# Python Automation Fundamentals

## 1. What Is Automation?

Automation means using a program to perform repetitive work automatically instead of doing the same task manually.

For example, imagine you receive 200 files every week:

```
report_01.pdf
report_02.pdf
report_03.pdf
...
report_200.pdf
```

Manually:

```
Find file
   ↓
Rename file
   ↓
Move file
   ↓
Repeat 200 times
```

With Python:

```
Python script
     ↓
Find files
     ↓
Rename files
     ↓
Create folders
     ↓
Move files
     ↓
Finish
```

The goal of automation is not simply "using Python."

The goal is:

> **Turn a predictable manual process into a reliable repeatable program.**

---

# 2. Why Python Is Useful for Automation

Python is particularly useful for automation because it has a large standard library and a simple syntax.

Python can interact with:

- files
- folders
- operating-system commands
- environment variables
- APIs
- databases
- spreadsheets
- JSON and CSV data
- email systems
- web services
- scheduled jobs

For example:

```
Files
  ↓
Python
  ↓
Process data
  ↓
Database
  ↓
API
  ↓
Report
```

This makes Python useful for both small scripts and larger automation systems.

---

# 3. Automation Mental Model

A useful automation model is:

```
INPUT
  ↓
READ
```

```
  ↓
PROCESS
  ↓
DECIDE
  ↓
ACTION
  ↓
VERIFY
  ↓
LOG
```

Example:

```
Input:
100 PDF files

Read:
Find all PDFs

Process:
Check filenames

Decide:
Which files are invoices?

Action:
Move invoices to /invoices

Verify:
Confirm files moved

Log:
Record what happened
```

This pattern appears in almost every useful automation script.

---

# 4. Start With the Manual Process

Before writing automation code, understand the manual workflow.

Suppose you manually organize downloads:

```
Downloads/
    photo.jpg
```

```
    report.pdf
    invoice.pdf
    song.mp3
    presentation.pptx
```

Your manual rules might be:

```
.jpg/.png → Images
.pdf      → Documents
.mp3      → Music
.pptx     → Presentations
```

Now the automation becomes:

```
Find files
   ↓
Read extension
   ↓
Choose destination
   ↓
Create destination
   ↓
Move file
```

This is much easier to implement than saying:

> "I want to automate my Downloads folder."

---

# 5. Identify Repetition

Good automation candidates usually contain repetition.

Look for tasks like:

```
Open 100 files
Rename 100 files
Copy 100 files
Check 100 records
Send 50 similar requests
Convert 200 files
Generate 100 reports
```

Ask:

1. Is the process repetitive?
2. Are the rules predictable?

3. Can Python access the required data?
4. Can the task be performed safely without constant human decisions?

If yes, automation may be appropriate.

---

# 6. Python Automation Tools

Python provides several useful tools.

Common examples include:

```
pathlib
os
shutil
subprocess
csv
json
sqlite3
datetime
logging
```

Third-party packages can extend this further.

The important point is that automation is not one Python feature.

It is a combination of Python capabilities.

---

# 7. Working With Paths

`pathlib` is one of the most useful tools for automation.

Python's documentation describes `pathlib` as an object-oriented interface for filesystem paths.

```
from pathlib import Path

folder = Path("downloads")

print(folder.exists())
print(folder.is_dir())
```

You can create paths:

```
file = folder / "report.pdf"
```

Instead of manually constructing:

```
"downloads/report.pdf"
```

This makes filesystem code clearer and more portable.

---

# 8. Finding Files

Suppose you want all PDF files:

```
from pathlib import Path

folder = Path("downloads")

for file in folder.glob("*.pdf"):
    print(file)
```

For recursive searching:

```
for file in folder.rglob("*.pdf"):
    print(file)
```

You can therefore automate tasks across an entire directory tree.

---

# 9. Checking File Information

Python can inspect files.

```
from pathlib import Path

file = Path("report.pdf")

print(file.exists())
print(file.name)
print(file.suffix)
print(file.parent)
```

You can also inspect metadata:

```
print(file.stat().st_size)
```

This can help automate decisions.

For example:

```
if file.stat().st_size > 10_000_000:
    print("Large file")
```

---

# 10. Creating Folders

```
from pathlib import Path

folder = Path("reports")

folder.mkdir(exist_ok=True)
```

For nested directories:

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

This is useful because automation should not assume that a destination folder already exists.

---

# 11. Renaming Files

```
from pathlib import Path

old_name = Path("report.pdf")
new_name = Path("final_report.pdf")

old_name.rename(new_name)
```

For multiple files:

```
from pathlib import Path

folder = Path("reports")

for file in folder.glob("*.txt"):
    new_name = file.with_name(
        f"processed_{file.name}"
    )

    file.rename(new_name)
```

Automation becomes powerful when one rule can process many files.

---

# 12. Moving Files

Python can move files using `shutil`.

```
from pathlib import Path
import shutil

source = Path("report.pdf")
destination = Path("documents/report.pdf")

destination.parent.mkdir(
    parents=True,
    exist_ok=True
)

shutil.move(source, destination)
```

The `shutil` module provides high-level operations for files and collections of files, including copying and removal.

---

# 13. Copying Files

```
import shutil

shutil.copy(
    "report.pdf",
    "backup/report.pdf"
)
```

For preserving additional metadata when appropriate:

```
shutil.copy2(
    "report.pdf",
    "backup/report.pdf"
)
```

Do not assume that every piece of filesystem metadata is preserved across platforms; Python's documentation notes limitations around file metadata when using these functions.

---

# 14. Deleting Files

A file can be removed with `Path.unlink()`:

```
from pathlib import Path

file = Path("old_report.pdf")
```

```
if file.exists():
    file.unlink()
```

This should be treated carefully.

Automation can execute actions much faster than a human.

A mistake like:

```
for file in folder.iterdir():
    file.unlink()
```

can delete many files.

Always test destructive automation on sample data first.

---

# 15. A File Organizer

Now combine the concepts.

Suppose:

```
downloads/
    photo.jpg
    report.pdf
    presentation.pptx
    song.mp3
```

We want:

```
downloads/
    Images/
    Documents/
    Presentations/
    Music/
```

A simple automation script:

```
from pathlib import Path
import shutil

downloads = Path("downloads")

categories = {
    ".jpg": "Images",
    ".jpeg": "Images",
    ".png": "Images",
    ".pdf": "Documents",
```

```
    ".pptx": "Presentations",
    ".mp3": "Music",
}

for file in downloads.iterdir():

    if not file.is_file():
        continue

    category = categories.get(file.suffix.lower())

    if category is None:
        continue

    destination_folder = downloads / category

    destination_folder.mkdir(
        exist_ok=True
    )

    shutil.move(
        file,
        destination_folder / file.name
    )
```

This is a real automation workflow:

```
Discover
   ↓
Classify
   ↓
Create destination
   ↓
Move
```

---

# 16. Why Functions Matter in Automation

Avoid putting the entire automation workflow into one huge script.

Instead:

```
def find_files():
    ...
```

```
def classify_file(file):
    ...


def move_file(file, destination):
    ...


def organize_files():
    ...
```

Then:

```
organize_files()
```

This makes automation easier to:

- understand
- test
- debug
- modify
- reuse

---

# 17. Separate Rules From Actions

Consider:

```
if file.suffix == ".pdf":
    destination = "Documents"
```

The rule is:

```
PDF → Documents
```

The action is:

```
Move file
```

Keep these concepts separate when possible.

```
def get_category(file):
    ...


def move_file(file, category):
    ...
```

This makes the system easier to change.

For example:

```
Old rule:
.pdf → Documents

New rule:
.pdf → PDFs
```

You change the classification rule instead of rewriting the entire automation.

---

# 18. Automating CSV Data

Python can automate structured data too.

Example:

```
import csv

with open("users.csv", newline="") as file:
    reader = csv.DictReader(file)

    for row in reader:
        print(row["name"])
```

Now imagine a CSV containing:

```
name,email,status
Hafsa,hafsa@example.com,active
Asim,asim@example.com,inactive
```

Python can:

```
Read CSV
   ↓
Filter users
   ↓
Process records
   ↓
Generate output
```

This pattern is useful for reports, migrations, data cleaning, and administrative workflows.

---

# 19. Automating JSON Data

Python's `json` module can process JSON.

```
import json

with open("config.json") as file:
    data = json.load(file)

print(data)
```

You can then transform the data:

```
for resource in data["resources"]:
    print(resource["title"])
```

Automation often means moving information between formats:

```
CSV
 ↓
Python
 ↓
JSON
 ↓
Database
```

---

# 20. Automating APIs

Python can also automate web services through APIs.

For example:

```
Python script
     ↓
API request
     ↓
JSON response
     ↓
Process data
     ↓
Save result
```

A typical workflow could be:

```
response = requests.get(url)

response.raise_for_status()
```

```
data = response.json()
```

Then:

```
for item in data:
    process(item)
```

This can automate tasks such as:

- retrieving data
- synchronizing systems
- generating reports
- checking service status
- submitting records
- processing API responses

---

# 21. Automation and Databases

Python automation can interact with databases too.

For example:

```
Database
   ↓
Python
   ↓
Find incomplete records
   ↓
Process them
   ↓
Update database
```

Example:

```
import sqlite3

connection = sqlite3.connect("app.db")

rows = connection.execute("""
    SELECT id, title
    FROM resources
    WHERE processed = 0
""").fetchall()
```

```
for resource_id, title in rows:
    print("Processing:", title)

connection.close()
```

This connects automation with the database concepts covered earlier.

---

# 22. Running Other Programs

Sometimes Python needs to execute an external command.

The `subprocess` module is designed for spawning processes and working with their input/output.

Example:

```
import subprocess

result = subprocess.run(
    ["python", "--version"],
    capture_output=True,
    text=True
)

print(result.stdout)
```

The important idea is:

```
Python
   ↓
Operating system
   ↓
External program
   ↓
Output
   ↓
Python
```

This is useful when an existing command-line tool already performs part of the job.

---

# 23. Avoid `shell=True` Without a Reason

Be careful with:

```
subprocess.run(
    command,
    shell=True
)
```

If untrusted input reaches a shell command, shell injection can become a security problem.

Python's documentation notes that when `shell=True` is explicitly used, the application is responsible for correctly handling shell metacharacters and quoting.

Prefer argument lists:

```
subprocess.run([
    "python",
    "script.py"
])
```

instead of constructing shell command strings from user input.

---

# 24. Checking Command Success

External commands return an exit status.

You can use:

```
result = subprocess.run(
    ["python", "--version"],
    capture_output=True,
    text=True
)

print(result.returncode)
```

A common pattern is:

```
subprocess.run(
    ["python", "script.py"],
    check=True
)
```

With `check=True`, a failed command causes Python to raise an exception instead of silently continuing.

This is important in automation because a failed step should not always be treated as success.

---

# 25. Environment Variables

Automation scripts often need configuration.

For example:

```
API URL
API key
database URL
environment
output directory
```

Read environment variables with:

```
import os

api_key = os.environ["API_KEY"]
```

This is safer than placing secrets directly in source code.

Automation should separate:

```
Code
+
Configuration
+
Secrets
```

---

# 26. Logging Automation

Automation should tell you what happened.

Instead of:

```
print("done")
```

use logging for more structured information.

```
import logging

logging.basicConfig(
    level=logging.INFO
)

logging.info("Automation started")
logging.info("Processing report.pdf")
logging.info("Automation finished")
```

Useful levels include:

```
DEBUG
INFO
WARNING
ERROR
CRITICAL
```

Logging becomes especially important when automation runs without someone watching it.

---

# 27. Error Handling

Automation should expect failures.

For example:

```
from pathlib import Path

file = Path("report.pdf")

try:
    content = file.read_text()
except FileNotFoundError:
    print("File does not exist")
```

For larger automation:

```
try:
    process_file(file)
except Exception as error:
    logging.exception(
        "Failed to process %s",
        file
    )
```

However, avoid blindly catching every exception.

If the program can safely continue, handle the specific failure.

---

# 28. Continue or Stop?

Suppose you process 1,000 files.

File 427 fails.

Should the entire automation stop?

Sometimes yes.

Sometimes no.

For independent files:

```
File 1 → success
File 2 → success
File 3 → failure
File 4 → success
...
```

you might log the failure and continue.

For dependent steps:

```
Create database
   ↓
Insert records
   ↓
Generate report
```

if the database step fails, continuing may produce invalid results.

Automation design therefore requires understanding dependencies.

---

# 29. Idempotent Automation

An important automation concept is **idempotency**.

An idempotent operation can be safely run multiple times without causing unintended repeated effects.

For example, this is safer:

```
folder.mkdir(exist_ok=True)
```

Running it repeatedly does not create duplicate folders.

But blindly doing:

```
shutil.copy(
    "report.pdf",
    "backup/report.pdf"
)
```

may overwrite or duplicate data depending on the situation.

Design automation so that running it again is predictable.

---

# 30. Dry Run Mode

A powerful safety feature is a dry-run mode.

Instead of immediately moving files:

```
print(
    f"Would move {file} → {destination}"
)
```

You can inspect the planned actions first.

Example:

```
DRY_RUN = True

if DRY_RUN:
    print(
        f"Would move {file} → {destination}"
    )
else:
    shutil.move(file, destination)
```

This is extremely useful for destructive or large-scale automation.

---

# 31. Validation Before Action

Before changing anything, validate your assumptions.

For example:

```
if not source.exists():
    raise FileNotFoundError(source)

if not destination.parent.exists():
    destination.parent.mkdir(
        parents=True
    )
```

For file organization:

```
Check source
   ↓
Check extension
   ↓
Determine destination
   ↓
Check destination
```

```
   ↓
Perform action
```

Don't let automation blindly act on assumptions.

---

# 32. Backups

If an automation script modifies important data, consider backups.

Example:

```
Original files
     ↓
Backup
     ↓
Automation
```

This gives you a recovery option if the script behaves incorrectly.

A backup is not a replacement for testing, but it reduces the impact of mistakes.

---

# 33. Scheduling Automation

A Python script normally runs when you execute it:

```
python automate.py
```

But automation often needs to run automatically.

Common scheduling options include:

```
Windows Task Scheduler
Linux cron
CI/CD pipelines
Cloud schedulers
Application task queues
```

The Python program itself does not necessarily need to remain open forever.

A scheduler can launch it when required.

---

# 34. Example Scheduled Workflow

Imagine a daily report:

```
Every day at 8:00 AM
        ↓
Scheduler launches Python
```

```
        ↓
Python retrieves API data
        ↓
Processes data
        ↓
Creates report
        ↓
Saves report
        ↓
Logs result
```

The scheduler controls **when**.

Python controls **what happens**.

---

# 35. Automation Project Structure

A tiny automation can be one file:

```
automation.py
```

A larger project might look like:

```
automation_project/
│
├── src/
│   ├── main.py
│   ├── files.py
│   ├── api.py
│   ├── processing.py
│   └── reporting.py
│
├── tests/
│   └── test_processing.py
│
├── logs/
│
├── .env
├── requirements.txt
└── README.md
```

Do not create a large architecture for a five-line script.

Start simple and add structure as complexity grows.

---

# 36. Testing Automation

Automation can cause real-world changes, so testing is important.

Instead of testing directly against:

```
C:\ImportantFiles
```

create:

```
test_data/
```

Then test against controlled files.

For example:

```
test_data/
    photo.jpg
    report.pdf
    notes.txt
```

Run the automation.

Expected:

```
test_data/
    Images/photo.jpg
    Documents/report.pdf
    notes.txt
```

Only after testing should you consider running it against real data.

---

# 37. Unit Testing Automation Logic

Separate logic from filesystem actions.

Instead of:

```
def organize():
    # everything
```

use:

```
def get_category(extension):
    categories = {
        ".pdf": "Documents",
        ".jpg": "Images",
    }
```

```
    return categories.get(extension)
```

Now it can be tested:

```
def test_get_category():
    assert get_category(".pdf") == "Documents"
```

This makes automation safer because important decisions can be tested without actually moving files.

---

# 38. A Better File Organizer

A more maintainable version:

```
from pathlib import Path
import shutil

CATEGORIES = {
    ".jpg": "Images",
    ".jpeg": "Images",
    ".png": "Images",
    ".pdf": "Documents",
    ".docx": "Documents",
    ".pptx": "Presentations",
}


def get_category(file):
    return CATEGORIES.get(
        file.suffix.lower()
    )


def move_file(file, folder):
    folder.mkdir(
        parents=True,
        exist_ok=True
    )

    destination = folder / file.name

    shutil.move(file, destination)
```

```
def organize(folder):
    for file in folder.iterdir():

        if not file.is_file():
            continue

        category = get_category(file)

        if category is None:
            continue

        destination = folder / category

        move_file(
            file,
            destination
        )


if __name__ == "__main__":
    organize(Path("downloads"))
```

Now the automation has separate responsibilities:

```
get_category()
      ↓
classification

move_file()
      ↓
action

organize()
      ↓
workflow
```

---

# 39. Automation Workflow Design

Before coding, write the workflow in plain language.

For example:

```
1. Find all PDF files.
2. Ignore files already inside /processed.
```

```
3. Check file size.
4. Rename files using a consistent format.
5. Move them to /processed.
6. Record successful operations.
7. Record failures.
```

Then convert each step into code.

This is often easier than immediately writing Python.

---

# 40. Common Automation Mistakes

## Mistake 1: Automating before understanding the process

If the manual process is unclear, the automation will be unclear.

---

## Mistake 2: No safety checks

Never assume files exist or destinations are safe.

---

## Mistake 3: Testing on real data

Use sample data first.

---

## Mistake 4: No logging

When something fails tomorrow, you need to know what happened.

---

## Mistake 5: One giant script

Separate reusable logic into functions.

---

## Mistake 6: Hard-coded paths

Avoid:

```
"C:\\Users\\Name\\Desktop\\files"
```

when a configurable path would work better.

---

## Mistake 7: Hard-coded secrets

Keep credentials in environment variables or a secure configuration system.

---

## Mistake 8: Ignoring repeated execution

Ask:

> What happens if this script runs twice?

---

### Mistake 9: Using shell commands unnecessarily

If Python can safely perform the operation directly, you may not need `subprocess`.

---

## 41. Automation Security

Automation can have significant privileges.

A script that can:

```
read files
write files
delete files
run commands
access APIs
access databases
```

should be treated carefully.

Follow the principle of least privilege:

```
Only give the script
the permissions it actually needs.
```

For example, a report generator may only need:

```
read input/
write reports/
```

It does not need permission to delete the entire filesystem.

---

## 42. Automation and `subprocess` Security

Avoid this pattern with untrusted input:

```
import subprocess

command = input("Command: ")

subprocess.run(
    command,
    shell=True
)
```

The user is effectively being allowed to construct a shell command.

Prefer controlled commands:

```
subprocess.run([
    "python",
    "generate_report.py"
])
```

Python's subprocess documentation specifically highlights shell injection concerns when using `shell=True`.

---

# 43. Real World Automation Examples

Python can automate:

### File Management

```
Downloads organizer
Backup system
File renamer
Duplicate detector
Archive creator
```

### Data

```
CSV cleanup
JSON transformation
Report generation
Data validation
Database updates
```

### APIs

```
API synchronization
Data collection
Status checks
Automated reporting
```

### Development

```
Build scripts
Testing commands
Code generation
```

```
Project setup
Deployment helpers
```

### Business Workflows

```
Invoice processing
Report generation
Data exports
Scheduled tasks
File processing
```

---

# 44. haas.dev Example

Imagine you have a folder containing resource PDFs:

```
resources/
    python_functions.pdf
    python_classes.pdf
    api_basics.pdf
    oop.pdf
```

You want to automatically generate metadata.

The workflow could be:

```
Find PDFs
   ↓
Extract filename
   ↓
Convert filename to title
   ↓
Determine category
   ↓
Create metadata
   ↓
Save JSON
```

For example:

```
{
  "title": "Python Functions",
  "file": "python_functions.pdf",
  "category": "Python"
}
```

Python could process hundreds of resources using the same rules.

This is a good example of automation because the task is:

- repetitive
- predictable
- rule-based
- easy to verify

---

# 45. Mini Project: Downloads Organizer

Build a production-style Downloads organizer.

## Requirements

Input:

```
downloads/
```

Create:

```
Images/
Documents/
Videos/
Audio/
Archives/
Other/
```

Rules:

```
.jpg .jpeg .png → Images
.pdf .docx .txt → Documents
.mp4 .mkv       → Videos
.mp3 .wav       → Audio
.zip .rar .7z   → Archives
anything else   → Other
```

Requirements:

1. Do not move directories.
2. Ignore unknown hidden/system files when appropriate.
3. Create folders automatically.
4. Add a dry-run mode.
5. Log every action.
6. Handle errors without silently hiding them.

7. Avoid overwriting files accidentally.
8. Make the script safe to run more than once.

---

# 46. Five Minute Challenge

Write a script that scans:

```
reports/
```

and finds every `.pdf`.

For each PDF, print:

```
filename
file size
parent directory
```

Example:

```
report.pdf
Size: 125 KB
Folder: reports
```

Do not modify any files.

The goal is to practice **read-only automation** before performing actions.

---

# 47. Exercises

## Exercise 1

Create a script that renames:

```
file1.txt
file2.txt
file3.txt
```

to:

```
document_1.txt
document_2.txt
document_3.txt
```

---

## Exercise 2

Create a backup script that copies all `.py` files from:

```
project/
```

to:

```
backup/
```

---

### Exercise 3

Read a CSV file and print only records whose status is:

```
pending
```

---

### Exercise 4

Create a JSON report containing:

```
total files
number of PDFs
number of images
number of large files
```

---

### Exercise 5

Build an automation that:

```
Reads resources from SQLite
        ↓
Finds unpublished resources
        ↓
Generates a JSON report
        ↓
Marks processed records
```

Use a transaction so that the database is not left in an inconsistent state.

---

# 48. Interview Questions

## Beginner

1. What is automation?
2. Why is Python useful for automation?
3. What types of tasks are good candidates for automation?
4. What is `pathlib`?
5. How can Python find files?

6. How can Python rename a file?
7. How can Python move a file?
8. What is `shutil`?
9. What is `subprocess`?
10. Why is logging useful in automation?

### Intermediate

11. What makes an automation script reliable?
12. What is idempotency?
13. Why should automation have validation?
14. What is a dry run?
15. How would you safely automate file deletion?
16. How would you handle a failure halfway through an automation?
17. How can environment variables help automation?
18. Why should automation scripts be tested with sample data?
19. What are the risks of `shell=True`?
20. How would you design an automation that runs every day?

### Practical

21. How would you design a Downloads organizer?
22. How would you prevent duplicate file processing?
23. How would you recover from an interrupted automation?
24. How would you log successful and failed operations?
25. How would you make an automation safe to run multiple times?

---

# 49. Automation Cheat Sheet

## Find files

```
from pathlib import Path

folder = Path("files")

for file in folder.glob("*.pdf"):
    print(file)
```

## Recursive search

```
for file in folder.rglob("*.pdf"):
    print(file)
```

## Create folder

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

## Rename

```
file.rename(new_name)
```

## Move

```
import shutil

shutil.move(
    source,
    destination
)
```

## Copy

```
shutil.copy(
    source,
    destination
)
```

## Delete

```
file.unlink()
```

## Run external command

```
import subprocess

subprocess.run(
    ["python", "script.py"],
    check=True
)
```

## Environment variable

```
import os

value = os.environ["API_KEY"]
```

### Logging

```
import logging

logging.basicConfig(
    level=logging.INFO
)

logging.info("Task completed")
```

---

# 50. Automation Checklist

Before running an automation, ask:

```
[ ] Do I understand the manual process?
[ ] Are the rules clearly defined?
[ ] Have I tested it on sample data?
[ ] Are important files backed up?
[ ] Are paths validated?
[ ] Are errors handled?
[ ] Is logging enabled?
[ ] Are secrets protected?
[ ] Can it safely run twice?
[ ] Do I have a dry-run mode?
[ ] Does it have only the permissions it needs?
```

---

# 51. Key Takeaways

- Automation means turning repeatable work into a program.
- Good automation begins with understanding the manual workflow.
- `pathlib` is useful for filesystem paths and file discovery.
- `shutil` provides high-level file and directory operations.
- `subprocess` lets Python communicate with external programs.
- Automation can work with files, APIs, databases, CSV, JSON, and external programs.
- Separate classification, processing, and actions into functions.
- Validate before modifying real data.

- Test with controlled sample data.
- Logging makes unattended automation observable.
- Dry-run modes make destructive automation safer.
- Design automation to behave predictably when run repeatedly.
- Protect credentials and avoid unsafe shell execution.
- Scheduling determines **when** automation runs; Python determines **what** it does.
- Reliable automation is more than code that "works once."

## The Core Mental Model

```
Understand the process
        ↓
Identify repetition
        ↓
Define rules
        ↓
Build small functions
        ↓
Validate inputs
        ↓
Perform actions
        ↓
Verify results
        ↓
Log everything important
        ↓
Schedule if needed
```

**Automation is not about making Python do everything.**

It is about finding a repeatable process, defining its rules clearly, and making the computer perform those rules consistently.

haas.dev

https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.