Automating Files and Folders

1. What Does File and Folder Automation Mean?

File and folder automation means using Python to perform repetitive filesystem tasks automatically.

Instead of manually doing:

```
Find files
   ↓
Rename files
   ↓
Create folders
   ↓
Move files
   ↓
Copy backups
   ↓
Delete unnecessary files
```

Python can perform the same workflow using rules.

For example:

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

Python can automatically turn this into:

```
downloads/
    Images/
        photo.jpg
    Documents/
        report.pdf
    Presentations/
        presentation.pptx
    Music/
        song.mp3
```

The `pathlib` and `shutil` modules provide many of the standard tools used for filesystem automation.

---

2. Why Automate Files and Folders?

File management becomes repetitive surprisingly quickly.

Common examples include:

- organizing Downloads
- renaming hundreds of files
- creating project folders
- moving files by extension
- creating backups
- archiving old files
- finding large files
- finding duplicate files
- deleting temporary files
- collecting files from multiple folders
- generating directory reports
- processing files in batches

The key question is:

> **Can I describe the task using predictable rules?**

If yes, Python can often automate it.

---

3. The File Automation Mental Model

A useful model is:

```
DISCOVER
    ↓
FILTER
    ↓
CLASSIFY
    ↓
VALIDATE
    ↓
ACT
    ↓
VERIFY
```

For example:

```
Find all PDFs
    ↓
Ignore directories
    ↓
Classify by year
    ↓
Check destination
    ↓
Move file
    ↓
Verify destination exists
```

This is safer than immediately modifying files.

---

4. `pathlib`: The Main Tool

For modern Python filesystem work, `pathlib` is one of the most useful modules.

```
from pathlib import Path
```

Create a path:

```
folder = Path("downloads")
```

Check it:

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

A `Path` object represents a filesystem path and provides methods for navigating and manipulating it.

---

5. Building Paths

Instead of manually joining strings:

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

use:

```
from pathlib import Path

path = Path("downloads") / "reports" / "report.pdf"
```

This is clearer and works appropriately with the operating system's path conventions.

For example:

```
downloads = Path("downloads")

report = downloads / "reports" / "report.pdf"
```

---

6. Inspecting a Path

```
from pathlib import Path

file = Path("downloads/report.pdf")

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

For:

```
downloads/report.pdf
```

the results conceptually are:

```
name   → report.pdf
stem   → report
suffix → .pdf
parent → downloads
```

These properties are extremely useful when creating automation rules.

---

7. Checking Files and Directories

```
path.exists()
```

checks whether the path exists.

```
path.is_file()
```

checks whether it is a file.

```
path.is_dir()
```

checks whether it is a directory.

Example:

```
from pathlib import Path

path = Path("downloads")

if path.exists():
    if path.is_dir():
        print("Folder exists")
```

Never assume that a path exists just because your script expects it to.

---

8. Listing Folder Contents

```
from pathlib import Path

folder = Path("downloads")

for item in folder.iterdir():
    print(item)
```

This gives you the immediate contents of the folder.

You can distinguish files from folders:

```
for item in folder.iterdir():

    if item.is_file():
        print("File:", item)

    elif item.is_dir():
        print("Folder:", item)
```

---

9. Finding Specific Files With `glob()`

Suppose you only want PDFs:

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

Find Python files:

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

Find files starting with `report`:

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

This is useful when automation should operate only on matching files.

---

10. Recursive Searching With `rglob()`

Suppose your structure is:

```
project/
    reports/
        2025/
            report.pdf
    backups/
        old/
            report.pdf
```

You can search all nested folders:

```
project = Path("project")

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

`rglob()` recursively searches through subdirectories. Python's documentation notes that filesystem traversal can also be performed with `Path.walk()` in modern Python versions.

---

11. `Path.walk()`

For more advanced directory-tree processing:

```
from pathlib import Path

root = Path("project")

for directory, directories, files in root.walk():
    print("Directory:", directory)

    for file in files:
        print("File:", file)
```

This is useful when you need control over both directories and files while traversing a tree. You can also prevent certain directories from being traversed by modifying the directory list during a top-down walk.

---

12. Creating Folders

Create one folder:

```
from pathlib import Path

folder = Path("reports")

folder.mkdir()
```

If it may already exist:

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

For nested directories:

```
folder = Path("reports/2026/python")

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

`parents=True` creates missing parent directories as needed.

---

13. Creating Files

You can create an empty file:

```
from pathlib import Path

Path("report.txt").touch()
```

Or write text:

```
Path("report.txt").write_text(
    "Automation completed."
)
```

For binary files, use:

```
Path("data.bin").write_bytes(data)
```

---

14. Renaming Files

Suppose:

```
report_old.pdf
```

should become:

```
report_final.pdf
```

Use:

```
from pathlib import Path

file = Path("report_old.pdf")

file.rename("report_final.pdf")
```

A safer path-based version:

```
new_name = file.with_name("report_final.pdf")

file.rename(new_name)
```

---

15. Renaming Many Files

Suppose:

```
image1.jpg
image2.jpg
image3.jpg
```

should become:

```
photo_1.jpg
photo_2.jpg
photo_3.jpg
```

You can automate it:

```
from pathlib import Path

folder = Path("images")

for number, file in enumerate(
    folder.glob("*.jpg"),
    start=1
):
    new_name = file.with_name(
        f"photo_{number}.jpg"
```

```
    )

    file.rename(new_name)
```

However, file iteration order should not be assumed to be meaningful. If numbering must follow a specific order, sort the paths first.

```
files = sorted(folder.glob("*.jpg"))

for number, file in enumerate(files, start=1):
    ...
```

---

16. Changing Parts of a Filename

Suppose:

```
draft_report.pdf
draft_notes.pdf
draft_summary.pdf
```

You want to remove `draft_`.

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

    new_name = file.name.replace(
        "draft_",
        "",
        1
    )

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

This is useful for batch cleanup.

---

17. Adding Dates to Filenames

You can generate filenames dynamically.

```
from datetime import date
from pathlib import Path

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

```
today = date.today()

new_name = file.with_name(
    f"{file.stem}_{today}{file.suffix}"
)

file.rename(new_name)
```

Result:

```
report_2026-09-27.pdf
```

This can be useful for reports and backups.

---

18. Moving Files With `shutil`

`shutil` provides high-level operations for copying, moving, and removing files and directory trees.

```
import shutil

shutil.move(
    "report.pdf",
    "documents/report.pdf"
)
```

With `Path`:

```
from pathlib import Path
import shutil

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

shutil.move(source, destination)
```

---

19. Moving a File Safely

Before moving:

```
from pathlib import Path
import shutil

source = Path("report.pdf")
destination_folder = Path("documents")
```

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

destination = destination_folder / source.name

if source.exists() and not destination.exists():
    shutil.move(source, destination)
```

This avoids blindly moving onto an existing destination.

The exact overwrite behavior of `shutil.move()` depends on the destination and underlying filesystem semantics, so explicit collision handling is preferable in important automation.

---

20. Copying Files

Copy a file:

```
import shutil

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

`copy()` copies file contents and permission mode.

`copy2()` attempts to preserve additional metadata such as modification times:

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

However, Python's documentation notes that even `copy2()` cannot preserve every kind of filesystem metadata on every operating system.

---

21. Copying an Entire Folder

```
import shutil

shutil.copytree(
    "project",
```

```
    "project_backup"
)
```

This can be useful for creating a directory-level backup.

Be careful with large folders because the operation can copy a substantial amount of data.

---

22. Creating a Backup

A simple backup workflow:

```
project/
    main.py
    config.json
    data/
```

Create:

```
backups/
    project_backup/
```

Python:

```
from pathlib import Path
import shutil

source = Path("project")
backup = Path("backups/project_backup")

shutil.copytree(
    source,
    backup
)
```

For production backup systems, you would normally need additional concerns such as naming, retention, verification, storage capacity, and recovery testing.

---

23. Avoiding Backup Collisions

Instead of repeatedly using:

```
project_backup
```

generate unique backup names:

```
from datetime import datetime
from pathlib import Path
import shutil

timestamp = datetime.now().strftime(
    "%Y%m%d_%H%M%S"
)

destination = Path(
    f"backups/project_{timestamp}"
)

shutil.copytree(
    "project",
    destination
)
```

You might get:

```
backups/
    project_20260927_021500/
    project_20260928_021500/
```

---

24. Organizing Files by Extension

A common automation task is organizing a messy folder.

```
from pathlib import Path
import shutil

folder = Path("downloads")

categories = {
    ".pdf": "Documents",
    ".docx": "Documents",
    ".txt": "Documents",
    ".jpg": "Images",
    ".jpeg": "Images",
    ".png": "Images",
    ".mp4": "Videos",
    ".mp3": "Audio",
}
```

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

    if not file.is_file():
        continue

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

    if category is None:
        continue

    destination_folder = folder / category

    destination_folder.mkdir(
        exist_ok=True
    )

    destination = (
        destination_folder / file.name
    )

    if not destination.exists():
        shutil.move(file, destination)
```

This is one of the most practical beginner automation projects.

---

25. Organizing by Filename

File extensions are not the only possible rule.

Suppose:

```
python_functions.pdf
python_classes.pdf
javascript_arrays.pdf
javascript_objects.pdf
```

You could organize them by prefix:

```
Python/
    python_functions.pdf
    python_classes.pdf

JavaScript/
```

```
    javascript_arrays.pdf
    javascript_objects.pdf
```

Example:

```
categories = {
    "python_": "Python",
    "javascript_": "JavaScript",
}

for file in folder.iterdir():

    for prefix, category in categories.items():

        if file.name.lower().startswith(prefix):
            destination_folder = (
                folder / category
            )

            destination_folder.mkdir(
                exist_ok=True
            )

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

            break
```

---

26. Organizing by File Size

You can classify files by size.

```
from pathlib import Path

folder = Path("downloads")

for file in folder.iterdir():

    if not file.is_file():
        continue

    size = file.stat().st_size
```

```
    if size > 100 * 1024 * 1024:
        print(
            file.name,
            "is larger than 100 MB"
        )
```

This can become an automation:

```
Small files
    ↓
Keep

Large files
    ↓
Review / archive
```

---

27. File Metadata

`Path.stat()` provides filesystem metadata.

```
info = file.stat()

print(info.st_size)
print(info.st_mtime)
```

Commonly useful information includes:

```
size
modified time
access information
filesystem metadata
```

You can use modification time to find old files.

---

28. Finding Old Files

For example, identify files older than a certain number of days:

```
from pathlib import Path
from datetime import datetime, timedelta

folder = Path("downloads")

cutoff = datetime.now() - timedelta(days=30)
```

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

    if not file.is_file():
        continue

    modified = datetime.fromtimestamp(
        file.stat().st_mtime
    )

    if modified < cutoff:
        print("Old:", file)
```

The automation could then move these files to:

```
archive/
```

instead of immediately deleting them.

---

29. Archive Instead of Delete

A safer workflow is:

```
Old files
    ↓
Archive
    ↓
Keep for review
    ↓
Delete later if necessary
```

Instead of:

```
Old files
    ↓
DELETE
```

This creates a recovery path.

---

30. Deleting Files

A file can be deleted using:

```
file.unlink()
```

Example:

```
from pathlib import Path

file = Path("temporary.txt")

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

For directories:

```
directory.rmdir()
```

but `rmdir()` requires the directory to be empty.

For recursive directory removal, `shutil.rmtree()` exists, but it is highly destructive and should be used carefully. Python's documentation explicitly warns about the dangers of recursive deletion.

---

31. Safe Deletion

Do not immediately delete files when developing an automation.

First:

```
print(
    f"Would delete: {file}"
)
```

Then inspect the output.

You can introduce a dry-run flag:

```
DRY_RUN = True

if DRY_RUN:
    print(f"Would delete: {file}")
else:
    file.unlink()
```

Only change:

```
DRY_RUN = False
```

after testing.

---

32. Finding Duplicate Files

One useful automation is finding duplicate files.

File names alone are not enough.

These could contain identical data:

```
report.pdf
report_copy.pdf
final_report.pdf
```

A stronger approach is to calculate a hash.

```
import hashlib

def file_hash(path):
    hasher = hashlib.sha256()

    with path.open("rb") as file:
        while chunk := file.read(1024 * 1024):
            hasher.update(chunk)

    return hasher.hexdigest()
```

Then:

```
File A
  ↓
SHA-256
  ↓
hash

File B
  ↓
SHA-256
  ↓
hash
```

If the hashes match, the files have matching contents with respect to that hash algorithm.

For important systems, remember that a hash comparison is a tool for detecting matching content, not proof that two files are conceptually the same document.

---

33. Building a Duplicate Finder

```
from pathlib import Path
import hashlib

def file_hash(path):
    hasher = hashlib.sha256()

    with path.open("rb") as file:
        while chunk := file.read(1024 * 1024):
            hasher.update(chunk)

    return hasher.hexdigest()


folder = Path("downloads")

hashes = {}

for file in folder.rglob("*"):

    if not file.is_file():
        continue

    digest = file_hash(file)

    if digest in hashes:
        print("Possible duplicate:")
        print(" ", hashes[digest])
        print(" ", file)
    else:
        hashes[digest] = file
```

This is a more meaningful duplicate check than comparing filenames.

---

34. Optimizing Duplicate Detection

Hashing every large file immediately can be expensive.

A common strategy is:

```
Compare file size
       ↓
Different size?
       ↓
Not duplicate
```

```
        ↓
Same size?
        ↓
Calculate hash
        ↓
Compare hash
```

This reduces unnecessary hashing.

You can also group by:

```
file size
extension
hash
```

before deciding what requires deeper comparison.

---

35. Creating Archives

Python's `shutil` provides high-level archive creation.

```
import shutil

shutil.make_archive(
    "project_backup",
    "zip",
    "project"
)
```

This creates a ZIP archive.

Python's `shutil` archive helpers support formats such as ZIP and several tar-based formats.

---

36. Extracting Archives

```
import shutil

shutil.unpack_archive(
    "project_backup.zip",
    "restored_project"
)
```

The destination folder can be specified explicitly.

Be careful when extracting archives from untrusted sources, particularly when archive contents may contain unexpected paths or files.

---

37. Batch File Processing

Suppose a folder contains:

```
reports/
    report1.txt
    report2.txt
    report3.txt
```

You can process them all:

```
from pathlib import Path

folder = Path("reports")

for file in folder.glob("*.txt"):
    content = file.read_text()

    print(
        file.name,
        len(content)
    )
```

This pattern can be used for:

- conversion
- validation
- extraction
- renaming
- classification
- metadata generation

---

38. Generating a Folder Report

You can create a report about a directory.

```
from pathlib import Path

folder = Path("project")

files = [
    file
    for file in folder.rglob("*")
```

```
    if file.is_file()
]

print("Total files:", len(files))

total_size = sum(
    file.stat().st_size
    for file in files
)

print("Total bytes:", total_size)
```

The result can then be written to:

```
report.txt
```

or:

```
report.json
```

---

39. Handling Filename Collisions

Imagine:

```
documents/report.pdf
archive/report.pdf
```

You want to move both into:

```
backup/
```

You cannot safely assume both should become:

```
backup/report.pdf
```

A collision strategy can generate:

```
backup/report.pdf
backup/report_1.pdf
backup/report_2.pdf
```

Example:

```
from pathlib import Path

def unique_path(destination):
    if not destination.exists():
```

```
        return destination

    counter = 1

    while True:
        candidate = destination.with_name(
            f"{destination.stem}_{counter}"
            f"{destination.suffix}"
        )

        if not candidate.exists():
            return candidate

        counter += 1
```

Then:

```
destination = unique_path(
    folder / file.name
)
```

This prevents accidental overwriting.

---

40. File Automation With Functions

Avoid putting all operations into one large loop.

Instead:

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


def classify_file(file):
    ...


def get_destination(file, category):
    ...


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

```
def organize(folder):
    ...
```

This gives each part of the automation one responsibility.

---

41. Example: Maintainable Organizer

```
from pathlib import Path
import shutil

CATEGORIES = {
    ".pdf": "Documents",
    ".docx": "Documents",
    ".jpg": "Images",
    ".png": "Images",
    ".mp4": "Videos",
}


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


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

    target = destination / file.name

    if target.exists():
        return False

    shutil.move(file, target)

    return True


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

```
        if not file.is_file():
            continue

        category = classify(file)

        if category is None:
            continue

        destination = folder / category

        move_file(file, destination)
```

The important separation is:

```
classify()
    ↓
What should happen?

move_file()
    ↓
How should it happen?

organize()
    ↓
When should it happen?
```

---

42. Logging File Operations

For larger automation, record operations.

```
import logging

logging.basicConfig(
    filename="file_automation.log",
    level=logging.INFO
)

logging.info(
    "Moved %s to %s",
    source,
    destination
)
```

A useful log can tell you:

```
2026-09-27 02:15
Moved report.pdf → Documents/report.pdf

2026-09-27 02:15
Skipped existing file

2026-09-27 02:16
Failed to move image.jpg
```

This is particularly useful when hundreds or thousands of files are processed.

---

43. Error Handling

Filesystem operations can fail because:

- file does not exist
- permission denied
- destination is unavailable
- file is locked
- path is invalid
- disk is full
- another process changes the filesystem

Handle expected failures:

```
try:
    shutil.move(source, destination)
except FileNotFoundError:
    print("Source file no longer exists")
except PermissionError:
    print("Permission denied")
```

Do not silently ignore all exceptions.

---

44. Dry Run Architecture

A good automation can separate planning from execution.

```
def move_file(
    source,
    destination,
    dry_run=True
```

```
):
    if dry_run:
        print(
            f"Would move {source} → {destination}"
        )
        return

    shutil.move(
        source,
        destination
    )
```

This allows:

```
Plan
 ↓
Review
 ↓
Execute
```

instead of:

```
Execute immediately
 ↓
Discover mistakes later
```

---

45. Avoid Following Unexpected Directories

Recursive automation needs careful boundaries.

For example:

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

can process far more data than expected if `folder` is incorrectly chosen.

Always establish a clear root:

```
ROOT = Path("test_data")
```

and verify it:

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

```
if not ROOT.is_dir():
    raise NotADirectoryError(ROOT)
```

Then operate only inside that root.

---

46. A Safer Automation Pattern

Use:

```
1. Define root directory
2. Verify root
3. Discover candidates
4. Filter candidates
5. Calculate intended destinations
6. Check collisions
7. Dry-run
8. Execute
9. Verify
10. Log
```

This pattern is more reliable than:

```
Find everything
 ↓
Change everything
```

---

47. Real World Example: Project Cleanup

Suppose:

```
project/
    main.py
    notes.txt
    screenshot.png
    debug.log
    old/
    __pycache__/
```

You want:

```
project/
    src/
    docs/
    assets/
```

```
    logs/
    archive/
```

Python can classify files and directories:

```
.py       → src/
.txt      → docs/
.png      → assets/
.log      → logs/
old/      → archive/
```

The important part is not the individual `move()` call.

The important part is defining the rules before implementing them.

---

48. Real World Example: haas.dev Resources

Imagine a folder containing:

```
resources/
    python_functions.pdf
    python_classes.pdf
    oop_basics.pdf
    javascript_arrays.pdf
    api_design.pdf
```

An automation could:

```
Discover PDFs
     ↓
Read filename
     ↓
Identify technology
     ↓
Create category folder
     ↓
Move PDF
     ↓
Generate metadata JSON
```

Result:

```
resources/
    Python/
        python_functions.pdf
```

```
        python_classes.pdf

    JavaScript/
        javascript_arrays.pdf

    Backend/
        api_design.pdf

    OOP/
        oop_basics.pdf
```

This becomes especially useful when a resource library grows to hundreds of files.

---

49. Mini Project: Smart File Organizer

Build a complete file organizer.

Input

```
downloads/
```

Possible files:

```
photo.jpg
report.pdf
presentation.pptx
song.mp3
video.mp4
archive.zip
unknown.xyz
```

Output

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

Requirements

Your program should:

1. Ignore directories.
2. Classify files by extension.
3. Create destination folders.
4. Detect filename collisions.
5. Support dry-run mode.
6. Log operations.
7. Handle permission and missing-file errors.
8. Never delete files automatically.
9. Be safe to run repeatedly.
10. Print a final summary.

Example:

```
Processed: 7
Moved: 6
Skipped: 1
Failed: 0
```

---

50. Five Minute Challenge

Create a script that scans:

```
test_files/
```

and prints:

```
Filename
Extension
Size
```

Example:

```
report.pdf
Extension: .pdf
Size: 125 KB
```

Do not move, rename, copy, or delete anything.

The goal is to practice **read-only filesystem automation** first.

---

51. Exercises

Exercise 1: Batch Rename

Rename:

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

to:

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

Exercise 2: Image Organizer

Move all:

```
.jpg
.jpeg
.png
```

files into:

```
Images/
```

Exercise 3: Old File Detector

Find all files older than 30 days.

Do not delete them.

Print them for review.

Exercise 4: Backup

Create a timestamped backup of:

```
project/
```

inside:

```
backups/
```

Exercise 5: Duplicate Finder

Find files with identical:

```
size + SHA-256 hash
```

and print possible duplicates.

Do not delete anything.

Exercise 6: Resource Organizer

Create a script that organizes:

```
resources/
```

into folders based on filename prefixes such as:

```
python_
javascript_
flutter_
```

---

52. Knowledge Check

1. What is `pathlib` used for?

Answer: Working with filesystem paths and filesystem operations using `Path` objects.

2. What does `glob()` do?

Answer: Finds paths matching a pattern.

3. What is `rglob()` useful for?

Answer: Recursively finding matching paths inside subdirectories.

4. What does `shutil.move()` do?

Answer: Moves a file or directory to another location.

5. What is the difference between `copy()` and `copy2()`?

Answer: `copy2()` attempts to preserve additional metadata such as modification times, while `copy()` copies data and permission mode.

6. Why should a file automation script have a dry-run mode?

Answer: It lets you inspect planned changes before actually modifying files.

7. Why should filename collisions be handled?

Answer: To prevent unintended overwriting or failed operations.

8. Why can hashing help find duplicates?

Answer: Matching hashes can identify files with matching contents, subject to the properties and limitations of the chosen hash.

---

53. Interview Questions

Beginner

1. What is `pathlib`?
2. How do you check whether a path exists?
3. How do you distinguish a file from a directory?
4. What does `Path.iterdir()` do?
5. What is `glob()`?
6. What is `rglob()`?
7. How do you create a directory with Python?
8. How do you rename a file?
9. How do you move a file?
10. What is `shutil`?

Intermediate

11. What is the difference between `copy()`, `copy2()`, and `copyfile()`?
12. How would you recursively search a directory?
13. How would you prevent filename collisions?
14. How would you safely delete files?
15. How would you find old files?
16. How would you identify duplicate files?
17. Why can file size be used before hashing?
18. How would you design a dry-run mode?
19. How would you create a timestamped backup?
20. Why should destructive filesystem automation be tested on sample data?

Practical

21. How would you build a Downloads organizer?
22. How would you organize files by date?
23. How would you design a batch-renaming tool?
24. How would you recover from a failed move operation?
25. How would you make a file automation safe to run repeatedly?

---

54. File Automation Cheat Sheet

Create a path

```
from pathlib import Path
```

```
path = Path("reports/report.pdf")
```

Check existence

```
path.exists()
```

Check file

```
path.is_file()
```

Check directory

```
path.is_dir()
```

List contents

```
path.iterdir()
```

Find matching files

```
path.glob("*.pdf")
```

Recursive search

```
path.rglob("*.pdf")
```

Create folder

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

Rename

```
path.rename(new_path)
```

Delete file

```
path.unlink()
```

Move

```
import shutil

shutil.move(
```

```
    source,
    destination
)
```

Copy

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

Copy with more metadata

```
shutil.copy2(
    source,
    destination
)
```

Copy directory

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

Create ZIP archive

```
shutil.make_archive(
    "backup",
    "zip",
    "project"
)
```

Extract archive

```
shutil.unpack_archive(
    "backup.zip",
    "restored"
)
```

---

55. Best Practices

1. Start with read-only operations

First inspect:

```
print(file)
```

Then modify.

2. Use `pathlib`

Prefer:

```
Path("reports") / "report.pdf"
```

over manual string path construction.

3. Define a clear root directory

Know exactly where your automation is allowed to operate.

4. Use dry runs

Preview changes before executing them.

5. Handle collisions

Never assume the destination is empty.

6. Log important actions

Especially when processing many files.

7. Back up important data

Do not experiment on irreplaceable files.

8. Separate logic from actions

Keep classification and filesystem changes independent.

9. Test on sample directories

Create:

```
test_data/
```

before using real data.

10. Make repeated execution predictable

Ask:

> What happens if I run this script twice?

---

56. Key Takeaways

- File automation turns repetitive filesystem work into repeatable Python programs.
- `pathlib` provides a clean interface for working with paths.

- `iterdir()` lists directory contents.
- `glob()` finds matching paths.
- `rglob()` searches recursively.
- `mkdir()` creates directories.
- `rename()` renames files and directories.
- `shutil` provides high-level copy, move, removal, and archive operations.
- `copy2()` can preserve more metadata than `copy()`, but filesystem metadata preservation is platform-dependent.
- File organization can be based on extension, filename, size, date, or other rules.
- Hashes can help identify files with matching contents.
- Backups and archives are safer alternatives to immediately deleting data.
- Filename collisions must be handled explicitly.
- Dry-run modes are valuable for destructive or large-scale automation.
- Recursive operations require carefully defined boundaries.
- Reliable automation should validate, act, verify, and log.

The Core Mental Model

```
Choose the root
      ↓
Discover files
      ↓
Filter candidates
      ↓
Classify them
      ↓
Validate the action
      ↓
Preview if possible
      ↓
Move / copy / rename / archive
      ↓
Verify result
      ↓
Log what happened
```

The goal is not simply to make Python manipulate files.

The goal is to make filesystem operations **predictable, safe, repeatable, and maintainable**.

haas.dev

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

Visit haas.dev for more resources and guides.