Letters in a port number.
- KeyboardInterrupt: User hit Ctrl+C.

9.3 The logging Module (Beyond print)

Scripts running in the background shouldn't print(). Where does the text go? They should **Log**.

```
import logging

# Configure formatting
logging.basicConfig(
  level=logging.INFO,
  format='%(asctime)s - %(levelname)s - %(message)s',
  filename='deploy.log' # Optional: Write to file
)

logging.debug("Starting connection...") # Won't show (level is INFO)
logging.info("Deploying v1.0")
logging.warning("Disk space low: 10%")
logging.error("Database connection failed")
logging.critical("SYSTEM DOWN")
```

Levels:

1. **DEBUG:** Detailed info for diagnosing problems.
2. **INFO:** Confirmation that things are working as expected.
3. **WARNING:** Indication that something unexpected happened.
4. **ERROR:** A more serious problem.
5. **CRITICAL:** A serious error, the program may be unable to continue.

9.4 Raising Errors (The Alarm)

Sometimes you *want* to crash (or signal failure to a higher function).

```
def deploy(version):
  if version < 1.0:
    raise ValueError("Cannot deploy beta version to Prod!")
  print("Deploying...")
```

DevOps Context: If a health check fails, raise SystemExit(1) to tell standard CI/CD pipelines (Jenkins/GitLab) that the job failed.

9.5 Custom Exceptions

Why use generic errors? Create your own domain-specific errors.

```
class DeployError(Exception):
  """Raised when deployment fails"""
  pass

class MaintenanceModeError(Exception):
  """Raised when server is in maintenance mode"""
  pass

# Usage
```

```
def verify_server(server):
    if server.status == "maintenance":
        raise MaintenanceModeError(f"{server.name} is in maintenance")
```

Benefit: You can catch *only* your specific errors.

```
try:
    verify_server(s)
except MaintenanceModeError:
    print("Skipping server...")
except DeployError:
    print("CRITICAL: Deployment halted!")
```

9.6 The traceback Module (Deep Debugging)

Sometimes you want to handle the error, but still see *where* it happened.

```
import traceback

try:
    risky_logic()
except Exception:
    print("Something went wrong, but I'm staying alive.")
    # Print the stack trace string without crashing
    print(traceback.format_exc())
```

This is vital for **Daemon** processes that must never crash but need to report bugs.

9.7 The finally Block (Cleanup)

Code that runs **no matter what** (Success or Failure). Crucial for closing connections or deleting temporary files.

```
lock_file = open("deploy.lock", "w")
try:
    lock_file.write("LOCKED")
    perform_long_task()
```

```
except:
    print("Failed!")
finally:
    # This runs even if we crashed above
    lock_file.close()
    print("Lock released.")
```

9.8 Under the Hood: The Stack Trace

When an error is raised, Python unwinds the **Call Stack**. It looks for a try block in the current function. Not found? It goes to the caller. Not found? It goes to main. Not found? **Crash & Print Traceback.**

9.9 The Laws of Handling

1. **The Law of Specificity:** Catch ValueError, not Exception.
2. **The Law of Silence:** Don't just pass in an except block. Log the error!
3. **The Law of Cleanup:** Use finally or with to release resources.
4. **The Law of Logging:** Production scripts log to files, not stdout.

[!] Common Mistakes

Error #1: The Empty Except

```
try:
    upload()
except:
    pass  # EVIL!
```

Why: If upload() fails because of a typo (NameError), you will never know. The script will just silently do nothing.

Error #2: Catching Interrupts

```
except Exception:
```

This usually doesn't catch Ctrl+C (KeyboardInterrupt). That's a good thing! You want to be able to stop your script.

Practice Problems

Problem 1: The Retry Loop

Write a function connect() that randomly raises ConnectionError. Write a loop that calls it. If it fails, logging.warning() and retry up to 3 times. If 3 failures, raise.

Problem 2: The Custom Validator

Define class ConfigError(Exception). Write a function that checks a dictionary for "API_KEY". If missing, raise ConfigError.

Chapter Summary

- **try/except:** Catching errors to prevent crashes.
- **logging:** Structured output for production monitoring.
- **Custom Exceptions:** making errors descriptive (DeployError).
- **Traceback:** Debugging without crashing.
- **Reliability:** Essential for long-running services.

Chapter 10: OOP - The Infrastructure Blueprints

"Code format reflects mental models."

The Goal: Modeling Complex Systems

In small scripts, functions are enough: restart_server(). In large systems (like writing a Cloud API wrapper), you need **Object-Oriented Programming (OOP)**.

OOP allows you to structure code by modeling real-world objects. You don't just have "data" and "logic" floating around; you have **Objects** that hold their own data and know how to perform their own actions.

10.1 Classes vs Objects (The Theory)

Think of a **Class** as a Blueprint (The Architect's Drawing). Think of an **Object** as the Building (The Physical Structure).

- **Class:** Server (Defines that servers have an IP and can restart).
- **Object:** web-01 (A specific server at 10.0.0.1).

```
class Server:
  # 1. The Constructor (__init__)
```

```
    # This runs automatically when you create a new object.
    # It sets up the initial state.
    def __init__(self, hostname, cpu_cores):
        self.hostname = hostname    # Attribute (Data)
        self.cpu_cores = cpu_cores  # Attribute (Data)
        self.is_on = False          # Attribute (Default State)

    # 2. Instance Methods (Behavior)
    # Functions inside a class are called Methods.
    # They MUST take 'self' as the first argument.
    def power_on(self):
        self.is_on = True
        print(f"{self.hostname} is booting...")

# Creating Instances (Instantiation)
s1 = Server("web-01", 4)
s2 = Server("db-01", 16)

# Each object has its own separate memory
print(s1.hostname)  # "web-01"
print(s2.hostname)  # "db-01"

s1.power_on()      # Only s1 turns on. s2 stays off.
```

10.2 Encapsulation (Protecting Data)

In DevOps, you have rules. "You cannot set CPU cores to -5." Encapsulation allows you to enforce these rules by hiding data behind a protective layer.

The @property Decorator (Getters and Setters)

Instead of letting users touch self.ram directly, we force them to go through a "Gatekeeper" method.

```
class VM:
    def __init__(self):
        self._ram = 0  # The "_" means "Internal/Protected" (Convention only)
    # 1. The Getter to read the value
    @property
    def ram(self):
        return f"{self._ram} GB"

    # 2. The Setter to write the value
    @ram.setter
    def ram(self, value):
        if value < 0:
            raise ValueError("RAM cannot be negative!")
        if value > 128:
            raise ValueError("Too much RAM for this host!")
        self._ram = value

# Usage
my_vm = VM()
my_vm.ram = 64      # Goes through Setter -> OK
print(my_vm.ram)    # Goes through Getter -> "64 GB"

# my_vm.ram = -10   # CRASH: ValueError (The logic protected the variable!)
```

10.3 Inheritance (The Hierarchy)

Don't Repeat Yourself (DRY). If you have WebServer and DatabaseServer, they share 90% of the same code (IP, Hostname, Restart). Put the shared code in a **Parent Class** (Server).

1. Single Inheritance

One Parent, One Child. The Child gets everything the Parent has, plus its own extras.

```
class Server:
    def ping(self): print("Pong from Server")

class WebServer(Server):
    def serve_html(self): print("Serving <html>...")

w = WebServer()
w.ping()       # Works! Inherited from Server.
```

```
w.serve_html()  # Works! Specific to WebServer.
```

2. Multilevel Inheritance

Grandparent -> Parent -> Child.

```
class Machine:
    def has_electricity(self): return True

class Server(Machine):
    def boot(self): print("Booting OS...")

class Database(Server):
    def query(self): print("SELECT * FROM data")

db = Database()
db.has_electricity() # From Machine
db.boot()         # From Server
db.query()        # From Database
```

10.4 Abstraction (The Interface)

Abstraction means enforcing a structure. If you are building a Cloud Tool, every provider (AWS, Azure) MUST have a connect() method. You can't let a developer forget to write it.

The Abstract Base Class (ABC)

We use the abc module to create a template. You cannot create an object from a template; you must inherit from it.

```
from abc import ABC, abstractmethod

# 1. The Abstract Contract
class CloudProvider(ABC):
    @abstractmethod
    def connect(self):
        pass

    @abstractmethod
    def get_status(self):
        pass

# 2. The Implementation (AWS)
class AWS(CloudProvider):
    def connect(self):
        print("Connecting to AWS via boto3...")

    def get_status(self):
        return "AWS is Online"

# 3. The Broken Implementation (Azure)
class Azure(CloudProvider):
    def connect(self):
        print("Connecting to Azure...")
    # Oops! We forgot 'get_status'

# Usage
aws = AWS()       # Works!
# azure = Azure()  # CRASH! TypeError: Can't instantiate abstract class Azure with abstract
method get_status
```

Why this matters: Abstraction guarantees that all your plugins/providers follow the exact same rules.

10.4 Multiple Inheritance & The Diamond Problem

This is where it gets complex. Python allows a class to have **Two Parents**. This is powerful (Mixins) but dangerous.

The Mixin Pattern (The Good Use Case)

A **Mixin** is a small class that adds one specific feature (like Logging) to any class.

```
class LoggableMixin:
    def log(self, msg):
        print(f"[LOG] {msg}")

class Server:
    def start(self): print("Starting...")

# Inherits from BOTH
class SecureServer(Server, LoggableMixin):
    pass

s = SecureServer()
s.start()       # From Server
s.log("Secure")  # From LoggableMixin
```

The Diamond Problem (The Danger)

What if **both** parents define the same method? Which one wins?

Structure:

```
     A (Root)
     / \
    B   C
     \ /
      D (Child)
 class A:
    def save(self): print("Saved to A (Root)")

class B(A):
    def save(self): print("Saved to B (File)")

class C(A):
    def save(self): print("Saved to C (Database)")

class D(B, C):
    pass
    # D inherits from B and C. Both have 'save()'.
    # Who wins?
```

The Solution: Method Resolution Order (MRO)

Python uses the **C3 Linearization Algorithm** to decide. **The Rule:** Children before Parents. Left Parents before Right Parents.

For class D(B, C):

1. Is it in D? No.
2. **Is it in B (Left)? Yes! (Stop here).**
3. (Ignores C).

```
d = D()
d.save()
# Output: "Saved to B (File)"

# Viewing the Decision Tree
print(D.mro())
# [<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>]
```

Why this matters: If you connect a class to two Libraries that both have a method called connect(), Python will silently pick the left one. You must know your MRO!

10.5 Polymorphism (Many Forms)

Polymorphism allows you to treat different objects as if they were the same. You don't care *what* the object is, only that it can do the job.

1. Method Overriding

We have a generic Server.restart(). The Database needs a special restart (flush to disk first). The Database **overrides** the parent method.

```
class Server:
  def restart(self):
    print("Rebooting OS...")

class Database(Server):
  def restart(self):
    print("Flushing SQL transactions...")
    print("Rebooting OS...")

# Polymorphic Loop
infrastructure = [Server(), Database(), Server()]

for unit in infrastructure:
  # "unit" changes type in every loop iteration!
  # Python figures out the correct method at Runtime.
  unit.restart()
```

2. Duck Typing

Python is dynamic. It doesn't check if you inherit from Server. It checks: **"Do you have a restart() method?"**

```
class Car:
    def restart(self): print("Turning ignition key...")
```

### 3. Compile Time vs Run Time Polymorphism (The Interview Question)

In languages like C++ or Java, there are two types:
1. **Compile Time (Static Binding):** Method **Overloading**. Same function name, different arguments. (e.g., `add(int, int)` vs `add(str, str)`).
2. **Run Time (Dynamic Binding):** Method **Overriding**. Parent has `restart()`, Child has new `restart()`.

Python's Reality:
Python is **Dynamically Typed**, so it does **NOT** support Compile Time Polymorphism (Overloading) in the traditional sense.
If you define two methods with the same name, the second one simply overwrites the first.

How to "Simulate" Overloading in Python:
Use default arguments or variable-length arguments (`*args`).

```python
class Calculator:
    # "Overloading" using default None
    def add(self, a, b, c=None):
        if c:
            return a + b + c
        return a + b

calc = Calculator()
print(calc.add(1, 2))      # 3
print(calc.add(1, 2, 3))   # 6
```

Key Takeaway: Python relies almost exclusively on **Run Time Polymorphism** (Overriding & Duck Typing). The interpreter figures out which method to call *while the code is running*, not beforehand.

10.6 Magic Methods (Connecting to Python)

How does len(my_object) work? Or my_object + other_object? You define **Magic Methods** (Double Underscore -> Dunder).

Method	Triggered By	Example
__init__	Creation	x = Server()
__str__	print()	print(x)
__len__	len()	len(x)
__add__	+ Operator	x + y
__eq__	== Operator	x == y

```
class LoadBalancer:
  def __init__(self):
    self.servers = []

  def __len__(self):
    # Telling Python how to count this object
    return len(self.servers)

  def __add__(self, other_lb):
    # Telling Python how to add two LoadBalancers
    new_lb = LoadBalancer()
    new_lb.servers = self.servers + other_lb.servers
    return new_lb
```

10.7 The Laws of Advanced OOP

1. **Liskov Substitution Principle:**
    a. **Rule:** If you inherit from a Parent, you must be able to replace the Parent without breaking anything.
    b. **Example:** If Server.restart() takes 0 arguments, Database.restart() cannot require 5 arguments.
2. **Favor Composition over Inheritance:**
    a. **Rule:** Multiple inheritance is messy. Instead of "Is A" (Inheritance), use "Has A" (Composition).
    b. **Good:** Server **has a** Logger.
    c. **Bad:** Server **is a** Logger.
3. **Know your MRO:**
    a. **Rule:** When in doubt with Multiple Inheritance, print ClassName.mro() to see exactly what Python will call.

Practice Problem: The Cloud API

Create a base class CloudResource with cost(). Create subclasses VM (cost = $10) and Storage (cost = $0.10/GB). Create a Project class that holds a list of resources. Add a magic method __int__ to Project that returns the total cost of all resources.

Chapter Summary

- **Classes/Objects:** Blueprints vs Instances.
- **Encapsulation:** @property protects your data.
- **Polymorphism:** restart() behaves differently for Web vs DB.
- **Multiple Inheritance:** Understand MRO (Left to Right) to solve collisions.
- **Magic Methods:** __str__, __len__ make your objects feel native.

Chapter 11: Algorithms - Efficiency at Scale

"Premature optimization is the root of all evil." — Donald Knuth

The Goal: Handling Scale

If you have 10 logs, any script is fast. If you have 10,000,000 logs (common in Prod), a bad script will hang for hours.

Algorithms are about solving problems efficiently. In DevOps, "Efficient" means:

1. **Time:** How long does it take? (CPU)
2. **Space:** How much RAM does it eat? (Memory)

11.1 Big O Notation (The Speedometer)

We don't measure speed in seconds (because hardware varies). We measure in **Operations** relative to Input Size (n).

The Big O Tier List

Notation	Name	Speed	Example
O(1)	Constant	⚡ Instant	Dictionary Lookup (config["ip"])
O(log n)	Logarithmic	🏎️ Very Fast	Binary Search (Sorted Logs)
O(n)	Linear	🏃 Fast-ish	Looping through a List
O(n log n)	Log Linear	🚶 Decent	Sorting a List
O(n^2)	Quadratic	🐢 Slow	Nested Loops (Compare All vs All)
O(2^n)	Exponential	🛑 Impossible	Cracking Passwords

1. O(1) - Constant Time

No matter how much data, it takes 1 step.

```
# Accessing an element by index or key
ip = servers[50]
config = settings["db_host"]
```

It takes the same time whether servers has 100 or 1,000,000 items.

2. O(n) - Linear Time

If you have N items, it takes N steps.

```
# Finding a value in an unsorted list
for server in servers_list:
  if server == "prod-db-01":
    print("Found!")
```

Scaling:

- 10 servers -> 10 ms
- 1 million servers -> 1,000 seconds (16 mins!)

3. O(n^2) - Quadratic Time (The Danger Zone)

Usually caused by **Nested Loops**.

```
# Finding duplicates (Bad Approach)
for line1 in log_lines:
  for line2 in log_lines:
    if line1 == line2:
      print("Duplicate!")
```

Scaling:

- 1,000 lines -> 1,000,000 steps.
- 1,000,000 lines -> **1 Trillion steps** (Script hangs forever).

DevOps Rule: Never use O(n^2) on Production logs.

11.2 Space Complexity (RAM is Expensive)

Time isn't the only cost. If you load a 50GB log file into RAM on a 4GB VM, your script dies (MemoryError or OOM Kill).

O(1) Space (Streaming)

You hold only one line in memory at a time.

```
with open("massive.log") as f:
  for line in f:  # Lazy Loading
    process(line)
```

O(n) Space (Loading All)

You load everything into a list.

```
with open("massive.log") as f:
    lines = f.readlines()  # BOOM! 50GB RAM needed.
```

11.3 Search Algorithms

Linear Search (Brute Force)

Start at the beginning, check every item.

- **Pros:** Works on unsorted data.
- **Cons:** Slow (O(n)).

```
def find_error(logs, target="ERROR"):
    for i, line in enumerate(logs):
        if target in line:
            return i
    return -1
```

Binary Search (Divide & Conquer)

Only works on **Sorted Data** (e.g., Logs sorted by Timestamp).

- **Pros:** Blazing FAST (O(log n)).
- **Cons:** Data MUST be sorted first.

How it works:

1. Pick the middle item.
2. Is target lower? Discard the right half.
3. Is target higher? Discard the left half.
4. Repeat.

Power of O(log n): Searching **4 Billion** items takes only **32 steps**.

11.4 DevOps Case Study: The IP Whitelist

Scenario: You have 10,000 Whitelisted IPs. You process 1,000,000 incoming requests.

Bad Solution (List - O(n))

```
whitelist = ["1.1.1.1", "2.2.2.2", ...] # List

for ip in incoming_requests:
    if ip in whitelist:  # O(n) scan inside the loop!
        allow(ip)
```

Total Complexity: O(n * m). 1M requests * 10k whitelist = **10 Billion Steps**.

Good Solution (Set - O(1))

```
whitelist_set = set(["1.1.1.1", "2.2.2.2", ...]) # Hash Map

for ip in incoming_requests:
    if ip in whitelist_set:  # O(1) Instant Lookup
        allow(ip)
```

Total Complexity: O(m). 1M requests * 1 step = **1 Million Steps**. **Result:** The script runs 10,000x faster.

11.5 The Laws of Optimization

1. **The Law of Measurement:** Don't guess which part is slow. Use time or cProfile to measure.
2. **The Law of Datatypes:** Dictionaries/Sets are O(1). Lists are O(n). Know the difference.
3. **The Law of Memory:** Streaming (O(1) Space) is always safer than Loading All (O(n) Space).

[!] Common Mistakes

Error #1: Premature Optimization

Spending 3 days optimizing a script that runs once a month and takes 5 seconds. **Fix:** Only optimize scripts that run frequently or handle massive data.

Error #2: The Hidden Loop

```
if request_ip in list_of_ips:
```

This looks like one line, but it is a **hidden loop** (O(n)). If list_of_ips is huge, this line is slow.

Practice Problems

Problem 1: The RAM Saver

Refactor this code to use O(1) Space:

```
# Bad
logs = open("server.log").read().splitlines()
errors = [line for line in logs if "ERROR" in line]
print(len(errors))
```

Problem 2: The Binary Search Implementation

Write a function binary_search(sorted_list, target) that returns the index. Handle the edge case where the target is not found.

Chapter Summary

- **Big O:** The standard language for describing performance.
- **O(1) vs O(n):** The difference between "Instant" and "Wait for it".
- **Space Complexity:** Managing RAM is just as important as CPU.
- **Sets/Dicts:** The secret weapon for O(1) lookups.

Chapter 12: Sorting and Recursion - Organization

"Chaos is merely order waiting to be deciphered."

The Goal: Ordered Data

Logs come in random order (due to network latency). To analyze an incident, you need to **Sort** them by timestamp. To find a file in a deep folder structure, you need **Recursion**.

12.1 Sorting in Python (sort vs sorted)

Python uses **Timsort**, a hybrid algorithm (O(n log n)). It's very fast.

1. sorted() function (Safe)

Returns a **new** sorted list. The original list is untouched.

```
ips = ["10.0.0.5", "10.0.0.1", "10.0.0.3"]
sorted_ips = sorted(ips)

print(sorted_ips) # ['10.0.0.1', '10.0.0.3', '10.0.0.5']
print(ips)      # ['10.0.0.5', '10.0.0.1', '10.0.0.3'] (Unchanged)
```

2. .sort() method (In-Place)

Modifies the **existing** list. Saves memory (no copy), but destroys original order.

```
ips.sort()
print(ips) # ['10.0.0.1', '10.0.0.3', '10.0.0.5']
```

12.2 Custom Sorting (Key Functions)

Default sorting works on strings/numbers. What if you want to sort items by specific criteria?

Scenario: Sort logs by severity (INFO < WARNING < ERROR).

```
logs = [
  {"msg": "Disk Full", "level": "ERROR"},
  {"msg": "Login OK", "level": "INFO"},
  {"msg": "High Latency", "level": "WARNING"}
]

# Helper function to define the order
def get_severity(item):
  priority = {"INFO": 0, "WARNING": 1, "ERROR": 2}
  return priority[item["level"]]

# Sort using the 'key'
sorted_logs = sorted(logs, key=get_severity, reverse=True)
# Result: ERROR first, then WARNING, then INFO
```

Lambda (Anonymous Function): For simple keys, use a lambda (one-line function).

```
# Sort servers by CPU usage
servers = [("web01", 80), ("db01", 10), ("cache01", 50)]
servers.sort(key=lambda s: s[1]) # Sort by the second item (CPU)
```

12.3 Recursion (Functions Calling Themselves)

Recursion is when a function calls itself to solve a smaller piece of the problem. **DevOps Use Case:** Walking through a directory tree (/var/www/html/...).

The Structure of Recursion

1. **Base Case:** The condition to STOP (e.g., "No more subfolders").
2. **Recursive Step:** The function calling itself.

```
import os

def walk_dir(path):
  print(f"Scanning: {path}")

  # Get all items in directory
  items = os.listdir(path)

  for item in items:
    full_path = os.path.join(path, item)

    if os.path.isdir(full_path):
      # RECURSION: Function calls itself for the subfolder
      walk_dir(full_path)
    else:
      print(f"Found File: {item}")

# walk_dir("/var/log")
```

Visualizing the Stack:

1. walk_dir("/log")
    a. walk_dir("/log/nginx")
        i. walk_dir("/log/nginx/old") ... returns
    b. ... returns
2. Done.

12.4 Recursive vs Iterative

Should you always use recursion? **No.** Recursion uses the **Call Stack** (O(n) Space). If the directory is 10,000 folders deep, Python will crash (RecursionError).

Iterative (Stack-Saving) Approach: Use a list as a "ToDo" stack.

```
def walk_iterative(start_path):
    todo = [start_path]

    while todo:
        current = todo.pop()
        # process current...
        # add subfolders to todo list
```

12.5 The Laws of Sorting

1. **The Law of Stability:** Python's sort is **Stable**. If two items have the same key, their original order is preserved.
2. **The Law of Keys:** Always use key= for complex objects (dicts/tuples).
3. **The Law of Depth:** Be careful with recursion limit (default 1000). Use sys.setrecursionlimit() if you dare.

[!] Common Mistakes

Error #1: Sorting None

```
x = ips.sort()
print(x) # None
```

Why: .sort() modifies in-place and returns None. **Do not assign the result.**

Error #2: Infinite Recursion

```
def loop():
    print("Run")
    loop() # No Base Case!
```

Result: RecursionError: maximum recursion depth exceeded.

Practice Problems

Problem 1: The Late Logs

Sort a list of log strings based on the timestamp at the beginning of the line. "2023-10-01 Error..." (Hint: key=lambda x: x.split()[0])

Problem 2: The Directory Size

Write a recursive function get_size(path) that calculates the total size (bytes) of a directory and all its subfolders.

Chapter Summary

- **Sorting:** sorted() (Copy) vs .sort() (In-Place).
- **Keys:** lambda functions define custom sort orders.
- **Recursion:** Elegant for hierarchical data (File systems).
- **Base Case:** The most important part of a recursive function.

Chapter 13: Stacks and Queues - Managing Flow

"First In, First Out. Or Last In, First Out. Pick one."

The Goal: Job Management

How do you handle 1,000 users trying to upload logs at once? You can't process them all instantly. You need a **Buffer**. In DevOps, these buffers are **Queues** (RabbitMQ, Kafka, Celery).

13.1 Stacks (LIFO - Last In, First Out)

Think of a stack of plates.

- You put a plate on top (**Push**).
- You take the top plate off (**Pop**). The **Last** one you put in is the **First** one out.

Python Implementation

Python lists are perfect stacks.

```
stack = []

# Push (Append)
stack.append("Page 1")
stack.append("Page 2")
stack.append("Page 3")

# Pop (Remove from end)
current = stack.pop()
print(current) # "Page 3"
```

DevOps Use Case: The "Undo" Button

Deployments often work like stacks.

1. Deploy Database.
2. Deploy API.
3. Deploy Frontend.

Error! Frontend fails. **Rollback:**

1. Undo Frontend (Pop).
2. Undo API (Pop).
3. Undo Database (Pop). (Reverse order of operations).

13.2 Queues (FIFO - First In, First Out)

Think of a line at a ticket counter.

- People join the back (**Enqueue**).
- People are served from the front (**Dequeue**).

The Performance Trap of Lists

Bad: Using list.pop(0). Removing item 0 forces Python to **shift** all other items left. This is slow **O(n)**.

Good: Using collections.deque (Double-Ended Queue). It allows O(1) adds and removes from both ends.

```
from collections import deque

job_queue = deque()

# Enqueue (Add to right)
job_queue.append("Job 1: Resize Image")
job_queue.append("Job 2: Send Email")
```

```
# Dequeue (Remove from left)
task = job_queue.popleft() # Fast! O(1)
print(task) # "Job 1..."
```

13.3 Priority Queues (VIP Lines)

Sometimes a "Critical Security Patch" should jump the line ahead of "Daily Backup". Python uses heapq for this.

```
import heapq

# List of (Priority, Task)
# Priority 1 is highest (served first)
tasks = []

heapq.heappush(tasks, (3, "Backup"))
heapq.heappush(tasks, (1, "Security Patch"))
heapq.heappush(tasks, (2, "Update Docs"))

# Process (heappop always gives the smallest number first)
while tasks:
    priority, task = heapq.heappop(tasks)
    print(f"Doing: {task}")

# Output:
# Doing: Security Patch
# Doing: Update Docs
# Doing: Backup
```

13.4 DevOps System Design: Worker Pools

Standard pattern for scaling: **Producers** and **Consumers**.

1. **Web Server (Producer):** Receives user request. Puts it in a **Queue** (Redis/RabbitMQ).
2. **Worker Server (Consumer):** Reads from Queue. Processes it.

If load increases, the Queue grows. The Worker keeps chugging at a steady pace. **Solution:** Auto-scale more Workers to drain the Queue faster.

13.5 The Laws of Queues

1. **The Law of Fairness:** FIFO Queues are fair. Everyone waits their turn.
2. **The Law of Performance:** Never use list.pop(0) for queues. Use deque.
3. **The Law of Backpressure:** If the queue grows faster than you can pop, you will run out of RAM. (Implement limits).

[!] Common Mistakes

Error #1: Using List as Queue

```
queue = []
queue.pop(0) # O(n) - Slow for large lists!
```

Fix: from collections import deque.

Error #2: Stack Overflow

Recursion uses the internal Call Stack. Too deep = Crash. A Stack Data Structure (on the heap) can hold millions of items without crashing.

Practice Problems

Problem 1: The Browser History

Implement a History class using a Stack. visit(url) pushes to stack. back() pops and returns the previous URL.

Problem 2: The Printer Spooler

Implement a print queue using deque. Add 5 documents. Print them in FIFO order.

Chapter Summary

- **Stack (LIFO):** Undo operations, Function calls. Use list.
- **Queue (FIFO):** Buffer for jobs/tasks. Use deque.
- **Priority Queue:** sorting tasks by urgency. Use heapq.

- **Architecture:** Decoupling systems using Queues is a core DevOps pattern.

Chapter 14: Linked Lists - Chains of Custody

"Strong links make a strong chain."

The Goal: Dynamic Structures

Array (List) in memory is a solid block. [1][2][3] If you want to insert a new item at the beginning [0][1][2][3], you have to shift EVERYTHING. This is O(n).

A **Linked List** is different. Items can be anywhere in RAM. Each item points to the next one. [1]->[2]->[3] To insert [0], you just change one pointer. [0]->[1]->[2]->[3] This is O(1).

14.1 The Node (The Building Block)

A Linked List is made of **Nodes**. Each Node has:

1. **Data** (The content).
2. **Next** (Pointer to the next node).

```
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None  # Initially points to nothing

  def __repr__(self):
    return f"Node({self.data})"
```

14.2 Building the Chain

```
# Create independent nodes
head = Node("Access Log")
middle = Node("Error Log")
tail = Node("Debug Log")

# Link them together
head.next = middle
middle.next = tail

# The Chain: Access -> Error -> Debug -> None
```

Traversing the List

To read the list, you start at the Head and follow the pointers until you hit None.

```
current = head

while current:
  print(current.data)
  current = current.next
```

14.3 DevOps Use Case: The Blockchain (Log Integrity)

Why uses Connected Lists in real life? **Filesystem Inodes:** Files are often fragmented on disk. One block points to the next. **Git Commits:** Each commit points to its parent. **Blockchain:** Each block includes the Hash of the previous block.

```
import hashlib

class Block:
  def __init__(self, data, previous_hash):
    self.data = data
    self.previous_hash = previous_hash
    self.hash = self.calculate_hash()

  def calculate_hash(self):
    # Unique ID based on data + parent
    content = f"{self.data}{self.previous_hash}".encode()
    return hashlib.sha256(content).hexdigest()

# Genesis Block
genesis = Block("Initial Commit", "0")
block1 = Block("Added Login", genesis.hash)
block2 = Block("Fixed Bug", block1.hash)

print(f"Block 2 points to: {block2.previous_hash}")
print(f"Block 1 Hash:    {block1.hash}")
# They Match! The chain is unbroken.
```

If a hacker changes block1.data, block1.hash changes. Then block2.previous_hash won't match. **The chain breaks.** This is how **Git** ensures history integrity.

14.4 Python's Reality Check

Does Python use Linked Lists? **No.** Python's list is a **Dynamic Array** (C Array).

- Access list[5] is O(1).
- Linked List node.next.next... is O(n).

You rarely implement Linked Lists manually in Python unless strictly for an algorithm interview or a very specific data structure (like a Graph).

14.5 The Laws of Linked Lists

1. **The Law of Pointers:** If you lose the head pointer, you lose the whole list (Garbage Collection eats it).
2. **The Law of Insertion:** Inserting at the beginning is O(1). (Lists are O(n)).
3. **The Law of Access:** Finding the 100th item is O(n). (Lists are O(1)).

[!] Common Mistakes

Error #1: Infinite Loops

```
node.next = node # Points to itself
```

If you traverse this, while current: never ends. Cycle Detection is a classic problem.

Error #2: Breaking the Chain

To remove B from A->B->C. **Correct:** A.next = C **Incorrect:** Just deleting B without updating A. (A still points to dead memory or keeps B alive).

Practice Problem: The Reversal

Write a function reverse_list(head) that takes a Linked List A->B->C and turns it into C->B->A. (Hint: You need 3 pointers: prev, curr, next).

Chapter Summary

- **Node:** Data + Pointer.
- **Linked List:** Chain of nodes.
- **Pros:** Fast insertions.
- **Cons:** Slow lookups.
- **Real World:** Git History, Blockchains, Filesystems.

Chapter 15: Trees - Hierarchical Data

"Roots, Trunks, Branches, Leaves. Nature knows best."

The Goal: Structured Data

Lists are flat. Real life is nested.

- **Filesystems:** / -> home -> user -> docs.
- **HTML/XML:** <body> -> <div> -> <p>.
- **JSON/YAML:** config -> network -> interfaces.

These are **Trees**.

15.1 The Value of Hierarchies

A Tree is just a Node (Root) that points to multiple Children. Each Child is also a Tree (Subtree).

```
class TreeNode:
  def __init__(self, name):
    self.name = name
    self.children = [] # List of Nodes

  def add(self, child_node):
    self.children.append(child_node)

  def __repr__(self):
    return f"Dir({self.name})"
```

15.2 Building a Directory Tree

Let's model a Linux filesystem.

```
root = TreeNode("/")
home = TreeNode("home")
bin_dir = TreeNode("bin")

root.add(home)
root.add(bin_dir)

user = TreeNode("ubuntu")
home.add(user)

# Structure:
# /
#  ├── home
# │   └── ubuntu
# └── bin
```

15.3 Traversing a Tree (Recursion Again!)

To print the whole tree, we visit the Node, then **Recursively** visit its children.

```
def print_tree(node, level=0):
    indent = "  " * level
    print(f"{indent}- {node.name}")

    for child in node.children:
        print_tree(child, level + 1)

print_tree(root)
# - /
#   - home
#     - ubuntu
#   - bin
```

DevOps Use Case: This is exactly how tree command or ls -R works.

15.4 Binary Search Trees (BST)

A special tree where:

- Left Child < Parent
- Right Child > Parent

Why? Searching is O(log n). Is 50 > 10? Go Right. Is 50 < 100? Go Left. found.

Database Indexes: Databases use B-Trees (Balanced Trees) to find "User ID 500" instantly without scanning the whole table.

15.5 Parsing JSON config (The Real Tree)

When you load a JSON file, Python creates a Tree of Dictionaries/Lists.

```
{
  "server": {
    "ports": [80, 443],
    "admin": {
      "name": "admin",
      "email": "root@localhost"
    }
  }
}
```

Searching unique keys in JSON: You can write a recursive function to find *any* key named "email", no matter how deep it is nested.

```
def find_key(data, target):
  if isinstance(data, dict):
    for key, value in data.items():
      if key == target:
        print(f"Found: {value}")
      find_key(value, target) # Recurse!

  elif isinstance(data, list):
    for item in data:
      find_key(item, target) # Recurse!
```

```
# find_key(config, "email")
```

15.6 The Laws of Trees

1. **The Law of Roots:** Every tree has exactly one Root.
2. **The Law of Leaves:** Nodes with no children are Leaves.
3. **The Law of Depth:** Deep trees risk Recursion Limits.
4. **The Law of Balance:** An unbalanced tree (a line) is just a Linked List (slow).

[!] Common Mistakes

Error #1: Cycles (Not a Tree)

If a child points back to the parent, it's not a Tree anymore. It's a **Graph**. Trees must be **Acyclic**.

Error #2: Modifying while Traversing

Deleting nodes while recursing through them is dangerous. Collect "nodes to delete" in a list, then delete them after the loop.

Practice Problems

Problem 1: The File Finder

Write a function that takes a TreeNode (directory) and searches for a file named "passwords.txt".

Problem 2: Depth Calculator

Write a function get_max_depth(node) that returns how many layers deep the tree goes.

Chapter Summary

- **Tree:** Hierarchical Nodes (Root -> Branches -> Leaves).
- **Recursion:** The standard way to walk trees.
- **DOM/JSON:** Real-world tree examples.
- **Databases:** Use Trees for fast indexing.

Chapter 16: Graphs - The Network Map

"Everything is connected."

The Goal: Modeling Connections

Trees have a strict hierarchy (Parent -> Child). Real networks are messy.

- Server A talks to Server B.
- Server B talks to Server A.
- Service C depends on Service D and E. This is a **Graph**.

16.1 What is a Graph?

1. **Vertices (Nodes):** The things (Servers, Cities, Tasks).
2. **Edges (Links):** The connections (Cables, Roads, Dependencies).

Types:

- **Undirected:** A cable connects A and B. (A <-> B).
- **Directed (Digraph):** Traffic flows one way. (A -> B).
- **Weighted:** Some links are "heavier" (Latency, Cost).

16.2 Representing Graphs (The Adjacency List)

The best way to store a graph in Python is a Dictionary of Lists (or Sets).

```
network = {
  "WebLB": ["Web01", "Web02"],
  "Web01": ["DB01", "Cache01"],
  "Web02": ["DB01"],
  "DB01":  [],
  "Cache01": []
}
```

Meaning: WebLB is connected to Web01 and Web02.

16.3 Graph Traversal (BFS vs DFS)

How do you find if WebLB can reach DB01?

Breadth-First Search (BFS) - The Wave

Scan all neighbors, then neighbors' neighbors. **Use Case:** Finding the **Shortest Path** (Fewest hops).

```
def bfs(graph, start, target):
    queue = [start]
    visited = set()

    while queue:
        current = queue.pop(0) # FIFO
        if current == target:
            return True

        visited.add(current)
        for neighbor in graph[current]:
            if neighbor not in visited:
                queue.append(neighbor)
    return False
```

Depth-First Search (DFS) - The Maze Runner

Go as deep as possible, then backtrack. **Use Case:** Solving puzzles, detecting loops.

```
def dfs(graph, current, visited=None):
    if visited is None: visited = set()
    print(f"Visiting: {current}")
    visited.add(current)

    for neighbor in graph[current]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

16.4 DevOps Use Case: Dependency Resolution

Scenario: You want to deploy "App A". "App A" needs "Lib B". "Lib B" needs "Lib C". This is a **Directed Acyclic Graph (DAG)**. Tools like **Terraform**, **Docker Compose**, and **Systemd** use graphs to determine startup order.

Topological Sort: An algorithm that flattens the graph into a linear list (Order of operations).

1. Find node with no dependencies (Lib C).

2. Remove it.
3. Repeat. Result: C -> B -> A.

16.5 The Shortest Path (Dijkstra)

If edges have weights (Latency in ms), BFS is not enough. **Dijkstra's Algorithm** finds the fastest path, not just fewest hops. It prioritizes exploring low-cost edges first (using a Priority Queue).

16.6 The Laws of Graphs

1. **The Law of Connectivity:** Not all nodes are reachable. Islands exist.
2. **The Law of Cycles:** Infinite loops are real. Always use a visited set to track where you've been.
3. **The Law of Complexity:** Graph algorithms can be computationally expensive (O(V + E)).

[!] Common Mistakes

Error #1: Getting Stuck in a Loop

```
A -> B -> A
```

If you don't check if neighbor in visited, your BFS/DFS will run forever.

Error #2: Assuming Direction

In a directed graph, if A -> B, it **does not** mean B can talk to A. (Firewall Rules!).

Practice Problems

Problem 1: The Social Network

Create a graph representing friends. Find if "Alice" is connected to "Kevin" within 3 degrees of separation (Degrees = Hops).

Problem 2: The Critical Path

Given a list of tasks and dependencies. {"Build": ["Lint"], "Deploy": ["Build"], "Lint": []} Print the order they must run.

Final Project: The Infrastructure Mapper

Write a script that takes a list of server connections and builds a Graph dictionary. Then, write a function ping_all(start_node) that uses BFS to print every reachable server from the Gateway.

Book Summary

You made it. You started with variables and loops. You built scripts with files and error handling. You modeled systems with OOP. You organized data with Trees and Graphs.

You are no longer a Scripter. You are a Developer.

Go build something amazing.

Chapter 17: Virtual Environments & Packages - The Sandbox

"Works on my machine" is not a valid excuse.

The Goal: Isolation

You have Project A requiring requests==2.0. You have Project B requiring requests==3.0. If you install them globally on your laptop, one will break.

Virtual Environments (venv) solve this by creating a self-contained folder for each project, with its own independent Python and libraries.

17.1 Creating a Virtual Environment

Navigate to your project folder and run:

```
# Linux/Mac
python3 -m venv .venv

# Windows
python -m venv .venv
```

This creates a hidden folder .venv/ containing a copy of the Python binary.

Activating the Environment

You must "enter" the sandbox to use it.

```
# Linux/Mac
source .venv/bin/activate

# Windows (PowerShell)
.venv\Scripts\Activate.ps1
```

How do you know it worked? Your terminal prompt will change: (.venv) user@host:~/project$

17.2 The Package Manager (pip)

pip is the tool to install external libraries from **PyPI** (The Python Package Index).

Installing Packages

```
pip install requests
pip install boto3
```

Listing Installed Packages

```
pip list
# Package    Version
# ---------- -------
# boto3      1.26.0
# requests   2.28.0
```

17.3 Dependency Management (requirements.txt)

You need to share your code with a teammate. They need the exact same libraries.

Freezing Dependencies

Save your current environment to a file:

```
pip freeze > requirements.txt
```

Content of requirements.txt:

```
boto3==1.26.0
requests==2.28.0
urllib3==1.26.13
```

Installing from Requirements

Your teammate clones your code and runs:

```
pip install -r requirements.txt
```

Now their environment matches yours exactly.

17.4 Project Structure

Professional Python projects follow a standard layout.

```
my_project/
├── .venv/          # The Sandbox (Gitignored!)
├── src/            # Source Code
│   ├── __init__.py
│   └── main.py
├── tests/          # Unit Tests
├── .gitignore      # Ignore .venv and __pycache__
├── README.md       # Documentation
└── requirements.txt   # Dependencies
```

The .gitignore File: Never commit your virtual environment to Git. It's huge and OS-specific. Add this swp to .gitignore:

```
.venv/
__pycache__/
*.pyc
```

17.5 Under the Hood: The sys.path

How does Python find imports? It looks in a list of directories called sys.path.

When you activate a venv, it modifies sys.path to look inside .venv/lib/python3.X/site-packages *before* looking in the system folders.

```
import sys
for path in sys.path:
    print(path)
```

17.6 The Laws of Packaging

1. **The Law of Isolation:** Never pip install globally (unless it's a system tool). Always use a venv.
2. **The Law of Freezing:** Always pin your versions in requirements.txt.
3. **The Law of Ignoring:** Add .venv to .gitignore immediately.

[!] Common Mistakes

Error #1: Forgetting to Activate

```
$ pip install requests
$ python main.py
ModuleNotFoundError: No module named 'requests'
```

Cause: You installed requests in the venv, but forgot to run source .venv/bin/activate.

Practice Problem: The Setup

1. Create a folder ops_tools.
2. Create a venv inside it.
3. Activate it.
4. Install requests.
5. Generate a requirements.txt.
6. Deactivate (deactivate command) and delete the folder.

Chapter Summary

- **venv:** Creates isolated Python sandboxes.
- **pip:** Installs packages from PyPI.

- **requirements.txt:** Tracks dependencies for reproducibility.
- **Isolation:** Essential for managing multiple projects on one machine.

Chapter 18: System Automation - Python as Glue

"Python is a better Bash."

The Goal: Replacing Shell Scripts

Bash is great for pipes (|). Python is great for logic. When your Bash script needs arrays, loops, or JSON parsing, switch to Python.

But how do you run Linux commands (ls, grep, systemctl) from Python? **Enter the subprocess module.**

18.1 Running Commands (subprocess.run)

The modern, safe way to execute commands.

```
import subprocess

# Simple execution
subprocess.run(["ls", "-l"])
```

Why the List? ["ls", "-l"] Accessing the shell is dangerous. If you used a string "ls -l", a hacker could inject commands. By passing a list, Python creates the process directly, bypassing the shell. **Safe.**

18.2 Capturing Output (stdout)

Usually, you want to store the output in a variable, not just print it.

```
result = subprocess.run(
   ["uname", "-a"],
   capture_output=True,
   text=True  # Decode bytes to string
)

print("Return Code:", result.returncode)
print("Output:", result.stdout)
```

- **capture_output=True**: Redirects stdout/stderr to memory.
- **text=True**: Converts generic Bytes (b'Linux\n') to String ("Linux\n").

18.3 Handling Errors (check=True)

If the command fails (e.g., ls /nonexistent), subprocess.run does *not* crash by default. It just sets returncode=1.

To force Python to raise an error (good for "Stop on Error" scripts):

```
try:
   subprocess.run(["ls", "/ghost"], check=True)
except subprocess.CalledProcessError as e:
   print(f"Command failed with code {e.returncode}")
```

Use check=True when the command **must** succeed for the script to continue.

18.4 Environment Variables (os.environ)

Scripts need secrets (API Keys, DB Passwords). **Never** hardcode them. Read them from the Environment.

```
import os

# Reading
api_key = os.environ.get("AWS_ACCESS_KEY_ID")

if not api_key:
   print("Error: AWS_ACCESS_KEY_ID is missing!")
   exit(1)

# Writing (for child processes only)
os.environ["TEMP_DIR"] = "/tmp/my_script"
subprocess.run(["./backup.sh"]) # backup.sh sees TEMP_DIR
```

18.5 The "Shell=True" Danger

Sometimes you need pipes (|) or wildcards (*). subprocess doesn't support them by default. You *can* use shell=True, but be careful.

```
# Dangerous if 'user_input' comes from the web!
subprocess.run(f"grep {user_input} file.txt", shell=True)
```

If user_input is "; rm -rf /", you are dead.

Safe Alternative (Do the pipe in Python!):

1. Run grep via subprocess.
2. Capture output.
3. Filter it in Python code.

18.6 DevOps Use Case: Service Restarter

A script that checks Nginx and restarts it if dead.

```
import subprocess
import time

def check_nginx():
    try:
        res = subprocess.run(
            ["systemctl", "is-active", "nginx"],
            capture_output=True, text=True
        )
        return res.stdout.strip() == "active"
    except FileNotFoundError:
        # Handles cases where systemctl is missing (e.g. inside a container)
        print("Error: 'systemctl' command not found.")
        return False

def restart_nginx():
    print("Restarting Nginx...")
    subprocess.run(["sudo", "systemctl", "restart", "nginx"], check=True)

if not check_nginx():
    restart_nginx()
    time.sleep(2)
    if check_nginx():
        print("Success: Nginx is back online.")
    else:
        print("Critical: Nginx failed to restart.")
else:
    print("Nginx is healthy.")
```

18.7 The Laws of Subprocess

1. **The Law of Lists:** Always pass commands as ["cmd", "arg"], not strings.
2. **The Law of Checking:** Use check=True to catch failures automatically.
3. **The Law of Sanitization:** Avoid shell=True unless absolutely necessary.
4. **The Law of Secrets:** Use os.environ, not hardcoded strings.

[!] Common Mistakes

Error #1: Bytes vs Strings

```
res = subprocess.run(["ls"], capture_output=True)
print(res.stdout) # b'file1\nfile2\n'
```

Fix: Pass text=True (or universal_newlines=True in older Python).

Error #2: Blocking

subprocess.run **waits** for the command to finish. If you run a command that never ends (like tail -f), your Python script hangs forever. **Fix:** Use subprocess.Popen for background tasks (Advanced).

Practice Problem: The Backup

Write a script that:

1. Uses tar (via subprocess) to compress a folder.
2. Uses mv (via subprocess) to move it to a /backup folder.
3. Reads BACKUP_DIR from an environment variable.

Chapter Summary

- **subprocess.run:** The standard way to run shell commands.
- **Safety:** Avoid shell injection by using list arguments.
- **Environment:** Use os.environ to configure your scripts dynamically.
- **Power:** Combine Linux tools (CLI) with Python logic (Loops/Libs).

Chapter 19: Building CLI Tools - The Interface

"A tool without documentation is a puzzle."

The Goal: Professional UX

You wrote a script backup.py. User A runs it: python backup.py -> Crashes. User B runs it: python backup.py /tmp -> "Unknown argument".

You want your script to behave like ls or git.

- --help menu.
- Flags like --verbose or --dry-run.
- Required vs Optional arguments.

argparse is the standard library for this.

19.1 specific vs Generic (argparse vs sys.argv)

The Amateur Way (sys.argv):

```
import sys
filename = sys.argv[1] # Crashes if user forgets the argument!
```

The Pro Way (argparse): Auto-generates help menus and handles errors gracefully.

19.2 Your First Robust CLI

```
import argparse

# 1. Create the Parser
parser = argparse.ArgumentParser(description="A robust backup tool.")

# 2. Add Arguments
parser.add_argument("source", help="Directory to back up")
parser.add_argument("dest", help="Destination folder")
parser.add_argument("--compress", action="store_true", help="Enable GZIP
compression")

# 3. Parse Args
args = parser.parse_args()

# 4. Use Logic
print(f"Backing up {args.source} to {args.dest}...")
if args.compress:
   print("Compression Enabled.")
```

Try running it:

- python cli.py --help -> Show nice menu.
- python cli.py src/ -> Error: the following arguments are required: dest.
- python cli.py src/ /bak --compress -> Works!

19.3 Argument Types

1. Positional Arguments

Mandatory. Order matters. (Like source and dest above).

2. Optional Flags (--name)

Optional. Order doesn't matter.

```
parser.add_argument("-v", "--verbose", action="store_true", help="Talkative mode")

Usage: python script.py -v
```

3. Application with Values

Taking an integer or a specific string choice.

```
parser.add_argument("--port", type=int, default=80, help="Server Port")
parser.add_argument("--env", choices=["dev", "prod"], default="dev")
```

Python automatically validates:

- --port ABC -> Error (expects int).
- --env mylaptop -> Error (must be dev/prod).

19.4 DevOps Use Case: The Deploy Tool

```
import argparse

def main():
  parser = argparse.ArgumentParser(description="Deploy App V1")

  # Target Selection
  parser.add_argument("service", help="Service name (e.g. 'web', 'db')")

  # Configuration
  parser.add_argument("--region", default="us-east-1", help="AWS Region")
  parser.add_argument("--replicas", type=int, default=1, help="Container Count")
  parser.add_argument("--dry-run", action="store_true", help="Simulate only")

  args = parser.parse_args()

  if args.dry_run:
    print(f"[DRY RUN] Would deploy {args.service} to {args.region} x {args.replicas}")
    return

  print(f"Deploying {args.service}...")
  # Real logic here

if __name__ == "__main__":
  main()
```

19.5 The Laws of CLIs

1. **The Law of Help:** Every tool must have a useful --help (argparse does this for you).
2. **The Law of Defaults:** Provide sensible defaults (e.g., port 80).
3. **The Law of Feedback:** If --verbose is on, tell the user what is happening.
4. **The Law of Validation:** Use type=int and choices=[] to stop bad input early.

[!] Common Mistakes

Error #1: Confusing - and --

- - (One dash) for single letters: -v.
- -- (Two dashes) for words: --verbose.

Error #2: Hardcoding Paths

Don't write filename = "backup.txt" inside your code. Make it an argument: parser.add_argument("--file", default="backup.txt"). Now your script is reusable!

Practice Problem: The Log Searcher

Create a tool search.py that takes:

1. file: Positional argument (Path to log).
2. --keyword: Optional argument (Default: "ERROR").
3. --limit: Optional Integer (Max lines to print).

It should print lines from file containing keyword, stopping after limit matches.

Chapter Summary

- **argparse:** The standard solution for Python CLIs.
- **Positional:** Mandatory inputs (cp source dest).
- **Optional:** Flags (ls -l).
- **Validation:** Automatic type checking and help generation.
- **Professionalism:** CLIs distinguish Scripts from Tools.

Chapter 20: Web APIs - Talking to the World

"The Internet is just other people's computers talking JSON."

The Goal: Integration

Your server needs to alert Slack. Your CI/CD pipeline needs to trigger a build on Jenkins. Your monitoring script needs to query AWS IP ranges.

All of these use **HTTP APIs**. Python's requests library is the gold standard for this.

20.1 GET Requests (Reading Data)

Browsers send GET requests to view pages. Scripts do the same to get JSON.

```
import requests

# 1. Send Request
response = requests.get("https://api.github.com/zen")

# 2. Check Status
print(f"Status: {response.status_code}") # 200 = OK

# 3. Read Content
print(f"Message: {response.text}")
```

20.2 Working with JSON Payloads

Most APIs return JSON. Requests has a built-in .json() decoder.

```
r = requests.get("https://jsonplaceholder.typicode.com/todos/1")

if r.status_code == 200:
   data = r.json() # Converts JSON string -> Python Dict
   print(f"Task: {data['title']}")
   print(f"Completed: {data['completed']}")
else:
   print("Error fetching data")
```

20.3 POST Requests (Sending Data)

To create data (e.g., Sending a Slack message), uses POST.

```
url = "https://hooks.slack.com/services/T000/B000/XXXX"
payload = {"text": "Deployment Started!"}

# Requests auto-sets Content-Type: application/json
r = requests.post(url, json=payload)

if r.status_code == 200:
    print("Message Sent!")
```

20.4 Handling Headers (Authentication)

Most APIs require an API Key. This goes in the **Header**.

```
headers = {
    "Authorization": "Bearer MY_SECRET_TOKEN",
    "User-Agent": "MyPythonScript/1.0"
}

r = requests.get("https://api.example.com/secure-data", headers=headers)
```

20.5 DevOps Use Case: The Health Check

A script that monitors a list of websites.

```
import requests
import time

urls = [
    "https://google.com",
    "https://github.com",
    "https://nonexistent-site-123.com"
]

def check_sites():
    for url in urls:
        try:
            r = requests.get(url, timeout=5) # Don't hang forever!
            if r.status_code == 200:
                print(f"[OK] {url}")
            else:
                print(f"[ERR] {url} returned {r.status_code}")
        except requests.exceptions.ConnectionError:
            print(f"[DOWN] {url} unreachable")
    except requests.exceptions.Timeout:
      print(f"[SLOW] {url} timed out")

check_sites()
```

20.6 The Laws of Requests

1. **The Law of Timeouts:** Always set timeout=5. Default is infinite! If the server hangs, your script hangs.
2. **The Law of Status:** Don't assume success. Check if r.status_code == 200.
3. **The Law of Secrets:** Never put API keys in the URL. Use Headers.
4. **The Law of Rate Limiting:** Don't loop a request 1,000 times a second. You will get banned.

[!] Common Mistakes

Error #1: Generic Except

```
try:
  requests.get(url)
except:
  print("Fail")
```

This hides bugs. Catch specific errors: requests.exceptions.RequestException.

Error #2: JSON Decode Error

If the server returns 500 (HTML Error Page), r.json() will crash because it's not JSON. **Fix:** Check status code first.

Practice Problem: The Weather Bot

1. Use a free weather API (like open-meteo.com).
2. Ask the user for a Latitude/Longitude (using input() or argparse).
3. Fetch the current temperature.
4. Print "Bring a Jacket" if temp < 15°C.

Chapter Summary

- **requests:** The most popular Python library (for a reason).
- **Methods:** GET (Read), POST (Write).
- **JSON:** The native language of APIs.
- **Status Codes:** 200 (Good), 404 (Missing), 500 (Server Broken).
- **Automation:** APIs allow scripts to control other systems.

Chapter 21: Python & Git - The DevOps Gatekeeper

"Bad code should never reach production."

The Goal: Automated Quality

You are tired of:

1. Running git commit with messy code.
2. Reviewing PRs with syntax errors.
3. Deploying broken scripts.

The solution is **CI/CD** (Continuous Integration/Deployment). And Python is the glue that makes it work.

21.1 Code Quality Tools (The Robot Reviewer)

Before you commit, you must clean your code.

1. Formatting with black

Stop arguing about spaces vs tabs. Let black decide.

```
pip install black
black my_script.py
# Output: "reformatted my_script.py. All done!"
```

2. Linting with pylint

Find bugs before running the code.

```
pip install pylint
pylint my_script.py
# Output: "Rate: 8.5/10. Error: Undefined variable 'x'"
```

21.2 Git Hooks (The Police)

Git has a hidden folder .git/hooks. You can put Python scripts there to **block bad commits**.

Example: The "No TODOs" Hook

Create .git/hooks/pre-commit (and chmod +x it):

```
#!/usr/bin/env python3
import sys
import subprocess

# 1. Get list of staged files
result = subprocess.run(
    ["git", "diff", "--cached", "--name-only"],
    capture_output=True, text=True
)
files = result.stdout.splitlines()

# 2. Check each file for "TODO"
for file in files:
    if file.endswith(".py"):
        with open(file, "r") as f:
            if "TODO" in f.read():
                print(f"Error: You left a TODO in {file}!")
                sys.exit(1) # Block the commit!

print("Code looks clean. Committing...")
sys.exit(0)
```

Now, if you try to commit a "TODO", Git will say **NO**.

21.3 CI Pipelines (GitHub Actions)

In the cloud (GitHub/GitLab), we define pipelines using YAML. Python scripts are the *steps* in that pipeline.

Example: .github/workflows/test.yml

```
name: Python DevOps Pipeline
```

```yaml
on: [push]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install Dependencies
        run: |
          pip install -r requirements.txt
          pip install pylint black

      - name: Check Formatting
        run: black --check .

      - name: Run Linter
        run: pylint src/

      - name: Run Tests
        run: python -m unittest discover tests/
```

If black or pylint fails (exit code 1), the **Merge Button turns Red**. This is how we protect Production.

21.4 DevOps Use Case: Version Bumping

A script to auto-increment your app version (Semantic Versioning). e.g., v1.0.2 -> v1.0.3.

```python
import argparse

def bump_version(path):
    with open(path, "r") as f:
        version = f.read().strip()

    major, minor, patch = map(int, version.split("."))
    new_version = f"{major}.{minor}.{patch + 1}"

    with open(path, "w") as f:
        f.write(new_version)

    print(f"Bumped: {version} -> {new_version}")

if __name__ == "__main__":
    bump_version("VERSION")
```

21.5 The Laws of CI/CD

1. **The Law of Formatting:** Use black locally. Use black --check in CI.
2. **The Law of Hooks:** Don't rely on humans to remember checks. Use pre-commit.
3. **The Law of Pipelines:** Deployment should be a button press (or Git Push), not a manual command.

[!] Common Mistakes

Error #1: Committing Secrets

You accidentally committed AWS_KEY="123". **Fix:** Use git-secrets or a Python hook to scan for regex patterns like AKIA....

Error #2: Ignoring CI Failures

"It works on my machine!" **Fix:** If CI fails, assume *your machine* is wrong (environment drift).

Chapter Summary

- **Linting:** Tools like pylint find bugs early.
- **Git Hooks:** Python scripts that run before commit or push.
- **CI/CD:** Using Python inside YAML pipelines (GitHub Actions) to automate testing.
- **Quality:** Automation protects you from your own mistakes.

Chapter 22: Python & Docker - The Container Handler

"It works on my machine... so we'll ship your machine."

The Goal: Container Automation

You can run docker run nginx. But what if you need to:

1. Build 50 images?
2. Wait for a database container to be healthy?
3. Clean up all stopped containers older than 1 hour?

You need **Python + Docker**.

22.1 The Docker SDK for Python

While you can use subprocess.run(["docker", ...]), the official library is cleaner.

```
pip install docker
```

Listing Containers

```
import docker

client = docker.from_env()

for container in client.containers.list():
    print(f"ID: {container.short_id} | Image: {container.image.tags} | Status:
{container.status}")
```

22.2 Running Containers

Let's launch a Redis server from Python.

```
try:
    container = client.containers.run(
        "redis:alpine",
        name="my-redis",
        ports={'6379/tcp': 6379},
        detach=True # Run in background
    )
    print(f"Redis started! ID: {container.short_id}")
```

```
except docker.errors.APIError as e:
    print(f"Docker Error: {e}")
```

22.3 Parsing Dockerfiles

Sometimes you need to analyze a Dockerfile as text.

```
def check_dockerfile(path="Dockerfile"):
    with open(path, "r") as f:
        lines = f.readlines()

    for line in lines:
        if line.startswith("FROM"):
            image = line.split()[1]
            if "latest" in image:
                print(f"[WARN] Using 'latest' tag in {image} is dangerous!")

# check_dockerfile()
```

22.4 DevOps Use Case: The Janitor Script

A script to clean up stopped containers and unused images (Pruning).

```
import docker

client = docker.from_env()

# Safety Check: Is Docker running?
try:
    client.ping()
except docker.errors.DockerException:
    print("Error: Could not connect to Docker. Is it running? Do you have permission?")
    exit(1)

def prune_system():
    # 1. Remove Stopped Containers
    pruned_containers = client.containers.prune()
    print(f"Deleted Containers: {pruned_containers['ContainersDeleted']}")

    # 2. Remove Dangling Images (None:None)
    pruned_images = client.images.prune()
    print(f"Deleted Images: {len(pruned_images.get('ImagesDeleted', []))}")

if __name__ == "__main__":
    confirm = input("Run Docker Prune? (y/n): ")
    if confirm.lower() == 'y':
        prune_system()
```

22.5 Integration Tests (Waiting for Services)

When testing, you need to wait for the DB to handle connections.

```
import time

def wait_for_db(container):
    print("Waiting for DB...")
    for i in range(10):
        if "Ready to accept connections" in container.logs().decode("utf-8"):
            print("DB is Ready!")
            return True
        time.sleep(1)
    return False

# container = client.containers.run("postgres", detach=True)
# wait_for_db(container)
```

22.6 The Laws of Container Scripts

1. **The Law of Binding:** Use docker.from_env() to connect to your local Docker socket.
2. **The Law of Cleanup:** Always container.stop() and container.remove() in your test scripts (use try/finally).

3. **The Law of Tags:** Never deploy latest. Parse your Dockerfiles to enforce specific versions.

[!] Common Mistakes

Error #1: Permission Denied

docker.errors.DockerException: Error while fetching server API version **Fix:** Your user is not in the docker group. Run sudo usermod -aG docker $USER.

Error #2: Resource Leaks

You started a container in a loop but verify crashed. Now you have 50 zombie containers eating RAM. **Fix:** Implementing a "Cleanup Hook" (using atexit or finally) is mandatory.

Practice Problem: The Image Builder

Write a script that:

1. Takes a directory path as input.
2. Checks if a Dockerfile exists.
3. Builds the image using client.images.build().
4. Tags it as myapp:v1.

Chapter Summary

- **Docker SDK:** A powerful Python library for controlling the Docker Daemon.
- **Automation:** Start, Stop, Build, and Prune containers programmatically.
- **Testing:** Use Python to spin up ephemeral environments for tests.
- **Sanity:** Scripts enable complex orchestrations (like "Start DB, Wait, Start App").

Chapter 23: Cloud & IaC - Infinite Scale

"Friends don't let friends click in the console."

The Goal: Infrastructure as Code (IaC)

You are done with manual server setup. You want to code your infrastructure.

1. **Cloud:** AWS/GCP/Azure via SDKs.
2. **Config:** Ansible/Terraform via Wrappers.

Python is the native language of the Cloud.

23.1 The AWS SDK (boto3)

boto3 is the standard Python library for Amazon Web Services. It allows you to create S3 buckets, EC2 instances, and IAM users.

Installation

```
pip install boto3
```

(Requires ~/.aws/credentials configuration).

Example: S3 Bucket Lister

```
import boto3

# Create S3 Client
s3 = boto3.client('s3')

# List Buckets
response = s3.list_buckets()

print("My Buckets:")
for bucket in response['Buckets']:
    print(f"- {bucket['Name']}")
```

23.2 Uploading Files to the Cloud

The DevOps classic: **The Backup Script**.

```
import boto3
from botocore.exceptions import NoCredentialsError

def upload_to_aws(local_file, bucket, s3_file):
    s3 = boto3.client('s3')

    try:
        s3.upload_file(local_file, bucket, s3_file)
        print("Upload Successful")
        return True
    except FileNotFoundError:
        print("The file was not found")
    except NoCredentialsError:
        print("Credentials not available")
    return False

# Usage
upload_to_aws('backup.zip', 'my-infra-backups', '2023-10-27-backup.zip')
```

23.3 Python & Ansible

Ansible is written in Python. You can write your own **Ansible Modules** in Python if the built-in ones aren't enough.

An Ansible Module Structure (Simplified)

An Ansible module is just a script that prints JSON to stdout.

```
#!/usr/bin/python
import json
import sys

def main():
    # 1. Read Arguments (from Ansible)
    # 2. Do Work (e.g. Check if a user exists)

    result = {
        "changed": True,
        "msg": "User created successfully"
    }

    # 3. Print JSON
    print(json.dumps(result))

if __name__ == '__main__':
    main()
```

Running Ansible from Python

Sometimes you want Python to trigger a Playbook.

```
import subprocess

def run_playbook(playbook_path, inventory):
    cmd = ["ansible-playbook", "-i", inventory, playbook_path]
    subprocess.run(cmd, check=True)
```

23.4 DevOps Use Case: The EC2 Manager

A script to find and stop expensive "Development" servers at night.

```
import boto3

ec2 = boto3.resource('ec2')

def stop_dev_instances():
    # Filter: Tag "Env=Dev" and State="running"
    filters = [
        {'Name': 'tag:Env', 'Values': ['Dev']},
        {'Name': 'instance-state-name', 'Values': ['running']}
    ]

    instances = ec2.instances.filter(Filters=filters)

    for instance in instances:
        print(f"Stopping {instance.id}...")
        instance.stop()

if __name__ == "__main__":
    stop_dev_instances()
```

23.5 The Laws of Cloud Scripts

1. **The Law of Regions:** Always specify your region (us-east-1). Defaults kill you.
2. **The Law of Tags:** Only touch resources with specific tags. Don't stop Prod by accident!

3. **The Law of Cost:** Remember that API calls (and resources) cost money. optimize your loops.
4. **The Law of Secrets:** Never commit your AWS_SECRET_ACCESS_KEY. Use Roles or Environment Variables.

[!] Common Mistakes

Error #1: Hardcoding Keys

boto3.client('s3', aws_access_key_id='AKIA...') # FIREABLE OFFENSE!

Fix: Let boto3 find them in ~/.aws/credentials or Environment Variables automatically.

Error #2: Pagination

AWS returns only 1000 items (e.g., S3 objects or EC2 instances). If you have 5000, your script only sees the first 1000. **Fix:** Use boto3 **Paginators**.

Chapter Summary

- **Cloud SDKs:** Python controls the cloud (boto3).
- **Automation:** Script backups, cleanups, and scaling events.
- **Ansible:** Python powers configuration management logic.
- **Risk:** With great power (ec2.terminate_instances) comes great responsibility.

Chapter 24: Capstone Project - The Inspector CLI

"Time to build."

The Mission

You have been hired as a DevOps Engineer. Your manager wants a single tool called inspector that can:

1. Check **Disk Space**.
2. Check **Website Health** (API Status).
3. Report results to **JSON** or **Terminal**.
4. Clean up **Old Logs**.

We will use everything we've learned: argparse, shutil, requests, logging, and OOP.

24.1 Project Structure

```
inspector/
├── main.py       # The CLI Entrypoint
├── checks/       # Module Folder
│   ├── __init__.py
│   ├── system.py  # Disk/CPU checks
│   └── web.py     # API checks
└── requirements.txt
```

requirements.txt:

```
requests==2.28.0
```

24.2 The System Check Module (checks/system.py)

Using shutil to check disk usage.

```python
import shutil

class SystemCheck:
    def check_disk(self, mount_point="/"):
        total, used, free = shutil.disk_usage(mount_point)
        percent_used = (used / total) * 100
        return {
            "check": "disk_usage",
            "mount": mount_point,
            "percent": round(percent_used, 2),
            "status": "ALARM" if percent_used > 80 else "OK"
        }
```

24.3 The Web Check Module (checks/web.py)

Using requests to monitor endpoints.

```python
import requests
```

```
class WebCheck:
    def check_site(self, url):
        try:
            r = requests.get(url, timeout=3)
            return {
                "check": "website",
                "url": url,
                "code": r.status_code,
                "latency": r.elapsed.total_seconds(),
                "status": "OK" if r.status_code == 200 else "ERROR"
            }
        except requests.exceptions.RequestException:
            return {
                "check": "website",
                "url": url,
                "status": "DOWN"
            }
```

24.4 The CLI Logic (main.py)

Using argparse to wire it together.

```
import argparse
import json
import sys
from checks.system import SystemCheck
from checks.web import WebCheck

def main():
    # 1. Setup Arguments
    parser = argparse.ArgumentParser(description="Inspector: DevOps Health Tool")
    parser.add_argument("--check-disk", action="store_true", help="Check Disk Usage")
    parser.add_argument("--check-web", help="Check a specific Website URL")
    parser.add_argument("--json", action="store_true", help="Output as JSON")

    args = parser.parse_args()
    results = []
    # 2. Run Checks
    if args.check_disk:
        sys_checker = SystemCheck()
        results.append(sys_checker.check_disk())

    if args.check_web:
        web_checker = WebCheck()
        results.append(web_checker.check_site(args.check_web))

    if not results:
        parser.print_help()
        sys.exit(1)

    # 3. Output
    if args.json:
        print(json.dumps(results, indent=2))
    else:
        print("--- REPORT ---")
        for res in results:
            status = res.get('status')
            print(f"[{status}] {res['check']} | Data: {res}")
        print("--------------")

if __name__ == "__main__":
    main()
```

24.5 Running The Tool

Scenario 1: Human Readable Check

```
$ python main.py --check-disk --check-web https://google.com

--- REPORT ---
[OK] disk_usage | Data: {'check': 'disk_usage', 'percent': 45.2, 'status': 'OK'}
[OK] website | Data: {'check': 'website', 'url': 'https://google.com', 'code': 200, 'status': 'OK'}
```

Scenario 2: JSON for Automation

```
$ python main.py --check-disk --json
[
 {
   "check": "disk_usage",
   "mount": "/",
   "percent": 45.2,
   "status": "OK"
 }
]
```

24.6 Challenge: Extensions

You built V1. Now build V2.

1. **Add Logging:** Write output to inspector.log.
2. **Add Email Alerts:** Use smtplib to email if Status == "ALARM".

3. Configuration via YAML

Hardcoding URLs is bad. Let's use PyYAML.

config.yaml

```
websites:
 - https://google.com
 - https://github.com
disk: /
```

Modified main.py

```
import yaml # pip install pyyaml

def load_config(path="config.yaml"):
   try:
      with open(path, "r") as f:
         return yaml.safe_load(f)
```

```
    except FileNotFoundError:
        print(f"Error: Config file '{path}' not found.")
        sys.exit(1)
    except yaml.YAMLError as e:
        print(f"Error: Invalid YAML format. {e}")
        sys.exit(1)

# Inside main():
config = load_config()
for url in config['websites']:
    # check_site(url)...
```

Book Conclusion

You started with print("Hello World"). You ended with a Modular, Object-Oriented, CLI Tool that performs real-world checks.

You are a Python DevOps Engineer. Go automate the world.

Appendix A: DevOps Python Best Practices

1. Code Logic

- **Avoid Global Variables:** Keep state inside functions or classes.
- **Return vs Print:** Functions should return values, not print them.
- **Main Guard:** Always use if __name__ == "__main__":.

2. Formatting & Style

- **Snake Case:** my_variable, not myVariable.
- **Docstrings:** Document every function.
- **Formatting:** Run black before committing.
- **Linting:** Run pylint to catch bugs.

3. Reliability

- **Error Handling:** Use try/except blocks for I/O and Network calls.
- **Timeouts:** Never make a network request without a timeout.
- **Logging:** Use logging module, not print statements in production.

4. Security

- **Secrets:** NEVER commit API Keys or Passwords.
- **Env Vars:** Use os.environ for credentials.
- **Sanitization:** Never use shell=True with user input.

5. Project Structure

- **Requirements:** Always include a requirements.txt.
- **Gitignore:** Always ignore .venv/, __pycache__/, and *.env.

Appendix B: DevOps Career Roadmap

Stage 1: The Scripter (Junior)

- **Skills:** Basic Python, Bash replacement, Cron jobs.
- **Projects:** Backup scripts, log parsers, health checks.
- **Focus:** Learning syntax and error handling.

Stage 2: The Automator (Mid-Level)

- **Skills:** Requests (APIs), Git Hooks, CI/CD Pipelines, Docker SDK.
- **Projects:** Deployment tools, Slack bots, automated testing.
- **Focus:** Reliability, logging, and extensive library usage.

Stage 3: The Architect (Senior)

- **Skills:** Infrastructure as Code (Boto3/Ansible), Advanced OOP, Packaging.
- **Projects:** Internal Platforms (IDPs), Custom Orchestrators, Security Compliance tools.
- **Focus:** Scalability, Maintainability, and Design Patterns.

Learning Path

1. Master **Core Python** (Chapters 1-10).
2. Understand **System Internals** (Chapters 17-18).
3. Build **Tools** not Scripts (Chapter 19+24).
4. Integrate with **Cloud/K8s** (Chapter 22-23).

Appendix C: Common Beginner Mistakes

"Experience is the name everyone gives to their mistakes." — Oscar Wilde

1. Mutable Default Arguments

The Bug:

```
def add_server(server, list=[]):
  list.append(server)
```

```
    return list

print(add_server("srv1")) # ['srv1']
print(add_server("srv2")) # ['srv1', 'srv2'] -- Wait, what?
```

The Why: The list is created *once* when the function is defined, not every time it runs. **The Fix:** Use None.

```
def add_server(server, list=None):
    if list is None:
        list = []
```

2. The Broad Exception

The Bug:

```
try:
    do_something()
except:
    pass # "Silence is golden"
```

The Why: This catches *everything*, including KeyboardInterrupt (Ctrl+C). You can't stop your script! **The Fix:** Catch specific errors.

```
except (ValueError, FileNotFoundError):
    logging.error("File missing")
```

3. Shell Injection

The Bug:

```
subprocess.run(f"ping {user_input}", shell=True)
```

The Why: If user inputs ; rm -rf /, your server is gone. **The Fix:** Use lists, not strings.

```
subprocess.run(["ping", user_input])
```

4. Shadowing Built-ins

The Bug:

```
list = [1, 2, 3] # You just killed the 'list()' function
str = "Hello"    # You just killed 'str()'
```

The Why: Python lets you overwrite standard functions. **The Fix:** Use synonyms like my_list or data_str.

5. Path Hardcoding

The Bug:

```
path = "C:\\Users\\Admin\\file.txt" # Breaks on Linux
```

The Why: Windows uses \, Linux uses /. **The Fix:** Use pathlib or os.path.join.

```
from pathlib import Path
path = Path.home() / "file.txt"
```

Appendix D: Code Review Checklist

"Code is a liability. The less of it you have, the better."

Use this checklist before merging any Pull Request.

1. Functionality & Logic

- **Does it work?** Has the author provided proof (screenshots/logs)?
- **Edge Cases:** What happens if the file is empty? If the API is down?
- **Complexity:** Can this 50-line function be 5 lines?
- **DRY (Don't Repeat Yourself):** Is there copied logic? Make it a function.

2. Style (PEP 8)

- **Naming:** snake_case for variables/functions, PascalCase for classes.
- **Docstrings:** Does every function explain *what* it does and *what* it returns?
- **Formatting:** Has black been run?
- **Imports:** Are imports at the top? Are unused imports removed?

3. Security

- **Secrets:** Are there API keys/passwords in the code? (REJECT IMMEDIATELY).
- **Input Validation:** Is user input sanitized?
- **Shell Safety:** Is shell=True avoided in subprocess?
- **Dependencies:** Are we pinning versions in requirements.txt?

4. Reliability & DevOps

- **Logging:** Are there print() statements? (Ask to change to logging).
- **Error Handling:** Are there bare except: blocks?
- **Configuration:** Are parameters (URLs, timeouts) configurable (Env Vars/YAML)?
- **Cleanup:** Does the script close files and DB connections (using with)?

5. Testing

- **Unit Tests:** Are there tests for the new logic?
- **Happy Path:** Does it work when everything is right?
- **Sad Path:** Does it fail gracefully when things go wrong?